guide-R

Guide pour l'analyse de données d'enquêtes avec R

Joseph Larmarange

10 juillet 2023

Table des matières

Préface

> ⚠ Guide en cours d'écriture
>
> Ce guide est encore incomplet. Le plan prévu est visible à cette adresse : https://github.com/larmarange/guide-R/issues/12.
> En attendant, nous vous conseillons de compléter votre lecture par le site analyse-R.

Ce guide porte sur l'analyse de données d'enquêtes avec le logiciel **R**, un logiciel libre de statitistiques et de traitement de données. Les exemples présentés ici relèvent principalement du champs des sciences sociales quantitatives et des sciences de santé. Ils peuvent néanmoins s'appliquer à d'autre champs disciplinaires. Cependant, comme tout ouvrage, ce guide ne peut être exhaustif.

Ce guide présente comment réaliser des analyses statistiques et diverses opérations courantes (comme la manipulation de données ou la production de graphiques) avec **R**. Il ne s'agit pas d'un cours de statistiques : les différents chapitres présupposent donc que vous avez déjà une connaissance des différentes techniques présentées. Si vous souhaitez des précisions théoriques / méthodologiques à propos d'un certain type d'analyses, nous vous conseillons d'utiliser votre moteur de recherche préféré. En effet, on trouve sur internet de très nombreux supports de cours (sans compter les nombreux ouvrages spécialisés disponibles en librairie).

De même, il ne s'agit pas d'une introduction ou d'un guide pour les utilisatrices et utilisateurs débutant · es. Si vous découvrez **R**, nous vous conseillons la lecture de l'*Introduction à R et au tidyverse* de Julien Barnier (https://juba.github.io/tidyverse/). Vous pouvez également lire les chapitres introductifs d'*analyse-R : Introduction à l'analyse d'enquêtes avec R et RStudio* (https:

//larmarange.github.io/analyse-R/). Néanmoins, quelques rappels sur les bases du langage sont fournis dans la section *Bases du langage*. Une bonne compréhension de ces dernières, bien qu'un peu ardue de prime abord, permet de comprendre le sens des commandes qu'on utilise et de pleinement exploiter la puissance que **R** offre en matière de manipulation de données.

R disposent de nombreuses extensions ou packages (plus de 16 000) et il existe souvent plusieurs manières de procéder pour arriver au même résultat. En particulier, en matière de manipulation de données, on oppose[1] souvent *base R* qui repose sur les fonctions disponibles en standard dans **R**, la majorité étant fournies dans les packages `{base}`, `{utils}` ou encore `{stats}`, qui sont toujours chargés par défaut, et le `{tidyverse}` qui est une collection de packages comprenant, entre autres, `{dplyr}`, `{tibble}`, `{tidyr}`, `{forcats}` ou encore `{ggplot2}`. Il y a un débat ouvert, parfois passionné, sur le fait de privilégier l'une ou l'autre approche, et les avantages et inconvénients de chacune dépendent de nombreux facteurs, comme la lisibilité du code ou bien les performances en temps de calcul. Dans ce guide, nous avons adopté un point de vue pragmatique et utiliserons, le plus souvent mais pas exclusivement, les fonctions du `{tidyverse}`, de même que nous avons privilégié d'autres packages, comme `{gtsummary}` ou `{questionr}` par exemple pour la statistique descriptive. Cela ne signifie pas, pour chaque point abordé, qu'il s'agit de l'unique manière de procéder. Dans certains cas, il s'agit simplement de préférences personnelles.

[1] Une comparaison des deux syntaxes est illustrée par une vignette dédiée de dplyr.

Bien qu'il en reprenne de nombreux contenus, ce guide ne se substitue pas au site analyse-R. Il s'agit plutôt d'une version complémentaire qui a vocation à être plus structurée et parfois plus sélective dans les contenus présentés.

En complément, on pourra également se référer aux webin-R, une série de vidéos avec partage d'écran, librement accessibles sur Youtube : https://www.youtube.com/c/webinR.

Cette version du guide a utilisé *R version 4.2.3 (2023-03-15 ucrt)*. Ce document est généré avec quarto et le code source est disponible sur GitHub. Pour toute suggestion ou correction, vous pouvez ouvrir un ticket GitHub. Pour d'autres questions, vous pouvez utiliser les forums de discussion disponibles en bas de chaque page sur la version web du guide. Ce document est

régulièrement mis à jour. La dernière version est consultable sur https://larmarange.github.io/guide-R/.

Remerciements

Ce document a bénéficié de différents apports provenant notamment de l'*Introduction à R* et de l'*Introduction à R et au tidyverse* de Julien Barnier et d'analyse-R : introduction à l'analyse d'enquêtes avec R et RStudio.

Merci donc à Julien Barnier, Julien Biaudet, François Briatte, Milan Bouchet-Valat, Ewen Gallic, Frédérique Giraud, Joël Gombin, Mayeul Kauffmann, Christophe Lalanne & Nicolas Robette.

Licence

Ce document est mis à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International.

partie I

Bases du langage

1 Packages

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 en anglais) pouvant être installées par l'utilisateur et fournissant alors diverses fonctionnalités supplémentaires.

Il existe un très grand nombre d'extensions (plus de 16 000 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.

On peut aussi citer le site *Awesome R* (https://github.com/qinwf/awesome-R) qui fournit une liste d'extensions choisies et triées par thématique.

1.1 Installation (CRAN)

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

```
install.packages("gtsummary")
```

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

Alternativement, on pourra avoir recours au package `{remotes}` et à sa fonction `remotes::install_cran()` :

```
remotes::install_cran("gtsummary")
```

> **Note**
>
> Le package `{remotes}` n'est pas disponible par défaut sous **R** et devra donc être installé classiquement avec `install.packages("remotes")`. À la différence de `install.packages()`, `remotes::install_cran()` vérifie si le package est déjà installé et, si oui, si la version installée est déjà la dernière version, avant de procéder à une installation complète si et seulement si cela est nécessaire.

1.2 Chargement

Une fois un package installé (c'est-à-dire que ses fichiers ont eté téléchargés et copiés sur votre ordinateur), ses fonctions et objets ne sont pas directement accessibles. Pour pouvoir les utiliser, il faut, **à chaque session de travail**, charger le package en mémoire avec la fonction `library()` ou la fonction `require()` :

```
library(gtsummary)
```

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.

Alternativement, pour accéder à un objet ou une fonction d'un package sans avoir à le charger en mémoire, on pourra avoir recours à l'opérateur `::`. Ainsi, l'écriture `p::f()` signifie la fonction `f()` du package `p`. Cette écriture sera notamment utilisée tout au long de ce guide pour indiquer à quel package appartient telle fonction : `remotes::install_cran()` indique que la fonction `install_cran()` provient du packages `{remotes}`.

> **!** Important
>
> Il est important de bien comprendre la différence entre `install.packages()` et `library()`. La première va chercher un package sur internet et l'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.

1.3 Mise à jour

Pour mettre à jour l'ensemble des pacakges installés, il suffit d'exécuter la fonction `update.packages()` :

```
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("gtsummary")
```

💡 Installer / Mettre à jour les packages utilisés par un projet

Après une mise à jour majeure de **R**, il est souvent nécessaire de réinstaller tous les packages utilisés. De même, on peut parfois souhaiter mettre à jour uniquement les packages utilisés par un projet donné sans avoir à mettre à jour tous les autres packages présents sur son PC.
Une astuce consiste à avoir recours à la fonction `renv::dependencies()` qui examine le code du projet courant pour identifier les packages utilisés, puis à passer cette liste de packages à `remotes::install_cran()` qui installera les packages manquants ou pour lesquels une mise à jour est disponible.
Il vous suffit d'exécuter la commande ci-dessous :

```
renv::dependencies() |>
  purrr::pluck("Package") |>
  remotes::install_cran()
```

1.4 Installation depuis GitHub

Certains packages ne sont pas disponibles sur **CRAN** mais seulement sur **GitHub**, une plateforme de développement informatique. Il s'agit le plus souvent de packages qui ne sont pas encore suffisament matures pour être diffusés sur **CRAN** (sachant que des vérifications strictes sont effectués avant qu'un package ne soit référencés sur **CRAN**).

Dans d'autres cas de figure, la dernière version stable d'un package est disponible sur **CRAN** tandis que la version en cours de développement est, elle, disponible sur **GitHub**. Il fuat être vigilant avec les versions de développement. Parfois, elle corrige un bug ou introduit une nouvelle fonctionnalité qui n'est pas encore dans la version stable. Mais les versions de développement peuvent aussi contenir de nouveaux bugs ou des fonctionnalités instables.

Sous Windows

Pour les utilisatrices et utilisateurs sous **Windows**, il faut être conscient que le code source d'un package doit être compilé afin de pouvoir être utilisé. **CRAN** fournit une version des packages déjà compilée pour **Windows** ce qui facilite l'installation.
Par contre, lorsque l'on installe un package depuis **GitHub**, **R** ne récupère que le code source et il est donc nécessaire de compiler localement le package. Pour cela, il est nécessaire que soit installé sur le PC un outil complémentaire appelé **RTools**. Il est téléchargeable à l'adresse https://cran.r-project.org/bin/windows/Rtools/.

Le code source du package `{labelled}` est disponible sur **GitHub** à l'adresse https://github.com/larmarange/labelled. Pour installer la version de développement de `{labelled}`,on aura recours à la fonction `remotes::install_github()` à laquelle on passera la partie située à droite de `https://github.com/` dans l'URL du package, à savoir :

```
remotes::install_github("larmarange/labelled")
```

1.5 Le tidyverse

Le terme `{tidyverse}` est une contraction de *tidy* (qu'on pourrait traduire par bien rangé) et de *universe*. Il s'agit en fait d'une collection de packages conçus pour travailler ensemble et basés sur une philosophie commune.

Ils abordent un très grand nombre d'opérations courantes dans **R** (la liste n'est pas exhaustive) :

- visualisation (`{ggplot2}`)
- manipulation des tableaux de données (`{dplyr}`, `{tidyr}`)
- import/export de données (`{readr}`, `{readxl}`, `{haven}`)
- manipulation de variables (`{forcats}`, `{stringr}`, `{lubridate}`)

- programmation (`{purrr}`, `{magrittr}`, `{glue}`)

Un des objectifs de ces extensions est de fournir des fonctions avec une syntaxe cohérente, qui fonctionnent bien ensemble, et qui retournent des résultats prévisibles. Elles sont en grande partie issues du travail d'Hadley Wickham, qui travaille désormais pour RStudio.

`{tidyverse}` est également le nom d'une extension générique qui permets d'installer en une seule commande l'ensemble des packages constituant le *tidyverse* :

```
install.packages("tidyverse")
```

Lorsque l'on charge le package `{tidyverse}` avec `library()`, cela charge également en mémoire les principaux packages du *tidyverse*[2].

```
library(tidyverse)
```

[2] Si on a besoin d'un autre package du *tidyverse* comme `{lubridate}`, il faudra donc le charger individuellement.

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.1     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.1     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

Figure 1.1: Packages chargés avec `library(tidyverse)`

2 Vecteurs

Les vecteurs sont l'objet de base de **R** et correspondent à une liste de valeurs. Leurs propriétés fondamentales sont :

- les vecteurs sont unidimensionnels (i.e. ce sont des objets à une seule dimension, à la différence d'une matrice par exemple) ;
- toutes les valeurs d'un vecteur sont d'un seul et même type ;
- les vecteurs ont une longueur qui correspond au nombre de valeurs contenues dans le vecteur.

2.1 Types et classes

Dans **R**, il existe plusieurs types fondamentaux de vecteurs et, en particulier, :

- les nombres réels (c'est-à-dire les nombres décimaux[3]), par exemple `5.23` ;
- les nombres entiers, que l'on saisi en ajoutant le suffixe `L`[4], par exemple `4L` ;
- les chaînes de caractères (qui correspondent à du texte), que l'on saisit avec des guillemets doubles (`"`) ou simples (`'`), par exemple `"abc"` ;
- les valeurs logiques ou valeurs booléennes, à savoir vrai ou faux, que l'on représente avec les mots `TRUE` et `FALSE` (en majuscules[5]).

En plus de ces types de base, il existe de nombreux autres types de vecteurs utilisés pour représenter toutes sortes de données, comme les facteurs (voir Chapitre **??**) ou les dates (voir Chapitre **??**).

[3] Pour rappel, **R** étant anglophone, le caractère utilisé pour indiqué les chiffres après la virgule est le point (`.`).

[4] **R** utilise 32 bits pour représenter des nombres entiers, ce qui correspond en informatique à des entiers longs ou *long integers* en anglais, d'où la lettre `L` utilisée pour indiquer un nombre entier.

[5] On peut également utiliser les raccourcis `T` et `F`. Cependant, pour une meilleure lisibilité du code, il est préférable d'utiliser les versions longues `TRUE` et `FALSE`.

La fonction `class()` renvoie la nature d'un vecteur tandis que la fonction `typeof()` indique la manière dont un vecteur est stocké de manière interne par **R**.

Table 2.1: Le type et la classe des principaux types de vecteurs

x	class(x)	typeof(x)
3L	integer	integer
5.3	numeric	double
TRUE	logical	logical
"abc"	character	character
factor("a")	factor	integer
as.Date("2020-01-01")	Date	double

> Astuce
>
> Pour un vecteur numérique, le type est `"double"` car **R** utilise une double précision pour stocker informatiquement les nombres réels.
> En interne, les facteurs sont représentés par un nombre entier auquel est attaché une étiquette, c'est pourquoi `typeof()` renvoie `"integer"`.
> Quand aux dates, elles sont stockées en interne sous la forme d'un nombre réel représentant le nombre de jours depuis le 1er janvier 1970, d'où le fait que `typeof()` renvoie `"double"`.

2.2 Création d'un vecteur

Pour créer un vecteur, on utilisera la fonction `c()` en lui passant la liste des valeurs à combiner[6].

[6] La lettre `c` est un raccourci du mot anglais *combine*, puisque cette fonction permet de combiner des valeurs individuelles dans un vecteur unique.

```
taille <- c(1.88, 1.65, 1.92, 1.76, NA, 1.72)
taille
```

```
[1] 1.88 1.65 1.92 1.76   NA 1.72
```

```r
sexe <- c("h", "f", "h", "f", "f", "f")
sexe
```

```
[1] "h" "f" "h" "f" "f" "f"
```

```r
urbain <- c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE)
urbain
```

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

Nous l'avons vu, toutes les valeurs d'un vecteur doivent obligatoirement être du même type. Dès lors, si on essaie de combiner des valeurs de différents types, **R** essaiera de les convertir au mieux. Par exemple :

```r
x <- c(2L, 3.14, "a")
x
```

```
[1] "2"    "3.14" "a"
```

```r
class(x)
```

```
[1] "character"
```

Dans le cas présent, toutes les valeurs ont été converties en chaînes de caractères.

Dans certaines situations, on peut avoir besoin de créer un vecteur d'une certaine longueur mais dont toutes les valeurs sont identiques. Cela se réalise facilement avec `rep()` à qui on indiquera la valeur à répéter puis le nombre de répétitions :

```r
rep(2, 10)
```

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

On peut aussi lui indiquer plusieurs valeurs qui seront alors répétées en boucle :

```r
rep(c("a", "b"), 3)
```

```
[1] "a" "b" "a" "b" "a" "b"
```

Dans d'autres situations, on peut avoir besoin de créer un vecteur contenant une suite de valeurs, ce qui se réalise aisément avec `seq()` à qui on précisera les arguments `from` (point de départ), `to` (point d'arrivée) et `by` (pas). Quelques exemples valent mieux qu'un long discours :

```r
seq(1, 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(5, 17, by = 2)
```

```
[1]  5  7  9 11 13 15 17
```

```r
seq(10, 0)
```

```
 [1] 10  9  8  7  6  5  4  3  2  1  0
```

```r
seq(100, 10, by = -10)
```

```
 [1] 100  90  80  70  60  50  40  30  20  10
```

```r
seq(1.23, 5.67, by = 0.33)
```

```
 [1] 1.23 1.56 1.89 2.22 2.55 2.88 3.21 3.54 3.87 4.20 4.53 4.86 5.19 5.52
```

L'opérateur `:` est un raccourci de la fonction `seq()` pour créer une suite de nombres entiers. Il s'utilise ainsi :

```r
1:5
```

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

```r
24:32
```

```
[1] 24 25 26 27 28 29 30 31 32
```

```r
55:43
```

```
 [1] 55 54 53 52 51 50 49 48 47 46 45 44 43
```

2.3 Longueur d'un vecteur

La longueur d'un vecteur correspond au nombre de valeurs qui le composent. Elle s'obtient avec `length()` :

```r
length(taille)
```

```
[1] 6
```

```r
length(c("a", "b"))
```

```
[1] 2
```

La longueur d'un vecteur vide (`NULL`) est zéro.

```r
length(NULL)
```

```
[1] 0
```

2.4 Combiner des vecteurs

Pour combiner des vecteurs, rien de plus simple. Il suffit d'utiliser `c()` ! Les valeurs des différents vecteurs seront mises bout à bout pour créer un unique vecteur.

```
x <- c(2, 1, 3, 4)
length(x)
```

```
[1] 4
```

```
y <- c(9, 1, 2, 6, 3, 0)
length(y)
```

```
[1] 6
```

```
z <- c(x, y)
z
```

```
 [1] 2 1 3 4 9 1 2 6 3 0
```

```
length(z)
```

```
[1] 10
```

2.5 Vecteurs nommés

Les différentes valeurs d'un vecteur peuvent être nommées. Une première manière de nommer les éléments d'un vecteur est de le faire à sa création :

```
sexe <- c(
  Michel = "h", Anne = "f",
  Dominique = NA, Jean = "h",
```

```
  Claude = NA, Marie = "f"
)
```

Lorsqu'on affiche le vecteur, la présentation change quelque peu.

```
sexe
```

```
   Michel      Anne Dominique      Jean    Claude     Marie
      "h"       "f"        NA       "h"        NA       "f"
```

La liste des noms s'obtient avec `names()`.

```
names(sexe)
```

```
[1] "Michel"    "Anne"      "Dominique" "Jean"      "Claude"    "Marie"
```

Pour ajouter ou modifier les noms d'un vecteur, on doit attribuer un nouveau vecteur de noms :

```
names(sexe) <- c("Michael", "Anna", "Dom", "John", "Alex", "Mary")
sexe
```

```
Michael    Anna     Dom    John    Alex    Mary
    "h"     "f"      NA     "h"      NA     "f"
```

Pour supprimer tous les noms, il y a la fonction `unname()` :

```
anonyme <- unname(sexe)
anonyme
```

```
[1] "h" "f" NA  "h" NA  "f"
```

2.6 Indexation par position

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 de valeurs en fonction de différents critères. Il existe trois types d'indexation : (i) l'indexation par position, (ii) l'indexation par nom et (iii) l'indexation par condition. Le principe est toujours le même : on indique entre crochets[7] (`[]`) ce qu'on souhaite garder ou non.

[7] Pour rappel, les crochets s'obtiennent sur un clavier français de type PC en appuyant sur la touche Alt Gr et la touche (ou).

Commençons par l'indexation par position encore appelée indexation directe. Ce mode le plus simple d'indexation consiste à indiquer la position des éléments à conserver.

Reprenons notre vecteur `taille` :

```
taille
```

```
[1] 1.88 1.65 1.92 1.76   NA 1.72
```

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

```
taille[1]
```

```
[1] 1.88
```

Si on souhaite les trois premiers éléments ou les éléments 2, 5 et 6 :

```
taille[1:3]
```

```
[1] 1.88 1.65 1.92
```

```
taille[c(2, 5, 6)]
```

```
[1] 1.65   NA 1.72
```

Si on veut le dernier élément :

```
taille[length(taille)]
```

```
[1] 1.72
```

Il est tout à fait possible de sélectionner les valeurs dans le désordre :

```
taille[c(5, 1, 4, 3)]
```

```
[1]   NA 1.88 1.76 1.92
```

Dans le cadre de l'indexation par position, il est également possible de spécifier des nombres négatifs, auquel cas cela signifiera toutes les valeurs sauf celles-là. Par exemple :

```
taille[c(-1, -5)]
```

```
[1] 1.65 1.92 1.76 1.72
```

À noter, si on indique une position au-delà de la longueur du vecteur, **R** renverra `NA`. Par exemple :

```
taille[23:25]
```

```
[1] NA NA NA
```

2.7 Indexation par nom

Lorsqu'un vecteur est nommé, il est dès lors possible d'accéder à ses valeurs à partir de leur nom. Il s'agit de l'indexation par nom.

```
sexe["Anna"]
```

```
Anna
 "f"
```

```r
sexe[c("Mary", "Michael", "John")]
```

```
   Mary Michael    John
    "f"     "h"     "h"
```

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

```r
sexe[names(sexe) != "Dom"]
```

… pour sélectionner tous les éléments sauf celui qui s'appelle Dom.

2.8 Indexation par condition

L'indexation par condition consiste à fournir un vecteur logique indiquant si chaque élément doit être inclus (si `TRUE`) ou exclu (si `FALSE`). Par exemple :

```r
sexe
```

```
Michael    Anna     Dom    John    Alex    Mary
    "h"     "f"      NA     "h"      NA     "f"
```

```r
sexe[c(TRUE, FALSE, FALSE, TRUE, FALSE, FALSE)]
```

```
Michael    John
    "h"     "h"
```

Écrire manuellement une telle condition n'est pas très pratique à l'usage. Mais supposons que nous ayons également à notre disposition les deux vecteurs suivants, également de longueur 6.

```
urbain <- c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE)
poids <- c(80, 63, 75, 87, 82, 67)
```

Le vecteur `urbain` est un vecteur logique. On peut directement l'utiliser pour avoir le sexe des enquêtés habitant en milieu urbain :

```
sexe[urbain]
```

```
Michael    Anna    Mary
    "h"     "f"     "f"
```

Supposons qu'on souhaite maintenant avoir la taille des individus pesant 80 kilogrammes ou plus. Nous pouvons effectuer une comparaison à l'aide des opérateurs de comparaison suivants :

Table 2.2: Opérateurs de comparaison

Opérateur de comparaison	Signification
`==`	égal à
`%in%`	appartient à
`!=`	différent de
`>`	strictement supérieur à
`<`	strictement inférieur à
`>=`	supérieur ou égal à
`<=`	inférieur ou égal à

Voyons tout de suite un exemple :

```
poids >= 80
```

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

Que s'est-il passé ? Nous avons fourni à **R** une condition et il nous a renvoyé un vecteur logique avec autant d'éléments qu'il y a d'observations et dont la valeur est `TRUE` si la condition

est remplie et `FALSE` dans les autres cas. Nous pouvons alors utiliser ce vecteur logique pour obtenir la taille des participants pesant 80 kilogrammes ou plus :

```
taille[poids >= 80]
```

```
[1] 1.88 1.76   NA
```

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

Table 2.3: Opérateurs logiques

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

Supposons que je veuille identifier les personnes pesant 80 kilogrammes ou plus **et** vivant en milieu urbain :

```
poids >= 80 & urbain
```

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

Les résultats sont différents si je souhaite isoler les personnes pesant 80 kilogrammes ou plus **ou** vivant milieu urbain :

```
poids >= 80 | urbain
```

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

> **!** Comparaison et valeur manquante
>
> 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.

```
taille
```

```
[1] 1.88 1.65 1.92 1.76   NA 1.72
```

```
taille > 1.8
```

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

On voit que le test `NA > 1.8` ne renvoie ni vrai ni faux, mais `NA`.
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()` :

```
is.na(taille > 1.8)
```

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

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 a donc des conséquences sur le résultat d'une indexation par comparaison.
Par exemple si je cherche à connaître le poids des personnes mesurant 1,80 mètre ou plus :

```
taille
```

```
[1] 1.88 1.65 1.92 1.76   NA 1.72
```

```
poids
```

```
[1] 80 63 75 87 82 67
```

```
poids[taille > 1.8]
```

```
[1] 80 75 NA
```

Les éléments pour lesquels la taille n'est pas connue ont été transformés en `NA`, ce qui n'influera pas le calcul d'une moyenne. Par contre, lorsqu'on utilisera assignation et indexation ensemble, cela peut créer des problèmes. Il est donc préférable lorsqu'on a des valeurs manquantes de les exclure ainsi :

```
poids[taille > 1.8 & !is.na(taille)]
```

```
[1] 80 75
```

2.9 Assignation par indexation

L'indexation peut être combinée avec l'assignation (opérateur `<-`) pour modifier seulement certaines parties d'un vecteur. Ceci fonctionne pour les différents types d'indexation évoqués précédemment.

```
v <- 1:5
v
```

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

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

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

```
sexe["Alex"] <- "non-binaire"
sexe
```

```
Michael          Anna           Dom          John          Alex
    "h"           "f"            NA           "h" "non-binaire"
   Mary
    "f"
```

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 :

```
sexe[c(1,3,4)] <- c("Homme", "Homme", "Homme")
sexe[c(1,3,4)] <- "Homme"
```

L'assignation par indexation peut aussi être utilisée pour ajouter une ou plusieurs valeurs à un vecteur :

```
length(sexe)
```

```
[1] 6
```

```
sexe[7] <- "f"
sexe
```

```
    Michael          Anna           Dom          John          Alex
   "Homme"           "f"        "Homme"       "Homme" "non-binaire"
       Mary
        "f"           "f"
```

```
length(sexe)
```

```
[1] 7
```

2.10 En résumé

- Un vecteur est un objet unidimensionnel contenant une liste de valeurs qui sont toutes du même type (entières, numériques, textuelles ou logiques).
- La fonction `class()` permet de connaître le type du vecteur et la fonction `length()` sa longueur, c'est-à-dire son nombre d'éléments.
- La fonction `c()` sert à créer et à combiner des vecteurs.

- Les valeurs manquantes sont représentées avec `NA`.
- Un vecteur peut être nommé, c'est-à-dire qu'un nom textuel a été associé à chaque élément. Cela peut se faire lors de sa création ou avec la fonction `names()`.
- L'indexation consiste à extraire certains éléments d'un vecteur. Pour cela, on indique ce qu'on souhaite extraire entre crochets (`[]`) juste après le nom du vecteur. Le type d'indexation dépend du type d'information transmise.
- S'il s'agit de nombres entiers, c'est l'indexation par position : les nombres représentent la position dans le vecteur des éléments qu'on souhaite extraire. Un nombre négatif s'interprète comme tous les éléments sauf celui-là.
- Si on indique des chaînes de caractères, c'est l'indexation par nom : on indique le nom des éléments qu'on souhaite extraire. Cette forme d'indexation ne fonctionne que si le vecteur est nommé.
- Si on transmet des valeurs logiques, le plus souvent sous la forme d'une condition, c'est l'indexation par condition : `TRUE` indique les éléments à extraire et `FALSE` les éléments à exclure. Il faut être vigilant aux valeurs manquantes (`NA`) dans ce cas précis.
- Enfin, il est possible de ne modifier que certains éléments d'un vecteur en ayant recours à la fois à l'indexation (`[]`) et à l'assignation (`<-`).

2.11 webin-R

On pourra également se référer au webin-R #02 (*les bases du langage R*) sur YouTube.

https://youtu.be/Eh8piunoqQc

3 Listes

Par nature, les vecteurs ne peuvent contenir que des valeurs de même type (numérique, textuel ou logique). Or, on peut avoir besoin de représenter des objets plus complexes composés d'éléments disparates. C'est ce que permettent les listes.

3.1 Propriétés et création

Une liste se crée tout simplement avec la fonction `list()` :

```
l1 <- list(1:5, "abc")
l1
```

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

[[2]]
[1] "abc"
```

Une liste est un ensemble d'objets, quels qu'ils soient, chaque élément d'une liste pouvant avoir ses propres dimensions. Dans notre exemple précédent, nous avons créé une liste `l1` composée de deux éléments : un vecteur d'entiers de longueur 5 et un vecteur textuel de longueur 1. La longueur d'une liste correspond aux nombres d'éléments qu'elle contient et s'obtient avec `length()` :

```
length(l1)
```

```
[1] 2
```

Comme les vecteurs, une liste peut être nommée et les noms des éléments d'une liste sont accessibles avec `names()` :

```
l2 <- list(
  minuscules = letters,
  majuscules = LETTERS,
  mois = month.name
)
l2
```

```
$minuscules
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

$majuscules
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

$mois
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

```
length(l2)
```

```
[1] 3
```

```
names(l2)
```

```
[1] "minuscules" "majuscules" "mois"
```

Que se passe-t-il maintenant si on effectue la commande suivante ?

```
l <- list(l1, l2)
```

À votre avis, quelle est la longueur de cette nouvelle liste `l` ? 5 ?

```
length(l)
```

```
[1] 2
```

Eh bien non ! Elle est de longueur 2 car nous avons créé une liste composée de deux éléments qui sont eux-mêmes des listes. Cela est plus lisible si on fait appel à la fonction `str()` qui permet de visualiser la structure d'un objet.

```
str(l)
```

```
List of 2
 $ :List of 2
  ..$ : int [1:5] 1 2 3 4 5
  ..$ : chr "abc"
 $ :List of 3
  ..$ minuscules: chr [1:26] "a" "b" "c" "d" ...
  ..$ majuscules: chr [1:26] "A" "B" "C" "D" ...
  ..$ mois      : chr [1:12] "January" "February" "March" "April" ...
```

Une liste peut contenir tous types d'objets, y compris d'autres listes. Pour combiner les éléments d'une liste, il faut utiliser la fonction `append()` :

```
l <- append(l1, l2)
length(l)
```

```
[1] 5
```

```
str(l)
```

```
List of 5
 $           : int [1:5] 1 2 3 4 5
 $           : chr "abc"
 $ minuscules: chr [1:26] "a" "b" "c" "d" ...
 $ majuscules: chr [1:26] "A" "B" "C" "D" ...
 $ mois      : chr [1:12] "January" "February" "March" "April" ...
```

> **Note**
>
> On peut noter en passant qu'une liste peut tout à fait n'être que partiellement nommée.

3.2 Indexation

Les crochets simples (`[]`) fonctionnent comme pour les vecteurs. On peut utiliser à la fois l'indexation par position, l'indexation par nom et l'indexation par condition.

```
l
```

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

[[2]]
[1] "abc"

$minuscules
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

$majuscules
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

$mois
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

```
l[c(1,3,4)]
```

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

$minuscules
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

$majuscules
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
l[c("majuscules", "minuscules")]
```

```
$majuscules
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

$minuscules
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
l[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
```

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

[[2]]
[1] "abc"

$mois
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

Même si on extrait un seul élément, l'extraction obtenue avec les crochets simples renvoie toujours une liste, ici composée d'un seul élément :

```
str(l[1])
```

```
List of 1
 $ : int [1:5] 1 2 3 4 5
```

Supposons que je souhaite calculer la moyenne des valeurs du premier élément de ma liste. Essayons la commande suivante :

```
mean(l[1])
```

```
Warning in mean.default(l[1]): l'argument n'est ni numérique, ni logique :
renvoi de NA
```

```
[1] NA
```

Nous obtenons un message d'erreur. En effet, **R** ne sait pas calculer une moyenne à partir d'une liste. Ce qu'il lui faut, c'est un vecteur de valeurs numériques. Autrement dit, ce que nous cherchons à obtenir c'est le contenu même du premier élément de notre liste et non une liste à un seul élément.

C'est ici que les doubles crochets (`[[]]`) vont rentrer en jeu. Pour ces derniers, nous pourrons utiliser l'indexation par position ou l'indexation par nom, mais pas l'indexation par condition. De plus, le critère qu'on indiquera doit indiquer **un et un seul** élément de notre liste. Au lieu de renvoyer une liste à un élément, les doubles crochets vont renvoyer l'élément désigné.

```
str(l[1])
```

```
List of 1
 $ : int [1:5] 1 2 3 4 5
```

```
str(l[[1]])
```

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

Maintenant, nous pouvons calculer notre moyenne :

```
mean(l[[1]])
```

```
[1] 3
```

Nous pouvons aussi utiliser l'indexation par nom.

```
l[["mois"]]
```

```
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

Mais il faut avouer que cette écriture avec doubles crochets et guillemets est un peu lourde. Heureusement, un nouvel acteur entre en scène : le symbole dollar (`$`). C'est un raccourci des doubles crochets pour l'indexation par nom qu'on utilise ainsi :

```
l$mois
```

```
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

Les écritures `l$mois` et `l[["mois"]]` sont équivalentes. Attention ! Cela ne fonctionne que pour l'indexation par nom.

```
l$1
```

```
Error: unexpected numeric constant in "l$1"
```

L'assignation par indexation fonctionne également avec les doubles crochets ou le signe dollar :

```
l[[2]] <- list(c("un", "vecteur", "textuel"))
l$mois <- c("Janvier", "Février", "Mars")
l
```

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

[[2]]
[[2]][[1]]
[1] "un"      "vecteur" "textuel"
```

```
$minuscules
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

$majuscules
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

$mois
[1] "Janvier" "Février" "Mars"
```

3.3 En résumé

- Les listes sont des objets unidimensionnels pouvant contenir tout type d'objet, y compris d'autres listes.
- Elles ont une longueur qu'on obtient avec `length()`.
- On crée une liste avec `list()` et on peut fusionner des listes avec `append()`.
- Tout comme les vecteurs, les listes peuvent être nommées et les noms des éléments s'obtiennent avec `base::names()`.
- Les crochets simples (`[]`) permettent de sélectionner les éléments d'une liste, en utilisant l'indexation par position, l'indexation par nom ou l'indexation par condition. Cela renvoie toujours une autre liste.
- Les doubles crochets (`[[]]`) renvoient directement le contenu d'un élément de la liste qu'on aura sélectionné par position ou par nom.
- Le symbole `$` est un raccourci pour facilement sélectionner un élément par son nom, `liste$nom` étant équivalent à `liste[["nom"]]`.

3.4 webin-R

On pourra également se référer au webin-R #02 (*les bases du langage R*) sur YouTube.

https://youtu.be/Eh8piunoqQc

4 Tableaux de données

Les tableaux de données, ou *data frame* en anglais, est un type d'objets essentiel pour les données d'enquêtes.

4.1 Propriétés et création

Dans **R**, les tableaux de données sont tout simplement des listes (voir Chapitre **??**) avec quelques propriétés spécifiques :

- les tableaux de données ne peuvent contenir que des vecteurs ;
- tous les vecteurs d'un tableau de données ont la même longueur ;
- tous les éléments d'un tableau de données sont nommés et ont chacun un nom unique.

Dès lors, un tableau de données correspond aux fichiers de données qu'on a l'habitude de manipuler dans d'autres logiciels de statistiques comme **SPSS** ou **Stata**. Les variables sont organisées en colonnes et les observations en lignes.

On peut créer un tableau de données avec la fonction `data.frame()` :

```
df <- data.frame(
  sexe =  c("f", "f", "h", "h"),
  age = c(52, 31, 29, 35),
  blond = c(FALSE, TRUE, TRUE, FALSE)
)
df
```

```
  sexe age blond
1    f  52 FALSE
```

```
2    f  31  TRUE
3    h  29  TRUE
4    h  35 FALSE
```

```
str(df)
```

```
'data.frame':   4 obs. of  3 variables:
 $ sexe : chr  "f" "f" "h" "h"
 $ age  : num  52 31 29 35
 $ blond: logi  FALSE TRUE TRUE FALSE
```

Un tableau de données étant une liste, la fonction `length()` renverra le nombre d'éléments de la liste, donc dans le cas présent le nombre de variables, et `names()` leurs noms :

```
length(df)
```

```
[1] 3
```

```
names(df)
```

```
[1] "sexe"  "age"   "blond"
```

Comme tous les éléments d'un tableau de données ont la même longueur, cet objet peut être vu comme bidimensionnel. Les fonctions `nrow()`, `ncol()` et `dim()` donnent respectivement le nombre de lignes, le nombre de colonnes et les dimensions de notre tableau.

```
nrow(df)
```

```
[1] 4
```

```
ncol(df)
```

```
[1] 3
```

```r
dim(df)
```

```
[1] 4 3
```

De plus, tout comme les colonnes ont un nom, il est aussi possible de nommer les lignes avec `row.names()` :

```r
row.names(df) <- c("Anna", "Mary-Ann", "Michael", "John")
df
```

```
         sexe age blond
Anna        f  52 FALSE
Mary-Ann    f  31  TRUE
Michael     h  29  TRUE
John        h  35 FALSE
```

4.2 Indexation

Les tableaux de données étant des listes, nous pouvons donc utiliser les crochets simples (`[]`), les crochets doubles (`[[]]`) et le symbole dollar (`$`) pour extraire des parties de notre tableau, de la même manière que pour n'importe quelle liste.

```r
df[1]
```

```
         sexe
Anna        f
Mary-Ann    f
Michael     h
John        h
```

```r
df[[1]]
```

```
[1] "f" "f" "h" "h"
```

```
df$sexe
```

```
[1] "f" "f" "h" "h"
```

Cependant, un tableau de données étant un objet bidimensionnel, il est également possible d'extraire des données sur deux dimensions, à savoir un premier critère portant sur les lignes et un second portant sur les colonnes. Pour cela, nous utiliserons les crochets simples (`[]`) en séparant nos deux critères par une virgule (`,`).

Un premier exemple :

```
df
```

```
         sexe age blond
Anna        f  52 FALSE
Mary-Ann    f  31  TRUE
Michael     h  29  TRUE
John        h  35 FALSE
```

```
df[3, 2]
```

```
[1] 29
```

Cette première commande indique que nous souhaitons la troisième ligne de la seconde colonne, autrement dit l'âge de Michael. Le même résultat peut être obtenu avec l'indexation par nom, l'indexation par condition, ou un mélange de tout ça.

```
df["Michael", "age"]
```

```
[1] 29
```

```
df[c(F, F, T, F), c(F, T, F)]
```

```
[1] 29
```

```
df[3, "age"]
```

```
[1] 29
```

```
df["Michael", 2]
```

```
[1] 29
```

Il est également possible de préciser un seul critère. Par exemple, si je souhaite les deux premières observations, ou les variables *sexe* et *blond* :

```
df[1:2,]
```

```
         sexe age blond
Anna        f  52 FALSE
Mary-Ann    f  31  TRUE
```

```
df[,c("sexe", "blond")]
```

```
         sexe blond
Anna        f FALSE
Mary-Ann    f  TRUE
Michael     h  TRUE
John        h FALSE
```

Il a suffi de laisser un espace vide avant ou après la virgule.

> **Avertissement**
>
> ATTENTION ! Il est cependant impératif de laisser la virgule pour indiquer à **R** qu'on souhaite effectuer une indexation à deux dimensions. Si on oublie la virgule, cela nous ramène au mode de fonctionnement des listes. Et le résultat n'est pas forcément le même :

```
  df[2, ]
```

```
         sexe age blond
Mary-Ann    f  31  TRUE
```

```
  df[, 2]
```

```
[1] 52 31 29 35
```

```
  df[2]
```

```
         age
Anna      52
Mary-Ann  31
Michael   29
John      35
```

Note

Au passage, on pourra noter quelques subtilités sur le résultat renvoyé.

```
  str(df[2, ])
```

```
'data.frame':   1 obs. of  3 variables:
 $ sexe : chr "f"
 $ age  : num 31
 $ blond: logi TRUE
```

```
  str(df[, 2])
```

```
 num [1:4] 52 31 29 35
```

```
  str(df[2])
```

```
'data.frame':   4 obs. of  1 variable:
 $ age: num  52 31 29 35
```

```
str(df[[2]])
```

```
 num [1:4] 52 31 29 35
```

`df[2, ]` signifie qu'on veut toutes les variables pour le second individu. Le résultat est un tableau de données à une ligne et trois colonnes. `df[2]` correspond au mode d'extraction des listes et renvoie donc une liste à un élément, en l'occurrence un tableau de données à quatre observations et une variable. `df[[2]]` quant à lui renvoie le contenu de cette variable, soit un vecteur numérique de longueur quatre. Reste `df[, 2]` qui renvoie toutes les observations pour la seconde colonne. Or l'indexation bidimensionnelle a un fonctionnement un peu particulier : par défaut elle renvoie un tableau de données mais s'il y a une seule variable dans l'extraction, c'est un vecteur qui est renvoyé. Pour plus de détails, on pourra consulter l'entrée d'aide `help("[.data.frame")`.

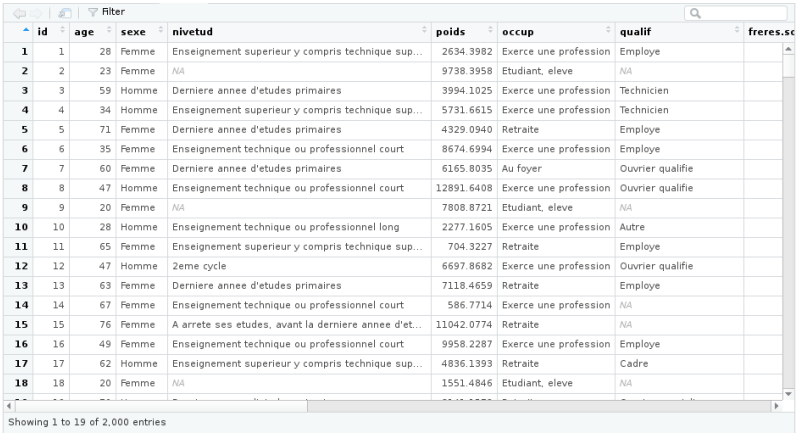
4.3 Afficher les données

Prenons un tableau de données un peu plus conséquent, en l'occurrence le jeu de données `?questionr::hdv2003` disponible dans l'extension `{questionr}` et correspondant à un extrait de l'enquête *Histoire de vie* réalisée par l'INSEE en 2003. Il contient 2000 individus et 20 variables.

```
library(questionr)
data(hdv2003)
```

Si on demande d'afficher l'objet `hdv2003` dans la console (résultat non reproduit ici), **R** va afficher l'ensemble du contenu de `hdv2003` à l'écran ce qui, sur un tableau de cette taille, ne sera pas très lisible. Pour une exploration visuelle, le plus simple est souvent d'utiliser la visionneuse intégrée à **RStudio** et qu'on peut appeler avec la fonction `View()`.

```
View(hdv2003)
```



Filter

	id	age	sexe	nivetud	poids	occup	qualif	freres.sc
1	1	28	Femme	Enseignement superieur y compris technique sup...	2634.3982	Exerce une profession	Employe	
2	2	23	Femme	NA	9738.3958	Etudiant, eleve	NA	
3	3	59	Homme	Derniere annee d'etudes primaires	3994.1025	Exerce une profession	Technicien	
4	4	34	Homme	Enseignement superieur y compris technique sup...	5731.6615	Exerce une profession	Technicien	
5	5	71	Femme	Derniere annee d'etudes primaires	4329.0940	Retraite	Employe	
6	6	35	Femme	Enseignement technique ou professionnel court	8674.6994	Exerce une profession	Employe	
7	7	60	Femme	Derniere annee d'etudes primaires	6165.8035	Au foyer	Ouvrier qualifie	
8	8	47	Homme	Enseignement technique ou professionnel court	12891.6408	Exerce une profession	Ouvrier qualifie	
9	9	20	Femme	NA	7808.8721	Etudiant, eleve	NA	
10	10	28	Homme	Enseignement technique ou professionnel long	2277.1605	Exerce une profession	Autre	
11	11	65	Femme	Enseignement superieur y compris technique sup...	704.3227	Retraite	Employe	
12	12	47	Homme	2eme cycle	6697.8682	Exerce une profession	Ouvrier qualifie	
13	13	63	Femme	Derniere annee d'etudes primaires	7118.4659	Retraite	Employe	
14	14	67	Femme	Enseignement technique ou professionnel court	586.7714	Exerce une profession	NA	
15	15	76	Femme	A arrete ses etudes, avant la derniere annee d'et...	11042.0774	Retraite	NA	
16	16	49	Femme	Enseignement technique ou professionnel court	9958.2287	Exerce une profession	Employe	
17	17	62	Homme	Enseignement superieur y compris technique sup...	4836.1393	Retraite	Cadre	
18	18	20	Femme	NA	1551.4846	Etudiant, eleve	NA	

Showing 1 to 19 of 2,000 entries

Figure 4.1: Interface View() de R RStudio

Les fonctions `head()` et `tail()`, qui marchent également sur les vecteurs, permettent d'afficher seulement les premières (respectivement les dernières) lignes d'un tableau de données :

```
head(hdv2003)
```

```
  id age  sexe                                                nivetud    poids
1  1  28 Femme Enseignement superieur y compris technique superieur 2634.398
2  2  23 Femme                                                  <NA> 9738.396
3  3  59 Homme                       Derniere annee d'etudes primaires 3994.102
4  4  34 Homme Enseignement superieur y compris technique superieur 5731.662
5  5  71 Femme                       Derniere annee d'etudes primaires 4329.094
6  6  35 Femme          Enseignement technique ou professionnel court 8674.699
                   occup     qualif freres.soeurs clso
1 Exerce une profession    Employe             8  Oui
2       Etudiant, eleve       <NA>             2  Oui
3 Exerce une profession Technicien             2  Non
4 Exerce une profession Technicien             1  Non
5              Retraite    Employe             0  Oui
6 Exerce une profession    Employe             5  Non
                         relig                   trav.imp    trav.satisf
1 Ni croyance ni appartenance               Peu important Insatisfaction
2 Ni croyance ni appartenance                        <NA>           <NA>
```

```
3 Ni croyance ni appartenance Aussi important que le reste      Equilibre
4  Appartenance sans pratique Moins important que le reste   Satisfaction
5         Pratiquant regulier                          <NA>           <NA>
6 Ni croyance ni appartenance            Le plus important      Equilibre
  hard.rock lecture.bd peche.chasse cuisine bricol cinema sport heures.tv
1       Non        Non          Non     Oui    Non    Non   Non         0
2       Non        Non          Non     Non    Non    Oui   Oui         1
3       Non        Non          Non     Non    Non    Non   Oui         0
4       Non        Non          Non     Oui    Oui    Oui   Oui         2
5       Non        Non          Non     Non    Non    Non   Non         3
6       Non        Non          Non     Non    Non    Oui   Oui         2
```

```r
tail(hdv2003, 2)
```

```
       id age  sexe                                        nivetud     poids
1999 1999  24 Femme Enseignement technique ou professionnel court 13740.810
2000 2000  66 Femme  Enseignement technique ou professionnel long  7709.513
                    occup  qualif freres.soeurs clso
1999 Exerce une profession Employe             2  Non
2000              Au foyer Employe             3  Non
                         relig                     trav.imp trav.satisf
1999 Appartenance sans pratique Moins important que le reste   Equilibre
2000 Appartenance sans pratique                          <NA>        <NA>
     hard.rock lecture.bd peche.chasse cuisine bricol cinema sport heures.tv
1999       Non        Non          Non     Non    Non    Oui   Non       0.3
2000       Non        Oui          Non     Oui    Non    Non   Non       0.0
```

L'extension `{dplyr}` propose une fonction `dplyr::glimpse()` (ce qui signifie aperçu en anglais) qui permet de visualiser rapidement et de manière condensée le contenu d'un tableau de données.

```r
library(dplyr)
glimpse(hdv2003)
```

```
Rows: 2,000
Columns: 20
$ id            <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~
```

```
$ age           <int> 28, 23, 59, 34, 71, 35, 60, 47, 20, 28, 65, 47, 63, 67, ~
$ sexe          <fct> Femme, Femme, Homme, Homme, Femme, Femme, Femme, Homme, ~
$ nivetud       <fct> "Enseignement superieur y compris technique superieur", ~
$ poids         <dbl> 2634.3982, 9738.3958, 3994.1025, 5731.6615, 4329.0940, 8~
$ occup         <fct> "Exerce une profession", "Etudiant, eleve", "Exerce une ~
$ qualif        <fct> Employe, NA, Technicien, Technicien, Employe, Employe, O~
$ freres.soeurs <int> 8, 2, 2, 1, 0, 5, 1, 5, 4, 2, 3, 4, 1, 5, 2, 3, 4, 0, 2,~
$ clso          <fct> Oui, Oui, Non, Non, Oui, Non, Oui, Non, Oui, Non, Oui, O~
$ relig         <fct> Ni croyance ni appartenance, Ni croyance ni appartenance~
$ trav.imp      <fct> Peu important, NA, Aussi important que le reste, Moins i~
$ trav.satisf   <fct> Insatisfaction, NA, Equilibre, Satisfaction, NA, Equilib~
$ hard.rock     <fct> Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, N~
$ lecture.bd    <fct> Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, N~
$ peche.chasse  <fct> Non, Non, Non, Non, Non, Non, Oui, Oui, Non, Non, Non, N~
$ cuisine       <fct> Oui, Non, Non, Oui, Non, Non, Oui, Oui, Non, Non, Oui, N~
$ bricol        <fct> Non, Non, Non, Oui, Non, Non, Non, Oui, Non, Non, Oui, O~
$ cinema        <fct> Non, Oui, Non, Oui, Non, Oui, Non, Non, Oui, Oui, Oui, N~
$ sport         <fct> Non, Oui, Oui, Oui, Non, Oui, Non, Non, Non, Oui, Non, O~
$ heures.tv     <dbl> 0.0, 1.0, 0.0, 2.0, 3.0, 2.0, 2.9, 1.0, 2.0, 2.0, 1.0, 0~
```

L'extension {labelled} propose une fonction `labelled::look_for()` qui permet de lister les différentes variables d'un fichier de données :

```
library(labelled)
look_for(hdv2003)
```

```
pos variable      label col_type missing
1   id            -     int      0
2   age           -     int      0
3   sexe          -     fct      0

4   nivetud       -     fct      112
```

```
5   poids          -     dbl      0
6   occup          -     fct      0

7   qualif         -     fct      347

8   freres.soeurs -      int      0
9   clso           -     fct      0

10  relig          -     fct      0

11  trav.imp       -     fct      952

12  trav.satisf    -     fct      952

13  hard.rock      -     fct      0

14  lecture.bd     -     fct      0

15  peche.chasse   -     fct      0

16  cuisine        -     fct      0

17  bricol         -     fct      0
```

```
18  cinema        -     fct      0

19  sport         -     fct      0

20  heures.tv     -     dbl      5
values


Homme
Femme
N'a jamais fait d'etudes
A arrete ses etudes, avant la derniere ann~
Derniere annee d'etudes primaires
1er cycle
2eme cycle
Enseignement technique ou professionnel co~
Enseignement technique ou professionnel lo~
Enseignement superieur y compris technique~

Exerce une profession
Chomeur
Etudiant, eleve
Retraite
Retire des affaires
Au foyer
Autre inactif
Ouvrier specialise
Ouvrier qualifie
Technicien
Profession intermediaire
Cadre
Employe
Autre

Oui
Non
Ne sait pas
Pratiquant regulier
Pratiquant occasionnel
Appartenance sans pratique
```

```
Ni croyance ni appartenance
Rejet
NSP ou NVPR
Le plus important
Aussi important que le reste
Moins important que le reste
Peu important
Satisfaction
Insatisfaction
Equilibre
Non
Oui
Non
Oui
Non
Oui
Non
Oui
Non
Oui
Non
Oui
Non
Oui
```

Lorsqu'on a un gros tableau de données avec de nombreuses variables, il peut être difficile de retrouver la ou les variables d'intérêt. Il est possible d'indiquer à `labelled::look_for()` un mot-clé pour limiter la recherche. Par exemple :

```
look_for(hdv2003, "trav")
```

```
pos variable    label col_type missing values
11  trav.imp    -     fct      952     Le plus important
                                       Aussi important que le reste
                                       Moins important que le reste
                                       Peu important
12  trav.satisf -     fct      952     Satisfaction
                                       Insatisfaction
```

Il est à noter que si la recherche n'est pas sensible à la casse (i.e. aux majuscules et aux minuscules), elle est sensible aux accents.

La méthode `summary()` qui fonctionne sur tout type d'objet permet d'avoir quelques statistiques de base sur les différentes variables de notre tableau, les statistiques affichées dépendant du type de variable.

```
summary(hdv2003)
```

```
      id              age           sexe
 Min.   :   1.0   Min.   :18.00   Homme: 899
 1st Qu.: 500.8   1st Qu.:35.00   Femme:1101
 Median :1000.5   Median :48.00
 Mean   :1000.5   Mean   :48.16
 3rd Qu.:1500.2   3rd Qu.:60.00
 Max.   :2000.0   Max.   :97.00

                                                 nivetud        poids
 Enseignement technique ou professionnel court       :463   Min.   :   78.08
 Enseignement superieur y compris technique superieur:441   1st Qu.: 2221.82
 Derniere annee d'etudes primaires                   :341   Median : 4631.19
 1er cycle                                           :204   Mean   : 5535.61
 2eme cycle                                          :183   3rd Qu.: 7626.53
 (Other)                                             :256   Max.   :31092.14
 NA's                                                :112
                   occup                                qualif    freres.soeurs
 Exerce une profession:1049   Employe                    :594   Min.   : 0.000
 Chomeur              : 134   Ouvrier qualifie           :292   1st Qu.: 1.000
 Etudiant, eleve      :  94   Cadre                      :260   Median : 2.000
 Retraite             : 392   Ouvrier specialise         :203   Mean   : 3.283
 Retire des affaires  :  77   Profession intermediaire:160   3rd Qu.: 5.000
 Au foyer             : 171   (Other)                    :144   Max.   :22.000
 Autre inactif        :  83   NA's                       :347
         clso                                relig
 Oui        : 936   Pratiquant regulier        :266
 Non        :1037   Pratiquant occasionnel     :442
```

```
Ne sait pas:  27   Appartenance sans pratique :760
                   Ni croyance ni appartenance:399
                   Rejet                       : 93
                   NSP ou NVPR                 : 40

                        trav.imp           trav.satisf  hard.rock  lecture.bd
Le plus important           : 29   Satisfaction  :480   Non:1986   Non:1953
Aussi important que le reste:259   Insatisfaction:117   Oui:  14   Oui:  47
Moins important que le reste:708   Equilibre     :451
Peu important               : 52   NA's          :952
NA's                        :952


peche.chasse cuisine    bricol     cinema     sport        heures.tv
Non:1776     Non:1119   Non:1147   Non:1174   Non:1277   Min.   : 0.000
Oui: 224     Oui: 881   Oui: 853   Oui: 826   Oui: 723   1st Qu.: 1.000
                                                         Median : 2.000
                                                         Mean   : 2.247
                                                         3rd Qu.: 3.000
                                                         Max.   :12.000
                                                         NA's   :5
```

On peut également appliquer `summary()` à une variable particulière.

```
summary(hdv2003$sexe)
```

```
Homme Femme
  899  1101
```

```
summary(hdv2003$age)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.00   35.00   48.00   48.16   60.00   97.00
```

4.4 En résumé

- Les tableaux de données sont des listes avec des propriétés particulières :
  i. tous les éléments sont des vecteurs ;
  ii. tous les vecteurs ont la même longueur ;
  iii. tous les vecteurs ont un nom et ce nom est unique.
- On peut créer un tableau de données avec `data.frame()`.
- Les tableaux de données correspondent aux fichiers de données qu'on utilise usuellement dans d'autres logiciels de statistiques : les variables sont représentées en colonnes et les observations en lignes.
- Ce sont des objets bidimensionnels : `ncol()` renvoie le nombre de colonnes et `nrow()` le nombre de lignes.
- Les doubles crochets (`[[]]`) et le symbole dollar (`$`) fonctionnent comme pour les listes et permettent d'accéder aux variables.
- Il est possible d'utiliser des coordonnées bidimensionnelles avec les crochets simples (`[]`) en indiquant un critère sur les lignes puis un critère sur les colonnes, séparés par une virgule (`,`).

4.5 webin-R

On pourra également se référer au webin-R #02 (*les bases du langage R*) sur YouTube.

https://youtu.be/Eh8piunoqQc

5 Tibbles

5.1 Le concept de tidy data

Le `{tidyverse}` est en partie fondé sur le concept de *tidy data*, développé à l'origine par Hadley Wickham dans un article de 2014 du *Journal of Statistical Software*.

Il s'agit d'un modèle d'organisation des données qui vise à faciliter le travail souvent long et fastidieux de nettoyage et de préparation préalable à la mise en oeuvre de méthodes d'analyse.

Les principes d'un jeu de données *tidy* sont les suivants :

1. chaque variable est une colonne
2. chaque observation est une ligne
3. chaque type d'observation est dans une table différente

Un chapitre dédié à `{tidyr}` (voir Chapitre **??**) présente comment définir et rendre des données *tidy* avec ce package.

Les extensions du `{tidyverse}`, notamment `{ggplot2}` et `{dplyr}`, sont prévues pour fonctionner avec des données *tidy*.

5.2 tibbles : des tableaux de données améliorés

Une autre particularité du `{tidyverse}` est que ces extensions travaillent avec des tableaux de données au format `tibble::tibble()`, qui est une évolution plus moderne du classique `data.frame` de **R** de base.

Ce format est fourni est géré par l'extension du même nom (`{tibble}`), qui fait partie du coeur du *tidyverse*. La plupart des

fonctions des extensions du *tidyverse* acceptent des *data.frames* en entrée, mais retournent un *tibble*.

Contrairement aux *data frames*, les *tibbles* :

- n'ont pas de noms de lignes (*rownames*)
- autorisent des noms de colonnes invalides pour les *data frames* (espaces, caractères spéciaux, nombres…) [8]
- s'affichent plus intelligemment que les *data frames* : seules les premières lignes sont affichées, ainsi que quelques informations supplémentaires utiles (dimensions, types des colonnes…)
- ne font pas de *partial matching* sur les noms de colonnes [9]
- affichent un avertissement si on essaie d'accéder à une colonne qui n'existe pas

[8] Quand on veut utiliser des noms de ce type, on doit les entourer avec des *backticks* (‘)

[9] Dans **R** base, si une table `d` contient une colonne `qualif`, `d$qual` retournera cette colonne.

Pour autant, les tibbles restent compatibles avec les *data frames*.

Il est possible de créer un *tibble* manuellement avec `tibble::tibble()`.

```
library(tidyverse)
tibble(
  x = c(1.2345, 12.345, 123.45, 1234.5, 12345),
  y = c("a", "b", "c", "d", "e")
)
```

```
# A tibble: 5 x 2
         x y
     <dbl> <chr>
1     1.23 a
2    12.3  b
3   123.   c
4  1234.   d
5 12345    e
```

On peut ainsi facilement convertir un *data frame* en tibble avec `tibble::as_tibble()` :

```
d <- as_tibble(mtcars)
d
```

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# i 22 more rows
```

D'ailleurs, quand on regarde la classe d'un tibble, on peut s'apercevoir qu'un tibble hérite de la classe `data.frame` mais possède en plus la classe `tbl_df`. Cela traduit bien le fait que les *tibbles* restent des *data frames*.

```
class(d)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Si le *data frame* d'origine a des *rownames*, on peut d'abord les convertir en colonnes avec `tibble::rownames_to_column()` :

```
d <- as_tibble(rownames_to_column(mtcars))
d
```

```
# A tibble: 32 x 12
   rowname       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 Mazda RX4    21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2 Mazda RX4 ~  21       6  160    110  3.9   2.88  17.0     0     1     4     4
```

```
 3 Datsun 710   22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4 Hornet 4 D~  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5 Hornet Spo~  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6 Valiant      18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7 Duster 360   14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8 Merc 240D    24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9 Merc 230     22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10 Merc 280     19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# i 22 more rows
```

À l'inverse, on peut à tout moment convertir un tibble en *data frame* avec `tibble::as.data.frame()` :

```
as.data.frame(d)
```

```
               rowname  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1            Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
2        Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
3           Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
4       Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
5    Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
6              Valiant 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
7           Duster 360 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
8            Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
9             Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
10            Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
11           Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
12          Merc 450SE 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
13          Merc 450SL 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
14         Merc 450SLC 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
15  Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
16 Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
17   Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
18            Fiat 128 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
19         Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
20      Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
21       Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
22    Dodge Challenger 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
23         AMC Javelin 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
24          Camaro Z28 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
```

```
25    Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
26           Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
27       Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
28        Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
29      Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
30        Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
31       Maserati Bora 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
32          Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Là encore, on peut convertir la colonne *rowname* en "vrais" *rownames* avec `tibble::column_to_rownames()` :

```
column_to_rownames(as.data.frame(d))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
```

```
Fiat X1-9          27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2      26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa       30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L     15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino       19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora      15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E         21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

> **Note**
>
> Les deux fonctions `tibble::column_to_rownames()` et `tibble::rownames_to_column()` acceptent un argument supplémentaire `var` qui permet d'indiquer un nom de colonne autre que le nom `rowname` utilisé par défaut pour créer ou identifier la colonne contenant les noms de lignes.

5.3 Données et tableaux imbriqués

Une des particularités des *tibbles* est qu'ils acceptent, à la différence des *data frames*, des colonnes composées de listes et, par extension, d'autres tibbles (qui sont des listes) !

```
d <- tibble(
  g = c(1, 2, 3),
  data = list(
    tibble(x = 1, y = 2),
    tibble(x = 4:5, y = 6:7),
    tibble(x = 10)
  )
)
d
```

```
# A tibble: 3 x 2
      g data
  <dbl> <list>
1     1 <tibble [1 x 2]>
2     2 <tibble [2 x 2]>
3     3 <tibble [1 x 1]>
```

```
d$data[[2]]
```

```
# A tibble: 2 x 2
      x     y
  <int> <int>
1     4     6
2     5     7
```

Cette fonctionalité, combinée avec les fonctions de `{tidyr}` et de `{purrr}`, s'avère très puissante pour réaliser des opérations multiples en peu de ligne de code.

Dans l'exemple ci-dessous, nous réalisons des régressions linéaires par sous-groupe et les présentons dans un même tableau. Pour le moment, le code présenté doit vous sembler complexe et un peu obscur. Pas de panique : tout cela sera clarifié dans les differents chapitres de ce guide. Ce qu'il y a à retenir pour le moment, c'est la possibilité de stocker, dans les colonnes d'un *tibble*, différent types de données, y compris des sous-tableaux, des résultats de modèles et même des tableaux mis en forme.

```
reg <-
  iris |>
  group_by(Species) |>
  nest() |>
  mutate(
    model = map(
      data,
      ~ lm(Sepal.Length ~ Petal.Length + Petal.Width, data = .)
    ),
    tbl = map(model, gtsummary::tbl_regression)
  )
reg
```

```
# A tibble: 3 x 4
# Groups:   Species [3]
  Species    data              model  tbl
  <fct>      <list>            <list> <list>
```

```
1 setosa     <tibble [50 x 4]> <lm>   <tbl_rgrs>
2 versicolor <tibble [50 x 4]> <lm>   <tbl_rgrs>
3 virginica  <tibble [50 x 4]> <lm>   <tbl_rgrs>
```

```
gtsummary::tbl_merge(
  reg$tbl,
  tab_spanner = paste0("**", reg$Species, "**")
)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	**setosa**			**versicolor**			**virginica**		
	Beta	**95% CI**	**p-value**	**Beta**	**95% CI**	**p-value**	**Beta**	**95% CI**	**p-value**
Petal.Length	0.40	-0.20, 0.99	0.2	0.93	0.59, 1.3	<0.001	1.0	0.81, 1.2	<0.001
Petal.Width	0.71	-0.27, 1.7	0.2	-0.32	-1.1, 0.49	0.4	0.01	-0.35, 0.37	>0.9

6 Attributs

Les objets **R** peuvent avoir des attributs qui correspondent en quelque sorte à des métadonnées associées à l'objet en question. Techniquement, un attribut peut être tout type d'objet **R** (un vecteur, une liste, une fonction…).

Parmi les attributs les plus courants, on retrouve nottament :

- `class` : la classe de l'objet
- `lenghth` : sa longueur
- `names` : les noms donnés aux éléments de l'objet
- `levels` : pour les facteurs, les étiquettes des différents niveaux
- `label` : une étiquette de variable

La fonction `attributes()` permet de lister tous les attributs associés à un objet.

```
attributes(iris)
```

```
$names
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

$class
[1] "data.frame"

$row.names
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
```

```
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150
```

Pour accéder à un attribut spécifique, on aura recours à `attr()` en spéficiant à la fois l'objet considéré et le nom de l'attribut souhaité.

```
iris |> attr("names")
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Pour les attributs les plus courants de **R**, il faut noter qu'il existe le plus souvent des fonctions spécifiques, comme `class()`, `names()` ou `row.names()`.

```
class(iris)
```

```
[1] "data.frame"
```

```
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

La fonction `attr()`, associée à l'opérateur d'assignation (`<-`) permet également de définir ses propres attributs.

```
attr(iris, "perso") <- "Des notes personnelles"
attributes(iris)
```

```
$names
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

$class
[1] "data.frame"

$row.names
```

```
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150

$perso
[1] "Des notes personnelles"
```

```r
attr(iris, "perso")
```

```
[1] "Des notes personnelles"
```

partie II

Manipulation de données

7 Le pipe

Il est fréquent d'enchainer des opérations en appelant successivement des fonctions sur le résultat de l'appel précédent.

Prenons un exemple. Supposons que nous ayons un vecteur numérique `v` dont nous voulons calculer la moyenne puis l'afficher via un message dans la console. Pour un meilleur rendu, nous allons arrondir la moyenne à une décimale, mettre en forme le résultat à la française, c'est-à-dire avec la virgule comme séparateur des décimales, créer une phrase avec le résultat, puis l'afficher dans la console. Voici le code correspondant, étape par étape.

```
v <- c(1.2, 8.7, 5.6, 11.4)
m <- mean(v)
r <- round(m, digits = 1)
f <- format(r, decimal.mark = ",")
p <- paste0("La moyenne est de ", f, ".")
message(p)
```

```
La moyenne est de 6,7.
```

Cette écriture, n'est pas vraiment optimale, car cela entraine la création d'un grand nombre de variables intermédiaires totalement inutiles. Nous pourrions dès lors imbriquer les différentes fonctions les unes dans les autres :

```
message(paste0("La moyenne est de ", format(round(mean(v),        digits = 1), decimal.mark
```

```
La moyenne est de 6,7.
```

Nous obtenons bien le même résultat, mais la lecture de cette ligne de code est assez difficile et il n'est pas aisé de bien identifier à quelle fonction est rattaché chaque argument.

Une amélioration possible serait d'effectuer des retours à la ligne avec une indentation adéquate pour rendre cela plus lisible.

```
message(
  paste0(
    "La moyenne est de ",
    format(
      round(
        mean(v),
        digits = 1),
      decimal.mark = ","
    ),
    "."
  )
)
```

```
La moyenne est de 6,7.
```

C'est déjà mieux, mais toujours pas optimal.

7.1 Le pipe natif de R : |>

Depuis la version 4.1, **R** a introduit ce que l'on nomme un *pipe* (tuyau en anglais), un nouvel opérateur noté `|>`.

Le principe de cet opérateur est de passer l'élément situé à sa gauche comme premier argument de la fonction située à sa droite. Ainsi, l'écriture `x |> f()` est équivalente à `f(x)` et l'écriture `x |> f(y)` à `f(x, y)`.

Parfois, on souhaite passer l'objet `x` à un autre endroit de la fonction `f()` que le premier argument. Depuis la version 4.2, **R** a introduit l'opérateur `_`,que l'on nomme un *placeholder*, pour indiquer où passer l'objet de gauche. Ainsi, `x |> f(y, a = _)` devient équivalent à `f(y, a = x)`. **ATTENTION :** le

placeholder doit impérativement être transmis à un argument nommé !

Tout cela semble encore un peu abstrait ? Reprenons notre exemple précédent et réécrivons le code avec le *pipe*.

```
v |>
  mean() |>
  round(digits = 1) |>
  format(decimal.mark = ",") |>
  paste0("La moyenne est de ", m = _, ".") |>
  message()
```

```
La moyenne est de 6,7.
```

Le code n'est-il pas plus lisible ?

Pour visualiser chaque étape du code, vous pouvez consulter le diaporama suivant : https://larmarange.github.io/guide-R/manipulation/ressources/flipbook-pipe.html

7.2 Le pipe du tidyverse : %>%

Ce n'est qu'à partir de la version 4.1 sortie en 2021 que **R** a proposé de manière native un *pipe*, en l'occurence l'opérateur `|>`.

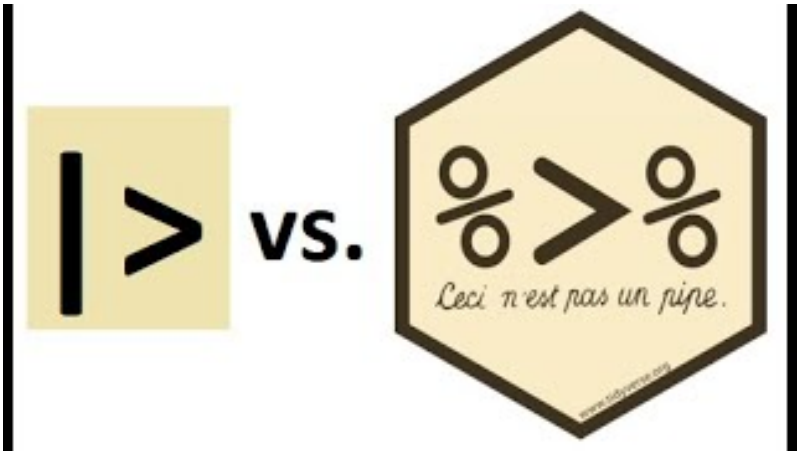
En cela, **R** s'est notamment inspiré d'un opérateur similaire introduit dès 2014 dans le *tidyverse*. Le pipe du *tidyverse* fonctionne de manière similaire. Il est implémenté dans le package `{magrittr}` qui doit donc être chargé en mémoire. Le *pipe* est également disponible lorsque l'on effecture `library(tidyverse)`.

Cet opérateur s'écrit `%>%` et il dispose lui aussi d'un *placeholder* qui est le `.`. La syntaxe du *placeholder* est un peu plus souple puisqu'il peut être passé à tout type d'argument, y compris un argument sans nom. Si l'on reprend notre exemple précédent.

```r
library(magrittr)
v %>%
  mean() %>%
  round(digits = 1) %>%
  format(decimal.mark = ",") %>%
  paste0("La moyenne est de ", ., ".") %>%
  message()
```

```
La moyenne est de 6,7.
```

7.3 Vaut-il mieux utiliser |> ou %>% ?



Bonne question. Si vous utilisez une version récente de **R** (4.2), il est préférable d'avoir recours au *pipe* natif de **R** dans la mesure où il est plus efficient en termes de temps de calcul car il fait partie intégrante du langage. Dans ce guide, nous privilégeons d'ailleurs l'utilisation de `|>`.

Si votre code nécessite de fonctionner avec différentes versions de **R**, par exemple dans le cadre d'un package, il est alors préférable, pour le moment, d'utiliser celui fourni par `{magrittr}` (`%>%`).

7.4 Accéder à un élément avec `purrr::pluck()` et `purrr::chuck()`

Il est fréquent d'avoir besoin d'accéder à un élément précis d'une liste, d'un tableau ou d'un vecteur, ce que l'on fait d'ordinaire avec la syntaxe `[[]]` ou `$` pour les listes ou `[]` pour les vecteurs. Cependant, cette syntaxe se combine souvent mal avec un enchaînement d'opérations utilisant le *pipe*.

Le package `{purrr}`, chargé par défaut avec `library(tidyverse)`, fournit une fonction `purrr::pluck()` qui, est l'équivalent de `[[]]`, et qui permet de récupérer un élément par son nom ou sa position. Ainsi, si l'on considère le tableau de données `iris`, `pluck(iris, "Petal.Witdh")` est équivalent à `iris$Petal.Width`. Voyons un example d'utilisation dans le cadre d'un enchaînement d'opérations.

```
iris |>
  purrr::pluck("Petal.Width") |>
  mean()
```

```
[1] 1.199333
```

Cette écriture est équivalente à :

```
mean(iris$Petal.Width)
```

```
[1] 1.199333
```

`purrr::pluck()` fonctionne également sur des vecteurs (et dans ce cas opère comme `[]`).

```
v <- c("a", "b", "c", "d")
v |> purrr::pluck(2)
```

```
[1] "b"
```

```
v[2]
```

```
[1] "b"
```

On peut également, dans un même appel à `purrr::pluck()`, enchaîner plusieurs niveaux. Les trois syntaxes ci-après sont ainsi équivalents :

```
iris |>
  purrr::pluck("Sepal.Width", 3)
```

```
[1] 3.2
```

```
iris |>
  purrr::pluck("Sepal.Width") |>
  purrr::pluck(3)
```

```
[1] 3.2
```

```
iris[["Sepal.Width"]][3]
```

```
[1] 3.2
```

Si l'on demande un élément qui n'existe pas, `purrr:pluck()` renverra l'élement vide (`NULL`). Si l'on souhaite plutôt que cela génère une erreur, on aura alors recours à `purrr::chuck()`.

```
iris |> purrr::pluck("inconnu")
```

```
NULL
```

```
iris |> purrr::chuck("inconnu")
```

```
Error in `purrr::chuck()`:
! Can't find name `inconnu` in vector.
```

```
v |> purrr::pluck(10)
```

```
NULL
```

```
v |> purrr::chuck(10)
```

```
Error in `purrr::chuck()`:
! Index 1 exceeds the length of plucked object (10 > 4).
```

8 dplyr

`{dplyr}` est l'un des packages les plus connus du *tidyverse*. Il facilite le traitement et la manipulation des tableaux de données (qu'il s'agisse de *data frame* ou de *tibble*). Il propose une syntaxe claire et cohérente, sous formes de verbes correspondant à des fonctions.

`{dplyr}` part du principe que les données sont *tidy* (chaque variable est une colonne, chaque observation est une ligne, voir Chapitre **??**). Les verbes de `{dplyr}` prennent en entrée un tableau de données[10] (*data frame* ou *tibble*) et renvoient systématiquement un *tibble*.

[10] Le package `{dbplyr}` permets d'étendre les verbes de `{dplyr}` à des tables de bases de données **SQL,** `{dtplyr}` à des tableaux de données du type `{data.table}` et `{srvyr}` à des données pondérées du type `{survey}`.

```
library(dplyr)
```

Dans ce qui suit on va utiliser le jeu de données `{nycflights13}`, contenu dans l'extension du même nom (qu'il faut donc avoir installée). Celui-ci correspond aux données de tous les vols au départ d'un des trois aéroports de New-York en 2013. Il a la particularité d'être réparti en trois tables :

- `nycflights13::flights` contient des informations sur les vols : date, départ, destination, horaires, retard…
- `nycflights13::airports` contient des informations sur les aéroports
- `nycflights13::airlines` contient des données sur les compagnies aériennes

On va charger les trois tables du jeu de données :

```
library(nycflights13)
## Chargement des trois tables du jeu de données
data(flights)
data(airports)
```

```
data(airlines)
```

Normalement trois objets correspondant aux trois tables ont dû apparaître dans votre environnement.

8.1 Opérations sur les lignes

8.1.1 filter()

`dplyr::filter()` sélectionne des lignes d'un tableau de données selon une condition. On lui passe en paramètre un test, et seules les lignes pour lesquelles ce test renvoit `TRUE` (vrai) sont conservées[11].

Par exemple, si on veut sélectionner les vols du mois de janvier, on peut filtrer sur la variable *month* de la manière suivante :

[11] Si le test renvoie faux (`FALSE`) ou une valeur manquante (`NA`), les lignes correspondantes ne seront donc pas sélectionnées.

```
filter(flights, month == 1)
```

```
# A tibble: 27,004 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 26,994 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Cela peut s'écrire plus simplement avec un pipe :

```
flights |> filter(month == 1)
```

```
# A tibble: 27,004 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 26,994 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Si l'on veut uniquement les vols avec un retard au départ (variable *dep_delay*) compris entre 10 et 15 minutes :

```
flights |>
  filter(dep_delay >= 10 & dep_delay <= 15)
```

```
# A tibble: 14,919 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      611            600        11      945            931
 2  2013     1     1      623            610        13      920            915
 3  2013     1     1      743            730        13     1107           1100
 4  2013     1     1      743            730        13     1059           1056
 5  2013     1     1      851            840        11     1215           1206
 6  2013     1     1      912            900        12     1241           1220
 7  2013     1     1      914            900        14     1058           1043
 8  2013     1     1      920            905        15     1039           1025
 9  2013     1     1     1011           1001        10     1133           1128
10  2013     1     1     1112           1100        12     1440           1438
```

```
# i 14,909 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Si l'on passe plusieurs arguments à `dplyr::filter()`, celui-ci rajoute automatiquement une condition **ET**. La ligne ci-dessus peut donc également être écrite de la manière suivante, avec le même résultat :

```
flights |>
  filter(dep_delay >= 10, dep_delay <= 15)
```

Enfin, on peut également placer des fonctions dans les tests, qui nous permettent par exemple de sélectionner les vols avec la plus grande distance :

```
flights |>
  filter(distance == max(distance))
```

```
# A tibble: 342 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      857            900        -3     1516           1530
 2  2013     1     2      909            900         9     1525           1530
 3  2013     1     3      914            900        14     1504           1530
 4  2013     1     4      900            900         0     1516           1530
 5  2013     1     5      858            900        -2     1519           1530
 6  2013     1     6     1019            900        79     1558           1530
 7  2013     1     7     1042            900       102     1620           1530
 8  2013     1     8      901            900         1     1504           1530
 9  2013     1     9      641            900      1301     1242           1530
10  2013     1    10      859            900        -1     1449           1530
# i 332 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Évaluation contextuelle

Il est important de noter que `{dplyr}` procède à une évaluation contextuelle des expressions qui lui sont passées. Ainsi, on peut indiquer directement le nom d'une variable et `{dplyr}` l'interprétera dans le contexte du tableau de données, c'est-à-dire regardera s'il existe une colonne portant ce nom dans le tableau.
Dans l'expression `flights |> filter(month == 1)`, `month` est interprété comme la colonne *month* du tableau `flights`, à savoir `flights$month`.
Il est également possible d'indiquer des objets extérieurs au tableau :

```
m <- 2
flights |>
  filter(month == m)
```

```
# A tibble: 24,951 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     2     1      456            500        -4      652            648
 2  2013     2     1      520            525        -5      816            820
 3  2013     2     1      527            530        -3      837            829
 4  2013     2     1      532            540        -8     1007           1017
 5  2013     2     1      540            540         0      859            850
 6  2013     2     1      552            600        -8      714            715
 7  2013     2     1      552            600        -8      919            910
 8  2013     2     1      552            600        -8      655            709
 9  2013     2     1      553            600        -7      833            815
10  2013     2     1      553            600        -7      821            825
# i 24,941 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Cela fonctionne car il n'y a pas de colonne *m* dans `flights`. Dès lors, `{dplyr}` regarde s'il existe un objet `m` dans l'environnement de travail.
Par contre, si une colonne existe dans le tableau, elle aura

priorité sur les objets du même nom dans l'environnement. Dans l'exemple ci-dessous, le résultat obtenu n'est pas celui voulu. Il est interprété comme sélectionner toutes les lignes où la colonne *mois* est égale à elle-même et donc cela sélectionne toutes les lignes du tableau.

```
month <- 3
flights |>
  filter(month == month)
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Afin de distinguer ce qui correspond à une colonne du tableau et à un objet de l'environnement, on pourra avoir recours à `.data` et `.env` (voir `help(".env", package = "rlang")`).

```
month <- 3
flights |>
  filter(.data$month == .env$month)
```

```
# A tibble: 28,834 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
```

```
 1  2013     3     1        4           2159       125      318             56
 2  2013     3     1       50           2358        52      526            438
 3  2013     3     1      117           2245       152      223           2354
 4  2013     3     1      454            500        -6      633            648
 5  2013     3     1      505            515       -10      746            810
 6  2013     3     1      521            530        -9      813            827
 7  2013     3     1      537            540        -3      856            850
 8  2013     3     1      541            545        -4     1014           1023
 9  2013     3     1      549            600       -11      639            703
10  2013     3     1      550            600       -10      747            801
# i 28,824 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

8.1.2 slice()

Le verbe `dplyr::slice()` sélectionne des lignes du tableau selon leur position. On lui passe un chiffre ou un vecteur de chiffres.

Si l'on souhaite sélectionner la 345e ligne du tableau `airports` :

```
airports |>
  slice(345)
```

```
# A tibble: 1 x 8
  faa   name                lat   lon   alt    tz dst   tzone
  <chr> <chr>             <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 CYF   Chefornak Airport  60.1 -164.    40    -9 A     America/Anchorage
```

Si l'on veut sélectionner les 5 premières lignes :

```
airports |>
  slice(1:5)
```

```
# A tibble: 5 x 8
```

```
  faa   name                            lat   lon   alt    tz dst   tzone
  <chr> <chr>                         <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 04G   Lansdowne Airport              41.1 -80.6  1044    -5 A     America/New~
2 06A   Moton Field Municipal Airport  32.5 -85.7   264    -6 A     America/Chi~
3 06C   Schaumburg Regional            42.0 -88.1   801    -6 A     America/Chi~
4 06N   Randall Airport                41.4 -74.4   523    -5 A     America/New~
5 09J   Jekyll Island Airport          31.1 -81.4    11    -5 A     America/New~
```

8.1.3 arrange()

`dplyr::arrange()` réordonne les lignes d'un tableau selon une ou plusieurs colonnes.

Ainsi, si l'on veut trier le tableau `flights` selon le retard au départ, dans l'ordre croissant :

```
flights |>
  arrange(dep_delay)
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013    12     7     2040           2123       -43       40           2352
 2  2013     2     3     2022           2055       -33     2240           2338
 3  2013    11    10     1408           1440       -32     1549           1559
 4  2013     1    11     1900           1930       -30     2233           2243
 5  2013     1    29     1703           1730       -27     1947           1957
 6  2013     8     9      729            755       -26     1002            955
 7  2013    10    23     1907           1932       -25     2143           2143
 8  2013     3    30     2030           2055       -25     2213           2250
 9  2013     3     2     1431           1455       -24     1601           1631
10  2013     5     5      934            958       -24     1225           1309
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

On peut trier selon plusieurs colonnes. Par exemple selon le mois, puis selon le retard au départ :

```
flights |>
  arrange(month, dep_delay)
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1    11     1900           1930       -30     2233           2243
 2  2013     1    29     1703           1730       -27     1947           1957
 3  2013     1    12     1354           1416       -22     1606           1650
 4  2013     1    21     2137           2159       -22     2232           2316
 5  2013     1    20      704            725       -21     1025           1035
 6  2013     1    12     2050           2110       -20     2310           2355
 7  2013     1    12     2134           2154       -20        4             50
 8  2013     1    14     2050           2110       -20     2329           2355
 9  2013     1     4     2140           2159       -19     2241           2316
10  2013     1    11     1947           2005       -18     2209           2230
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Si l'on veut trier selon une colonne par ordre décroissant, on lui applique la fonction `dplyr::desc()` :

```
flights |>
  arrange(desc(dep_delay))
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     9      641            900      1301     1242           1530
 2  2013     6    15     1432           1935      1137     1607           2120
 3  2013     1    10     1121           1635      1126     1239           1810
 4  2013     9    20     1139           1845      1014     1457           2210
 5  2013     7    22      845           1600      1005     1044           1815
 6  2013     4    10     1100           1900       960     1342           2211
 7  2013     3    17     2321            810       911      135           1020
 8  2013     6    27      959           1900       899     1236           2226
 9  2013     7    22     2257            759       898      121           1026
```

```
10  2013    12     5      756           1700       896     1058           2020
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Combiné avec `dplyr::slice()`, `dplyr::arrange()` permet par exemple de sélectionner les trois vols ayant eu le plus de retard :

```
flights |>
  arrange(desc(dep_delay)) |>
  slice(1:3)
```

```
# A tibble: 3 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     9      641            900      1301     1242           1530
2  2013     6    15     1432           1935      1137     1607           2120
3  2013     1    10     1121           1635      1126     1239           1810
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

8.1.4 slice_sample()

`dplyr::slice_sample()` permet de sélectionner aléatoirement un nombre de lignes ou une fraction des lignes d'un tableau. Ainsi si l'on veut choisir 5 lignes au hasard dans le tableau `airports` :

```
airports |>
  slice_sample(n = 5)
```

```
# A tibble: 5 x 8
  faa   name                          lat    lon   alt    tz dst   tzone
  <chr> <chr>                       <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
1 LSE   La Crosse Municipal          43.9  -91.3   654    -6 A     America/Chi~
```

```
2 UES   Waukesha County Airport       43.0  -88.2   911    -6 A     America/Chi~
3 SPW   Spencer Muni                  43.2  -95.2  1339    -6 A     America/Chi~
4 FAR   Hector International Airport  46.9  -96.8   902    -6 A     America/Chi~
5 KPR   Port Williams Seaplane Base   58.5 -153.      0    -9 A     America/Anc~
```

Si l'on veut tirer au hasard 10% des lignes de `flights` :

```
flights |>
  slice_sample(prop = .1)
```

```
# A tibble: 33,677 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     8     8     1748           1713        35     2013           1919
 2  2013    10     8     1040           1046        -6     1335           1330
 3  2013     3    14      624            630        -6      754            811
 4  2013     8    10     2153           2130        23       34           2359
 5  2013    10    23     2235           2245       -10     2335           2356
 6  2013     1    17     1230           1235        -5     1530           1606
 7  2013     6     6     1000            955         5     1124           1110
 8  2013     5    26      704            705        -1      938            956
 9  2013    10     1      712            720        -8      934           1000
10  2013     3     1     1816           1820        -4     2000           2031
# i 33,667 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Ces fonctions sont utiles notamment pour faire de l'"échantillonnage" en tirant au hasard un certain nombre d'observations du tableau.

8.1.5 distinct()

`dplyr::distinct()` filtre les lignes du tableau pour ne conserver que les lignes distinctes, en supprimant toutes les lignes en double.

```r
flights |>
  select(day, month) |>
  distinct()
```

```
# A tibble: 365 x 2
     day month
   <int> <int>
 1     1     1
 2     2     1
 3     3     1
 4     4     1
 5     5     1
 6     6     1
 7     7     1
 8     8     1
 9     9     1
10    10     1
# i 355 more rows
```

On peut lui spécifier une liste de variables : dans ce cas, pour toutes les observations ayant des valeurs identiques pour les variables en question, `dplyr::distinct()` ne conservera que la première d'entre elles.

```r
flights |>
  distinct(month, day)
```

```
# A tibble: 365 x 2
   month   day
   <int> <int>
 1     1     1
 2     1     2
 3     1     3
 4     1     4
 5     1     5
 6     1     6
 7     1     7
 8     1     8
 9     1     9
```

```
10     1    10
# i 355 more rows
```

L'option `.keep_all` permet, dans l'opération précédente, de conserver l'ensemble des colonnes du tableau :

```
flights |>
  distinct(month, day, .keep_all = TRUE)
```

```
# A tibble: 365 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     2       42           2359        43      518            442
 3  2013     1     3       32           2359        33      504            442
 4  2013     1     4       25           2359        26      505            442
 5  2013     1     5       14           2359        15      503            445
 6  2013     1     6       16           2359        17      451            442
 7  2013     1     7       49           2359        50      531            444
 8  2013     1     8      454            500        -6      625            648
 9  2013     1     9        2           2359         3      432            444
10  2013     1    10        3           2359         4      426            437
# i 355 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

8.2 Opérations sur les colonnes

8.2.1 select()

`dplyr::select()` permet de sélectionner des colonnes d'un tableau de données. Ainsi, si l'on veut extraire les colonnes `lat` et `lon` du tableau `airports` :

```
airports |>
  select(lat, lon)
```

```
# A tibble: 1,458 x 2
     lat    lon
   <dbl>  <dbl>
 1  41.1  -80.6
 2  32.5  -85.7
 3  42.0  -88.1
 4  41.4  -74.4
 5  31.1  -81.4
 6  36.4  -82.2
 7  41.5  -84.5
 8  42.9  -76.8
 9  39.8  -76.6
10  48.1 -123.
# i 1,448 more rows
```

Si on fait précéder le nom d'un `-`, la colonne est éliminée plutôt que sélectionnée :

```
airports |>
  select(-lat, -lon)
```

```
# A tibble: 1,458 x 6
   faa   name                            alt    tz dst   tzone
   <chr> <chr>                         <dbl> <dbl> <chr> <chr>
 1 04G   Lansdowne Airport              1044    -5 A     America/New_York
 2 06A   Moton Field Municipal Airport   264    -6 A     America/Chicago
 3 06C   Schaumburg Regional             801    -6 A     America/Chicago
 4 06N   Randall Airport                 523    -5 A     America/New_York
 5 09J   Jekyll Island Airport            11    -5 A     America/New_York
 6 0A9   Elizabethton Municipal Airport 1593    -5 A     America/New_York
 7 0G6   Williams County Airport         730    -5 A     America/New_York
 8 0G7   Finger Lakes Regional Airport   492    -5 A     America/New_York
 9 0P2   Shoestring Aviation Airfield   1000    -5 U     America/New_York
10 0S9   Jefferson County Intl           108    -8 A     America/Los_Angeles
# i 1,448 more rows
```

`dplyr::select()` comprend toute une série de fonctions facilitant la sélection de multiples colonnes. Par exemple, `dplyr::starts_with()`, `dplyr::ends_width()`,

`dplyr::contains()` ou `dplyr::matches()` permettent d'exprimer des conditions sur les noms de variables :

```
flights |>
  select(starts_with("dep_"))
```

```
# A tibble: 336,776 x 2
   dep_time dep_delay
      <int>     <dbl>
 1      517         2
 2      533         4
 3      542         2
 4      544        -1
 5      554        -6
 6      554        -4
 7      555        -5
 8      557        -3
 9      557        -3
10      558        -2
# i 336,766 more rows
```

La syntaxe `colonne1:colonne2` permet de sélectionner toutes les colonnes situées entre *colonne1* et *colonne2* incluses[12] :

[12] À noter que cette opération est un peu plus "fragile" que les autres, car si l'ordre des colonnes change elle peut renvoyer un résultat différent.

```
flights |>
  select(year:day)
```

```
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
```

```
10  2013     1     1
# i 336,766 more rows
```

`dplyr::all_of()` et `dplyr::any_of()` permettent de fournir une liste de variables à extraire sous forme de vecteur textuel. Alors que `dplyr::all_of()` renverra une erreur si une variable n'est pas trouvée dans le tableau de départ, `dplyr::any_of()` sera moins stricte.

```
flights |>
  select(all_of(c("year", "month", "day")))
```

```
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# i 336,766 more rows
```

```
flights |>
  select(all_of(c("century", "year", "month", "day")))
```

```
Error in `all_of()`:
! Can't subset columns that don't exist.
x Column `century` doesn't exist.
```

```
Erreur : Can't subset columns that don't exist.
x Column `century` doesn't exist.
```

```r
flights |>
  select(any_of(c("century", "year", "month", "day")))
```

```
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# i 336,766 more rows
```

`dplyr::where()` permets de sélectionner des variables à partir d'une fonction qui renvoie une valeur logique. Par exemple, pour sélectionner seulement les variables textuelles :

```r
flights |>
  select(where(is.character))
```

```
# A tibble: 336,776 x 4
   carrier tailnum origin dest
   <chr>   <chr>   <chr>  <chr>
 1 UA      N14228  EWR    IAH
 2 UA      N24211  LGA    IAH
 3 AA      N619AA  JFK    MIA
 4 B6      N804JB  JFK    BQN
 5 DL      N668DN  LGA    ATL
 6 UA      N39463  EWR    ORD
 7 B6      N516JB  EWR    FLL
 8 EV      N829AS  LGA    IAD
 9 B6      N593JB  JFK    MCO
10 AA      N3ALAA  LGA    ORD
# i 336,766 more rows
```

`dplyr::select()` peut être utilisée pour réordonner les colonnes d'une table en utilisant la fonction `dplyr::everything()`, qui sélectionne l'ensemble des colonnes non encore sélectionnées. Ainsi, si l'on souhaite faire passer la colonne *name* en première position de la table `airports`, on peut faire :

```
airports |>
  select(name, everything())
```

```
# A tibble: 1,458 x 8
   name                           faa     lat    lon   alt    tz dst   tzone
   <chr>                          <chr> <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
 1 Lansdowne Airport              04G    41.1  -80.6  1044    -5 A     America/~
 2 Moton Field Municipal Airport  06A    32.5  -85.7   264    -6 A     America/~
 3 Schaumburg Regional            06C    42.0  -88.1   801    -6 A     America/~
 4 Randall Airport                06N    41.4  -74.4   523    -5 A     America/~
 5 Jekyll Island Airport          09J    31.1  -81.4    11    -5 A     America/~
 6 Elizabethton Municipal Airport 0A9    36.4  -82.2  1593    -5 A     America/~
 7 Williams County Airport        0G6    41.5  -84.5   730    -5 A     America/~
 8 Finger Lakes Regional Airport  0G7    42.9  -76.8   492    -5 A     America/~
 9 Shoestring Aviation Airfield   0P2    39.8  -76.6  1000    -5 U     America/~
10 Jefferson County Intl          0S9    48.1 -123.    108    -8 A     America/~
# i 1,448 more rows
```

8.2.2 relocate()

Pour réordonner des colonnes, on pourra aussi avoir recours à `dplyr::relocate()` en indiquant les premières variables. Il n'est pas nécessaire d'ajouter `everything()` car avec `dplyr::relocate()` toutes les variables sont conservées.

```
airports |>
  relocate(lon, lat, name)
```

```
# A tibble: 1,458 x 8
      lon   lat name                           faa     alt    tz dst   tzone
    <dbl> <dbl> <chr>                          <chr> <dbl> <dbl> <chr> <chr>
 1  -80.6  41.1 Lansdowne Airport              04G    1044    -5 A     America/~
 2  -85.7  32.5 Moton Field Municipal Airport  06A     264    -6 A     America/~
```

```
 3  -88.1  42.0 Schaumburg Regional            06C     801    -6 A     America/~
 4  -74.4  41.4 Randall Airport                06N     523    -5 A     America/~
 5  -81.4  31.1 Jekyll Island Airport          09J      11    -5 A     America/~
 6  -82.2  36.4 Elizabethton Municipal Airport 0A9    1593    -5 A     America/~
 7  -84.5  41.5 Williams County Airport        0G6     730    -5 A     America/~
 8  -76.8  42.9 Finger Lakes Regional Airport  0G7     492    -5 A     America/~
 9  -76.6  39.8 Shoestring Aviation Airfield   0P2    1000    -5 U     America/~
10 -123.   48.1 Jefferson County Intl          0S9     108    -8 A     America/~
# i 1,448 more rows
```

8.2.3 rename()

Une variante de `dplyr::select()` est `dplyr::rename()`[13], qui permet de renommer facilement des colonnes. On l'utilise en lui passant des paramètres de la forme `nouveau_nom = ancien_nom`. Ainsi, si on veut renommer les colonnes *lon* et *lat* de `airports` en *longitude* et *latitude* :

[13] Il est également possible de renommer des colonnes directement avec `select()`, avec la même syntaxe que pour `rename()`.

```
airports |>
  rename(longitude = lon, latitude = lat)
```

```
# A tibble: 1,458 x 8
   faa   name                           latitude longitude   alt    tz dst   tzone
   <chr> <chr>                             <dbl>     <dbl> <dbl> <dbl> <chr> <chr>
 1 04G   Lansdowne Airport                   41.1     -80.6  1044    -5 A     Amer~
 2 06A   Moton Field Municipal Airpo~       32.5     -85.7   264    -6 A     Amer~
 3 06C   Schaumburg Regional                42.0     -88.1   801    -6 A     Amer~
 4 06N   Randall Airport                    41.4     -74.4   523    -5 A     Amer~
 5 09J   Jekyll Island Airport              31.1     -81.4    11    -5 A     Amer~
 6 0A9   Elizabethton Municipal Airp~       36.4     -82.2  1593    -5 A     Amer~
 7 0G6   Williams County Airport            41.5     -84.5   730    -5 A     Amer~
 8 0G7   Finger Lakes Regional Airpo~       42.9     -76.8   492    -5 A     Amer~
 9 0P2   Shoestring Aviation Airfield       39.8     -76.6  1000    -5 U     Amer~
10 0S9   Jefferson County Intl              48.1    -123.    108    -8 A     Amer~
# i 1,448 more rows
```

Si les noms de colonnes comportent des espaces ou des caractères spéciaux, on peut les entourer de guillemets (") ou de quotes inverses (`) :

```r
flights |>
  rename(
    "retard départ" = dep_delay,
    "retard arrivée" = arr_delay
  ) |>
  select(`retard départ`, `retard arrivée`)
```

```
# A tibble: 336,776 x 2
   `retard départ` `retard arrivée`
             <dbl>            <dbl>
 1               2               11
 2               4               20
 3               2               33
 4              -1              -18
 5              -6              -25
 6              -4               12
 7              -5               19
 8              -3              -14
 9              -3               -8
10              -2                8
# i 336,766 more rows
```

8.2.4 rename_with()

La fonction `dplyr::rename_with()` permets de renommer plusieurs colonnes d'un coup en transmettant une fonction, par exemple `toupper()` qui passe tous les caractères en majuscule.

```r
airports |>
  rename_with(toupper)
```

```
# A tibble: 1,458 x 8
   FAA   NAME                           LAT    LON   ALT    TZ DST   TZONE
   <chr> <chr>                        <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
 1 04G   Lansdowne Airport             41.1  -80.6  1044    -5 A     America/~
 2 06A   Moton Field Municipal Airport  32.5  -85.7   264    -6 A     America/~
 3 06C   Schaumburg Regional            42.0  -88.1   801    -6 A     America/~
```

```
 4 06N   Randall Airport                 41.4  -74.4   523    -5 A     America/~
 5 09J   Jekyll Island Airport           31.1  -81.4    11    -5 A     America/~
 6 0A9   Elizabethton Municipal Airport  36.4  -82.2  1593    -5 A     America/~
 7 0G6   Williams County Airport         41.5  -84.5   730    -5 A     America/~
 8 0G7   Finger Lakes Regional Airport   42.9  -76.8   492    -5 A     America/~
 9 0P2   Shoestring Aviation Airfield    39.8  -76.6  1000    -5 U     America/~
10 0S9   Jefferson County Intl           48.1 -123.    108    -8 A     America/~
# i 1,448 more rows
```

On pourra notamment utiliser les fonctions du package `snakecase` et, en particulier, `snakecase::to_snake_case()` que je recommande pour nommer de manière consistante les variables[14].

[14] Le ***snake case*** est une convention typographique en informatique consistant à écrire des ensembles de mots, généralement, en minuscules en les séparant par des tirets bas.

8.2.5 pull()

La fonction `dplyr::pull()` permet d'accéder au contenu d'une variable. C'est un équivalent aux opérateurs `$` ou `[[]]`. On peut lui passer un nom de variable ou bien sa position.

```
airports |>
  pull(alt) |>
  mean()
```

```
[1] 1001.416
```

> **Note**
>
> `dplyr::pull()` ressemble à la fonction `purrr::chuck()` que nous avons déjà abordée (cf. Section **??**). Cependant, `dplyr::pull()` ne fonctionne que sur des tableaux de données tandis que `purrr::chuck()` est plus générique et peut s'appliquer à tous types de listes.

8.2.6 mutate()

`dplyr::mutate()` permet de créer de nouvelles colonnes dans le tableau de données, en général à partir de variables existantes.

Par exemple, la table `airports` contient l'altitude de l'aéroport en pieds. Si l'on veut créer une nouvelle variable *alt_m* avec l'altitude en mètres, on peut faire :

```
airports <-
  airports |>
  mutate(alt_m = alt / 3.2808)
```

On peut créer plusieurs nouvelles colonnes en une seule fois, et les expressions successives peuvent prendre en compte les résultats des calculs précédents. L'exemple suivant convertit d'abord la distance en kilomètres dans une variable *distance_km*, puis utilise cette nouvelle colonne pour calculer la vitesse en km/h.

```
flights <-
  flights |>
  mutate(
    distance_km = distance / 0.62137,
    vitesse = distance_km / air_time * 60
)
```

8.3 Opérations groupées

8.3.1 group_by()

Un élément très important de `{dplyr}` est la fonction `dplyr::group_by()`. Elle permet de définir des groupes de lignes à partir des valeurs d'une ou plusieurs colonnes. Par exemple, on peut grouper les vols selon leur mois :

```
flights |>
  group_by(month)
```

```
# A tibble: 336,776 x 21
# Groups:   month [12]
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
```

```
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 336,766 more rows
# i 13 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>, distance_km <dbl>,
#   vitesse <dbl>
```

Par défaut ceci ne fait rien de visible, à part l'apparition d'une mention *Groups* dans l'affichage du résultat. Mais à partir du moment où des groupes ont été définis, les verbes comme `dplyr::slice()` ou `dplyr::mutate()` vont en tenir compte lors de leurs opérations.

Par exemple, si on applique `dplyr::slice()` à un tableau préalablement groupé, il va sélectionner les lignes aux positions indiquées *pour chaque groupe*. Ainsi la commande suivante affiche le premier vol de chaque mois, selon leur ordre d'apparition dans le tableau :

```
flights |>
  group_by(month) |>
  slice(1)
```

```
# A tibble: 12 x 21
# Groups:   month [12]
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     2     1      456            500        -4      652            648
 3  2013     3     1        4           2159       125      318             56
 4  2013     4     1      454            500        -6      636            640
 5  2013     5     1        9           1655       434      308           2020
```

```
 6  2013     6     1        2           2359         3      341            350
 7  2013     7     1        1           2029       212      236           2359
 8  2013     8     1       12           2130       162      257             14
 9  2013     9     1        9           2359        10      343            340
10  2013    10     1      447            500       -13      614            648
11  2013    11     1        5           2359         6      352            345
12  2013    12     1       13           2359        14      446            445
# i 13 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>, distance_km <dbl>,
#   vitesse <dbl>
```

Idem pour `dplyr::mutate()` : les opérations appliquées lors du calcul des valeurs des nouvelles colonnes sont appliquée groupe de lignes par groupe de lignes. Dans l'exemple suivant, on ajoute une nouvelle colonne qui contient le retard moyen *du mois correspondant* :

```
flights |>
  group_by(month) |>
  mutate(mean_delay_month = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 336,776 x 22
# Groups:   month [12]
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 336,766 more rows
# i 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>, distance_km <dbl>,
```

```
#   vitesse <dbl>, mean_delay_month <dbl>
```

Ceci peut permettre, par exemple, de déterminer si un retard donné est supérieur ou inférieur au retard moyen du mois en cours.

`dplyr::group_by()` peut aussi être utile avec `dplyr::filter()`, par exemple pour sélectionner les vols avec le retard au départ le plus important *pour chaque mois* :

```
flights |>
  group_by(month) |>
  filter(dep_delay == max(dep_delay, na.rm = TRUE))
```

```
# A tibble: 12 x 21
# Groups:   month [12]
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     9      641            900      1301     1242           1530
 2  2013    10    14     2042            900       702     2255           1127
 3  2013    11     3      603           1645       798      829           1913
 4  2013    12     5      756           1700       896     1058           2020
 5  2013     2    10     2243            830       853      100           1106
 6  2013     3    17     2321            810       911      135           1020
 7  2013     4    10     1100           1900       960     1342           2211
 8  2013     5     3     1133           2055       878     1250           2215
 9  2013     6    15     1432           1935      1137     1607           2120
10  2013     7    22      845           1600      1005     1044           1815
11  2013     8     8     2334           1454       520      120           1710
12  2013     9    20     1139           1845      1014     1457           2210
# i 13 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>, distance_km <dbl>,
#   vitesse <dbl>
```

Attention : la clause `dplyr::roup_by()` marche pour les verbes déjà vus précédemment, *sauf* pour `dplyr::arrange()`, qui par défaut trie la table sans tenir compte des groupes. Pour obtenir un tri par groupe, il faut lui ajouter l'argument `.by_group = TRUE`.

On peut voir la différence en comparant les deux résultats suivants :

```
flights |>
  group_by(month) |>
  arrange(desc(dep_delay))
```

```
# A tibble: 336,776 x 21
# Groups:   month [12]
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     9      641            900      1301     1242           1530
 2  2013     6    15     1432           1935      1137     1607           2120
 3  2013     1    10     1121           1635      1126     1239           1810
 4  2013     9    20     1139           1845      1014     1457           2210
 5  2013     7    22      845           1600      1005     1044           1815
 6  2013     4    10     1100           1900       960     1342           2211
 7  2013     3    17     2321            810       911      135           1020
 8  2013     6    27      959           1900       899     1236           2226
 9  2013     7    22     2257            759       898      121           1026
10  2013    12     5      756           1700       896     1058           2020
# i 336,766 more rows
# i 13 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>, distance_km <dbl>,
#   vitesse <dbl>
```

```
flights |>
  group_by(month) |>
  arrange(desc(dep_delay), .by_group = TRUE)
```

```
# A tibble: 336,776 x 21
# Groups:   month [12]
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     9      641            900      1301     1242           1530
 2  2013     1    10     1121           1635      1126     1239           1810
 3  2013     1     1      848           1835       853     1001           1950
 4  2013     1    13     1809            810       599     2054           1042
```

```
 5  2013     1    16     1622            800       502     1911            1054
 6  2013     1    23     1551            753       478     1812            1006
 7  2013     1    10     1525            900       385     1713            1039
 8  2013     1     1     2343           1724       379      314            1938
 9  2013     1     2     2131           1512       379     2340            1741
10  2013     1     7     2021           1415       366     2332            1724
# i 336,766 more rows
# i 13 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>, distance_km <dbl>,
#   vitesse <dbl>
```

8.3.2 summarise()

`dplyr::summarise()` permet d'agréger les lignes du tableau en effectuant une opération résumée sur une ou plusieurs colonnes. Il s'agit de toutes les fonctions qui prennent en entrée un ensemble de valeurs et renvoie une valeur unique, comme la moyenne (`mean()`). Par exemple, si l'on souhaite connaître les retards moyens au départ et à l'arrivée pour l'ensemble des vols du tableau `flights` :

```r
flights |>
  summarise(
    retard_dep = mean(dep_delay, na.rm=TRUE),
    retard_arr = mean(arr_delay, na.rm=TRUE)
  )
```

```
# A tibble: 1 x 2
  retard_dep retard_arr
       <dbl>      <dbl>
1       12.6       6.90
```

Cette fonction est en général utilisée avec `dplyr::group_by()`, puisqu'elle permet du coup d'agréger et de résumer les lignes du tableau groupe par groupe. Si l'on souhaite calculer le délai maximum, le délai minimum et le délai moyen au départ pour chaque mois, on pourra faire :

```r
flights |>
  group_by(month) |>
  summarise(
    max_delay = max(dep_delay, na.rm=TRUE),
    min_delay = min(dep_delay, na.rm=TRUE),
    mean_delay = mean(dep_delay, na.rm=TRUE)
  )
```

```
# A tibble: 12 x 4
   month max_delay min_delay mean_delay
   <int>     <dbl>     <dbl>      <dbl>
 1     1      1301       -30      10.0
 2     2       853       -33      10.8
 3     3       911       -25      13.2
 4     4       960       -21      13.9
 5     5       878       -24      13.0
 6     6      1137       -21      20.8
 7     7      1005       -22      21.7
 8     8       520       -26      12.6
 9     9      1014       -24       6.72
10    10       702       -25       6.24
11    11       798       -32       5.44
12    12       896       -43      16.6
```

`dplyr::summarise()` dispose d'une fonction spéciale `dplyr::n()`, qui retourne le nombre de lignes du groupe. Ainsi si l'on veut le nombre de vols par destination, on peut utiliser :

```r
flights |>
  group_by(dest) |>
  summarise(n = n())
```

```
# A tibble: 105 x 2
   dest      n
   <chr> <int>
 1 ABQ     254
 2 ACK     265
 3 ALB     439
```

```
 4 ANC       8
 5 ATL   17215
 6 AUS    2439
 7 AVL     275
 8 BDL     443
 9 BGR     375
10 BHM     297
# i 95 more rows
```

`dplyr::n()` peut aussi être utilisée avec `dplyr::filter()` et `dplyr::mutate()`.

8.3.3 count()

À noter que quand l'on veut compter le nombre de lignes par groupe, on peut utiliser directement la fonction `dplyr::count()`. Ainsi le code suivant est identique au précédent :

```
flights |>
  count(dest)
```

```
# A tibble: 105 x 2
   dest      n
   <chr> <int>
 1 ABQ     254
 2 ACK     265
 3 ALB     439
 4 ANC       8
 5 ATL   17215
 6 AUS    2439
 7 AVL     275
 8 BDL     443
 9 BGR     375
10 BHM     297
# i 95 more rows
```

8.3.4 Grouper selon plusieurs variables

On peut grouper selon plusieurs variables à la fois, il suffit de les indiquer dans la clause du `dplyr::group_by()` :

```
flights |>
  group_by(month, dest) |>
  summarise(nb = n()) |>
  arrange(desc(nb))
```

```
`summarise()` has grouped output by 'month'. You can override using the
`.groups` argument.
```

```
# A tibble: 1,113 x 3
# Groups:   month [12]
   month dest     nb
   <int> <chr> <int>
 1     8 ORD    1604
 2    10 ORD    1604
 3     5 ORD    1582
 4     9 ORD    1582
 5     7 ORD    1573
 6     6 ORD    1547
 7     7 ATL    1511
 8     8 ATL    1507
 9     8 LAX    1505
10     7 LAX    1500
# i 1,103 more rows
```

On peut également compter selon plusieurs variables :

```
flights |>
  count(origin, dest) |>
  arrange(desc(n))
```

```
# A tibble: 224 x 3
   origin dest      n
   <chr>  <chr> <int>
```

```
 1 JFK    LAX   11262
 2 LGA    ATL   10263
 3 LGA    ORD    8857
 4 JFK    SFO    8204
 5 LGA    CLT    6168
 6 EWR    ORD    6100
 7 JFK    BOS    5898
 8 LGA    MIA    5781
 9 JFK    MCO    5464
10 EWR    BOS    5327
# i 214 more rows
```

On peut utiliser plusieurs opérations de groupage dans le même *pipeline*. Ainsi, si l'on souhaite déterminer le couple origine/destination ayant le plus grand nombre de vols selon le mois de l'année, on devra procéder en deux étapes :

- d'abord grouper selon mois, origine et destination pour calculer le nombre de vols
- puis grouper uniquement selon le mois pour sélectionner la ligne avec la valeur maximale.

Au final, on obtient le code suivant :

```
flights |>
  group_by(month, origin, dest) |>
  summarise(nb = n()) |>
  group_by(month) |>
  filter(nb == max(nb))
```

```
`summarise()` has grouped output by 'month', 'origin'. You can override using
the `.groups` argument.
```

```
# A tibble: 12 x 4
# Groups:   month [12]
   month origin dest     nb
   <int> <chr>  <chr> <int>
 1     1 JFK    LAX     937
 2     2 JFK    LAX     834
 3     3 JFK    LAX     960
```

```
 4     4 JFK    LAX     935
 5     5 JFK    LAX     960
 6     6 JFK    LAX     928
 7     7 JFK    LAX     985
 8     8 JFK    LAX     979
 9     9 JFK    LAX     925
10    10 JFK    LAX     965
11    11 JFK    LAX     907
12    12 JFK    LAX     947
```

Lorsqu'on effectue un `dplyr::group_by()` suivi d'un `dplyr::summarise()`, le tableau résultat est automatiquement dégroupé *de la dernière variable de regroupement*. Ainsi le tableau généré par le code suivant est groupé par *month* et *origin*[15] :

[15] Comme expliqué dans le message affiché dans la console, cela peut être contrôlé avec l'argument `.groups` de `dplyr::summarise()`, dont les options sont décrites dans l'aide de la fonction.

```
  flights |>
    group_by(month, origin, dest) |>
    summarise(nb = n())
```

```
`summarise()` has grouped output by 'month', 'origin'. You can override using
the `.groups` argument.
```

```
# A tibble: 2,313 x 4
# Groups:   month, origin [36]
   month origin dest     nb
   <int> <chr>  <chr> <int>
 1     1 EWR    ALB      64
 2     1 EWR    ATL     362
 3     1 EWR    AUS      51
 4     1 EWR    AVL       2
 5     1 EWR    BDL      37
 6     1 EWR    BNA     111
 7     1 EWR    BOS     430
 8     1 EWR    BQN      31
 9     1 EWR    BTV     100
10     1 EWR    BUF     119
# i 2,303 more rows
```

Cela peut permettre d'enchaîner les opérations groupées. Dans l'exemple suivant, on calcule le pourcentage des trajets pour chaque destination par rapport à tous les trajets du mois :

```
  flights |>
    group_by(month, dest) |>
    summarise(nb = n()) |>
    mutate(pourcentage = nb / sum(nb) * 100)
```

```
`summarise()` has grouped output by 'month'. You can override using the
`.groups` argument.
```

```
# A tibble: 1,113 x 4
# Groups:   month [12]
   month dest     nb pourcentage
   <int> <chr> <int>       <dbl>
 1     1 ALB      64     0.237
 2     1 ATL    1396     5.17
 3     1 AUS     169     0.626
 4     1 AVL       2     0.00741
 5     1 BDL      37     0.137
 6     1 BHM      25     0.0926
 7     1 BNA     399     1.48
 8     1 BOS    1245     4.61
 9     1 BQN      93     0.344
10     1 BTV     223     0.826
# i 1,103 more rows
```

On peut à tout moment dégrouper un tableau à l'aide de `dplyr::ungroup()`. Ce serait par exemple nécessaire, dans l'exemple précédent, si on voulait calculer le pourcentage sur le nombre total de vols plutôt que sur le nombre de vols par mois :

```
  flights |>
    group_by(month, dest) |>
    summarise(nb = n()) |>
    ungroup() |>
    mutate(pourcentage = nb / sum(nb) * 100)
```

```
`summarise()` has grouped output by 'month'. You can override using the
`.groups` argument.
```

```
# A tibble: 1,113 x 4
   month dest     nb pourcentage
   <int> <chr> <int>       <dbl>
 1     1 ALB      64    0.0190
 2     1 ATL    1396    0.415
 3     1 AUS     169    0.0502
 4     1 AVL       2    0.000594
 5     1 BDL      37    0.0110
 6     1 BHM      25    0.00742
 7     1 BNA     399    0.118
 8     1 BOS    1245    0.370
 9     1 BQN      93    0.0276
10     1 BTV     223    0.0662
# i 1,103 more rows
```

À noter que `dplyr::count()`, par contre, renvoit un tableau non groupé :

```
flights |>
  count(month, dest)
```

```
# A tibble: 1,113 x 3
   month dest      n
   <int> <chr> <int>
 1     1 ALB      64
 2     1 ATL    1396
 3     1 AUS     169
 4     1 AVL       2
 5     1 BDL      37
 6     1 BHM      25
 7     1 BNA     399
 8     1 BOS    1245
 9     1 BQN      93
10     1 BTV     223
# i 1,103 more rows
```

8.4 Cheatsheet

8.5 webin-R

On pourra également se référer au webin-R #04 (*manipuler les données avec dplyr*) sur YouTube.

https://youtu.be/aFvBhgmawcs

9 Facteurs et `forcats`

Dans **R**, les fecteurs sont utilisés pour représenter des variables catégorielles, c'est-à-dire des variables qui ont un nombre fixé et limité de valeurs possibles (par exemple une variable *sexe* ou une variable *niveau d'éducation*).

De telles variables sont parfois représentées sous forme textuelle (vecteurs de type `character`). Cependant, cela ne permets pas d'indiquer un ordre spécifique aux modalités, à la différence des facteurs.

> **Note**
>
> Lorsque l'on importe des données d'enquêtes, il est fréquent que les variables catégorielles sont codées sous la forme d'un code numérique (par exemple 1 pour *femme* et 2 pour *homme*) auquel est associé une *étiquette de valeur*. C'est notamment le fonctionnement usuel de logiciels tels que **SPSS**, **Stata** ou **SAS**. Les étiquettes de valeurs seront abordés dans un prochain chapitre (voir Chapitre **??**).
> Au moment de l'analyse (tableaux statistiques, graphiques, modèles de régression…), il sera nécessaire de transformer ces vecteurs avec étiquettes en facteurs.

9.1 Création d'un facteur

Le plus simple pour créer un facteur est de partir d'un vecteur textuel et d'utiliser la fonction `factor()`.

```
x <- c("nord", "sud", "sud", "est", "est", "est")
x |>
```

```
  factor()
```

```
[1] nord sud  sud  est  est  est
Levels: est nord sud
```

Par défaut, les niveaux du facteur obtenu correspondent aux valeurs uniques du fecteur textuel, triés par ordre alphabétique. Si l'on veut contrôler l'ordre des niveaux, et éventuellement indiquer un niveau absent des données, on utilisera l'argument `levels` de `factor()`.

```
x |>
  factor(levels = c("nord", "est", "sud", "ouest"))
```

```
[1] nord sud  sud  est  est  est
Levels: nord est sud ouest
```

Si une valeur observée dans les données n'est pas indiqué dans `levels`, elle sera siliencieusement convertie en valeur manquante (`NA`).

```
x |>
  factor(levels = c("nord", "sud"))
```

```
[1] nord sud  sud  <NA> <NA> <NA>
Levels: nord sud
```

Si l'on veut être averti par un warning dans ce genre de situation, on pourra avoir plutôt recours à la fonction `readr::parse_factor()` du package `{readr}`, qui, le cas échéant, renverra un tableau avec les problèmes rencontrés.

```
x |>
  readr::parse_factor(levels = c("nord", "sud"))
```

```
Warning: 3 parsing failures.
row col           expected actual
  4  -- value in level set    est
  5  -- value in level set    est
  6  -- value in level set    est

[1] nord sud  sud  <NA> <NA> <NA>
attr(,"problems")
# A tibble: 3 x 4
    row   col expected           actual
  <int> <int> <chr>              <chr>
1     4    NA value in level set est
2     5    NA value in level set est
3     6    NA value in level set est
Levels: nord sud
```

Une fois un facteur créé, on peut accéder à la liste de ses étiquettes avec `levels()`.

```
f <- factor(x)
levels(f)
```

```
[1] "est"  "nord" "sud"
```

Dans certaines situations (par exemple pour la réalisation d'une régression logistique ordinale), on peut avoir avoir besoin d'indiquer que les modalités du facteur sont ordonnées héarchiquement. Dans ce cas là, on aura simplement recours à `ordered()` pour créer/convertir notre facteur.

```
c("supérieur", "primaire", "secondaire", "primaire", "supérieur") |>
  ordered(levels = c("primaire", "secondaire", "supérieur"))
```

```
[1] supérieur  primaire   secondaire primaire   supérieur
Levels: primaire < secondaire < supérieur
```

Techniquement, les valeurs d'un facteur sont stockés de manière interne à l'aide de nombres entiers, dont la valeur représente la position de l'étiquette correspondante dans l'attribut

`levels`. Ainsi, un facteur à `n` modalités sera toujours codé avec les nombre entiers allant de 1 à `n`.

```
class(f)
```

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

```
typeof(f)
```

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

```
as.integer(f)
```

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

```
as.character(f)
```

```
[1] "nord" "sud"  "sud"  "est"  "est"  "est"
```

9.2 Changer l'ordre des modalités

Le pacakge `{forcats}`, chargé par défaut lorsque l'on exécute la commande `library(tidyverse)`, fournie plusieurs fonctions pour manipuler des facteurs. Pour donner des exemples d'utilisation de ces différentes fonctions, nous allons utiliser le jeu de données `hdv2003` du package `{questionr}`.

```
library(tidyverse)
data("hdv2003", package = "questionr")
```

Considérons la variable *qualif* qui indique le niveau de qualification des enquêtés. On peut voir la liste des niveaux de ce facteur, et leur ordre, avec `levels()`, ou en effectuant un tri à plat avec la fonction `questionr::freq()`.

```
hdv2003$qualif |>
  levels()
```

```
[1] "Ouvrier specialise"       "Ouvrier qualifie"
[3] "Technicien"               "Profession intermediaire"
[5] "Cadre"                    "Employe"
[7] "Autre"
```

```
hdv2003$qualif |>
  questionr::freq()
```

```
                           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
```

Parfois, on a simplement besoin d'inverser l'ordre des facteurs, ce qui peut se faire facilement avec la fonction `forcats::fct_rev()`. Elle renvoie le facteur fourni en entrée en ayant inverser l'ordre des modalités (mais sans modifier l'ordre des valeurs dans le vecteur).

```
hdv2003$qualif |>
  fct_rev() |>
  questionr::freq()
```

```
                           n    % val%
Autre                     58  2.9  3.5
Employe                  594 29.7 35.9
Cadre                    260 13.0 15.7
Profession intermediaire 160  8.0  9.7
Technicien                86  4.3  5.2
```

```
Ouvrier qualifie         292 14.6 17.7
Ouvrier specialise       203 10.2 12.3
NA                       347 17.3   NA
```

Pour plus de contrôle, on utilisera `forcats::fct_relevel()` où l'on indique l'ordre souhaité des modalités. On peut également seulement indiquer les premières modalités, les autres seront ajoutées à la fin sans changer leur ordre.

```
hdv2003$qualif |>
  fct_relevel("Cadre", "Autre", "Technicien", "Employe") |>
  questionr::freq()
```

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

La fonction `forcats::fct_infreq()` ordonne les modalités de celle la plus fréquente à celle la moins fréquente (nombre d'observations) :

```
hdv2003$qualif |>
  fct_infreq() |>
  questionr::freq()
```

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

Pour inverser l'ordre, on combinera `forcats::fct_infreq()` avec `forcats::fct_rev()`.

```
hdv2003$qualif |>
  fct_infreq() |>
  fct_rev() |>
  questionr::freq()
```

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

Dans certains cas, on souhaite créer un facteur dont les modalités sont triées selon leur ordre d'apparation dans le jeu de données. Pour cela, on aura recours à `forcats::fct_inorder()`.

```
v <- c("c", "a", "d", "b", "a", "c")
factor(v)
```

```
[1] c a d b a c
Levels: a b c d
```

```
fct_inorder(v)
```

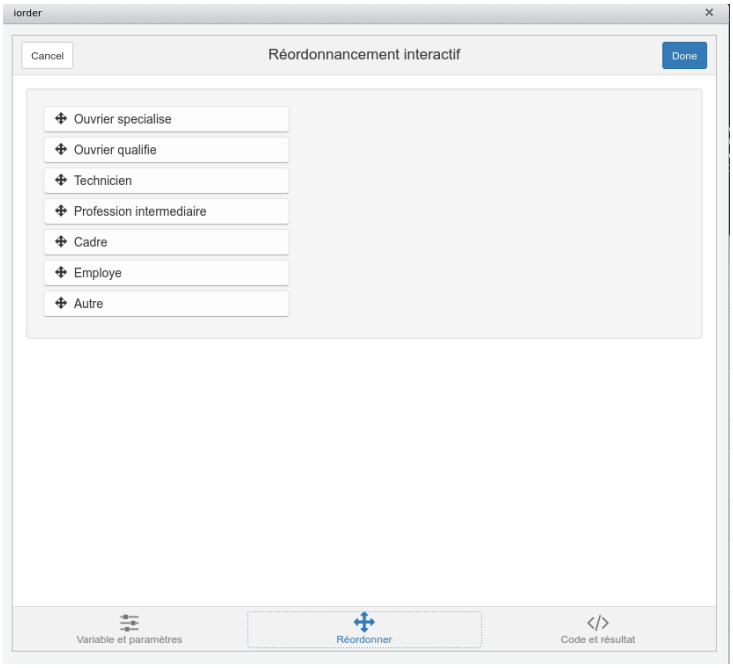
```
[1] c a d b a c
Levels: c a d b
```

La fonction `forcats::fct_reorder()` permets de trier les modalités en fonction d'une autre variable. Par exemple, si je souhaite trier les modalités de la variable *qualif* en fonction de l'âge moyen (dans chaque modalité) :

```
hdv2003$qualif_tri_age <-
  hdv2003$qualif |>
  fct_reorder(hdv2003$age, .fun = mean)
hdv2003 |>
  dplyr::group_by(qualif_tri_age) |>
  dplyr::summarise(age_moyen = mean(age))
```

```
# A tibble: 8 x 2
  qualif_tri_age           age_moyen
  <fct>                        <dbl>
1 Technicien                    45.9
2 Employe                       46.7
3 Autre                         47.0
4 Ouvrier specialise            48.9
5 Profession intermediaire      49.1
6 Cadre                         49.7
7 Ouvrier qualifie              50.0
8 <NA>                          47.9
```

> **Astuce**
>
> `{questionr}` propose une interface graphique afin de faciliter les opérations de réordonnancement manuel. Pour la lancer, sélectionner le menu *Addins* puis *Levels ordering*, ou exécuter la fonction `questionr::iorder()` en lui passant comme paramètre le facteur à réordonner.



Une démonstration en vidéo de cet *add-in* est disponible dans le webin-R #05 (*recoder des variables*) sur [YouTube](https://youtu.be/CokvTbtWdwc?t=3934).
https://youtu.be/CokvTbtWdwc

9.3 Modifier les modalités

Pour modifier le nom des modalités, on pourra avoir recours à `forcats::fct_recode()` avec une syntaxe de la forme `"nouveau nom" = "ancien nom"`.

```
hdv2003$sexe |>
  questionr::freq()
```

```
         n  % val%
Homme  899 45   45
Femme 1101 55   55
```

```
hdv2003$sexe <-
  hdv2003$sexe |>
  fct_recode(f = "Femme", m = "Homme")
hdv2003$sexe |>
  questionr::freq()
```

```
     n  % val%
m  899 45   45
f 1101 55   55
```

On peut également fusionner des modalités ensemble en leur attribuant le même nom.

```
hdv2003$nivetud |>
  questionr::freq()
```

```
                                                              n    %  val%
N'a jamais fait d'etudes                                     39  2.0   2.1
A arrete ses etudes, avant la derniere annee d'etudes primaires  86  4.3   4.6
Derniere annee d'etudes primaires                           341 17.0  18.1
1er cycle                                                   204 10.2  10.8
2eme cycle                                                  183  9.2   9.7
Enseignement technique ou professionnel court               463 23.2  24.5
Enseignement technique ou professionnel long                131  6.6   6.9
Enseignement superieur y compris technique superieur        441 22.0  23.4
NA                                                          112  5.6    NA
```

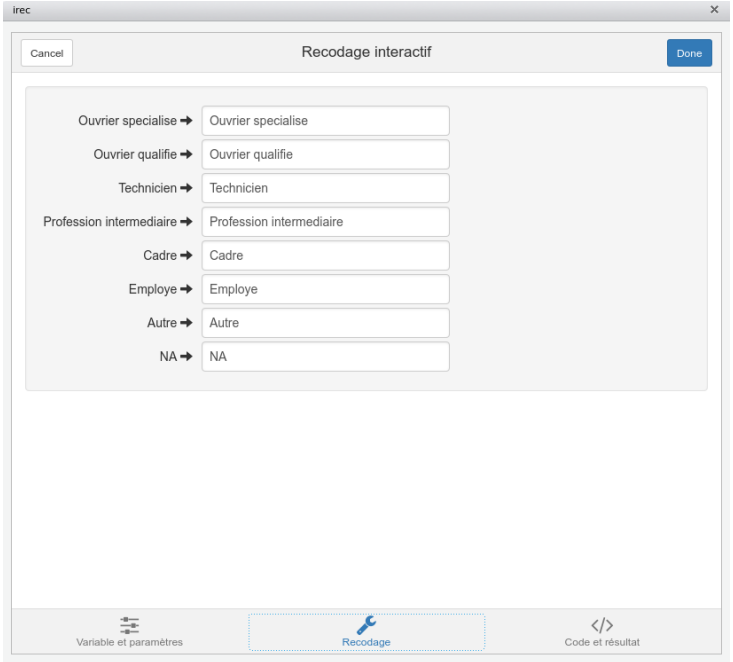
```
hdv2003$instruction <-
  hdv2003$nivetud |>
  fct_recode(
    "primaire" = "N'a jamais fait d'etudes",
    "primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
    "primaire" = "Derniere annee d'etudes primaires",
    "secondaire" = "1er cycle",
    "secondaire" = "2eme cycle",
    "technique/professionnel" = "Enseignement technique ou professionnel court",
    "technique/professionnel" = "Enseignement technique ou professionnel long",
    "supérieur" = "Enseignement superieur y compris technique superieur"
```

```
  )
hdv2003$instruction |>
  questionr::freq()
```

```
                           n    % val%
primaire                 466 23.3 24.7
secondaire               387 19.4 20.5
technique/professionnel  594 29.7 31.5
supérieur                441 22.0 23.4
NA                       112  5.6   NA
```

Interface graphique

Le package`{questionr}` propose une interface graphique facilitant le recodage des modalités d'une variable qualitative. L'objectif est de permettre à la personne qui l'utilise de saisir les nouvelles valeurs dans un formulaire, et de générer ensuite le code R correspondant au recodage indiqué.
Pour utiliser cette interface, sous **RStudio** vous pouvez aller dans le menu *Addins* (présent dans la barre d'outils principale) puis choisir *Levels recoding*. Sinon, vous pouvez lancer dans la console la fonction `questionr::irec()` en lui passant comme paramètre la variable à recoder.



> **Astuce**
>
> Une démonstration en vidéo de cet *add-in* est disponible dans le webin-R #05 (*recoder des variables*) sur [YouTube](https://youtu.be/CokvTbtWdwc?t=3387).
> https://youtu.be/CokvTbtWdwc

La fonction `forcats::fct_collapse()` est une variante de `forcats::fct_recode()` pour indiquer les fusions de modalités. La même recodification s'écrirait alors :

```
hdv2003$instruction <-
  hdv2003$nivetud |>
  fct_collapse(
    "primaire" = c(
      "N'a jamais fait d'etudes",
      "A arrete ses etudes, avant la derniere annee d'etudes primaires",
      "Derniere annee d'etudes primaires"
```

```
    ),
    "secondaire" = c(
      "1er cycle",
      "2eme cycle"
    ),
    "technique/professionnel" = c(
      "Enseignement technique ou professionnel court",
      "Enseignement technique ou professionnel long"
    ),
    "supérieur" = "Enseignement superieur y compris technique superieur"
  )
```

Pour transformer les valeurs manquantes (`NA`) en une modalité explicite, on pourra avoir recours à `forcats::fct_na_value_to_level()`[16]. Cette fonction s'appelait précédemment `forcats::fct_explicit_na()` et a été renommée depuis la version 1.0.0 de `{forcats}`.

[16]

```
hdv2003$instruction <-
  hdv2003$instruction |>
  fct_na_value_to_level(level = "(manquant)")
hdv2003$instruction |>
  questionr::freq()
```

```
                          n    % val%
primaire                466 23.3 23.3
secondaire              387 19.4 19.4
technique/professionnel 594 29.7 29.7
supérieur               441 22.0 22.0
(manquant)              112  5.6  5.6
```

Plusieurs fonctions permettent de regrouper plusieurs modalités dans une modalité *autres*.

Par exemple, avec `forcats::fct_other()`, on pourra indiquer les modalités à garder.

```
hdv2003$qualif |>
  questionr::freq()
```

```
                          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
```

```
hdv2003$qualif |>
  fct_other(keep = c("Technicien", "Cadre", "Employe")) |>
  questionr::freq()
```

```
             n    % val%
Technicien  86  4.3  5.2
Cadre      260 13.0 15.7
Employe    594 29.7 35.9
Other      713 35.6 43.1
NA         347 17.3   NA
```

La fonction `forcats::fct_lump_n()` permets de ne conserver que les modalités les plus fréquentes et de regrouper les autres dans une modalité *autres*.

```
hdv2003$qualif |>
  fct_lump_n(n = 4, other_level = "Autres") |>
  questionr::freq()
```

```
                    n    % val%
Ouvrier specialise 203 10.2 12.3
Ouvrier qualifie   292 14.6 17.7
Cadre              260 13.0 15.7
Employe            594 29.7 35.9
Autres             304 15.2 18.4
NA                 347 17.3   NA
```

Et `forcats::fct_lump_min()` celles qui ont un minimum d'observations.

```
hdv2003$qualif |>
  fct_lump_min(min = 200, other_level = "Autres") |>
  questionr::freq()
```

```
                     n    % val%
Ouvrier specialise 203 10.2 12.3
Ouvrier qualifie   292 14.6 17.7
Cadre              260 13.0 15.7
Employe            594 29.7 35.9
Autres             304 15.2 18.4
NA                 347 17.3   NA
```

Il peut arriver qu'une des modalités d'un facteur ne soit pas représentée dans les données.

```
v <- factor(
  c("a", "a", "b", "a"),
  levels = c("a", "b", "c")
)
questionr::freq(v)
```

```
  n  % val%
a 3 75   75
b 1 25   25
c 0  0    0
```

Pour calculer certains tests statistiques ou faire tourner un modèle, ces modalités sans observation peuvent être problématiques. `forcats::fct_drop()` permet de supprimer les modalités qui n'apparaissent pas dans les données.

```
v
```

```
[1] a a b a
Levels: a b c
```

```
v |> fct_drop()
```

```
[1] a a b a
Levels: a b
```

À l'inverse, `forcats::fct_expand()` permet d'ajouter une ou plusieurs modalités à un facteur.

```
v
```

```
[1] a a b a
Levels: a b c
```

```
v |> fct_expand("d", "e")
```

```
[1] a a b a
Levels: a b c d e
```

9.4 Découper une variable numérique en classes

Il est fréquent d'avoir besoin de découper une variable numérique en une variable catgéorielles (un facteur) à plusieurs modalités, par exemple pour créer des groupes d'âges à partir d'une variable *age*.

On utilise pour cela la fonction `cut()` qui 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 la variable *age* en cinq classes :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(age, 5))
hdv2003$groupe_ages |> questionr::freq()
```

```
              n    % val%
(17.9,33.8] 454 22.7 22.7
(33.8,49.6] 628 31.4 31.4
(49.6,65.4] 556 27.8 27.8
(65.4,81.2] 319 16.0 16.0
(81.2,97.1]  43  2.1  2.1
```

Par défaut **R** nous a bien créé cinq classes d'amplitudes égales. La première classe va de 17,9 à 33,8 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 :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(age, c(18, 20, 40, 60, 80, 97)))
hdv2003$groupe_ages |> questionr::freq()
```

```
          n    % val%
(18,20]  55  2.8  2.8
(20,40] 660 33.0 33.3
(40,60] 780 39.0 39.3
(60,80] 436 21.8 22.0
(80,97]  52  2.6  2.6
NA       17  0.9   NA
```

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, 18, 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 17, soit utiliser l'option `include.lowest=TRUE` :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(
    age,
    c(18, 20, 40, 60, 80, 97),
    include.lowest = TRUE
  ))
hdv2003$groupe_ages |> questionr::freq()
```

```
          n    % val%
[18,20]  72  3.6  3.6
(20,40] 660 33.0 33.0
(40,60] 780 39.0 39.0
(60,80] 436 21.8 21.8
(80,97]  52  2.6  2.6
```

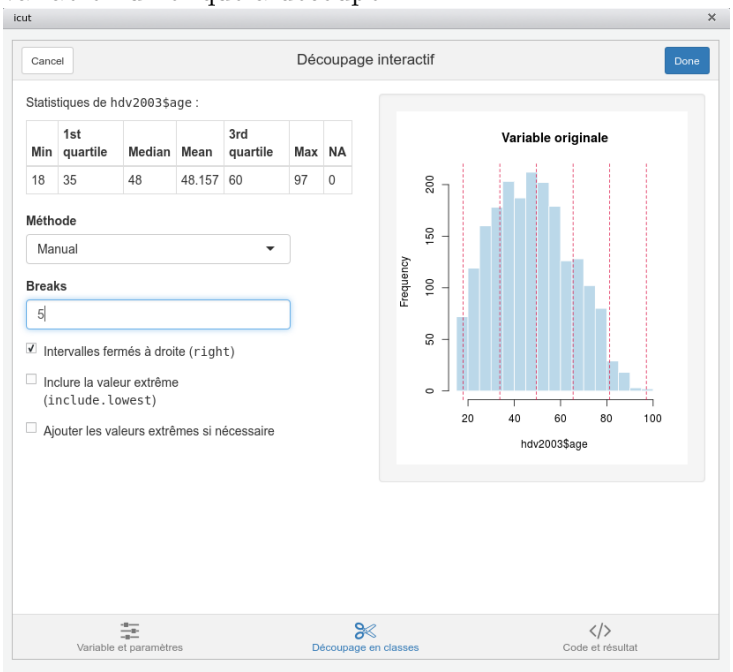
On peut également modifier le sens des intervalles avec l'option `right=FALSE` :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(
    age,
    c(18, 20, 40, 60, 80, 97),
    include.lowest = TRUE,
    right = FALSE
  ))
hdv2003$groupe_ages |> questionr::freq()
```

```
          n    % val%
[18,20)  48  2.4  2.4
[20,40) 643 32.1 32.1
[40,60) 793 39.6 39.6
[60,80) 454 22.7 22.7
[80,97]  62  3.1  3.1
```

Interface graphique

Il n'est pas nécessaire de connaître toutes les options de `cut()`. Le package `{questionr}` propose là encore une interface graphique permettant de visualiser l'effet des différents paramètres et de générer le code **R** correspondant. Pour utiliser cette interface, sous **RStudio** vous pouvez aller dans le menu *Addins* (présent dans la barre d'outils principale) puis choisir *Numeric range dividing*. Sinon, vous pouvez lancer dans la console la fonction `questionr::icut()` en lui passant comme paramètre la variable numérique à découper.



Une démonstration en vidéo de cet *add-in* est disponible dans le webin-R #05 (*recoder des variables*) sur [YouTube](https://youtu.be/CokvTbtWdwc?t=2795).
https://youtu.be/CokvTbtWdwc

10 Combiner plusieurs variables

Parfois, on a besoin de créer une nouvelle variable en partant des valeurs d'une ou plusieurs autres variables. Dans ce cas on peut utiliser les fonctions `dplyr::if_else()` pour les cas les plus simples, ou `dplyr::case_when()` pour les cas plus complexes.

Une fois encore, nous utiliser le jeu de données `hdv2003` pour illustrer ces différentes fonctions.

```
library(tidyverse)
data("hdv2003", package = "questionr")
```

10.1 if_else()

`dplyr::if_else()` prend trois arguments : un test, les valeurs à renvoyer si le test est vrai, et les valeurs à renvoyer si le test est faux.

Voici un exemple simple :

```
v <- c(12, 14, 8, 16)
if_else(v > 10, "Supérieur à 10", "Inférieur à 10")
```

```
[1] "Supérieur à 10" "Supérieur à 10" "Inférieur à 10" "Supérieur à 10"
```

La fonction devient plus intéressante avec des tests combinant plusieurs variables. Par exemple, imaginons qu'on souhaite créer une nouvelle variable indiquant les hommes de plus de 60 ans :

```
hdv2003 <-
  hdv2003 |>
  mutate(
    statut = if_else(
      sexe == "Homme" & age > 60,
      "Homme de plus de 60 ans",
      "Autre"
    )
  )
hdv2003 |>
  pull(statut) |>
  questionr::freq()
```

```
                           n    % val%
Autre                   1778 88.9 88.9
Homme de plus de 60 ans  222 11.1 11.1
```

Il est possible d'utiliser des variables ou des combinaisons de variables au sein du `dplyr::if_else()`. Suppons une petite enquête menée auprès de femmes et d'hommes. Le questionnaire comportait une question de préférence posée différemment aux femmes et aux hommes et dont les réponses ont ainsi été collectées dans deux variables différentes, *pref_f* et *pref_h*, que l'on souhaite combiner en une seule variable. De même, une certaine mesure quantitative a été réalisée, mais une correction est nécessaire pour normaliser ce score (retirer 0.4 aux scores des hommes et 0.6 aux scores des femmes). Cela peut être réalisé avec le code ci-dessous.

```
df <- tibble(
  sexe = c("f", "f", "h", "h"),
  pref_f = c("a", "b", NA, NA),
  pref_h = c(NA, NA, "c", "d"),
  mesure = c(1.2, 4.1, 3.8, 2.7)
)
df
```

```
# A tibble: 4 x 4
  sexe  pref_f pref_h mesure
```

```
  <chr> <chr>  <chr>   <dbl>
1 f     a      <NA>      1.2
2 f     b      <NA>      4.1
3 h     <NA>   c         3.8
4 h     <NA>   d         2.7
```

```
df <-
  df |>
  mutate(
    pref = if_else(sexe == "f", pref_f, pref_h),
    indicateur = if_else(sexe == "h", mesure - 0.4, mesure - 0.6)
  )
df
```

```
# A tibble: 4 x 6
  sexe  pref_f pref_h mesure pref  indicateur
  <chr> <chr>  <chr>   <dbl> <chr>      <dbl>
1 f     a      <NA>      1.2 a            0.6
2 f     b      <NA>      4.1 b            3.5
3 h     <NA>   c         3.8 c            3.4
4 h     <NA>   d         2.7 d            2.3
```

! if_else() et ifelse()

La fonction `dplyr::if_else()` ressemble à la fonction `ifelse()` en base **R**. Il y a néanmoins quelques petites différences :

- `dplyr::if_else()` vérifie que les valeurs fournies pour `true` et celles pour false sont du même type et de la même classe et renvoie une erreur dans le cas contraire, là où `ifelse()` sera plus permissif ;
- si un vecteur a des attributs (cf. Chapitre **??**), ils seront préservés par `dplyr::if_else()` (et pris dans le vecteur `true`), ce que ne fera pas `if_else()` ;
- `dplyr::if_else()` propose un argument optionnel supplémentaire `missing` pour indiquer les valeurs à retourner lorsque le test renvoie `NA`.

10.2 case_when()

`dplyr::case_when()` est une généralisation de `dplyr::if_else()` qui permet d'indiquer plusieurs tests et leurs valeurs associées.

Imaginons que l'on souhaite créer une nouvelle variable permettant d'identifier les hommes de plus de 60 ans, les femmes de plus de 60 ans, et les autres. On peut utiliser la syntaxe suivante :

```
hdv2003 <-
  hdv2003 |>
  mutate(
    statut = case_when(
      age >= 60 & sexe == "Homme" ~ "Homme, 60 et plus",
      age >= 60 & sexe == "Femme" ~ "Femme, 60 et plus",
      TRUE ~ "Autre"
    )
  )
hdv2003 |>
  pull(statut) |>
  questionr::freq()
```

```
                     n    % val%
Autre             1484 74.2 74.2
Femme, 60 et plus  278 13.9 13.9
Homme, 60 et plus  238 11.9 11.9
```

`dplyr::case_when()` prend en arguments une série d'instructions sous la forme `condition ~ valeur`. Il les exécute une par une, et dès qu'une `condition` est vraie, il renvoit la `valeur` associée.

La clause `TRUE ~ "Autre"` permet d'assigner une valeur à toutes les lignes pour lesquelles aucune des conditions précédentes n'est vraie.

! Important

Attention : comme les conditions sont testées l'une après l'autre et que la valeur renvoyée est celle correspondant à la première condition vraie, l'ordre de ces conditions est très important. Il faut absolument aller du plus spécifique au plus général.
Par exemple le recodage suivant ne fonctionne pas :

```
hdv2003 <-
  hdv2003 |>
  mutate(
    statut = case_when(
      sexe == "Homme" ~ "Homme",
      age >= 60 & sexe == "Homme" ~ "Homme, 60 et plus",
      TRUE ~ "Autre"
    )
  )
hdv2003 |>
  pull(statut) |>
  questionr::freq()
```

```
       n  % val%
Autre 1101 55   55
Homme  899 45   45
```

Comme la condition `sexe == "Homme"` est plus générale que `sexe == "Homme" & age > 60`, cette deuxième condition n'est jamais testée ! On n'obtiendra jamais la valeur correspondante.
Pour que ce recodage fonctionne il faut donc changer l'ordre des conditions pour aller du plus spécifique au plus général :

```
hdv2003 <-
  hdv2003 |>
  mutate(
    statut = case_when(
      age >= 60 & sexe == "Homme" ~ "Homme, 60 et plus",
      sexe == "Homme" ~ "Homme",
      TRUE ~ "Autre"
    )
  )
hdv2003 |>
  pull(statut) |>
  questionr::freq()
```

```
                     n    % val%
Autre             1101 55.0 55.0
Homme              661 33.1 33.1
Homme, 60 et plus  238 11.9 11.9
```

C'est pour cela que l'on peut utiliser, en toute dernière condition, la valeur `TRUE` pour indiquer dans tous les autres cas.

10.3 recode_if()

Parfois, on n'a besoin de ne modifier une variable que pour certaines observations. Prenons un petit exemple :

```
df <- tibble(
  pref = factor(c("bleu", "rouge", "autre", "rouge", "autre")),
  autre_details = c(NA, NA, "bleu ciel", NA, "jaune")
)
df
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 bleu  <NA>
```

```
2 rouge <NA>
3 autre bleu ciel
4 rouge <NA>
5 autre jaune
```

Nous avons demandé aux enquêtés d'indiquer leur couleur préférée. Ils pouvaient répondre bleu ou rouge et avait également la possibilité de choisir autre et d'indiquer la valeur de leur choix dans un champs textuel libre.

Une des personnes enquêtées a choisi autre et a indiqué dans le champs texte la valeur bleu ciel. Pour les besoins de l'analyse, on peut considérer que cette valeur bleu ciel pour être tout simplement recodée en bleu.

En syntaxe **R** classique, on pourra simplement faire :

```
df$pref[df$autre_details == "bleu ciel"] <- "bleu"
```

Avec `dplyr::if_else()`, on serait tenté d'écrire :

```
df |>
  mutate(pref = if_else(autre_details == "bleu ciel", "bleu", pref))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <chr> <chr>
1 <NA>  <NA>
2 <NA>  <NA>
3 bleu  bleu ciel
4 <NA>  <NA>
5 autre jaune
```

On obtient une erreur, car `dplyr::if_else()` exige les valeurs fournie pour `true` et `false` soient de même type. Essayons alors :

```
df |>
  mutate(pref = if_else(autre_details == "bleu ciel", factor("bleu"), pref))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 <NA>  <NA>
2 <NA>  <NA>
3 bleu  bleu ciel
4 <NA>  <NA>
5 autre jaune
```

Ici nous avons un autre problème, signalé par un message d'avertissement (*warning*) : `dplyr::if_else()` ne préserve que les attributs du vecteur passé en `true` et non ceux passés à `false`. Or l'ensemble des modalités (niveaux du facteur) de la variable *pref* n'ont pas été définis dans `factor("bleu")` et sont ainsi perdus, générant une perte de données (valeurs manquantes `NA`).

Pour obtenir le bon résultat, il faudrait inverser la condition :

```
df |>
  mutate(pref = if_else(
    autre_details != "bleu ciel",
    pref,
    factor("bleu")
  ))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 <NA>  <NA>
2 <NA>  <NA>
3 bleu  bleu ciel
4 <NA>  <NA>
5 autre jaune
```

Mais ce n'est toujours pas suffisant. En effet, la variable *autre_details* a des valeurs manquantes pour lesquelles le test `autre_details != "bleu ciel"` renvoie `NA` ce qui une fois encore génère des valeurs manquantes non souhaitées.

Dès lors, il nous faut soit définir l'argument `missing` de `dplyr::if_else()`, soit être plus précis dans notre test.

```
  df |>
    mutate(pref = if_else(
      autre_details != "bleu ciel",
      pref,
      factor("bleu"),
      missing = pref
    ))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

```
  df |>
    mutate(pref = if_else(
      autre_details != "bleu ciel" | is.na(autre_details),
      pref,
      factor("bleu")
    ))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

Bref, on peut s'en sortir avec `dplyr::if_else()` mais ce n'est pas forcément le plus pratique dans le cas présent. La syntaxe

en base **R** fonctionne très bien, mais ne peut pas être intégrée à un enchainement d'opérations utilisant le *pipe*.

Dans ce genre de situation, on pourra être intéressé par la fonction `labelled::recode_if()` disponible dans le package `{labelled}`. Elle permet de ne modifier que certaines observations d'un vecteur en fonction d'une condition. Si la condition vaut `FALSE` ou `NA`, les observations concernées restent inchangées. Voyons comment cela s'écrit :

```
df <-
  df |>
  mutate(
    pref = pref |>
      labelled::recode_if(autre_details == "bleu ciel", "bleu")
  )
df
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

C'est tout de suite plus intuitif !

11 Étiquettes de variables

11.1 Principe

Les étiquettes de variable permettent de donner un nom long, plus explicite, aux différentes colonnes d'un tableau de données (ou encore directement à un vecteur autonome). Dans le champs des grandes enquêtes, il est fréquent de nommer les variables *q101*, *q102*, etc. pour refléter le numéro de la question et d'indiquer ce qu'elle réprésente (groupe d'âges, milieur de résidence…) avec une étiquette.

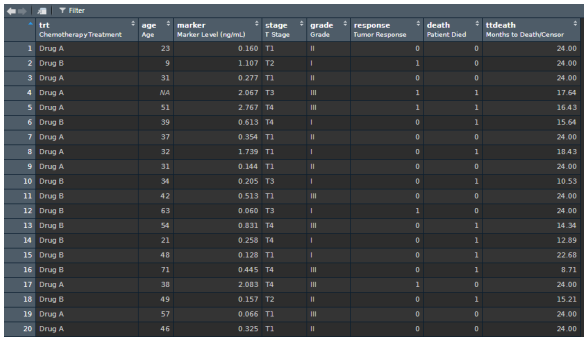
Un usage, introduit par le package `{haven}`, et repris depuis par de nombreux autres packages dont `{gtsummary}` que nous aborderons dans de prochains chapitres, consiste à stocker les étiquettes de variables sous la forme d'un attribut[17] `"label"` attaché au vecteur / à la colonne du tableau.

[17] Pour plus d'information sur les attributs, voir Chapitre **??**.

Le package `{labelled}` permet de manipuler aisément ces étiquettes de variables.

La visonneuse de données de **RStudio** sait reconnaître et afficher ces étiquettes de variable lorsqu'elles existent. Prenons pour exemple le jeu de données `gtsummary::trial` dont les colonnes ont des étiquettes de variable. La commande `View(gtsummary::trial)` permet d'ouvrir la visionneuse de données de **RStudio**. Comme on peut le constater, une étiquette de variable est bien présente sous le nom des différentes colonnes.

La fonction `labelled::look_for()` du package `{labelled}` permet de lister l'ensemble des variables d'un tableau de données et affiche notamment les étiquettes de variable associées.



	trt Chemotherapy Treatment	age Age	marker Marker Level (ng/mL)	stage T Stage	grade Grade	response Tumor Response	death Patient Died	ttdeath Months to Death/Censor
1	Drug A	23	0.160	T1	II	0	0	24.00
2	Drug B	9	1.107	T2	I	1	0	24.00
3	Drug A	31	0.277	T1	II	0	0	24.00
4	Drug A	NA	2.067	T3	III	1	1	17.64
5	Drug A	51	2.767	T4	III	1	1	16.43
6	Drug B	39	0.613	T4	I	0	1	15.64
7	Drug A	37	0.354	T1	II	0	0	24.00
8	Drug A	32	1.739	T1	I	0	1	18.43
9	Drug A	31	0.144	T1	II	0	0	24.00
10	Drug B	34	0.205	T3	I	0	1	10.53
11	Drug B	42	0.513	T1	III	0	0	24.00
12	Drug B	63	0.060	T3	I	1	0	24.00
13	Drug B	54	0.831	T4	III	0	1	14.34
14	Drug B	21	0.258	T4	I	0	1	12.89
15	Drug B	48	0.128	T1	I	0	1	22.68
16	Drug B	71	0.445	T4	III	0	1	8.71
17	Drug A	38	2.083	T4	III	1	0	24.00
18	Drug B	49	0.157	T2	II	0	1	15.21
19	Drug A	57	0.066	T1	III	0	0	24.00
20	Drug A	46	0.325	T1	II	0	0	24.00

Figure 11.1: Présentation du tableau `gtsummary::trial` dans la visionneuse de **RStudio**

```
library(labelled)
gtsummary::trial |>
  look_for()
```

```
pos variable label                   col_type missing values
1   trt      Chemotherapy Treatment  chr      0
2   age      Age                     dbl      11
3   marker   Marker Level (ng/mL)    dbl      10
4   stage    T Stage                 fct      0       T1
                                                     T2
                                                     T3
                                                     T4
5   grade    Grade                   fct      0       I
                                                     II
                                                     III
6   response Tumor Response          int      7
7   death    Patient Died            int      0
8   ttdeath  Months to Death/Censor  dbl      0
```

La fonction `labelled::look_for()` permet également de rechercher des variables en tenant compte à la fois de leur nom et de leur étiquette.

```
gtsummary::trial |>
  look_for("months")
```

```
pos variable label                  col_type missing values
8   ttdeath  Months to Death/Censor dbl      0
```

Astuce

Comme on le voit, la fonction `labelled::look_for()` est tout à fait adaptée pour générer un dictionnaire de codification. Ses différentes options sont détaillées dans une vignette dédiée. Les résultats renvoyés par `labelled::look_for()` sont récupérables dans un tableau de données que l'on pourra ainsi manipuler à sa guise.

```
gtsummary::trial |>
  look_for() |>
  dplyr::as_tibble()
```

```
# A tibble: 8 x 7
    pos variable label                  col_type missing levels     value_labels
  <int> <chr>    <chr>                  <chr>      <int> <named li> <named list>
1     1 trt      Chemotherapy Treatment chr            0 <NULL>     <NULL>
2     2 age      Age                    dbl           11 <NULL>     <NULL>
3     3 marker   Marker Level (ng/mL)   dbl           10 <NULL>     <NULL>
4     4 stage    T Stage                fct            0 <chr [4]>  <NULL>
5     5 grade    Grade                  fct            0 <chr [3]>  <NULL>
6     6 response Tumor Response         int            7 <NULL>     <NULL>
7     7 death    Patient Died           int            0 <NULL>     <NULL>
8     8 ttdeath  Months to Death/Censor dbl            0 <NULL>     <NULL>
```

11.2 Manipulation sur un vecteur / une colonne

La fonction `labelled::var_label()` permets de voir l'étiquette de variable attachée à un vecteur (renvoie `NULL` s'il n'y en a pas) mais également d'ajouter/modifier une étiquette.

Le fait d'ajouter une étiquette de variable à un vecteur ne modifie en rien son type ni sa classe. On peut associer une étiquette

de variable à n'importe quel type de variable, qu'elle soit numérique, textuelle, un facteur ou encore des dates.

```
v <- c(1, 5, 2, 4, 1)
v |> var_label()
```

```
NULL
```

```
var_label(v) <- "Mon étiquette"
var_label(v)
```

```
[1] "Mon étiquette"
```

```
str(v)
```

```
 num [1:5] 1 5 2 4 1
 - attr(*, "label")= chr "Mon étiquette"
```

```
var_label(v) <- "Une autre étiquette"
var_label(v)
```

```
[1] "Une autre étiquette"
```

```
str(v)
```

```
 num [1:5] 1 5 2 4 1
 - attr(*, "label")= chr "Une autre étiquette"
```

Pour supprimer une étiquette, il suffit d'attribuer la valeur `NULL`.

```
var_label(v) <- NULL
str(v)
```

```
num [1:5] 1 5 2 4 1
```

On peut appliquer `labelled::var_label()` directement sur une colonne de tableau.

```
var_label(iris$Petal.Length) <- "Longueur du pétale"
var_label(iris$Petal.Width) <- "Largeur du pétale"
var_label(iris$Species) <- "Espèce"
iris |>
  look_for()
```

```
pos variable     label              col_type missing values
1   Sepal.Length -                  dbl      0
2   Sepal.Width  -                  dbl      0
3   Petal.Length Longueur du pétale dbl      0
4   Petal.Width  Largeur du pétale  dbl      0
5   Species      Espèce             fct      0       setosa
                                                     versicolor
                                                     virginica
```

11.3 Manipulation sur un tableau de données

La fonction `labelled::set_variable_labels()` permets de manipuler les étiquettes de variable d'un tableau de données avec une syntaxe du type `{dplyr}`.

```
iris <-
  iris |>
  set_variable_labels(
    Species = NULL,
    Sepal.Length = "Longeur du sépale"
  )
iris |>
  look_for()
```

```
pos variable     label              col_type missing values
1   Sepal.Length Longeur du sépale  dbl      0
2   Sepal.Width  -                  dbl      0
```

```
3   Petal.Length Longueur du pétale dbl      0
4   Petal.Width  Largeur du pétale  dbl      0
5   Species      -                 fct      0       setosa
                                                    versicolor
                                                    virginica
```

11.4 Préserver les étiquettes

Certaines fonctions de **R** ne préservent pas les attributs et risquent donc d'effacer les étiquettes de variables que l'on a définit. Un exemple est la fonction générique **subset()** qui permet de sélectionner certaines lignes remplissant une certaines conditions.

```
iris |>
  look_for()
```

```
pos variable     label              col_type missing values
1   Sepal.Length Longueur du sépale  dbl      0
2   Sepal.Width  -                   dbl      0
3   Petal.Length Longueur du pétale  dbl      0
4   Petal.Width  Largeur du pétale   dbl      0
5   Species      -                   fct      0       setosa
                                                      versicolor
                                                      virginica
```

```
iris |>
  subset(Species == "setosa") |>
  look_for()
```

```
pos variable     label col_type missing values
1   Sepal.Length -     dbl      0
2   Sepal.Width  -     dbl      0
3   Petal.Length -     dbl      0
4   Petal.Width  -     dbl      0
5   Species      -     fct      0       setosa
                                        versicolor
                                        virginica
```

On pourra, dans ce cas précis, préférer la fonction `dplyr::filter()` qui préserve les attributs et donc les étiquettes de variables.

```
iris |>
  dplyr::filter(Species == "setosa") |>
  look_for()
```

```
pos variable     label              col_type missing values
1   Sepal.Length Longeur du sépale  dbl      0
2   Sepal.Width  -                  dbl      0
3   Petal.Length Longueur du pétale dbl      0
4   Petal.Width  Largeur du pétale  dbl      0
5   Species      -                  fct      0       setosa
                                                    versicolor
                                                    virginica
```

On pourra également tirer parti de la fonction `labelled::copy_labels_from()` qui permet de copier les étiquettes d'un tabeau à un autre.

```
iris |>
  subset(Species == "setosa") |>
  copy_labels_from(iris) |>
  look_for()
```

```
pos variable     label              col_type missing values
1   Sepal.Length Longeur du sépale  dbl      0
2   Sepal.Width  -                  dbl      0
3   Petal.Length Longueur du pétale dbl      0
4   Petal.Width  Largeur du pétale  dbl      0
5   Species      -                  fct      0       setosa
                                                    versicolor
                                                    virginica
```

12 Étiquettes de valeurs

Dans le domaine des grandes enquêtes, il est fréquent de coder les variables catégorielles avec des codes numériques auxquels on associé une certaines valeurs. Par exemple, une variable *milieu de résidence* pourrait être codée 1 pour urbain, 2 pour semi-urbain, 3 pour rural et 9 pour indiquer une donnée manquante. Une variable binaire pourrait quant à elle être codée 0 pour non et 1 pour oui. Souvent, chaque enquête définit ses propres conventions.

Les logiciels statistiques propriétaires **SPSS**, **Stata** et **SAS** ont tous les trois un système d'étiquettes de valeurs pour représenter ce type de variables catégorielles.

R n'a pas, de manière native, de système d'étiquettes de valeurs. Le format utilisé en interne pour représenter les variables catégorielles est celui des facteurs (cf. Chapitre **??**). Cependant, ce dernier ne permet de contrôler comment sont associées une étiquette avec une valeur numérique précise.

12.1 La classe `haven_labelled`

Afin d'assurer une importation complète des données depuis **SPSS**, **Stata** et **SAS**, le package `{haven}` a introduit un nouveau type de vecteurs, la classe `haven_labelled`, qui permet justement de rendre compte de ces vecteurs labellisés (i.e. avec des étiquettes de valeurs). Le package `{labelled}` fournie un jeu de fonctions pour faciliter la manipulation des vecteurs labellisés.

> ! Important
>
> Les vecteurs labellisés sont un format intermédiaire qui permets d'importer les données telles qu'elles ont été définies dans le fichier source. Il n'est pas destiné à être utilisé pour l'analyse statistique.
> Pour la réalisation de tableaux, graphiques, modèles, **R** attend que les variables catégorielles soit codées sous formes de facteurs, et que les variables continues soient numériques. On aura donc besoin, à un moment ou à un autre, de convertir les vecteurs labellisés en facteurs ou en variables numériques classiques.

12.2 Manipulation sur un vecteur / une colonne

Pour définir des étiquettes, la fonction de base est `labelled::val_labels()`. Il est possible de définir des étiquettes de valeurs pour des vecteurs numériques, d'entiers et textuels. On indiquera les étiquettes sous la forme `étiquette = valeur`. Cette fonction s'utilise de la même manière que `labelled::var_label()` abordée au chapitre précédent (cf. Chapitre **??**). Un appel simple renvoie les étiquettes de valeur associées au vecteur, `NULL` s'il n'y en n'a pas. Combiner avec l'opérateur d'assignation (`<-`), on peut ajouter/modifier les étiquettes de valeurs associées au vecteur.

```
library(labelled)
v <- c(1, 2, 1, 9)
v
```

```
[1] 1 2 1 9
```

```
class(v)
```

```
[1] "numeric"
```

```
  val_labels(v)
```

```
NULL
```

```
  val_labels(v) <- c(non = 1, oui = 2)
  val_labels(v)
```

```
non oui
  1   2
```

```
  v
```

```
<labelled<double>[4]>
[1] 1 2 1 9

Labels:
 value label
     1   non
     2   oui
```

```
  class(v)
```

```
[1] "haven_labelled" "vctrs_vctr"     "double"
```

Comme on peut le voir avec cet exemple simple :

- l'ajout d'étiquettes de valeurs modifie la class de l'objet (qui est maintenant un vecteur de la classe `haven_labelled`) ;
- l'objet obtenu est multi-classes, la classe `double` indiquant ici qu'il s'agit d'un vecteur numérique ;
- il n'est pas obligatoire d'associer une étiquette de valeurs à toutes les valeurs observées dans le vecteur (ici, nous n'avons pas défini d'étiquettes pour la valeur `9`).

La fonction `labelled::val_label()` (notez l'absence d'un s à la fin du nom de la fonction) permet d'accéder / de modifier l'étiquette associée à une valeur spécifique.

```
  val_label(v, 1)
```

```
[1] "non"
```

```
  val_label(v, 9)
```

```
NULL
```

```
  val_label(v, 9) <- "(manquant)"
  val_label(v, 2) <- NULL
  v
```

```
<labelled<double>[4]>
[1] 1 2 1 9

Labels:
 value      label
     1        non
     9 (manquant)
```

Pour supprimer, toutes les étiquettes de valeurs, on attribuera `NULL` avec `labelled::val_labels()`.

```
  val_labels(v) <- NULL
  v
```

```
[1] 1 2 1 9
```

```
  class(v)
```

```
[1] "numeric"
```

On remarquera que, lorsque toutes les étiquettes de valeurs sont supprimées, la nature de l'objet change à nouveau et il redevient un simple vecteur numérique.

Mise en garde

Il est essentiel de bien comprendre que l'ajout d'étiquettes de valeurs ne change pas fondamentalement la nature du vecteur. **Cela ne le transforme pas en variable catégorielle.** À ce stade, le vecteur n'a pas été transformé en facteur. Cela reste un vecteur numérique qui est considéré comme tel par **R**. On peut ainsi en calculer une moyenne, ce qui serait impossible avec un facteur.

```
v <- c(1, 2, 1, 2)
val_labels(v) <- c(non = 1, oui = 2)
mean(v)
```

```
[1] 1.5
```

```
f <- factor(v, levels = c(1, 2), labels = c("non", "oui"))
mean(f)
```

```
Warning in mean.default(f): l'argument n'est ni numérique, ni logique : renvoi
de NA

[1] NA
```

Les fonctions `labelled::val_labels()` et `labelled::val_label()` peuvent également être utilisées sur les colonnes d'un tableau.

```
df <- dplyr::tibble(
  x = c(1, 2, 1, 2),
  y = c(3, 9, 9, 3)
)
val_labels(df$x) <- c(non = 1, oui = 2)
val_label(df$y, 9) <- "(manquant)"
df
```

```
# A tibble: 4 x 2
  x         y
  <dbl+lbl> <dbl+lbl>
1 1 [non]   3
```

```
2 2 [oui]   9 [(manquant)]
3 1 [non]   9 [(manquant)]
4 2 [oui]   3
```

On pourra noter, que si notre tableau est un *tibble*, les étiquettes sont rendues dans la console quand on affiche le tableau.

La fonction `labelled::look_for()` est également un bon moyen d'afficher les étiquettes de valeurs.

```
df |>
  look_for()
```

```
 pos variable label col_type missing values
 1   x        -     dbl+lbl  0       [1] non
                                     [2] oui
 2   y        -     dbl+lbl  0       [9] (manquant)
```

12.3 Manipulation sur un tableau de données

{labelled} fournie 3 fonctions directement applicables sur un tableau de données : `labelled::set_value_labels()`, `labelled::add_value_labels()` et `labelled::remove_value_labels()`. La première remplace l'ensemble des étiquettes de valeurs associées à une variable, la seconde ajoute des étiquettes de valeurs (et conserve celles déjà définies), la troisième supprime les étiquettes associées à certaines valeurs spécifiques (et laisse les autres inchangées).

```
df |>
  look_for()
```

```
 pos variable label col_type missing values
 1   x        -     dbl+lbl  0       [1] non
                                     [2] oui
 2   y        -     dbl+lbl  0       [9] (manquant)
```

```r
df <- df |>
  set_value_labels(
    x = c(yes = 2),
    y = c("a répondu" = 3, "refus de répondre" = 9)
  )
df |>
  look_for()
```

```
pos variable label col_type missing values
1   x        -     dbl+lbl  0       [2] yes
2   y        -     dbl+lbl  0       [3] a répondu
                                    [9] refus de répondre
```

```r
df <- df |>
  add_value_labels(
    x = c(no = 1)
  ) |>
  remove_value_labels(
    y = 9
  )
df |>
  look_for()
```

```
pos variable label col_type missing values
1   x        -     dbl+lbl  0       [2] yes
                                    [1] no
2   y        -     dbl+lbl  0       [3] a répondu
```

12.4 Conversion

12.4.1 Quand convertir les vecteurs labellisés ?

La classe `haven_labelled` permets d'ajouter des métadonnées aux variables sous la forme d'étiquettes de valeurs. Lorsque les données sont importées depuis **SAS**, **SPSS** ou **Stata**, cela permet notamment de conserver le codage original du fichier importé.

Mais il faut noter que ces *étiquettes de valeur* n'indique pas pour autant de manière systématique le type de variable (catégorielle ou continue). Les vecteurs labellisés n'ont donc pas vocation à être utilisés pour l'analyse, notamment le calcul de modèles statistiques. Ils doivent être convertis en facteurs (pour les variables catégorielles) ou en vecteurs numériques (pour les variables continues).

La question qui peut se poser est donc de choisir à quel moment cette conversion doit avoir lieu dans un processus d'analyse. On peut considérer deux approches principales.

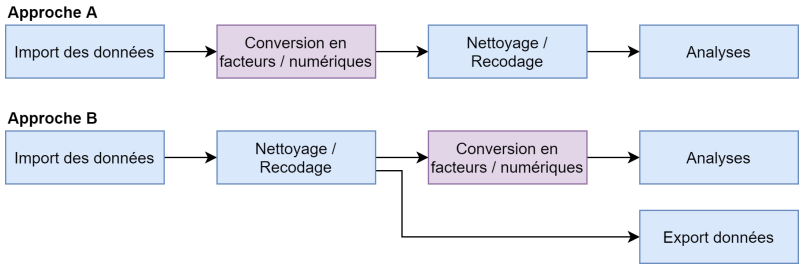


Figure 12.1: Deux approches possibles pour la conversion des étiquettes de valeurs

Dans l'**approche A**, les vecteurs labellisés sont convertis juste après l'import des données, en utilisant les fonctions `labelled::unlabelled()`, `labelled::to_factor()` ou `base::unclass()` qui sont présentées ci-après. Dès lors, toute la partie de nettoyage et de recodage des données se fera en utilisant les fonctions classiques de **R**. Si l'on n'a pas besoin de conserver le codage original, cette approche a l'avantage de s'inscrire dans le fonctionnement usuel de **R**.

Dans l'**approche B**, les vecteurs labellisés sont conservés pour l'étape de nettoyage et de recodage des données. Dans ce cas là, on pourra avoir recours aux fonctions de l'extension `{labelled}` qui facilitent la gestion des données labellisées. Cette approche est particulièrement intéressante quand (i) on veut pouvoir se référer au dictionnaire de codification fourni avec les données sources et donc on veut conserver le codage original et/ou (ii) quand les données devront faire l'objet d'un réexport après transformation. Par contre, comme dans

l'approche A, il faudra prévoir une conversion des variables labellisées au moment de l'analyse.

> **Avertissement**
>
> Dans tous les cas, il est recommandé d'adopter l'une ou l'autre approche, mais d'éviter de mélanger les différents types de vecteur. Une organisation rigoureuse de ses données et de son code est essentielle !

12.4.2 Convertir un vecteur labellisé en facteur

Il est très facile de convertir un vecteur labellisé en facteur à l'aide la fonction `labelled::to_factor()` du package `{labelled}`[18].

[18] On priviligiera la fonction `labelled::to_factor()` à la fonction `haven::as_factor()` de l'extension `{haven}`, la première ayant plus de possibilités et un comportement plus consistent.

```
v <- c(1,2,9,3,3,2,NA)
val_labels(v) <- c(
  oui = 1, "peut-être" = 2,
  non = 3, "ne sait pas" = 9
)
v
```

```
<labelled<double>[7]>
[1]  1  2  9  3  3  2 NA

Labels:
 value       label
     1         oui
     2   peut-être
     3         non
     9 ne sait pas
```

```
to_factor(v)
```

```
[1] oui         peut-être   ne sait pas non         non         peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

Il possible d'indiquer si l'on souhaite, comme étiquettes du facteur, utiliser les étiquettes de valeur (par défaut), les valeurs elles-mêmes, ou bien les étiquettes de valeurs préfixées par la valeur d'origine indiquée entre crochets.

```
to_factor(v, 'l')
```

```
[1] oui         peut-être   ne sait pas non         non         peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

```
to_factor(v, 'v')
```

```
[1] 1    2    9    3    3    2    <NA>
Levels: 1 2 3 9
```

```
to_factor(v, 'p')
```

```
[1] [1] oui         [2] peut-être   [9] ne sait pas [3] non
[5] [3] non         [2] peut-être   <NA>
Levels: [1] oui [2] peut-être [3] non [9] ne sait pas
```

Par défaut, les modalités du facteur seront triées selon l'ordre des étiquettes de valeur. Mais cela peut être modifié avec l'argument `sort_levels` si l'on préfère trier selon les valeurs ou selon l'ordre alphabétique des étiquettes.

```
to_factor(v, sort_levels = 'v')
```

```
[1] oui         peut-être   ne sait pas non         non         peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

```
to_factor(v, sort_levels = 'l')
```

```
[1] oui         peut-être   ne sait pas non         non         peut-être
[7] <NA>
Levels: ne sait pas non oui peut-être
```

12.4.3 Convertir un vecteur labellisé en numérique ou en texte

Pour rappel, il existe deux types de vecteurs labellisés : des vecteurs numériques labellisés (`x` dans l'exemple ci-dessous) et des vecteurs textuels labellisés (`y` dans l'exemple ci-dessous).

```
x <- c(1, 2, 9, 3, 3, 2, NA)
val_labels(x) <- c(
  oui = 1, "peut-être" = 2,
  non = 3, "ne sait pas" = 9
)

y <- c("f", "f", "h", "f")
val_labels(y) <- c(femme = "f", homme = "h")
```

Pour leur retirer leur caractère labellisé et revenir à leur classe d'origine, on peut utiliser la fonction `unclass()`.

```
unclass(x)
```

```
[1]  1  2  9  3  3  2 NA
attr(,"labels")
        oui   peut-être         non ne sait pas
          1           2           3           9
```

```
unclass(y)
```

```
[1] "f" "f" "h" "f"
attr(,"labels")
femme homme
  "f"   "h"
```

À noter que dans ce cas-là, les étiquettes sont conservées comme attributs du vecteur.

Une alternative est d'utiliser `labelled::remove_labels()` qui supprimera toutes les étiquettes, y compris les étiquettes de variable. Pour conserver les étiquettes de variables et ne supprimer que les étiquettes de valeurs, on indiquera `keep_var_label = TRUE`.

```
var_label(x) <- "Etiquette de variable"
remove_labels(x)
```

```
[1]  1  2  9  3  3  2 NA
```

```
remove_labels(x, keep_var_label = TRUE)
```

```
[1]  1  2  9  3  3  2 NA
attr(,"label")
[1] "Etiquette de variable"
```

```
remove_labels(y)
```

```
[1] "f" "f" "h" "f"
```

Dans le cas d'un vecteur numérique labellisé que l'on souhaiterait convertir en variable textuelle, on pourra utiliser `labelled::to_character()` à la place de `labelled::to_factor()` qui, comme sa grande soeur, utilisera les étiquettes de valeurs.

```
to_character(x)
```

```
[1] "oui"         "peut-être"   "ne sait pas" "non"         "non"
[6] "peut-être"   NA
attr(,"label")
[1] "Etiquette de variable"
```

12.4.4 Conversion conditionnelle en facteurs

Il n'est pas toujours possible de déterminer la nature d'une variable (continue ou catégorielle) juste à partir de la présence ou l'absence d'étiquettes de valeur. En effet, on peut utiliser des étiquettes de valeur dans le cadre d'une variable continue pour indiquer certaines valeurs spécifiques.

Une bonne pratique est de vérifier chaque variable inclue dans une analyse, une à une.

Cependant, une règle qui fonctionne dans 90% des cas est de convertir un vecteur labellisé en facteur si et seulement si toutes les valeurs observées dans le vecteur disposent d'une étiquette de valeur correspondante. C'est ce que propose la fonction `labelled::unlabelled()` qui peut même être appliqué à tout un tableau de données. Par défaut, elle fonctionne ainsi :

1. les variables non labellisées restent inchangées (variables f et g dans l'exemple ci-dessous);
2. si toutes les valeurs observées d'une variable labellisées ont une étiquette, elles sont converties en facteurs (variables b et c);
3. sinon, on leur applique `base::unclass()` (variables a, d et e).

```
df <- dplyr::tibble(
  a = c(1, 1, 2, 3),
  b = c(1, 1, 2, 3),
  c = c(1, 1, 2, 2),
  d = c("a", "a", "b", "c"),
  e = c(1, 9, 1, 2),
  f = 1:4,
  g = as.Date(c(
    "2020-01-01", "2020-02-01",
    "2020-03-01", "2020-04-01"
  ))
) |>
  set_value_labels(
    a = c(No = 1, Yes = 2),
```

```
    b = c(No = 1, Yes = 2, DK = 3),
    c = c(No = 1, Yes = 2, DK = 3),
    d = c(No = "a", Yes = "b"),
    e = c(No = 1, Yes = 2)
  )
df |> look_for()
```

```
pos variable label col_type missing values
1   a        -     dbl+lbl  0       [1] No
                                    [2] Yes
2   b        -     dbl+lbl  0       [1] No
                                    [2] Yes
                                    [3] DK
3   c        -     dbl+lbl  0       [1] No
                                    [2] Yes
                                    [3] DK
4   d        -     chr+lbl  0       [a] No
                                    [b] Yes
5   e        -     dbl+lbl  0       [1] No
                                    [2] Yes
6   f        -     int      0
7   g        -     date     0
```

```
to_factor(df) |> look_for()
```

```
pos variable label col_type missing values
1   a        -     fct      0       No
                                    Yes
                                    3
2   b        -     fct      0       No
                                    Yes
                                    DK
3   c        -     fct      0       No
                                    Yes
                                    DK
4   d        -     fct      0       No
                                    Yes
                                    c
```

```
5   e        -     fct      0       No
                                    Yes
                                    9
6   f        -     int      0
7   g        -     date     0
```

```
unlabelled(df) |> look_for()
```

```
pos variable label col_type missing values
1   a        -     dbl      0
2   b        -     fct      0       No
                                    Yes
                                    DK
3   c        -     fct      0       No
                                    Yes
                                    DK
4   d        -     chr      0
5   e        -     dbl      0
6   f        -     int      0
7   g        -     date     0
```

On peut indiquer certaines options, par exemple `drop_unused_labels = TRUE` pour supprimer des facteurs créés les niveaux non observées dans les données (voir la variable c).

```
unlabelled(df, drop_unused_labels = TRUE) |>
  look_for()
```

```
pos variable label col_type missing values
1   a        -     dbl      0
2   b        -     fct      0       No
                                    Yes
                                    DK
3   c        -     fct      0       No
                                    Yes
4   d        -     chr      0
5   e        -     dbl      0
6   f        -     int      0
7   g        -     date     0
```

```
unlabelled(df, levels = "prefixed") |>
  look_for()
```

```
pos variable label col_type missing values
1   a        -     dbl      0
2   b        -     fct      0       [1] No
                                    [2] Yes
                                    [3] DK
3   c        -     fct      0       [1] No
                                    [2] Yes
                                    [3] DK
4   d        -     chr      0
5   e        -     dbl      0
6   f        -     int      0
7   g        -     date     0
```

13 Valeurs manquantes

Dans **R** base, les valeurs manquantes sont indiquées par la valeurs logiques `NA` que l'on peut utiliser dans tous types de vecteurs.

Dans certains cas, par exemple dans la fonction `dplyr::if_else()` qui vérifie que les deux options sont du même type, on peut avoir besoin de spécifier une valeur manquante d'un certains types précis (numérique, entier, textuel…) ce que l'on peut faire avec les constantes `NA_real_`, `NA_integer_` ou encore `NA_character_`.

De base, il n'existe qu'un seul type de valeurs manquantes dans **R**. D'autres logiciels statistiques ont mis en place des systèmes pour distinguer plusieurs types de valeurs manquantes, ce qui peut avoir une importance dans le domaine des grandes enquêtes, par exemple pour distinguer des ne sait pas d'un refus de répondre ou d'un oubli enquêteur.

Ainsi, **Stata** et **SAS** ont un système de valeurs manquantes étiquettées ou *tagged NAs*, où les valeurs manquantes peuvent recevoir une étiquette (une lettre entre a et z). De son côté, **SPSS** permet d'indiquer, sous la forme de métadonnées, que certaines valeurs devraient être traitées comme des valeurs manquantes (par exemple que la valeur 8 correspont à des refus et que la valeur 9 correspond à des ne sait pas). Il s'agit alors de valeurs manquantes définies par l'utilisateur ou *user NAs*.

Dans tous les cas, il appartient à l'analyste de décider au cas par cas comment ces valeurs manquantes doivent être traitées. Dans le cadre d'une variable numérique, il est essentiel d'exclure ces valeurs manquantes pour le calcul de statistiques telles que la moyenne ou l'écart-type. Pour des variables catégorielles, les pourcentages peuvent être calculées sur l'ensemble de l'échantillon (les valeurs manquantes étant alors traitées comme des modalités à part entière) ou bien

uniquement sur les réponses valides, en fonction du besoin de l'analyse et de ce que l'on cherche à montrer.

Afin d'éviter toute perte d'informations lors d'un import de données depuis **Stata**, **SAS** et **SPSS**, le package `{haven}` propose une implémentation sous **R** des *tagged NAs* et des *user NAs*. Le package {labelled} fournit quant à lui différentes fonctions pour les manipuler aisément.

```
library(labelled)
```

13.1 Valeurs manquantes étiquettées (*tagged NAs*)

13.1.1 Création et test

Les *tagged NAs* sont de véritables valeurs manquantes (`NA`) au sens de **R**, auxquelles a été attachées sur étiquette, une lettre unique minuscule (a-z) ou majuscule (A-Z). On peut les créer avec `labelled::tagged_na()`.

```
x <- c(1:3, tagged_na("a"), tagged_na("z"), NA)
```

Pour la plupart des fonctions de **R**, les *tagged NAs* sont juste considérées comme des valeurs manquantes régulières (*regumar NAs*). Dès lors, par défaut, elles sont justes affichées à l'écran comme n'importe quelle valeur manquante et la fonction `is.na()` renvoie `TRUE`.

```
x
```

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

```
is.na(x)
```

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

Pour afficher les étiquettes associées à ces valeurs manquantes, il faut avoir recours à `labelled::na_tag()`, `labelled::print_tagged_na()` ou encore `labelled::format_tagged_na()`.

```
na_tag(x)
```

```
[1] NA  NA  NA  "a" "z" NA
```

```
print_tagged_na(x)
```

```
[1]     1     2     3 NA(a) NA(z)    NA
```

```
format_tagged_na(x)
```

```
[1] "    1" "    2" "    3" "NA(a)" "NA(z)" "   NA"
```

Pour tester si une certaine valeur manquante est une *regular NA* ou une *tagged NA*, on aura recours à `labelled::is_regular_na()` et à `labelled::is_tagged_na()`.

```
is.na(x)
```

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

```
is_regular_na(x)
```

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

```
is_tagged_na(x)
```

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

Il est possible de tester une étiquette particulière en passant un deuxième argument à `labelled::is_tagged_na()`.

```
is_tagged_na(x, "a")
```

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

> **Note**
>
> Il n'est possible de définir des *tagged NAs* seulement pour des vecteurs numériques (*double*). Si l'on ajoute une *tagged NA* à un vecteur d'entiers, ce vecteur sera converti en vecteur numérique. Si on l'ajoute à un vecteur textuel, la valeur manquante sera convertie en *regular NA*.

```
y <- c("a", "b", tagged_na("z"))
y
```

```
[1] "a" "b" NA
```

```
is_tagged_na(y)
```

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

```
format_tagged_na(y)
```

```
Error: `x` must be a double vector
```

```
z <- c(1L, 2L, tagged_na("a"))
typeof(z)
```

```
[1] "double"
```

```
format_tagged_na(z)
```

```
[1] "    1" "    2" "NA(a)"
```

13.1.2 Valeurs uniques, doublons et tris

Par défaut, les fonctions classiques de **R** `unique()`, `duplicated()`, `ordered()` ou encore `sort()` traiteront les *tagged NAs* comme des valeurs manquantes tout ce qu'il y a de plus classique, et ne feront pas de différences entre des *tagged NAs* ayant des étiquettes différentes.

Pour traiter des *tagged NAs* ayant des étiquettes différentes comme des valeurs différentes, on aura recours aux fonctions `labelled::unique_tagged_na()`, `labelled::duplicated_tagged_na()`, `labelled::order_tagged_na()` ou encore `labelled::sort_tagged_na()`.

```
x <- c(1, 2, tagged_na("a"), 1, tagged_na("z"), 2, tagged_na("a"), NA)
x |>
  print_tagged_na()
```

```
[1]     1     2 NA(a)     1 NA(z)     2 NA(a)    NA
```

```
x |>
  unique() |>
  print_tagged_na()
```

```
[1]     1     2 NA(a)
```

```
x |>
  unique_tagged_na() |>
  print_tagged_na()
```

```
[1]     1     2 NA(a) NA(z)    NA
```

```
x |>
  duplicated()
```

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

```
x |>
  duplicated_tagged_na()
```

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

```
x |>
  sort(na.last = TRUE) |>
  print_tagged_na()
```

```
[1]     1     1     2     2 NA(a) NA(z) NA(a)    NA
```

```
x |>
  sort_tagged_na()  |>
  print_tagged_na()
```

```
[1]     1     1     2     2 NA(a) NA(a) NA(z)    NA
```

13.1.3 Tagged NAs et étiquettes de valeurs

Il est tout à fait possible d'associer une étiquette de valeurs (cf. Chapitre **??**) à des *tagged NAs*.

```
x <- c(
  1, 0,
  1, tagged_na("r"),
  0, tagged_na("d"),
  tagged_na("z"), NA
)
val_labels(x) <- c(
  no = 0,
  yes = 1,
  "don't know" = tagged_na("d"),
  refusal = tagged_na("r")
)
x
```

```
<labelled<double>[8]>
[1]     1     0     1 NA(r)     0 NA(d) NA(z)    NA

Labels:
 value      label
     0         no
     1        yes
 NA(d) don't know
 NA(r)    refusal
```

Lorsqu'un vecteur labellisé est converti en facteur avec `labelled::to_factor()`, les *tagged NAs* sont, par défaut convertis en en valeurs manquantes classiques (*regular NAs*). Il n'est pas possible de définir des *tagged NAs* pour des facteurs.

```
x |> to_factor()
```

```
[1] yes  no   yes  <NA> no   <NA> <NA> <NA>
Levels: no yes
```

L'option `explicit_tagged_na` de `labelled::to_factor()` permets de convertir les *tagged NAs* en modalités explicites du facteur.

```
x |>
  to_factor(explicit_tagged_na = TRUE)
```

```
[1] yes        no         yes        refusal    no         don't know NA(z)
[8] <NA>
Levels: no yes don't know refusal NA(z)
```

```
x |>
  to_factor(
    levels = "prefixed",
    explicit_tagged_na = TRUE
  )
```

```
[1] [1] yes            [0] no             [1] yes            [NA(r)] refusal
[5] [0] no             [NA(d)] don't know [NA(z)] NA(z)      <NA>
Levels: [0] no [1] yes [NA(d)] don't know [NA(r)] refusal [NA(z)] NA(z)
```

13.1.4 Conversion en user NAs

La fonction `labelled::tagged_na_to_user_na()` permets de convertir des *tagged NAs* en *user NAs*.

```
x |>
  tagged_na_to_user_na()
```

```
<labelled_spss<double>[8]>
[1]  1  0  1  3  0  2  4 NA
Missing range:  [2, 4]

Labels:
 value      label
     0         no
     1        yes
     2 don't know
     3    refusal
     4      NA(z)
```

```
x |>
  tagged_na_to_user_na(user_na_start = 10)
```

```
<labelled_spss<double>[8]>
[1]  1  0  1 11  0 10 12 NA
Missing range:  [10, 12]

Labels:
 value      label
     0         no
     1        yes
    10 don't know
    11    refusal
    12      NA(z)
```

La fonction `labelled::tagged_na_to_regular_na()` convertit les *tagged NAs* en valeurs manquantes classiques (*regular NAs*).

```
x |>
  tagged_na_to_regular_na()
```

```
<labelled<double>[8]>
[1]  1  0  1 NA  0 NA NA NA

Labels:
 value label
     0    no
     1   yes
```

```
x |>
  tagged_na_to_regular_na() |>
  is_tagged_na()
```

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

13.2 Valeurs manquantes définies par l'utilisateurs (*user NAs*)

Le package `{haven}` a introduit la classe `haven_labelled_spss`, une extension de la classe `haven_labelled` permettant d'indiquer des valeurs à considérer comme manquantes à la manière de **SPSS**.

> **!** Important
>
> Cela revient à associer à un vecteur des attributs (cf. Chapitre **??**) additionnels pour indiquer des valeurs que l'utilisateur pourrait/devrait considérer comme manquante. Cependant, il ne s'agit que de métadonnées et en interne ces valeurs ne sont pas stockées sous forme de `NA` mais restent des valeurs valides.

Il convient de garder en mémoire que la très grande majorité des fonctions de **R** ne prendront pas en compte ces métadonnées et traiteront donc ces valeurs comme des valeurs valides. C'est donc à l'utilisateur de convertir, au besoin, ces les valeurs indiquées comme manquantes en réelles valeurs manquantes (`NA`).

13.2.1 Création

Il est possible d'indiquer des valeurs à considérer comme manquantes (*user NAs*) de deux manières :

- soit en indiquant une liste de valeurs individuelles avec `labelled::na_values()` (on peut indiquer `NULL` pour supprimer les déclarations existantes) ;
- soit en indiquant deux valeurs représentant une plage de valeurs à considérées comme manquantes avec `labelled::na_range()` (seront considérées comme manquantes toutes les valeurs supérieures ou égale au premier chiffre et inférieures ou égales au second chiffre[19]).

[19] On peut utiler `-Inf` et `Inf` qui représentent respectivement moins l'infini et l'infini.

```
v <- c(1, 2, 3, 9, 1, 3, 2, NA)
val_labels(v) <- c(
  faible = 1,
  fort = 3,
  "ne sait pas" = 9
)
na_values(v) <- 9
v
```

```
<labelled_spss<double>[8]>
[1]  1  2  3  9  1  3  2 NA
Missing values: 9

Labels:
 value       label
     1      faible
     3        fort
     9 ne sait pas
```

```
na_values(v) <- NULL
v
```

```
<labelled<double>[8]>
[1]  1  2  3  9  1  3  2 NA

Labels:
 value       label
     1      faible
     3        fort
     9 ne sait pas
```

```
na_range(v) <- c(5, Inf)
v
```

```
<labelled_spss<double>[8]>
[1]  1  2  3  9  1  3  2 NA
Missing range:  [5, Inf]

Labels:
 value       label
     1      faible
     3        fort
     9 ne sait pas
```

On peut noter que les *user NAs* peuvent cohabiter avec des *regular NAs* ainsi qu'avec des étiquettes de valeurs (*value labels*, cf. Chapitre **??**).

Pour manipuler les variables d'un tableau de données, on peut également avoir recours à `labelled::set_na_values()` et `labelled::set_na_range()`.

```
df <-
  dplyr::tibble(
    s1 = c("M", "M", "F", "F"),
    s2 = c(1, 1, 2, 9)
  ) |>
```

```
  set_na_values(s2 = 9)
df$s2
```

```
<labelled_spss<double>[4]>
[1] 1 1 2 9
Missing values: 9
```

```
df <-
  df |>
  set_na_values(s2 = NULL)
df$s2
```

```
<labelled<double>[4]>
[1] 1 1 2 9
```

13.2.2 Tests

La fonction `is.na()` est l'une des rares fonctions de base **R** à reconnaître les *user NAs* et donc à renvoyer `TRUE` dans ce cas. Pour des tests plus spécifiques, on aura recours à `labelled::is_user_na()` et `labelled::is_regular_na()`.

```
v
```

```
<labelled_spss<double>[8]>
[1]  1  2  3  9  1  3  2 NA
Missing range:  [5, Inf]

Labels:
 value       label
     1      faible
     3        fort
     9 ne sait pas
```

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

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

```
v |> is_user_na()
```

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

```
v |> is_regular_na()
```

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

13.2.3 Conversion

Comme dit précédemment, pour la plupart des fonctions de **R**, les *users NAs* sont toujours des valeurs valides.

```
x <- c(1:5, 11:15)
na_range(x) <- c(10, Inf)
x
```

```
<labelled_spss<integer>[10]>
 [1]  1  2  3  4  5 11 12 13 14 15
Missing range:  [10, Inf]
```

```
mean(x)
```

```
[1] 8
```

On aura alors recours à `labelled::user_na_to_regular_na()` pour convertir les *users NAs* en véritables valeurs manquantes avant de procéder à un calcul statistique.

```
x |>
  user_na_to_na()
```

```
<labelled<integer>[10]>
 [1]  1  2  3  4  5 NA NA NA NA NA
```

```
x |>
  user_na_to_na() |>
  mean(na.rm = TRUE)
```

```
[1] 3
```

Une alternative consiste à transformer les *user NAs* en *tagged NAs* avec `labelled::user_na_to_tagged_na()`.

```
x |>
  user_na_to_tagged_na() |>
  print_tagged_na()
```

```
'x' has been converted into a double vector.
```

```
 [1]     1     2     3     4     5 NA(a) NA(b) NA(c) NA(d) NA(e)
```

```
x |>
  user_na_to_tagged_na() |>
  mean(na.rm = TRUE)
```

```
'x' has been converted into a double vector.
```

```
[1] 3
```

Pour supprimer les métadonnées relatives aux *user NAs* sans les convertir en valeurs manquantes, on aura recours à `labelled::remove_user_na()`.

```
x |>
  remove_user_na()
```

```
<labelled<integer>[10]>
 [1]  1  2  3  4  5 11 12 13 14 15
```

```
x |>
  remove_user_na() |>
  mean()
```

```
[1] 8
```

Enfin, lorsque l'on convertit un vecteur labellisé en facteur avec `labelled::to_factor()`, on pourra utiliser l'argument `user_na_to_na` pour indiquer si les *users NAs* doivent être convertis ou non en valeurs manquantes classiques (`NA`).

```
x <- c(1, 2, 9, 2)
val_labels(x) <- c(oui = 1, non = 2, refus = 9)
na_values(x) <- 9
x |>
  to_factor(user_na_to_na = TRUE)
```

```
[1] oui  non  <NA> non
Levels: oui non
```

```
x |>
  to_factor(user_na_to_na = FALSE)
```

```
[1] oui   non   refus non
Levels: oui non refus
```

14 Import & Export de données

14.1 Importer un fichier texte

Les fichiers texte constituent un des formats les plus largement supportés par la majorité des logiciels statistiques. Presque tous permettent d'exporter des données dans un format texte, y compris les tableurs comme **Libre Office**, **Open Office** ou **Excel**.

Cependant, il existe une grande variétés de format texte, qui peuvent prendre différents noms selon les outils, tels que texte tabulé ou *texte (séparateur : tabulation)*, **CSV** (pour *comma-separated value*, sachant que suivant les logiciels le séparateur peut être une virgule ou un point-virgule).

14.1.1 Structure d'un fichier texte

Dès lors, avant d'importer un fichier texte dans **R**, il est indispensable de regarder comment ce dernier est structuré. Il importe de prendre note des éléments suivants :

- La première ligne contient-elle le nom des variables ?
- Quel est le caractère séparateur entre les différentes variables (encore appelé séparateur de champs) ? Dans le cadre d'un fichier **CSV**, il aurait pu s'agir d'une virgule ou d'un point-virgule.
- Quel est le caractère utilisé pour indiquer les décimales (le séparateur décimal) ? Il s'agit en général d'un point (à l'anglo-saxonne) ou d'une virgule (à la française).
- Les valeurs textuelles sont-elles encadrées par des guillemets et, si oui, s'agit-il de guillements simple (`'`) ou de guillemets doubles (`"`) ?

- Pour les variables textuelles, y a-t-il des valeurs manquantes et si oui comment sont-elles indiquées ? Par exemple, le texte `NA` est parfois utilisé.

Il ne faut pas hésitez à ouvrir le fichier avec un éditeur de texte pour le regarder de plus près.

14.1.2 Interface graphique avec RStudio

RStudio fournit une interface graphique pour faciliter l'import d'un fichier texte. Pour cela, il suffit d'aller dans le menu *File > Import Dataset* et de choisir l'option *From CSV*[20]. Cette option est également disponible via l'onglet *Environment* dans le quadrant haut-droite.

[20] L'option CSV fonctionne pour tous les fichiers de type texte, même si votre fichier a une autre extension, `.txt` par exemple

Pour la suite, nous allons utiliser ce fichier texte à titre d'exemple.

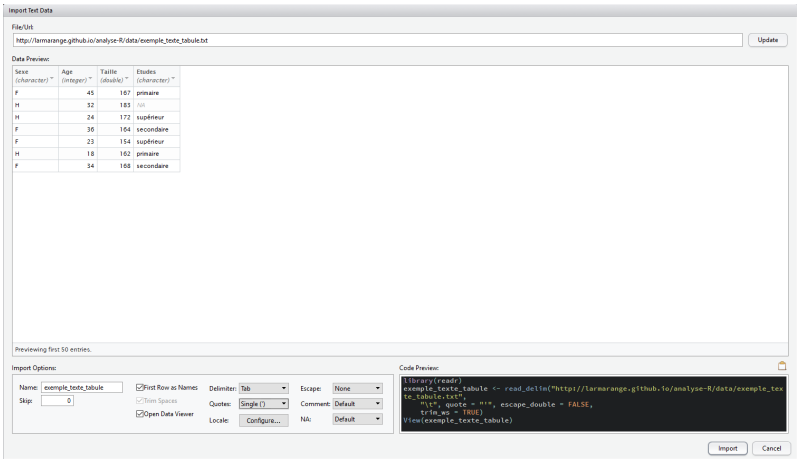


Figure 14.1: Importer un fichier texte avec RStudio

L'interface de **RStudio** vous présente sous *Import Options* les différentes options d'import disponible. La section *Data Preview* vous permet de voir en temps réel comment les données sont importées. La section *Code Preview* vous indique le code **R** correspondant à vos choix. Il n'y a plus qu'à le copier/coller dans un de vos scripts ou à cliquer sur **Import** pour l'exécuter.

Vous pourrez remarquer que **RStudio** fait appel à l'extension `{readr}` du tidyverse pour l'import des données via la fonction `readr::read_csv()`.

`{readr}` essaie de deviner le type de chacune des colonnes, en se basant sur les premières observations. En cliquant sur le nom d'une colonne, il est possible de modifier le type de la variable importée. Il est également possible d'exclure une colonne de l'import (*skip*).

14.1.3 Dans un script

L'interface graphique de **RStudio** fournit le code d'import. On peut également l'adapter à ces besoins en consultant la page d'aide de `readr::read_csv()` pour plus de détails. Par exemple :

```
library(readr)
d <- read_delim(
  "http://larmarange.github.io/analyse-R/data/exemple_texte_tabule.txt",
  delim = "\t",
  quote = "'"
)
```

On peut indiquer le chemin local vers un fichier (le plus courant) ou bien directement l'URL d'un fichier sur Internet.

`{readr}` propose plusieurs fonctions proches : `readr::read_delim()`, `readr::read_csv()`, `readr::read_csv2()` et `readr::read_tsv()`. Elles fonctionnent toutes de manière identique et ont les mêmes arguments. Seule différence, les valeurs par défaut de certains paramètres.

> **Fichiers de très grande taille**
>
> Si vous travaillez sur des données de grandes dimensions, les formats texte peuvent être lents à exporter et importer. Dans ce cas là, on pourra jeter un œil au package `{vroom}` et/ou aux fonctions `data.table::fread()` et `data.table::fwrite()`.

Dans des manuels ou des exemples en ligne, vous trouverez parfois mention des fonctions `utils::read.table()`, `utils::read.csv()`, `utils::read.csv2()`, `utils::read.delim()` ou encore `utils::read.delim2()`. Il s'agit des fonctions natives et historiques de **R** (extension `{utils}`) dédiées à l'import de fichiers textes. Elles sont similaires à celles de `{readr}` dans l'idée générale mais diffèrent dans leurs détails et les traitements effectués sur les données (pas de détection des dates par exemple). Pour plus d'information, vous pouvez vous référer à la page d'aide de ces fonctions.

14.2 Importer un fichier Excel

Une première approche pour importer des données **Excel** dans **R** consiste à les exporter depuis **Excel** dans un fichier texte (texte tabulé ou **CSV**) puis de suivre la procédure d'importation d'un fichier texte.

Une feuille **Excel** peut également être importée directement avec l'extension `{readxl}` du *tidyverse*.

La fonction `readxl::read_excel()` permet d'importer à la fois des fichiers `.xls` (**Excel** 2003 et précédents) et `.xlsx` (**Excel** 2007 et suivants).

```
library(readxl)
donnees <- read_excel("data/fichier.xlsx")
```

Une seule feuille de calculs peut être importée à la fois. On pourra préciser la feuille désirée avec `sheet` en indiquant soit le nom de la feuille, soit sa position (première, seconde, …).

```
donnees <- read_excel("data/fichier.xlsx", sheet = 3)
donnees <- read_excel("data/fichier.xlsx", sheet = "mes_donnees")
```

On pourra préciser avec `col_names` si la première ligne contient le nom des variables.

Par défaut, `readxl::read_excel()` va essayer de deviner le type (numérique, textuelle, date) de chaque colonne. Au besoin,

on pourra indiquer le type souhaité de chaque colonne avec `col_types`.

RStudio propose également pour les fichiers **Excel** un assitant d'importation, similaire à celui pour les fichiers texte, permettant de faciliter l'import.

14.3 Importer depuis des logiciels de statistique

Le package `{haven}` du *tidyverse* a été développé spécifiquement pour permettre l'importation de données depuis les formats des logiciels **Stata**, **SAS** et **SPSS**.

Il vise à offrir une importation unifiée depuis ces trois logiciels (là où le package `{foreign}` distribué en standard avec **R** adopte des conventions différentes selon le logiciel source).

Afin de ne pas perdre d'information lors de l'import, `{haven}` a introduit la notion d'étiquettes de variables (cf. Chapitre **??**), une classe de vecteurs pour la gestion des étiquettes de valeurs (cf. Chapitre **??**), des mécanismes pour reproduire la gestion des valeurs manquantes de ces trois logiciels (cf. Chapitre **??**), mais également une gestion et un import correct des dates, dates-heures et des variables horaires (cf. le package `{hms}`).

À noter que **RStudio** intègre égalementg une interface graphique pour l'import des fichiers **Stata**, **SAS** et **SPSS**.

14.3.1 SPSS

Les fichiers générés par **SPSS** sont de deux types : les fichiers **SPSS natifs** (extension `.sav`) et les fichiers au format **SPSS export** (extension `.por`).

Dans les deux cas, on aura recours à la fonction `haven::read_spss()` :

```
library(haven)
donnees <- read_spss("data/fichier.sav", user_na = TRUE)
```

Valeurs manquantes

Dans **SPSS**, il est possible de définir des valeurs à considérées comme manquantes ou *user NAs*, voir Chapitre **??**. Par défaut, `haven::read_spss()` convertir toutes ces valeurs en `NA` lors de l'import.
Or, il est parfois important de garder les différentes valeurs originelles. Dans ce cas, on appellera `haven::read_spss()` avec l'option `user_na = TRUE`.

14.3.2 SAS

Les fichiers **SAS** se présentent en général sous deux format : format **SAS export** (extension `.xport` ou `.xpt`) ou format **SAS natif** (extension `.sas7bdat`).

Les fichiers **SAS natifs** peuvent être importées directement avec `haven::read_sas()` de l'extension `{haven}` :

```
library(haven)
donnees <- read_sas("data/fichier.sas7bdat")
```

Au besoin, on pourra préciser en deuxième argument le nom d'un fichier **SAS catalogue** (extension `.sas7bcat`) contenant les métadonnées du fichier de données.

```
library(haven)
donnees <- read_sas(
  "data/fichier.sas7bdat",
  catalog_file = "data/fichier.sas7bcat"
)
```

Note

Les fichiers au format **SAS export** peuvent être importés via la fonction `foreign::read.xport()` de l'extension `{foreign}`. Celle-ci s'utilise très simplement, en lui passant le nom du fichier en argument :

```
library(foreign)
donnees <- read.xport("data/fichier.xpt")
```

14.3.3 Stata

Pour les fichiers **Stata** (extension `.dta`), on aura recours aux fonctions `haven::read_dta()` et `haven::read_stata()` de l'extension `{haven}`. Ces deux fonctions sont identiques.

```
library(haven)
donnees <- read_dta("data/fichier.dta")
```

> **!** Important
>
> **Gestion des valeurs manquantes**
> Dans **Stata**, il est possible de définir plusieurs types de valeurs manquantes, qui sont notées sous la forme `.a` à `.z`. Elles sont importées par `{haven}` sous formes de *tagged NAs*, cf. Chapitre **??**.

14.3.4 dBase

L'Insee et d'autres producteur de données diffusent leurs fichiers au format **dBase** (extension `.dbf`). Ceux-ci sont directement lisibles dans **R** avec la fonction `foreign::read.dbf()` de l'extension `{foreign}`.

```
library(foreign)
donnees <- read.dbf("data/fichier.dbf")
```

14.4 Sauver ses données

R dispose également de son propre format pour sauvegarder et échanger des données. On peut sauver n'importe quel objet créé avec **R** et il est possible de sauver plusieurs objets dans

un même fichier. L'usage est d'utiliser l'extension `.RData` pour les fichiers de données **R**. La fonction à utiliser s'appelle tout simplement `save()`.

Par exemple, si l'on souhaite sauvegarder son tableau de données `d` ainsi que les objets `tailles` et `poids` dans un fichier `export.RData` :

```
save(d, tailles, poids, file = "export.RData")
```

À tout moment, il sera toujours possible de recharger ces données en mémoire à l'aide de la fonction `load()` :

```
load("export.RData")
```

> Mise en garde
>
> Si entre temps vous aviez modifié votre tableau `d`, vos modifications seront perdues. En effet, si lors du chargement de données, un objet du même nom existe en mémoire, ce dernier sera remplacé par l'objet importé.

La fonction `save.image()` est un raccourci pour sauvergarder tous les objets de la session de travail dans le fichier `.RData` (un fichier un peu étrange car il n'a pas de nom mais juste une extension). Lors de la fermeture de **RStudio**, il vous sera demandé si vous souhaitez enregistrer votre session. Si vous répondez *Oui*, c'est cette fonction `save.image()` qui sera appliquée.

```
save.image()
```

Un autre mécanisme possible est le format **RDS** de **R**. La fonction `saveRDS()` permet de sauvegarder **un et un seul** objet **R** dans un fichier.

```
saveRDS(d, file = "mes_donnees.rds")
```

Cet objet pourra ensuite être lu avec la fonction `readRDS()`. Mais au lieu d'être directement chargé dans la mémoire de l'environnement de travail, l'objet lu sera retourné par la fonction `readRDS()` et ce sera à l'utilisateur de le sauvegarder.

```r
donnees <- readRDS("mes_donnees.rds")
```

14.5 Export de tableaux de données

On peut avoir besoin d'exporter un tableau de données **R** vers différents formats. La plupart des fonctions d'import disposent d'un équivalent permettant l'export de données. On citera notamment :

- `readr::write_csv()` et `readr::write_tsv()` permettent d'exporter au format **CSV** et texte tabulé respectivement, `readr::write_delim()` offrant de multiples options pour l'exort au format texte ;
- `haven::write_sas()` permet d'exporter au format **SAS ;**
- `haven::write_sav()` au format **SPSS ;**
- `haven::write_dta()` au format **Stata ;**
- `foreign::write.dbf()` au format **dBase**.

L'extension `readxl` ne fournit pas de fonction pour exporter au format **Excel**. Par contre, on pourra passer par la fonction `openxlsx::write.xlsx()` du package `{openxlsx}` ou la fonction `xlsx::write.xlsx()` de l'extension `{xlsx}`. L'intérêt de `{openxlsx}` est de ne pas dépendre de **Java** à la différence de `{xlsx}`.

15 Mettre en forme des nombres

Dans les chapitres suivants, nous aurons régulièrement besoin, pour produire des tableaux ou des figures propres, de **fonctions de formatage** qui permettent de transformer des valeurs numériques en chaînes de texte.

La fonction **R** de base est `format()` mais de nombreux autres packages proposent des variations pour faciliter cette opération. Le plus complet est probablement `{scales}` et, notamment, ses fonctions `scales::label_number()` et `scales::number()`.

Elles ont l'air très similaires et partagent un grand nombre de paramètres en commun. La différence est que `scales::number()` a besoin d'un vecteur numérique en entrée qu'elle va mettre en forme, tandis que que `scales::label_number()` renvoie une fonction que l'on pourra ensuite appliquer à un vecteur numérique.

```
library(scales)
x <- c(0.0023, .123, 4.567, 874.44, 8957845)
number(x)
```

```
[1] "0.00"         "0.12"         "4.57"         "874.44"       "8 957 845.00"
```

```
f <- label_number()
f(x)
```

```
[1] "0.00"         "0.12"         "4.57"         "874.44"       "8 957 845.00"
```

```
label_number()(x)
```

```
[1] "0.00"         "0.12"         "4.57"         "874.44"       "8 957 845.00"
```

Dans de nombreux cas de figure (par exemple pour un graphique `{ggplot2}` ou un tableau `{gtsummary}`), il sera demandé de fournir une fonction de formatage, auquel cas on aura recours aux fonctions de `{scales}` préfixées par `label_*()` qui permettent donc de générer une fonction personnalisée.

15.1 `label_number()`

`scales::label_number()` est la fonction de base de mise en forme de nombres dans `{scales}`, une majorité des autres fonctions faisant appel à `scales::label_number()` et partageant les mêmes arguments.

Le paramètre `accurary` permets de définir le niveau d'arrondi à utiliser. Par exemple, `.1` pour afficher une seule décimale. Il est aussi possible d'indiquer un nombre qui n'est pas une puissance de 10 (par exemple `.25`). Si on n'indique rien (`NULL`), alors `scales::label_number()` essaiera de deviner un nombre de décimales pertinent en fonction des valeurs du vecteur de nombres à mettre en forme.

```
label_number(accuracy = NULL)(x)
```

```
[1] "0.00"         "0.12"         "4.57"         "874.44"       "8 957 845.00"
```

```
label_number(accuracy = .1)(x)
```

```
[1] "0.0"         "0.1"         "4.6"         "874.4"       "8 957 845.0"
```

```
label_number(accuracy = .25)(x)
```

```
[1] "0.0"         "0.0"         "4.5"         "874.5"       "8 957 845.0"
```

```
  label_number(accuracy = 10)(x)
```

```
[1] "0"         "0"         "0"         "870"       "8 957 840"
```

L'option `scale` permets d'indiquer un facteur multiplicatif à appliquer avant de mettre en forme. On utilisera le plus souvent les options `prefix` et `suffix` en même temps pour indiquer les unités.

```
  label_number(scale = 100, suffix = "%")(x) # pour cent
```

```
[1] "0%"           "12%"          "457%"         "87 444%"      "895 784 500%"
```

```
  label_number(scale = 1000, suffix = "\u2030")(x) # pour mille
```

```
[1] "2‰"             "123‰"           "4 567‰"         "874 440‰"
[5] "8 957 845 000‰"
```

```
  label_number(scale = .001, suffix = " milliers", accuracy = .1)(x)
```

```
[1] "0.0 milliers"     "0.0 milliers"     "0.0 milliers"     "0.9 milliers"
[5] "8 957.8 milliers"
```

Les arguments `decimal.mark` et `big.mark` permettent de définir, respectivement, le séparateur de décimale et le séparateur de milliers. Ainsi, pour afficher des nombres à la française (virgule pour les décimales, espace pour les milliers) :

```
  label_number(decimal.mark = ",", big.mark = " ")(x)
```

```
[1] "0,00"         "0,12"         "4,57"         "874,44"       "8 957 845,00"
```

Note : il est possible d'utiliser `small.interval` et `small.mark` pour ajouter des séparateurs parmi les décimales.

```
label_number(accuracy = 10^-9, small.mark = "|", small.interval = 3)(x)
```

```
[1] "0.002|300|000"         "0.123|000|000"         "4.567|000|000"
[4] "874.440|000|000"       "8 957 845.000|000|000"
```

Les options `style_positive` et `style_negative` permettent de personnaliser la manière dont les valeurs positives et négatives sont mises en forme.

```
y <- c(-1.2, -0.3, 0, 2.4, 7.2)
label_number(style_positive = "plus")(y)
```

```
[1] "-1.2" "-0.3" "0.0"  "+2.4" "+7.2"
```

```
label_number(style_negative = "parens")(y)
```

```
[1] "(1.2)" "(0.3)" "0.0"   "2.4"   "7.2"
```

15.2 Les autres fonctions de {scales}

15.2.1 label_comma()

`scales::label_comma()` (et `scales::comma()`) est une variante de `scales::label_number()` qui, par défaut, affiche les nombres à l'américaine, avec une virgule comme séparateur de milliers.

```
label_comma()(x)
```

```
[1] "0.00"         "0.12"         "4.57"         "874.44"       "8,957,845.00"
```

15.2.2 `label_percent()`

`scales::label_percent()` (et `scales::percent()`) est une variante de `scales::label_number()` qui affiche les nombres sous formes de pourcentages (les options par défaut sont `scale = 100, suffix = "%"`).

```
  label_percent()(x)
```

```
[1] "0%"           "12%"          "457%"         "87 444%"      "895 784 500%"
```

On peut utiliser cette fonction pour afficher des résultats en pour mille (le code unicode du symbole ‰ étant u2030) :

```
  label_percent(scale = 1000, suffix = "\u2030")(x)
```

```
[1] "2‰"             "123‰"           "4 567‰"         "874 440‰"
[5] "8 957 845 000‰"
```

15.2.3 `label_dollar()`

`scales::label_dollar()` est adapté à l'affichage des valeurs monétaires.

```
  label_dollar()(x)
```

```
[1] "$0"         "$0"         "$5"         "$874"       "$8,957,845"
```

```
  label_dollar(prefix = "", suffix = " €", accuracy = .01, big.mark = " ")(x)
```

```
[1] "0.00 €"         "0.12 €"         "4.57 €"         "874.44 €"
[5] "8 957 845.00 €"
```

L'option `style_negative` permet d'afficher les valeurs négatives avec des parenthèses, convention utilisée dans certaines disciplines.

```
  label_dollar()(c(12.5, -4, 21, -56.36))
```

```
[1] "$12.50"  "-$4.00"  "$21.00"  "-$56.36"
```

```
  label_dollar(style_negative = "parens")(c(12.5, -4, 21, -56.36))
```

```
[1] "$12.50"   "($4.00)"  "$21.00"   "($56.36)"
```

15.2.4 `label_pvalue()`

`scales::label_pvalue()` est adapté pour la mise en forme de p-valeurs.

```
  label_pvalue()(c(0.000001, 0.023, 0.098, 0.60, 0.9998))
```

```
[1] "<0.001" "0.023"  "0.098"  "0.600"  ">0.999"
```

```
  label_pvalue(accuracy = .01, add_p = TRUE)(c(0.000001, 0.023, 0.098, 0.60))
```

```
[1] "p<0.01" "p=0.02" "p=0.10" "p=0.60"
```

15.2.5 `label_number_si()`

`scales::label_number_si()` cherche le préfixe du Système international d'unités le plus proche et arrondi chaque valeur en fonction, en ajoutant la précision correspondante.

```
  label_number_si(unit = "g")(c(.00000145, .0034, 5, 12478, 14569787))
```

```
Warning: `label_number_si()` was deprecated in scales 1.2.0.
i Please use the `scale_cut` argument of `label_number()` instead.
```

```
[1] "0.0000 g" "0.0034 g" "5.0000 g" "12 Kg"    "15 Mg"
```

15.2.6 `label_scientific()`

`scales::label_scientific()` affiche les nombres dans un format scientifique (avec des puissances de 10).

```
label_scientific(unit = "g")(c(.00000145, .0034, 5, 12478, 14569787))
```

```
[1] "1.45e-06" "3.40e-03" "5.00e+00" "1.25e+04" "1.46e+07"
```

15.2.7 `label_bytes()`

`scales::label_bytes()` mets en forme des tailles exprimées en octets, utilisant au besoin des multiples de 1024.

```
b <- c(478, 1235468, 546578944897)
label_bytes()(b)
```

```
[1] "478 B"  "1 MB"   "547 GB"
```

```
label_bytes(units = "auto_binary")(b)
```

```
[1] "478 iB"  "1 MiB"   "509 GiB"
```

15.2.8 `label_ordinal()`

`scales::label_bytes()` permets d'afficher des rangs ou nombres ordinaux. Plusieurs langues sont disponibles.

```
label_ordinal()(1:5)
```

```
[1] "1st" "2nd" "3rd" "4th" "5th"
```

```
label_ordinal(rules = ordinal_french())(1:5)
```

```
[1] "1er" "2e"  "3e"  "4e"  "5e"
```

```
label_ordinal(rules = ordinal_french(gender = "f", plural = TRUE))(1:5)
```

```
[1] "1res" "2es"  "3es"  "4es"  "5es"
```

15.2.9 `label_date()`, `label_date_short()` & `label_time()`

`scales::label_date()`, `scales::label_date_short()` et `scales::label_time()` peuvent être utilisées pour la mise en forme de dates.

```
label_date()(as.Date("2020-02-14"))
```

```
[1] "2020-02-14"
```

```
label_date(format = "%d/%m/%Y")(as.Date("2020-02-14"))
```

```
[1] "14/02/2020"
```

```
label_date_short()(as.Date("2020-02-14"))
```

```
[1] "14\nfévr.\n2020"
```

La mise en forme des dates est un peu complexe. Ne pas hésiter à consulter le fichier d'aide de la fonction `base::strptime()` pour plus d'informations.

15.2.10 `label_wrap()`

La fonction `scales::label_wrap()` est un peu différente. Elle permets d'insérer des retours à la ligne (`\n`) dans des chaines de caractères. Elle tient compte des espaces pour identifier les mots et éviter ainsi des coupures au milieu d'un mot.

```
x <- "Ceci est un texte assez long et que l'on souhaiterait afficher sur plusieurs lignes. C
label_wrap(80)(x)
```

```
[1] "Ceci est un texte assez long et que l'on souhaiterait afficher sur plusieurs\nlignes. Cepe
```

```
label_wrap(80)(x) |> message()
```

```
Ceci est un texte assez long et que l'on souhaiterait afficher sur plusieurs
lignes. Cependant, on souhaite éviter que des coupures apparaissent au milieu
d'un mot.
```

```
label_wrap(40)(x) |> message()
```

```
Ceci est un texte assez long et que
l'on souhaiterait afficher sur
plusieurs lignes. Cependant, on
souhaite éviter que des coupures
apparaissent au milieu d'un mot.
```

15.3 Les fonctions de formatage de {gtsummary}

Véritable couteau-suisse du statisticien, le package {gtsummary} sera largement utilisé dans les prochains chapitres pour produire des tableaux statistiques prêts à être publiés.

Ce package utilise par défaut ses propres fonctions de formatage mais, au besoin, il sera toujours possible de lui transmettre des fonctions de formatage créées avec {scales}.

15.3.1 `style_number()`

Fonction de base, `gtsummary::style_number()` accepte les paramètres `big.mark` (séparateur de milliers), `decimal.mark` (séparateur de décimales) et `scale` (facteur d'échelle). Le nombre de décimales se précisera quant à lui avec `digits` où l'on indiquera le nombre de décimales souhaité.

```
library(gtsummary)
```

#Uighur

```
x <- c(0.123, 0.9, 1.1234, 12.345, -0.123, -0.9, -1.1234, -132.345)
style_number(x, digits = 1)
```

```
[1] "0.1"    "0.9"    "1.1"    "12.3"   "-0.1"   "-0.9"   "-1.1"   "-132.3"
```

> **Astuce**
>
> Nous verrons dans le chapitre sur les statistiques univariées (cf. Section **??**) la fonction `gtsummary::theme_gtsummary_language()` qui permet de fixer globalement le séparateur de milliers et celui des décimales, afin de changer les valeurs par défaut de l'ensemble des fonctions de formatage de `{gtsummary}`.
> Il est important de noter que cela n'a aucun effet sur les fonctions de formatage de `{scales}`.

> **Mise en garde**
>
> `gtsummary::style_number()` est directement une fonction de formatage (comme `scales::number()`) et non une fonction qui générère une fonction de formatage (comme `scales::label::number()`).
> Pour créer une fonction de formatage personnalisée, on pourra avoir recours à `purrr::partial()` qui permet d'appeller partiellement une fonctio et qui renvoie une nouvelle fonction avec des paramètres par défaut person-

nalisés.

```
fr <- style_number |>
  purrr::partial(decimal.mark = ",", digits = 1)
fr(x)
```

```
[1] "0,1"    "0,9"    "1,1"    "12,3"   "-0,1"   "-0,9"   "-1,1"   "-132,3"
```

15.3.2 `style_sigfig()`

Variante de `gtsummary::style_number()`, `gstummary::style_sigfig()` arrondi les valeurs transmises pour n'afficher qu'un nombre choisi de chiffres significatifs. Le nombre de décimales peut ainsi varier.

```
style_sigfig(x)
```

```
[1] "0.12"  "0.90"  "1.1"   "12"    "-0.12" "-0.90" "-1.1"  "-132"
```

```
style_sigfig(x, digits = 3)
```

```
[1] "0.123"  "0.900"  "1.12"   "12.3"   "-0.123" "-0.900" "-1.12"  "-132"
```

15.3.3 `style_percent()`

La fonction `gtsummary::style_percent()` a un fonctionnement un peu différent de celui de `scales::label_percent()`. Par défaut, le symbole `%` n'est pas affiché (mais paramétrable avec `symbol = TRUE`. Par défaut, une décimale est affichée pour les valeurs inférieures à 10% et aucune pour celles supérieures à 10%. Un symbole `<` est ajouté devant les valeurs strictement positives inférieures à 0,1%.

```
v <- c(0, 0.0001, 0.005, 0.01, 0.10, 0.45356, 0.99, 1.45)
label_percent(accuracy = .1)(v)
```

```
[1] "0.0%"   "0.0%"   "0.5%"   "1.0%"   "10.0%"  "45.4%"  "99.0%"  "145.0%"
```

```
style_percent(v)
```

```
[1] "0"    "<0.1" "0.5"  "1.0"  "10"   "45"   "99"   "145"
```

```
style_percent(v, symbol = TRUE)
```

```
[1] "0%"    "<0.1%" "0.5%"  "1.0%"  "10%"   "45%"   "99%"   "145%"
```

```
style_percent(v, digits = 1)
```

```
[1] "0"     "0.01"  "0.50"  "1.00"  "10.0"  "45.4"  "99.0"  "145.0"
```

15.3.4 `style_pvalue()`

La fonction `gtsummary::style_pvalue()` est similaire à `scales::label_pvalue()` mais adapte le nombre de décimales affichées,

```
p <- c(0.000001, 0.023, 0.098, 0.60, 0.9998)
label_pvalue()(p)
```

```
[1] "<0.001" "0.023"  "0.098"  "0.600"  ">0.999"
```

```
style_pvalue(p)
```

```
[1] "<0.001" "0.023"  "0.10"   "0.6"    ">0.9"
```

```
style_pvalue(p, prepend_p = TRUE)
```

```
[1] "p<0.001" "p=0.023" "p=0.10"  "p=0.6"   "p>0.9"
```

15.3.5 `style_ratio()`

Enfin, `gtsummary::style_ratio()` est adaptée à l'affichage de ratios.

```
r <- c(0.123, 0.9, 1.1234, 12.345, 101.234, -0.123, -0.9, -1.1234, -12.345, -101.234)
style_ratio(r)
```

```
 [1] "0.12"  "0.90"  "1.12"  "12.3"  "101"   "-0.12" "-0.90" "-1.12" "-12.3"
[10] "-101"
```

15.4 Bonus : `signif_stars()` de {ggstats}

La fonction `ggstats::signif_stars()` de {ggstats} permet d'afficher des p-valeurs sous forme d'étoiles de significativité, Par défaut, trois astérisques si p < 0,001, deux si p < 0,01, une si p < 0,05 et un point si p < 0,10. Les valeurs sont bien sur paramétrables.

```
p <- c(0.5, 0.1, 0.05, 0.01, 0.001)
ggstats::signif_stars(p)
```

```
[1] ""    "."   "*"   "**"  "***"
```

```
ggstats::signif_stars(p, one = .15, point = NULL)
```

```
[1] ""    "*"   "*"   "**"  "***"
```

15.5

16 Couleurs & Palettes

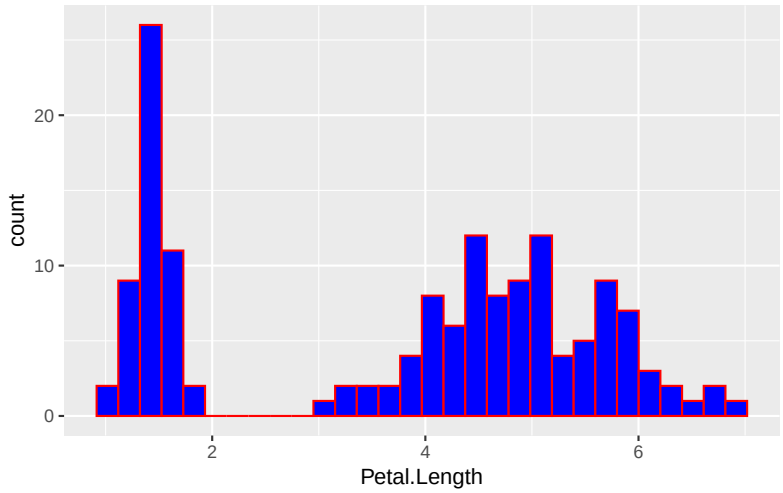
Dans les prochains chapitres, notamment lorsque nous ferons des graphiques, nous aurons besoin de spécier à **R** les couleurs souhaitées.

Le choix d'une palette de couleurs adaptée à sa représentation graphique est également un élément essentiel avec quelques règles de base : un dégradé est adpaté pour représentée une variable continue tandis que pour une variable catégorielle non ordonnée on aura recours à une palette contrastée.

16.1 Noms de couleur

Lorsque l'on doit indiquer à **R** une couleur, notamment dans les fonctions graphiques, on peut mentionner certaines couleurs en toutes lettres (en anglais) comme `"red"` ou `"blue"`. La liste des couleurs reconnues par **R** est disponible sur http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf.

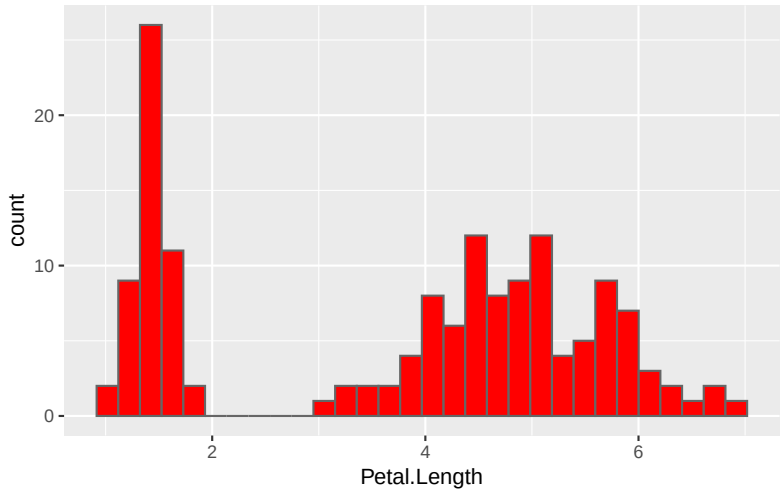
```
library(tidyverse)
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram(colour = "red", fill = "blue")
```



16.2 Couleurs RVB et code hexadécimal

En informatique, les couleurs sont usuellement codées en Rouge/Vert/Bleu (voir https://fr.wikipedia.org/wiki/Rouge_vert_bleu) et représentées par un code hexadécimal à 6 caractères (chiffres 0 à 9 et/ou lettres A à F), précédés du symbole `#`. Ce code est reconnu par **R**. On pourra par exemple indiquer `"#FF0000"` pour la couleur rouge ou `"#666666"` pour un gris foncé. Le code hexadécimal des différentes couleurs peut s'obtenir aisément sur internet, de nombreux sites étant consacrés aux palettes de couleurs.

```
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram(colour = "#666666", fill = "#FF0000")
```



Parfois, au lieu du code hexadécimal, les couleurs RVB sont indiquées avec trois chiffres entiers compris entre 0 et 255. La conversion en hexadécimal se fait avec la fonction `grDevices::rgb()`.

```
rgb(255, 0, 0, maxColorValue = 255)
```

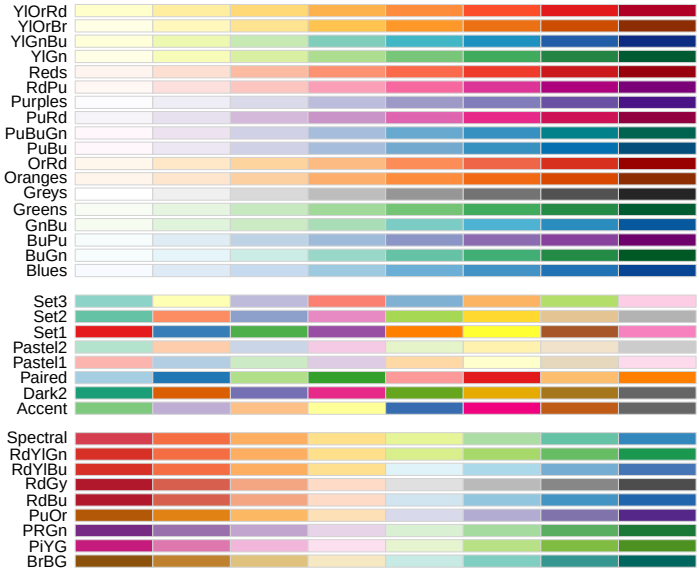
```
[1] "#FF0000"
```

16.3 Palettes de couleurs

16.3.1 Color Brewer

Le projet **Color Brewer** a développé des palettes cartographiques, à la fois séquentielles, divergentes et catégorielles, présentées en détail sur http://colorbrewer2.org/. Pour chaque type de palette, et en fonction du nombre de classes, est indiqué sur ce site si la palette est adaptée aux personnes souffrant de daltonisme, si elle est rendra correctement sur écran, en cas d'impression couleur et en cas d'impression en noir et blanc.

Voici un aperçu des différentes palettes disponibles :



L'extension `{RColorBrewer}` permets d'accéder à ces palettes sous **R**.

Si on utilise `{ggplot2}`, les palettes Color Brewer sont directement disponibles via les fonctions `ggplot2::scale_fill_brewer()` et `ggplot2::scale_colour_brewer()`.

> **Mise en garde**
>
> Les palettes Color Brewer sont seulement implémentées pour des variables catégorielles. Il est cependant possible de les utiliser avec des variables continues en les combinants avec `ggplot2::scale_fill_gradientn()` ou `ggplot2::scale_coulour_gradientn()` (en remplaçant `"Set1"` par le nom de la palette désirée) :
>
> ```
> scale_fill_gradientn(values = RColorBrewer::brewer.pal(6, "Set1"))
> ```

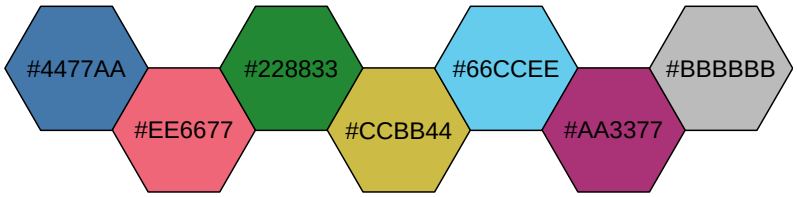
16.3.2 Palettes de Paul Tol

Le physicien Paul Tol a développé plusieurs palettes de couleurs adaptées aux personnes souffrant de déficit de perception des couleurs (daltonisme). À titre personnel, il s'agit des palettes de couleurs que j'utilise le plus fréquemment.

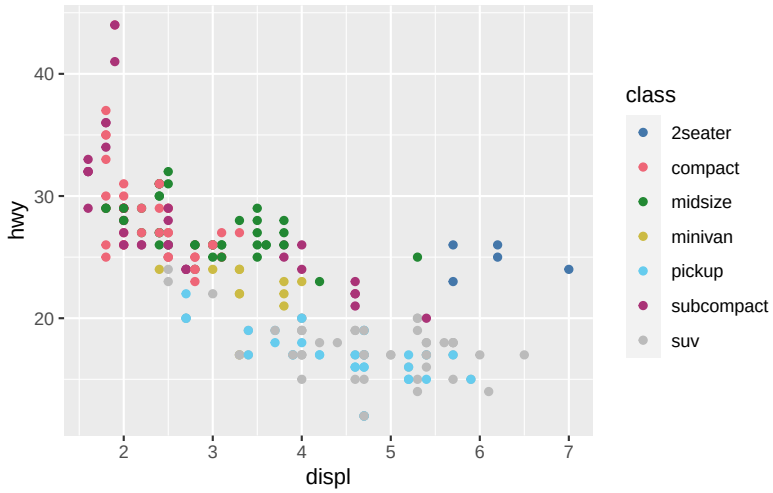
Le détail de ses travaux est présenté sur https://personal.sron.nl/~pault/.

Le pacakge `{khroma}` implémente ces palettes de couleurs proposées par Paul Tol afin de pouvoir les utilisées directement dans **R** et avec `{ggplot}`.

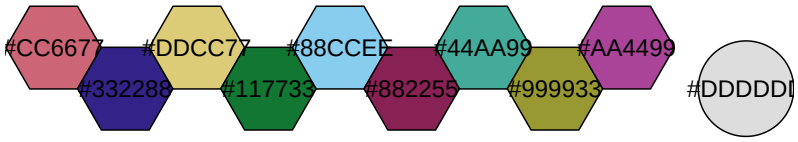
```
library(khroma)
plot_scheme(colour("bright")(7), colours = TRUE)
```



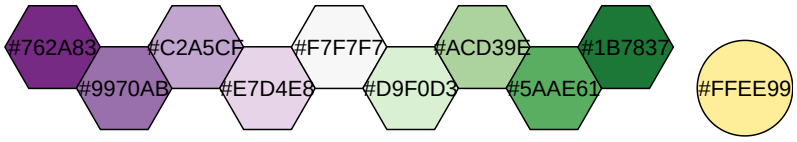

```
ggplot(mpg) +
  aes(x = displ, y = hwy, colour = class) +
  geom_point() +
  khroma::scale_colour_bright()
```




```
plot_scheme(colour("muted")(9), colours = TRUE)
```




```
plot_scheme(colour("PRGn")(9), colours = TRUE, size = 0.9)
```



Pour la liste complète des palettes disponibles, voir https://packages.tesselle.org/khroma/articles/tol.html.

16.3.3 Interface unifiée avec `{paletteer}`

L'extension `{paletteer}` vise à proposer une interface unifiée pour l'utilisation de palettes de couleurs fournies par d'autres packages (dont `{khroma}`, mais aussi par exemple `{ggsci}` qui fournit les palettes utilisées par certaines revues scientifiques). Plus de 2 500 palettes sont ainsi disponibles.

On peut afficher un aperçu des principales palettes disponibles dans `{paletteer}` avec la commande suivante :

```
gt::info_paletteer()
```

Pour afficher la liste complète des palettes discrètes et continues, on utilisera les commandes suivantes :

```
palettes_d_names |> View()
palettes_c_names |> View()
```

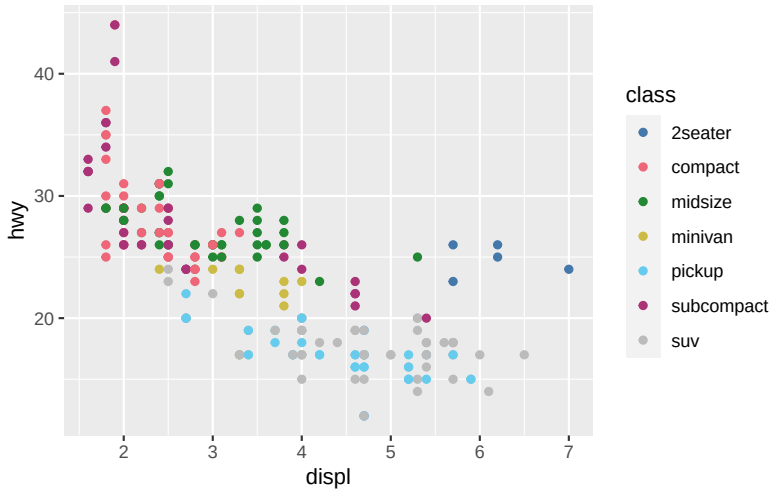
La fonction `paletteer::paletteer_d()` permet d'obtenir les codes hexadécimaux d'une parlette discrète en précisant le nombre de couleurs attendues. Les fonctions

`paletteer::scale_color_paletteer_d()` et `paletteer::scale_fill_paletteer_d()` permettront d'utiliser une palette donnée avec {ggplot2}.

```
library(paletteer)
paletteer_d("khroma::bright", n = 5)
```

```
<colors>
#4477AAFF #EE6677FF #228833FF #CCBB44FF #66CCEEFF
```

```
ggplot(mpg) +
  aes(x = displ, y = hwy, colour = class) +
  geom_point() +
  scale_color_paletteer_d("khroma::bright")
```

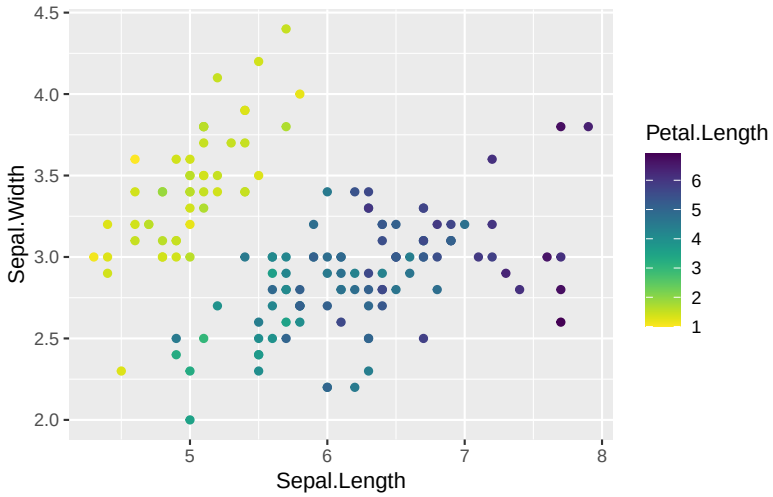


L'équivalent existe pour les palettes continues, avec `paletteer::paletteer_c()`, `paletteer::scale_color_paletteer_c()` et `paletteer::scale_fill_paletteer_c()` .

```
paletteer_c("viridis::viridis", n = 6)
```

```
<colors>
#440154FF #414487FF #2A788EFF #22A884FF #7AD151FF #FDE725FF
```

```
ggplot(iris) +
  aes(x = Sepal.Length, y = Sepal.Width, colour = Petal.Length) +
  geom_point() +
  scale_colour_paletteer_c("viridis::viridis", direction = -1)
```



partie III

Analyses

17 Graphiques avec ggplot2

Le package {ggplot2} fait partie intégrante du *tidyverse*. Développé par Hadley Wickham, ce package met en œuvre la grammaire graphique théorisée par Leland Wilkinson. Il devient vite indispensable lorsque l'on souhaite réaliser des graphiques un peu complexe.

17.1 Ressources

Il existe de très nombreuses ressources traitant de {ggplot2}.

Pour une introduction en français, on pourra se référer au chapitre Visualiser avec ggplot2 de l'*Introduction à R et au tidyverse* de Julien Barnier, au chapitre Introduction à ggplot2, la grammaire des graphiques du site *analyse-R* et adapté d'une séance de cours de François Briatte, ou encore au chapitre Graphiques du cours *Logiciel R et programmation* d'Ewen Gallic.

Pour les anglophones, la référence reste encore l'ouvrage *ggplot2: Elegant Graphics for Data Analysis* d'Hadley Wickham lui-même, dont la troisième édition est librement accessible en ligne (https://ggplot2-book.org/). D'un point de vue pratique, l'ouvrage *R Graphics Cookbook: practical recipes for visualizing data* de Winston Chang est une mine d'informations, ouvrage là encore librement accessible en ligne (https://r-graphics.org/).

17.2 Les bases de ggplot2

{ggplot2} nécessite que les données du graphique soient sous la forme d'un tableau de données (*data.frame* ou *tibble*) au format

tidy, c'est-à-dire avec une ligne par observation et les différentes valeurs à représenter sous forme de variables du tableau.

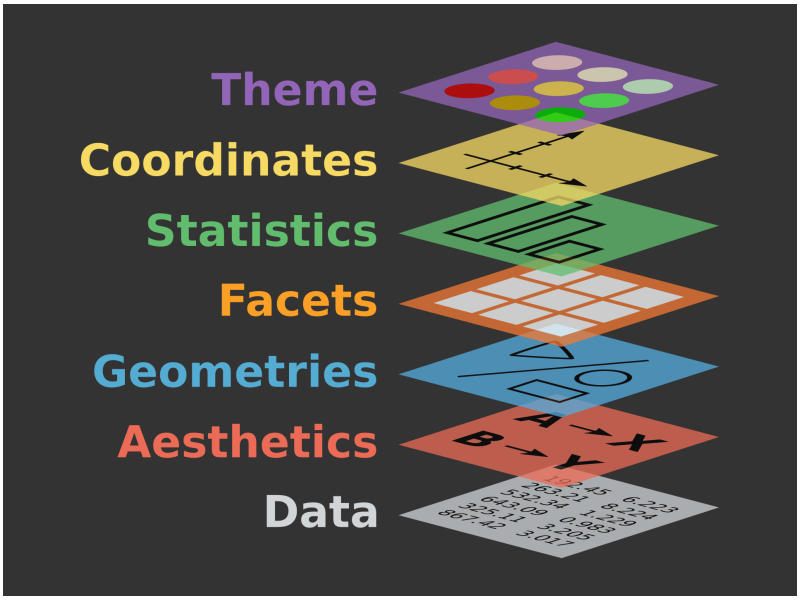


Figure 17.1: La grammaire des graphiques

Tous les graphiques avec `{ggplot2}` suivent une même logique. En **premier** lieu, on appelera la fonction `ggplot2::ggplot()` en lui passant en paramètre le fichier de données.

`{ggplot2}` nomme *esthétiques* les différentes propriétés visuelles d'un graphique, à savoir l'axe des x (`x`), celui des y (`y`), la couleur des lignes (`colour`), celle de remplissage des polygones (`fill`), le type de lignes (`linetype`), la forme des points (`shape`), etc. Une représentation graphique consiste donc à représenter chacune de nos variables d'intérêt selon une esthétique donnée. En **second** lieu, on appelera donc la fonction `ggplot2::aes()` pour indiquer la correspondance entre les variables de notre fichier de données et les esthétiques du graphique.

A minima, il est nécessaire d'indiquer en **troisième** lieu une *géométrie*, autrement dit la manière dont les éléments seront représentés visuellement. À chaque géométrie corresponds une fonction commençant par `geom_`, par exemple `ggplot2::geom_point()` pour dessiner

des points, `ggplot2::geom_line()` pour des lignes, `ggplot2::geom_bar()` pour des barres ou encore `ggplot2::geom_area()` pour des aires. Il existe de nombreuses géométries différentes[21], chacune prenant en compte certaines esthétiques, certaines étant requises pour cette géométrie et d'autres optionnelles. La liste des esthétiques prises en compte par chaque géométrie est indiquée dans l'aide en ligne de cette dernière.

[21] On trouvera une liste dans la *cheat sheet* de `{ggplot2}`, voir Section **??**.

Voici un exemple minimal de graphique avec `{ggplot2}` :

```
library(ggplot2)
p <-
  ggplot(iris) +
  aes(
    x = Petal.Length,
    y = Petal.Width,
    colour = Species
  ) +
  geom_point()
p
```

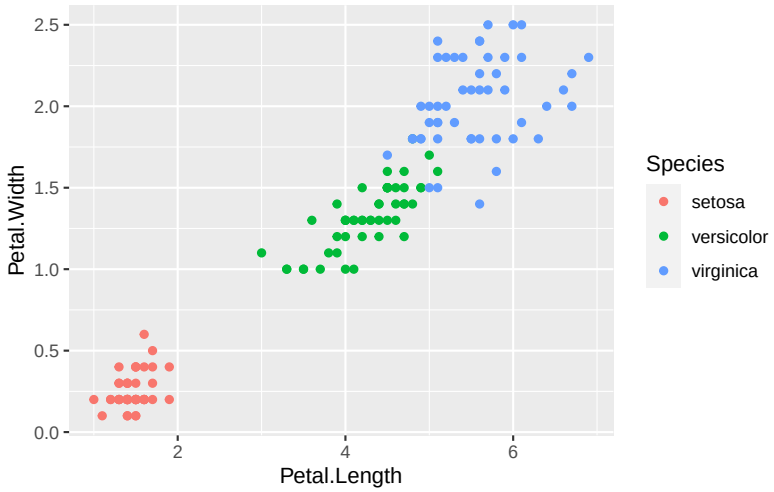


Figure 17.2: Un exemple simple de nuage de points avec `ggplot2`

> **!** Syntaxe additive
>
> Le développement de `{ggplot2}` a débuté avant celui du *tidyverse* et la généralisation du *pipe*. Dès lors, on ne sera pas étonné que la syntaxe de `{ggplot2}` n'ait pas recours à ce dernier mais repose sur une approche *additive*. Un graphique est dès lors initialisé avec la fonction `ggplot2::ggplot()` et l'on ajoutera successivement des éléments au graphique en appelant différentes fonctions et en utilisant l'opérateur `+`.

Il est ensuite possible de personnaliser de nombreux éléments d'un graphique et notamment :

- les *étiquettes* ou *labs* (titre, axes, légendes) avec `ggplot2::ggtitle()`, `ggplot2::xlab()`, `ggplot2::ylab()` ou encore la fonction plus générique `ggplot2::labs()` ;
- les *échelles* (*scales*) des différentes esthétiques avec les fonctions commençant par `scale_` ;
- le système de *coordonnées* avec les fonctions commençant par `coord_` ;
- les *facettes* (*facets*) avec les fonctions commençant par `facet_` ;
- la *légende* (*guides*) avec les fonctions commençant par `guide_` ;
- le *thème* du graphiques (mise en forme des différents éléments) avec `ggplot2::theme()`.

```
p +
  labs(
    x = "Longueur du pétale",
    y = "Largeur du pétale",
    colour = "Espèce"
  ) +
  ggtitle(
    "Relation entre longueur et largeur des pétales",
    subtitle = "Jeu de données Iris"
  ) +
  scale_x_continuous(breaks = 1:7) +
  scale_y_continuous(
```

```
  labels = scales::label_number(decimal.mark = ",")
) +
coord_equal() +
facet_grid(cols = vars(Species)) +
guides(
  color = guide_legend(nrow = 2)
) +
theme_light() +
theme(
  legend.position = "bottom",
  axis.title = element_text(face = "bold")
)
```

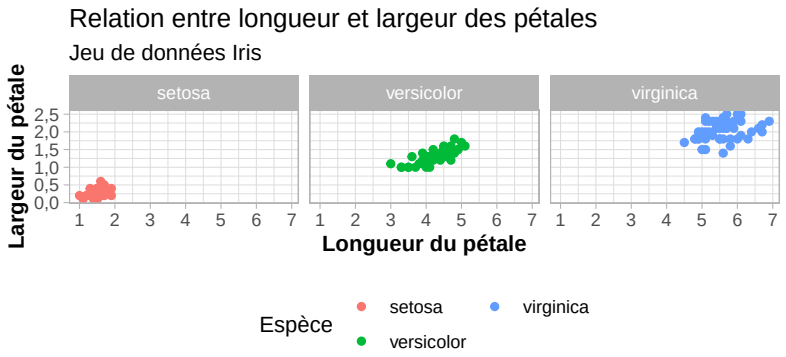


Figure 17.3: Un exemple avancé de nuage de points avec `ggplot2`

Pour visualiser chaque étape du code, vous pouvez consulter le diaporama suivant : https://larmarange.github.io/guide-R/analyses/ressources/flipbook-ggplot2.html

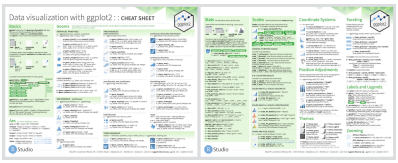


Figure 17.4: Cheatsheet ggplot2

17.3 Cheatsheet

17.4 Exploration visuelle avec `esquisse`

Le package `{esquisse}` propose un *addin* offrant une interface visuelle pour la création de graphiques `{ggplot2}`. Après installation du package, on pourra lancer `{esquisse}` directement à partir du menu *addins* de **RStudio**.

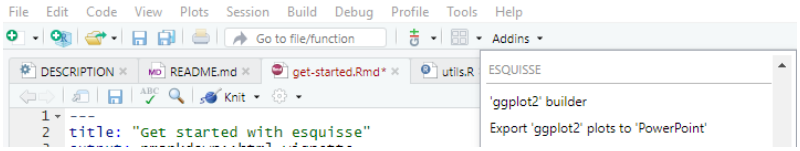


Figure 17.5: Lancement d'`esquisse` à partir du menu *Addins* de **RStudio**

Au lancement de l'*addin*, une interface permettra de choisir le tableau de données à partir duquel générer le graphique. Le plus simple est de choisir un tableau présent dans l'environnement. Mais `{esquisse}` offre aussi la possibilité d'importer des fichiers externes, voir de procéder à quelques modificatiosn des données.

Le principe général d'`{esquisse}` consiste à associer des variables à des esthétiques par glisser/déposer[22]. L'outil déterminera automatiquement une géométrie adaptée en fonction de la nature des variables (continues ou catgéorielles). Un clic sur le nom de la géométrie en haut à gauche permet de sélectionner une autre géométrie.

Les menus situés en bas de l'écran permettent d'ajouter/modifier des étiquettes, de modifier certaines options du graphiques, de

[22] Si une esthétique n'est pas visible à l'écran, on pourra cliquer en haut à droite sur l'icône en forme de roue dentée afin de choisir d'afficher plus d'esthétiques.

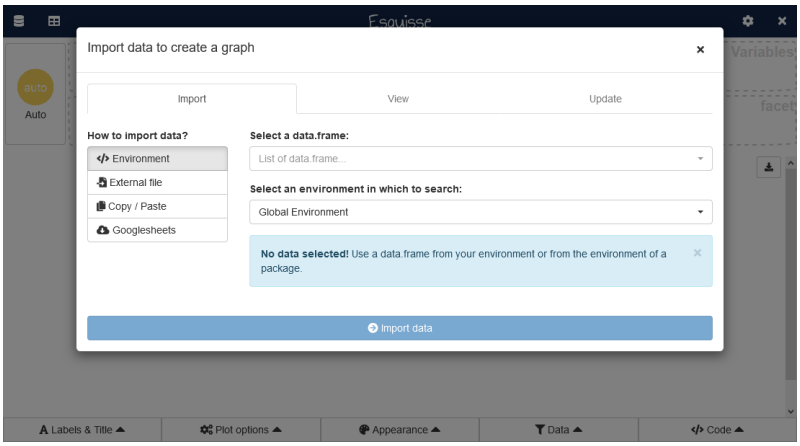


Figure 17.6: Import de données au lancement d'`esquisse`

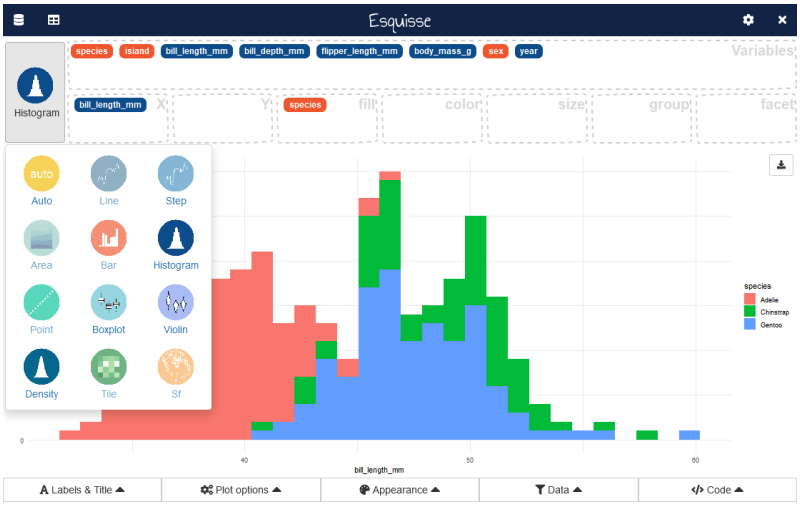


Figure 17.7: Choix d'une géométrie dans `esquisse`

modifier les échelles de couleurs et l'apparence du graphique, et de filtrer les observations inclues dans le graphique.

Le menu **Code** permet de récupérer le code correspondant au graphique afin de pouvoir le copier/coller dans un script.

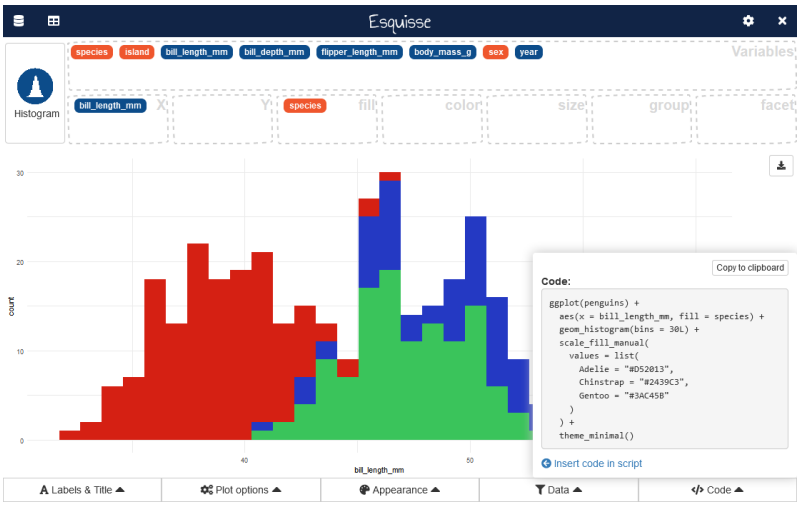


Figure 17.8: Obtenir le code du graphique obtenu avec `esquisse`

`{esquisse}` offre également la possibilité d'exporter le graphique obtenu dans différents formats.

17.5 webin-R

L'utilisation d'`{esquisse}` est présentée dans le webin-R #03 (*statistiques descriptives avec gtsummary et esquisse*) sur YouTube.

https://youtu.be/oEF_8GXyP5c

`{ggplot2}` est abordé plus en détails dans le webin-R #08 (*ggplot2 et la grammaire des graphiques*) sur YouTube.

https://youtu.be/msnwENny_cg

17.6 Combiner plusieurs graphiques

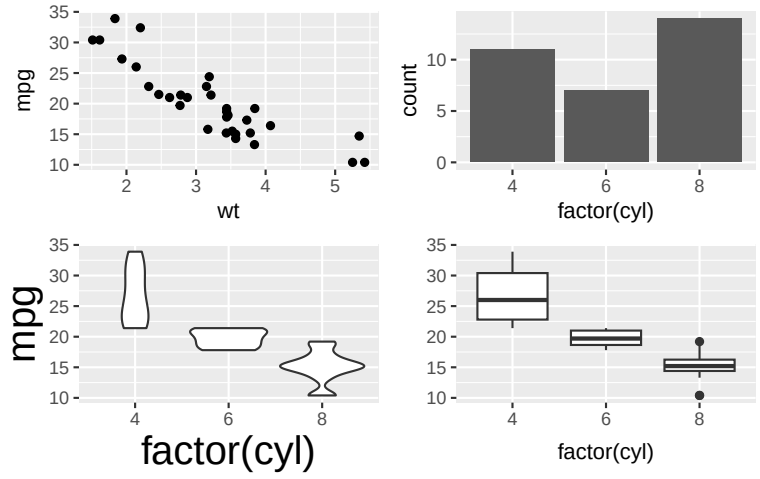
Plusieurs packages proposent des fonctions pour combiner ensemble des graphiques `{ggplot2}`, comme `{patchwork}`, `{ggpubr}`, `{egg}` ou `{cowplot}`. Ici, nous privilégierons le package `{patchwork}` car, bien qu'il ne fasse pas partie du *tidyverse*, est développé et maintenant par les mêmes auteurs que `{ggplot2}`.

Commençons par créer quelques graphiques avec `{ggplot2}`.

```
p1 <- ggplot(mtcars) +
  aes(x = wt, y = mpg) +
  geom_point()
p2 <- ggplot(mtcars) +
  aes(x = factor(cyl)) +
  geom_bar()
p3 <- ggplot(mtcars) +
  aes(x = factor(cyl), y = mpg) +
  geom_violin() +
  theme(axis.title = element_text(size = 20))
p4 <- ggplot(mtcars) +
  aes(x = factor(cyl), y = mpg) +
  geom_boxplot() +
  ylab(NULL)
```

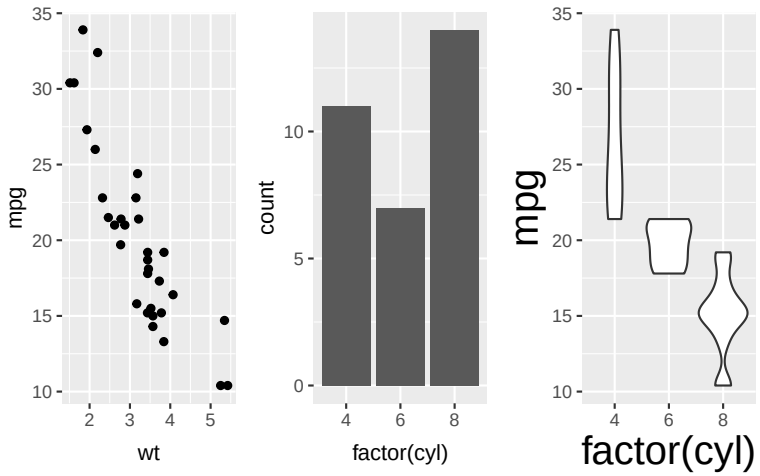
Le symbole `+` permet de combiner des graphiques entre eux. Le package `{patchwork}` déterminera le nombre de lignes et de colonnes en fonction du nombre de graphiques. On pourra noter que les axes des graphiques sont alignés les uns par rapports aux autres.

```
library(patchwork)
p1 + p2 + p3 + p4
```

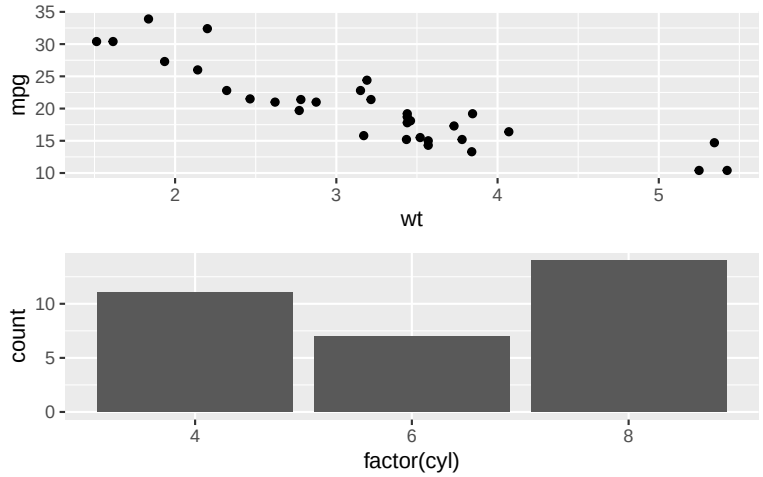


Les symboles | et / permettent d'indiquer une disposition côte à côte ou les uns au-dessus des autres.

```
p1 | p2 | p3
```

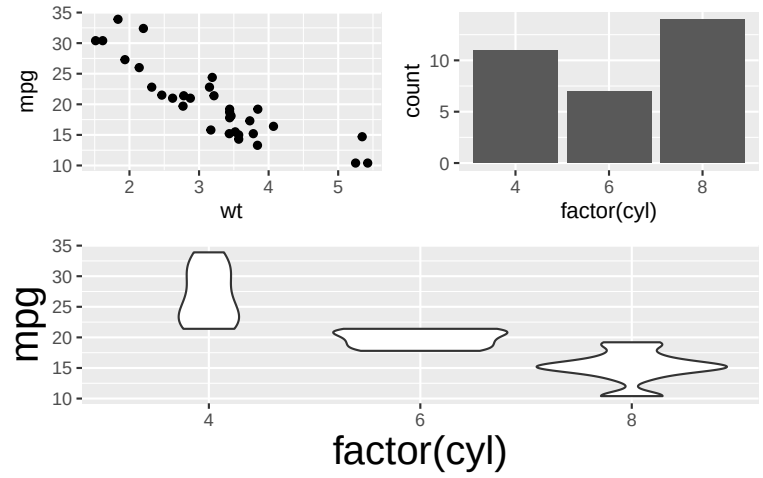



```
p1 / p2
```

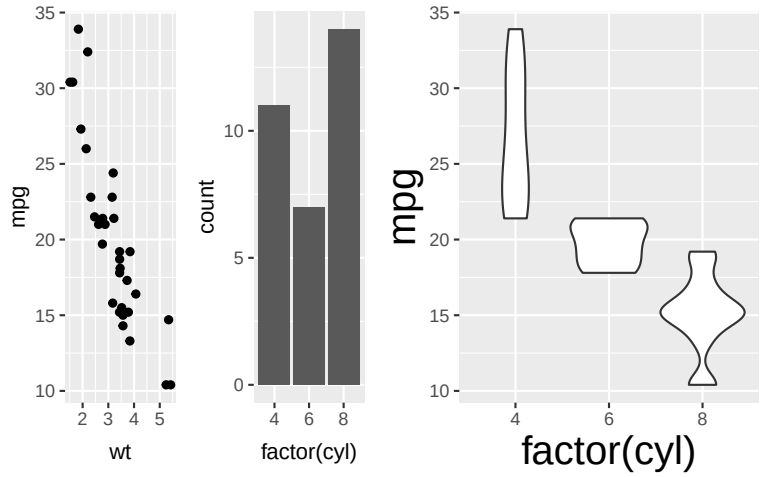


On peut utiliser les parenthèses pour indiquer des arrangements plus complexes.

```
(p1 + p2) / p3
```

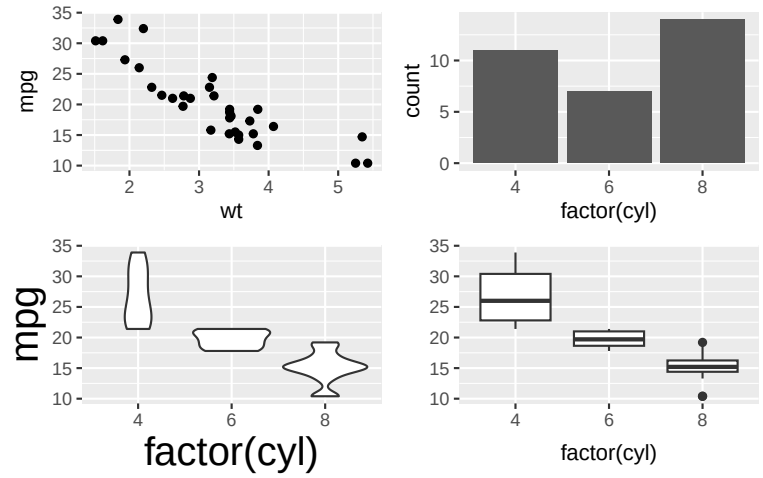



```
(p1 + p2) | p3
```



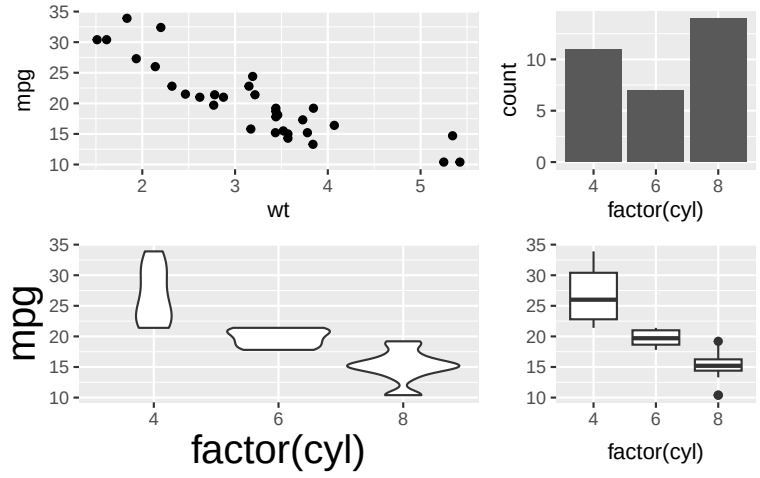
Si l'on a une liste de graphiques, on pourra appeler `patchwork::wrap_plots()`.

```
list(p1, p2, p3, p4) |>
  wrap_plots()
```



La fonction `patchwork::plot_layout()` permet de controler les hauteurs / largeurs relatives des lignes / colonnes.

```
p1 + p2 + p3 + p4 + plot_layout(widths = c(2, 1))
```

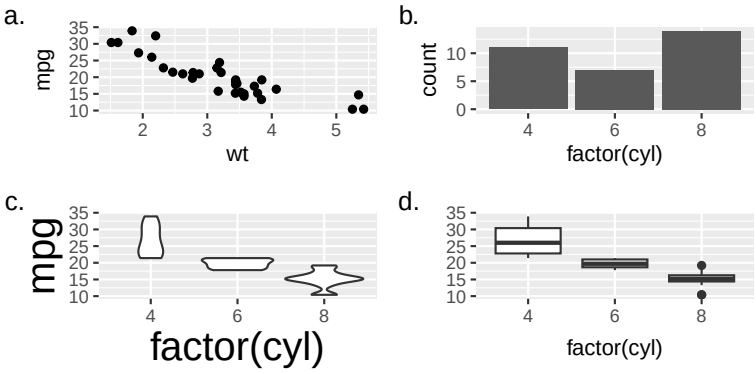


On peut également ajouter un titre ou des étiquettes avec `patchwork::plot_annotation()`.

```
p1 + p2 + p3 + p4 +
  plot_annotation(
    title = "Titre du graphique",
    subtitle = "sous-titre",
    caption = "notes additionelles",
    tag_levels = "a",
    tag_suffix = "."
  )
```

Titre du graphique

sous−titre



notes additionelles

18 Statistique univariée & Intervalles de confiance

On entend par statistique univariée l'étude d'une seule variable, que celle-ci soit continue (quantitative) ou catégorielle (qualitative). La statistique univariée fait partie de la statistique descriptive.

18.1 Exploration graphique

Une première approche consiste à explorer visuelle la variable d'intérêt, notamment à l'aide de l'interface proposée par `{esquisse}` (cf Section **??**).

Nous indiquons ci-après le code correspond aux graphiques `{ggplot2}` les plus courants.

```
library(ggplot2)
```

18.1.1 Variable continue

Un histogramme est la représentation graphique la plus commune pour représenter la distribution d'une variable, par exemple ici la longueur des pétales (variable `Petal.Length`) du fichier de données `datasets::iris`. Il s'obtient avec la géométrie `ggplot2::geom_histogram()`.

```
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

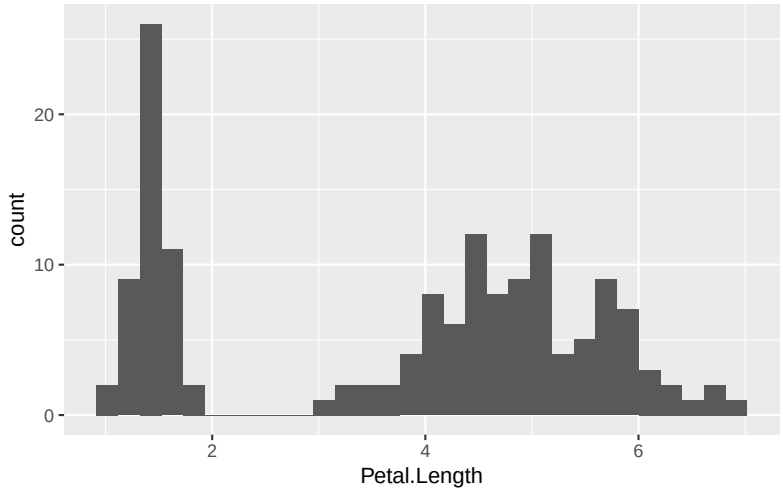


Figure 18.1: un histogramme simple

> **Astuce**
>
> Il faut noter qu'il nous a suffit d'associer simplement la variable `Petal.Length` à l'esthétique **x**, sans avoir eu besoin d'indiquer une variable pour l'esthétique **y**.
> En fait, `{ggplot2}` associe par défaut à toute géométrie une certaine statistique. Dans le cas de `ggplot2::geom_histogram()`, il s'agit de la statistique `ggplot2::stat_bin()` qui divise la variable continue en classes de même largeur et compte le nombre d'observation dans chacunes. `ggplot2::stat_bin()` renvoie un certain nombre de variables calculées (la liste complète est indiquée dans la documentation dans la section *Compute variables*), dont la variable `count` qui correspond au nombre d'observations la classe. On peut associer cette variable calculée à une esthétique grace à la fonction `ggplot2::after_stat()`, par exemple `aes(y = after_stat(count))`. Dans le cas présent, ce n'est pas nécessaire car `{ggplot2}` fait cette association automatiquement si l'on n'a pas déjà attribué une variable à l'esthétique **y**.

On peut personnaliser la couleur de remplissage des rectangles en indiquant une valeur fixe pour l'esthétique `fill` dans l'appel de `ggplot2::geom_histogram()` (et non via la fonction `ggplot2::aes()` puisqu'il ne s'agit pas d'une variable du tableau de données). L'esthétique `colour` permet de spécifier la couleur du trait des rectangles. Enfin, le paramètre `binwidth` permet de spécifier la largeur des barres.

```
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram(
    fill ="lightblue",
    colour = "black",
    binwidth = 1
  ) +
  xlab("Longeur du pétale") +
  ylab("Effectifs")
```

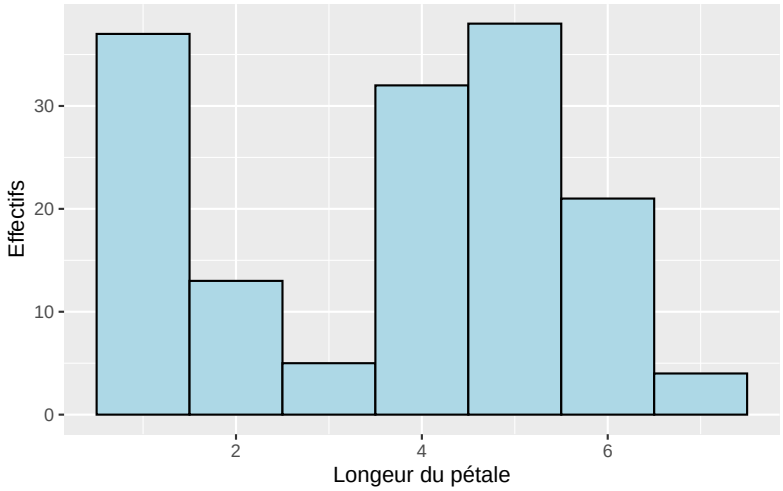


Figure 18.2: un histogramme personnalisé

On peut alternativement indiquer un nombre de classes avec `bins`.

```
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram(bins = 10, colour = "black")
```

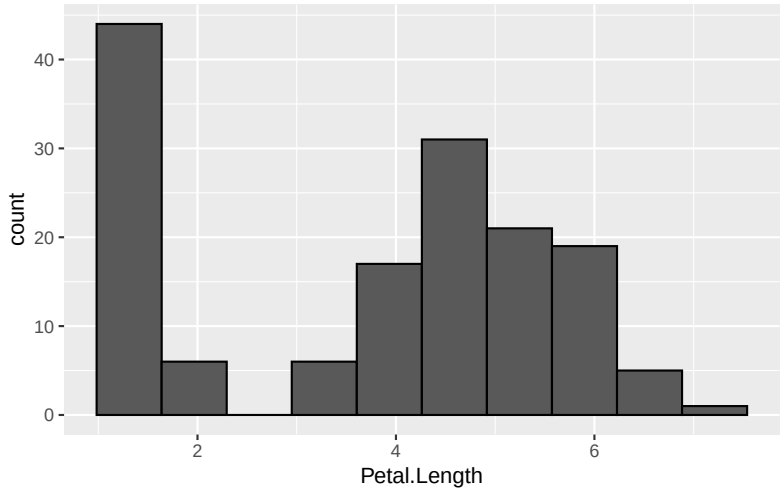


Figure 18.3: un histogramme en 10 classes

Une représentation alternative de la distribution d'une variable peut être obtenue avec une courbe de densité, dont la particularité est d'avoir une surface sous la courbe égale à 1. Une telle courbe s'obtient avec `ggplot2::geom_density()`. Le paramètre `adjust` permet d'ajuster le niveau de lissage de la courbe.

```
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_density(adjust = .5)
```

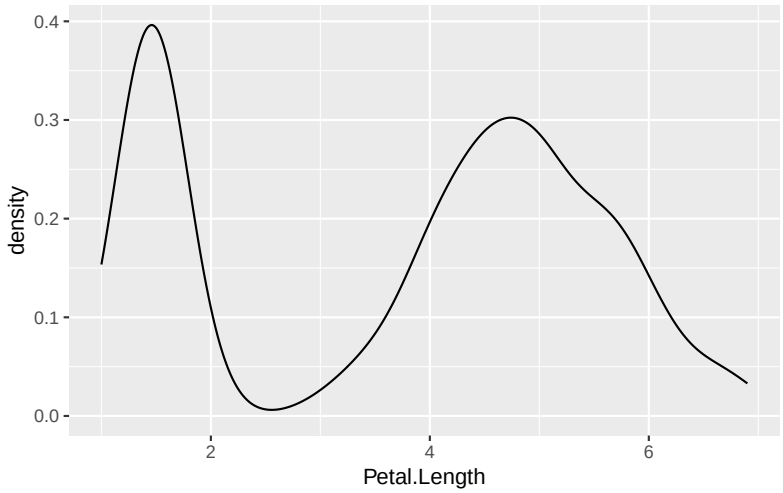


Figure 18.4: une courbe de densité

18.1.2 Variable catégorielle

Pour représenter la répartition des effectifs parmi les modalités d'une variable catégorielle, on a souvent tendance à utiliser des diagrammes en secteurs (camemberts). Or, ce type de représentation graphique est très rarement appropriée : l'oeil humain préfère comparer des longueurs plutôt que des surfaces[23].

[23] Voir en particulier https://www.data-to-viz.com/caveat/pie.html pour un exemple concret.

Dans certains contextes ou pour certaines présentations, on pourra éventuellement considérer un diagramme en donut, mais le plus souvent, rien ne vaut un bon vieux diagramme en barres avec `ggplot2::geom_bar()`. Prenons pour l'exemple la variable `occup` du jeu de données `hdv2003` du package `{questionr}`.

```
data("hdv2003", package = "questionr")
ggplot(hdv2003) +
  aes(x = occup) +
  geom_bar()
```

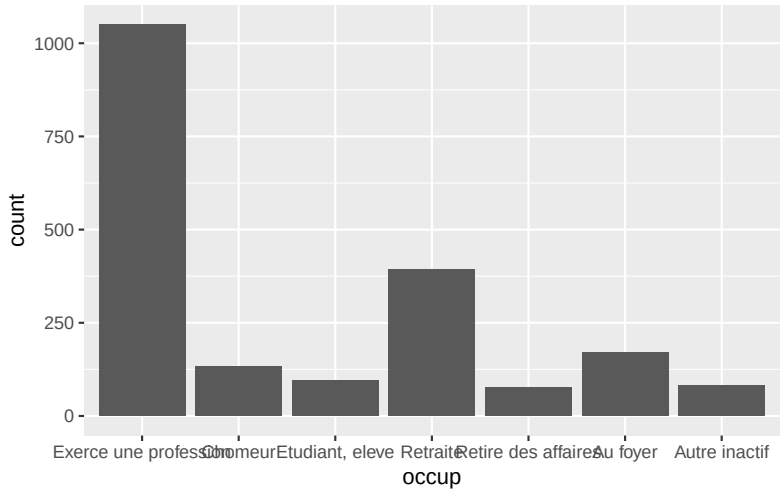


Figure 18.5: un diagramme en barres simple

> **Astuce**
>
> Là encore, `{ggplot2}` a calculé de lui-même le nombre d'observations de chaque modalité, en utilisant cette fois la statistique `ggplot2::stat_count()`.

Si l'on souhaite représenter des pourcentages plutôt que des effectifs, le plus simple est d'avoir recours à la statistique `ggstats::stat_prop()` du package `{ggstats}`[24]. Pour appeler cette statistique, on utilisera simplement `stat = "prop"` dans les géométries concernées.

[24] Cette statistique est également disponible via le package `{GGally}`.

Cette statistique, qui sera également bien utile pour des graphiques plus complexes, nécessite qu'on lui indique une esthétique **by** pour dans quels sous-groupes calculés des proportions. Ici, nous avons un seul groupe considéré et nous souhaitons des pourcentages du total. On indiquera simplement `by = 1`.

Pour formater l'axe vertical avec des pourcentages, on pourra avoir recours à la fonction `scales::label_percent()` que l'on appelera via `ggplot2::scale_y_continuous()`.

```
library(ggstats)
ggplot(hdv2003) +
```

```
  aes(x = occup, y = after_stat(prop), by = 1) +
  geom_bar(stat = "prop") +
  scale_y_continuous(labels = scales::label_percent())
```

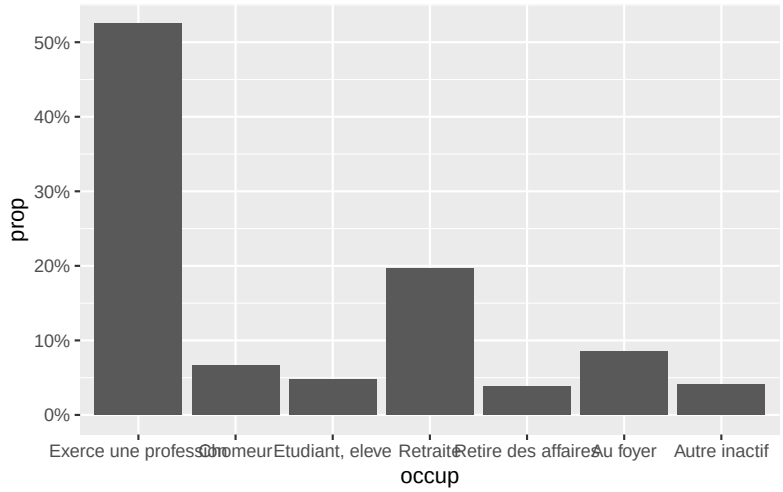


Figure 18.6: un diagramme en barres épuré

Pour une publication ou une communication, il ne faut surtout pas hésiter à **épurer** vos graphiques (*less is better!*), voire à trier les modalités en fonction de leur fréquence pour faciliter la lecture (ce qui se fait aisément avec `forcats::fct_infreq()`).

```
ggplot(hdv2003) +
  aes(x = forcats::fct_infreq(occup),
      y = after_stat(prop), by = 1) +
  geom_bar(stat = "prop",
           fill = "#4477AA", colour = "black") +
  geom_text(
    aes(label = after_stat(prop) |>
          scales::percent(accuracy = .1)),
    stat = "prop",
    nudge_y = .02
  ) +
  theme_minimal() +
```

```
  theme(
    panel.grid = element_blank(),
    axis.text.y = element_blank()
  ) +
  xlab(NULL) + ylab(NULL) +
  ggtitle("Occupation des personnes enquêtées")
```

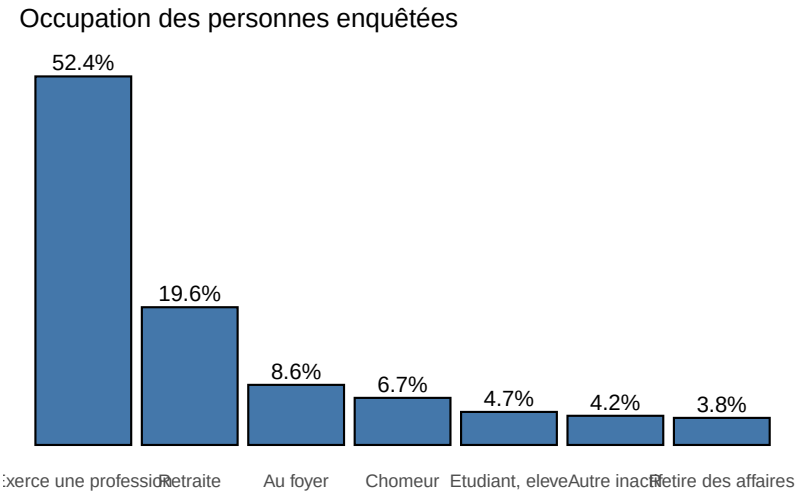


Figure 18.7: un diagramme en barres épuré

Pour visualiser chaque étape du code, vous pouvez consulter le diaporama suivant : https://larmarange.github.io/guide-R/analyses/ressources/flipbook-geom_bar-univarie.html

18.2 Tableaux et tris à plat

Le package `{gtsummary}` constitue l'une des boites à outils de l'analyste quantitativiste, car il permet de réaliser très facilement des tableaux quasiment publiables en l'état. En matière de statistique univarisée, la fonction clé est `gtsummary::tbl_summary()`.

Commençons avec un premier exemple rapide. On part d'un tableau de données et on indique, avec l'argument `include`, les variables à afficher dans le tableau statistique (si on

n'indique rien, toutes les variables du tableau de données sont considérées). Il faut noter que l'argument `include` de `gtsummary::tbl_summary()` utilise la même syntaxe dite *tidy select* que `dplyr::select()` (cf. Section **??**). On peut indiquer tout autant des variables catégorielles que des variables continues.

```
library(gtsummary)
```

```
#BlackLivesMatter
```

```
hdv2003 |>
  tbl_summary(include = c(age, occup))
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.1: un tableau simple

Characteristic	**N = 2,000**
age	48 (35, 60)
occup	
Exerce une profession	1,049 (52%)
Chomeur	134 (6.7%)
Etudiant, eleve	94 (4.7%)
Retraite	392 (20%)
Retire des affaires	77 (3.9%)
Au foyer	171 (8.6%)
Autre inactif	83 (4.2%)

! Remarque sur les types de variables et les sélecteurs associés

`{gtsummary}` permets de réaliser des tableaux statistiques combinant plusieurs variables, l'affichage des résultats pouvant dépendre du type de variables.

Par défaut, `{gtsummary}` considère qu'une variable est **catégorielle** s'il s'agit d'un facteur, d'une variable textuelle ou d'une variable numérique ayant moins de 10 valeurs différentes.
Une variable sera considérée comme **dichotomique** (variable catégorielle à seulement deux modalités) s'il s'agit d'un vecteur logique (`TRUE`/`FALSE`), d'une variable textuelle codée `yes`/`no` ou d'une variable numérique codée `0`/`1`.
Dans les autres cas, une variable numérique sera considérée comme **continue**.
Si vous utilisez des vecteurs labellisés (cf. Chapitre **??**), vous devez les convertir, en amont, en facteurs ou en variables numériques. Voir l'extension `{labelled}` et les fonctions `labelled::to_factor()`, `labelled::unlabelled()` et `unclass()`.
Au besoin, il est possible de forcer le type d'une variable avec l'argument `type` de `gtsummary::tbl_summary()`.
`{gtsummary}` fournit des sélecteurs qui peuvent être utilisés dans les options des différentes fonctions, en particulier `gtsummary::all_continuous()` pour les variables continues, `gtsummary::all_dichotolous()` pour les variables dichotomiques et `gtsummary::all_categorical()` pour les variables catégorielles. Cela inclue les variables dichotomiques. Il faut utiliser `all_categorical(dichotomous = FALSE)` pour sélectionner les variables catégorielles en excluant les variables dichotomiques.

18.2.1 Thème du tableau

`{gtsummary}` fournit plusieurs fonctions préfixées `theme_gtsummary_*()` permettant de modifier l'affichage par défaut des tableaux. Vous aurez notez que, par défaut, `{gtsummary}` est anglophone.

La fonction `gtsummary::theme_gtsummary_journal()` permets d'adopter les standards de certaines grandes revues scientifiques telles que *JAMA* (*Journal of the American Medical Association*), *The Lancet* ou encore le *NEJM* (*New England Journal of Medicine*).

La fonction `gtsummary::theme_gtsummary_language()` permet de modifier la langue utilisée par défaut dans les tableaux. Les options `decimal.mark` et `big.mark` permettent de définir respectivement le séparateur de décimales et le séparateur des milliers. Ainsi, pour présenter un tableau en français, on appliquera en début de script :

```
theme_gtsummary_language(
  language = "fr",
  decimal.mark = ",",
  big.mark = " "
)
```

```
Setting theme `language: fr`
```

Ce thème sera appliqué à tous les tableaux ultérieurs.

```
hdv2003 |>
  tbl_summary(include = c(age, occup))
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.2: un tableau simple en français

Caractéristique	**N = 2 000**
age	48 (35 – 60)
occup	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

18.2.2 Étiquettes des variables

`gtsummary`, par défaut, prends en compte les étiquettes de variables (cf. Chapitre **??**), si elles existent, et sinon utilisera le nom de chaque variable dans le tableau. Pour rappel, les étiquettes de variables peuvent être manipulées avec l'extension `{labelled}` et les fonctions `labelled::var_label()` et `labelled::set_variable_labels()`.

Il est aussi possible d'utiliser l'option `label` de `gtsummary::tbl_summary()` pour indiquer des étiquettes personnalisées.

```
hdv2003 |>
  labelled::set_variable_labels(
    occup = "Occupation actuelle"
  ) |>
  tbl_summary(
    include = c(age, occup, heures.tv),
    label = list(age ~ "Âge médian")
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.3: un tableau étiquetté

Caractéristique	**N = 2 000**
Âge médian	48 (35 – 60)
Occupation actuelle	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)
heures.tv	2,00 (1,00 – 3,00)
Manquant	5

Pour modifier les modalités d'une variable catégorielle, il faut modifier en amont les niveaux du facteur correspondant.

> ! Remarque sur la syntaxe des options
>
> De nombreuses options des fonctions de `{gtsummary}` peuvent s'appliquer seulement à une ou certaines variables. Pour ces options-là, `{gtsummary}` attends une formule de la forme `variables concernées ~ valeur de l'option` ou bien une liste de formules ayant cette forme. Par exemple, pour modifier l'étiquette associée à une certaine variable, on peut utiliser l'option `label` de `gtsummary::tbl_summary()`.
>
> ```
> trial |>
>   tbl_summary(label = age ~ "Âge")
> ```
>
> Lorsque l'on souhaite passer plusieurs options pour plusieurs variables différentes, on utilisera une `list()`.
>
> ```
> trial |>
>   tbl_summary(label = list(age ~ "Âge", trt ~ "Traitement"))
> ```
>
> `{gtsummary}` est très flexible sur la manière d'indiquer la ou les variables concernées. Il peut s'agir du nom de la variable, d'une chaîne de caractères contenant le nom de la variable, ou d'un vecteur contenant le nom de la variable. Les syntaxes ci-dessous sont ainsi équivalentes.
>
> ```
> trial |>
>   tbl_summary(label = age ~ "Âge")
> trial |>
>   tbl_summary(label = "age" ~ "Âge")
> v <- "age"
> trial |>
>   tbl_summary(label = v ~ "Âge")
> ```
>
> Pour appliquer le même changement à plusieurs variables, plusieurs syntaxes sont acceptées pour lister plusieurs variables.

```
trial |>
  tbl_summary(label = c("age", "trt") ~ "Une même étiquette")
trial |>
  tbl_summary(label = c(age, trt) ~ "Une même étiquette")
```

Il est également possible d'utiliser la syntaxe `{tidyselect}` et les sélecteurs de `{tidyselect}` comme `tidyselect::everything()`, `tidyselect::starts_with()`, `tidyselect::contains()` ou `tidyselect::all_of()`. Ces différents sélecteurs peuvent être combinés au sein d'un `c()`.

```
trial |>
  tbl_summary(
    label = everything() ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = starts_with("a") ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = c(everything(), -age, -trt) ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = age:trt ~ "Une même étiquette"
  )
```

Bien sûr, il est possible d'utiliser les sélecteurs propres à `{gtsummary}`.

```
trial |>
  tbl_summary(
    label = all_continuous() ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = list(
      all_continuous() ~ "Variable continue",
      all_dichotomous() ~ "Variable dichotomique",
      all_categorical(dichotomous = FALSE) ~ "Variable catégorielle"
    )
  )
```

Enfin, si l'on ne précise rien à gauche du `~`, ce sera considéré comme équivalent à `everything()`. Les deux syntaxes ci-dessous sont donc équivalentes.

```
trial |>
  tbl_summary(label = ~ "Une même étiquette")
trial |>
  tbl_summary(
    label = everything() ~ "Une même étiquette"
  )
```

18.2.3 Statistiques affichées

Le paramètre `statistic` permets de sélectionner les statistiques à afficher pour chaque variable. On indiquera une chaîne de caractères dont les différentes statistiques seront indiquées entre accolades (`{}`).

Pour une **variable continue**, on pourra utiliser `{median}` pour la médiane, `{mean}` pour la moyenne, `{sd}` pour l'écart type, `{var}` pour la variance, `{min}` pour le minimum, `{max}` pour le maximum, ou encore `{p##}` (en remplacant `##` par un nombre entier entre 00 et 100) pour le percentile correspondant (par exemple `p25` et `p75` pour le premier et le troisième quartile). Utilisez `gtsummary::all_continous()` pour sélectionner toutes les variables continues.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    statistic =
      all_continuous() ~ "Moy. : {mean} [min-max : {min} - {max}]"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.4: statisques personnalisées pour une variable continue

Caractéristique	**N = 2 000**
age	Moy. : 48 [min-max : 18 - 97]
heures.tv	Moy. : 2,25 [min-max : 0,00 - 12,00]
Manquant	5

Il est possible d'afficher des statistiques différentes pour chaque variable.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    statistic = list(
      age ~ "Méd. : {median} [{p25} - {p75}]",
      heures.tv ~ "Moy. : {mean} ({sd})"
    )
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.5: statisques personnalisées pour une variable continue (2)

Caractéristique	**N = 2 000**
age	Méd. : 48 [35 - 60]
heures.tv	Moy. : 2,25 (1,78)
Manquant	5

Pour les variables continues, il est également possible d'indiquer le nom d'une fonction personnalisée qui prends un vecteur et renvoie une valeur résumée. Par exemple, pour afficher la moyenne des carrés :

```
moy_carres <- function(x) {
  mean(x^2, na.rm = TRUE)
}
hdv2003 |>
  tbl_summary(
    include = heures.tv,
    statistic = ~ "MC : {moy_carres}"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.6: statisques personnalisées pour une variable continue (3)

Caractéristique	**N = 2 000**
heures.tv	MC : 8,20
Manquant	5

Pour une **variable catégorielle**, les statistiques possibles sont `{n}` le nombre d'observations, `{N}` le nombre total d'observations, et `{p}` le pourcentage correspondant. Utilisez `gtsummary::all_categorical()` pour sélectionner toutes les variables catégorielles.

```
hdv2003 |>
  tbl_summary(
    include = occup,
    statistic = all_categorical() ~ "{p} % ({n}/{N})"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.7: statisques personnalisées pour une variable catégorielle

Caractéristique	**N = 2 000**
occup	
Exerce une profession	52 % (1 049/2 000)
Chomeur	6,7 % (134/2 000)
Etudiant, eleve	4,7 % (94/2 000)
Retraite	20 % (392/2 000)
Retire des affaires	3,9 % (77/2 000)
Au foyer	8,6 % (171/2 000)
Autre inactif	4,2 % (83/2 000)

Il est possible, pour une variable catégorielle, de trier les modalités de la plus fréquente à la moins fréquente avec le paramètre `sort`.

```
hdv2003 |>
  tbl_summary(
    include = occup,
    sort = all_categorical() ~ "frequency"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.8: variable catégorielle triée par fréquence

Caractéristique	**N = 2 000**
occup	
Exerce une profession	1 049 (52%)
Retraite	392 (20%)
Au foyer	171 (8,6%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Autre inactif	83 (4,2%)
Retire des affaires	77 (3,9%)

Pour toutes les variables (catégorielles et continues), les statistiques suivantes sont également disponibles :

- `{N_obs}` le nombre total d'observations,
- `{N_miss}` le nombre d'observations manquantes (`NA`),
- `{N_nonmiss}` le nombre d'observations non manquantes,
- `{p_miss}` le pourcentage d'observations manquantes (i.e. `N_miss / N_obs`) et
- `{p_nonmiss}` le pourcentage d'observations non manquantes (i.e. `N_nonmiss / N_obs`).

18.2.4 Affichage du nom des statistiques

Lorsque l'on affiche de multiples statistiques, la liste des statistiques est regroupée dans une note de tableau qui peut vite devenir un peu confuse.

```
tbl <- hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv, occup),
    statistic = list(
      age ~ "{mean} ({sd})",
      heures.tv ~ "{median} [{p25} - {p75}]"
    )
  )
tbl
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.9: tableau par défaut

Caractéristique	**N = 2 000**
age	48 (17)
heures.tv	2,00 [1,00 - 3,00]
Manquant	5
occup	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

La fonction `gtsummary::add_stat_label()` permets d'indiquer le type de statistique à côté du nom des variables ou bien dans une colonne dédiée, plutôt qu'en note de tableau.

```
tbl |>
  add_stat_label()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.10: ajout du nom des statistiques

Caractéristique	**N = 2 000**
age, Moyenne (ET)	48 (17)
heures.tv, Médiane [EI]	2,00 [1,00 - 3,00]
Manquant	5

Caractéristique	N = 2 000
occup, n (%)	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

```
tbl |>
  add_stat_label(location = "column")
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.11: ajout du nom des statistiques dans une colonne séparée

Caractéristique	Statistique	N = 2 000
age	Moyenne (ET)	48 (17)
heures.tv	Médiane [EI]	2,00 [1,00 - 3,00]
Manquant	n	5
occup		
Exerce une profession	n (%)	1 049 (52%)
Chomeur	n (%)	134 (6,7%)
Etudiant, eleve	n (%)	94 (4,7%)
Retraite	n (%)	392 (20%)
Retire des affaires	n (%)	77 (3,9%)
Au foyer	n (%)	171 (8,6%)
Autre inactif	n (%)	83 (4,2%)

18.2.5 Forcer le type de variable

Comme évoqué plus haut, `{gtsummary}` détermine automatiquement le type de chaque variable. Par défaut, la variabe `age`

du tableau de données `trial` est traitée comme variable continue, `death` comme dichotomique (seule la valeur 1 est affichée) et `grade` comme variable catégorielle.

```
trial |>
  tbl_summary(
    include = c(grade, age, death)
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.12: types de variable par défaut

Caractéristique	**N = 200**
Grade	
I	68 (34%)
II	68 (34%)
III	64 (32%)
Age	47 (38 – 57)
Manquant	11
Patient Died	112 (56%)

Il est cependant possible de forcer un certain type avec l'argument `type`. Précision : lorsque l'on force une variable en dichotomique, il faut indiquer avec `value` la valeur à afficher (les autres sont alors masquées).

```
trial |>
  tbl_summary(
    include = c(grade, death),
    type = list(
      grade ~ "dichotomous",
      death ~ "categorical"
    ),
    value = grade ~ "III",
    label = grade ~ "Grade III"
```

```
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.13: types de variable personnalisés

Caractéristique	**N = 200**
Grade III	64 (32%)
Patient Died	
0	88 (44%)
1	112 (56%)

18.2.6 Afficher des statistiques sur plusieurs lignes (variables continues)

Pour les variables continues, `{gtsummary}` a introduit un type de variable `"continuous2"`, qui doit être attribué manuellement via `type`, et qui permets d'afficher plusieurs lignes de statistiques (en indiquant plusieurs chaînes de caractères dans `statistic`). À noter le sélecteur dédié `gtsummary::all_continuous2()`.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    type = age ~ "continuous2",
    statistic =
      all_continuous2() ~ c(
        "{median} ({p25} - {p75})",
        "{mean} ({sd})",
        "{min} - {max}"
      )
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
```

```
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.14: des statistiques sur plusieurs lignes (variables continues)

Caractéristique	**N = 2 000**
age	
Médiane (EI)	48 (35 - 60)
Moyenne (ET)	48 (17)
Étendue	18 - 97
heures.tv	2,00 (1,00 – 3,00)
Manquant	5

18.2.7 Mise en forme des statistiques

L'argument `digits` permet de spécifier comment mettre en forme les différentes statistiques. Le plus simple est d'indiquer le nombre de décimales à afficher. Il est important de tenir compte que plusieurs statistiques peuvent être affichées pour une même variable. On peut alors indiquer une valeur différente pour chaque statistique.

```
hdv2003 |>
  tbl_summary(
    include = c(age, occup),
    digits = list(
      all_continuous() ~ 1,
      all_categorical() ~ c(0, 1)
    )
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.15: personnalisation du nombre de décimales

Caractéristique	**N = 2 000**
age	48,0 (35,0 – 60,0)
occup	
Exerce une profession	1 049 (52,4%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (19,6%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

Au lieu d'un nombre de décimales, on peut indiquer plutôt une fonction à appliquer pour mettre en forme le résultat. Par exemple, `{gtsummary}` fournit les fonctions suivantes : `gtsummary::style_number()` pour les nombres de manière générale, `gtsummary::style_percent()` pour les pourcentages (les valeurs sont multipliées par 100, mais le symbole % n'est pas ajouté), `gtsummary::style_pvalue()` pour les p-valeurs, `gtsummary::style_sigfig()` qui n'affiche, par défaut, que deux chiffres significatifs, ou encore `gtsummary::style_ratio()` qui est une variante de `gtsummary::``style_sigfig()` pour les ratios (comme les *odds ratios*) que l'on compare à 1.

Il faiut bien noter que ce qui est attendu par `digits`, c'est une fonction et non le résultat d'une fonction. On indiquera donc le nom de la fonction sans parenthèse, comme dans l'exemple ci-dessous (même si pas forcément pertinent ;-)).

```
hdv2003 |>
  tbl_summary(
    include = age,
    digits =
      all_continuous() ~ c(style_percent, style_sigfig, style_ratio)
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
```

```
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.16: personnalisation de la mise en forme des nombres

Caractéristique	**N = 2 000**
age	4 800 (35 – 60,0)

Comme `digits` s'attends à recevoir une fonction (et non le résultat) d'une fonction, on ne peut pas passer directement des arguments aux fonctions `style_*()` de `{gtsummary}`. Pour cela il faut créer une fonction à la levée :

```
trial |>
  tbl_summary(
    include = marker,
    statistic = ~ "{mean} pour 100",
    digits = ~ function(x){style_percent(x, digits = 1)}
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.17: passer une fonction personnalisée à digits (syntaxe 1)

Caractéristique	**N = 200**
Marker Level (ng/mL)	91,6 pour 100
Manquant	10

Depuis **R 4.1**, il existe une syntaxe raccourcie équivalente, avec le symbole `\` à la place de `function`.

```
trial |>
  tbl_summary(
```

```
    include = marker,
    statistic = ~ "{mean} pour 100",
    digits = ~ \(x){style_percent(x, digits = 1)}
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.18: passer une fonction personnalisée à digits (syntaxe 2)

Caractéristique	**N = 200**
Marker Level (ng/mL)	91,6 pour 100
Manquant	10

Une syntaxe alternative consiste à avoir recours à la fonction `purrr::partial()` qui permet d'appeler partiellement une fonction et de renvoyer une nouvelle fonction.

```
trial |>
  tbl_summary(
    include = marker,
    statistic = ~ "{mean} pour 100",
    digits = ~ purrr::partial(style_percent, digits = 1)
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.19: passer une fonction personnalisée à digits (syntaxe 3)

Caractéristique	**N = 200**
Marker Level (ng/mL)	91,6 pour 100
Manquant	10

À noter dans l'exemple précédent que les fonctions `style_*()` de `{gtsummary}` tiennent compte du thème défini (ici la virgule comme séparateur de décimale).

Pour une mise en forme plus avancée des nombres, il faut se tourner vers l'extension `{scales}` et ses diverses fonctions de mise en forme comme `scales::label_number()` ou `scales::label_percent()`.

ATTENTION : les fonctions de `{scales}` n'héritent pas des paramètres du thème `{gtsummary}` actif. Il faut donc personnaliser le séparateur de décimal dans l'appel à la fonction.

```
trial |>
  tbl_summary(
    include = marker,
    statistic = ~ "{mean}",
    digits = ~ scales::label_number(
      accuracy = .01,
      suffix = " ng/mL",
      decimal.mark = ","
    )
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.20: passer une fonction personnalisée à digits (syntaxe 4)

Caractéristique	N = 200
Marker Level (ng/mL)	0,92 ng/mL
Manquant	10

18.2.8 Données manquantes

Le paramètre `missing` permets d'indiquer s'il faut afficher le nombre d'observations manquantes (c'est-à-dire égales à `NA`) : `"ifany"` (valeur par défaut) affiche ce nombre seulement s'il y en a, `"no"` masque ce nombre et `"always"` force l'affichage de ce nombre même s'il n'y pas de valeur manquante. Le paramètre `missing_text` permets de personnaliser le texte affiché.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    missing = "always",
    missing_text = "Nbre observations manquantes"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.21: forcer l'affichage des valeurs manquantes

Caractéristique	N = 2 000
age	48 (35 – 60)
Nbre observations manquantes	0
heures.tv	2,00 (1,00 – 3,00)
Nbre observations manquantes	5

Il est à noter, pour les variables catégorielles, que les valeurs manquantes ne sont jamais pris en compte pour le

calcul des pourcentages. Pour les inclure dans le calcul, il faut les transformer en valeurs explicites, par exemple avec `forcats::fct_na_value_to_level()` de `{forcats}`.

```
hdv2003 |>
  dplyr::mutate(
    trav.imp.explicit = trav.imp |>
      forcats::fct_na_value_to_level("(non renseigné)")
  ) |>
  tbl_summary(
    include = c(trav.imp, trav.imp.explicit)
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.22: valeurs manquantes explicites (variable catégorielle)

Caractéristique	**N = 2 000**
trav.imp	
Le plus important	29 (2,8%)
Aussi important que le reste	259 (25%)
Moins important que le reste	708 (68%)
Peu important	52 (5,0%)
Manquant	952
trav.imp.explicit	
Le plus important	29 (1,5%)
Aussi important que le reste	259 (13%)
Moins important que le reste	708 (35%)
Peu important	52 (2,6%)
(non renseigné)	952 (48%)

18.2.9 Ajouter les effectifs observés

Lorsque l'on masque les manquants, il peut être pertinent d'ajouter une colonne avec les effectifs observés pour chaque variable à l'aide de la fonction `gtsummary::add_n()`.

```
hdv2003 |>
  tbl_summary(
    include = c(heures.tv, trav.imp),
    missing = "no"
  ) |>
  add_n()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.23: ajouter une colonne avec les effectifs observés

Caractéristique	**N**	**N = 2 000**
heures.tv	1 995	2,00 (1,00 – 3,00)
trav.imp	1 048	
Le plus important		29 (2,8%)
Aussi important que le reste		259 (25%)
Moins important que le reste		708 (68%)
Peu important		52 (5,0%)

18.3 Calcul manuel

18.3.1 Variable continue

R fournit de base toutes les fonctions nécessaires pour le calcul des différentes statistiques descriptives :

- `mean()` pour la moyenne
- `sd()` pour l'écart-type
- `min()` et `max()` pour le minimum et le maximum
- `range()` pour l'étendue
- `median()` pour la médiane

Si la variable contient des valeurs manquantes (`NA`), ces fonctions renverront une valeur manquante, sauf si on leur précise `na.rm = TRUE`.

```
hdv2003$heures.tv |> mean()
```

```
[1] NA
```

```
hdv2003$heures.tv |> mean(na.rm = TRUE)
```

```
[1] 2.246566
```

```
hdv2003$heures.tv |> sd(na.rm = TRUE)
```

```
[1] 1.775853
```

```
hdv2003$heures.tv |> min(na.rm = TRUE)
```

```
[1] 0
```

```
hdv2003$heures.tv |> max(na.rm = TRUE)
```

```
[1] 12
```

```
hdv2003$heures.tv |> range(na.rm = TRUE)
```

```
[1]  0 12
```

```
hdv2003$heures.tv |> median(na.rm = TRUE)
```

```
[1] 2
```

La fonction `quantile()` permets de calculer tous types de quantiles.

```
hdv2003$heures.tv |> quantile(na.rm = TRUE)
```

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

```
hdv2003$heures.tv |>
  quantile(
    probs = c(.2, .4, .6, .8),
    na.rm = TRUE
  )
```

```
20% 40% 60% 80%
  1   2   2   3
```

La fonction `summary()` renvoie la plupart de ces indicateurs en une seule fois, ainsi que le nombre de valeurs manquantes.

```
hdv2003$heures.tv |> summary()
```

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

18.3.2 Variable catégorielle

Les fonctions de base pour le calcul d'un tri à plat sont les fonctions `table()` et `xtabs()`. Leur syntaxe est quelque peu différente. On passe un vecteur entier à `table()` alors que la syntaxe de `xtabs()` se rapproche de celle d'un modèle linéaire : on décrit le tableau attendu à l'aide d'une formule et on indique le tableau de données. Les deux fonctions renvoient le meme résultat.

```
tbl <- hdv2003$trav.imp |> table()
tbl <- xtabs(~ trav.imp, data = hdv2003)
tbl <- hdv2003 |> xtabs(~ trav.imp, data = _)
tbl
```

```
trav.imp
           Le plus important Aussi important que le reste
                          29                          259
Moins important que le reste               Peu important
                         708                           52
```

Comme on le voit, il s'agit du tableau brut des effectifs, sans les valeurs manquantes, et pas vraiment lisible dans la console de **R**.

Pour calculer les proportions, on appliquera `prop.table()` sur la table des effectifs bruts.

```
prop.table(tbl)
```

```
trav.imp
           Le plus important Aussi important que le reste
                  0.02767176                  0.24713740
Moins important que le reste               Peu important
                  0.67557252                  0.04961832
```

Pour la réalisation rapide d'un tri à plat, on pourra donc préférer la fonction `questionr::freq()` qui affiche également le nombre de valeurs manquantes et les pourcentages, en un seul appel.

```
hdv2003$trav.imp |>
  questionr::freq(total = TRUE)
```

```
                               n     %  val%
Le plus important             29   1.5   2.8
Aussi important que le reste  259  13.0  24.7
Moins important que le reste  708  35.4  67.6
Peu important                  52   2.6   5.0
NA                            952  47.6    NA
Total                        2000 100.0 100.0
```

18.4 Intervalles de confiance

18.4.1 Avec gtsummary

La fonction `gtsummary::add_ci()` permet d'ajouter des intervalles de confiance à un tableau créé avec `gtsummary::tbl_summary()`.

> ⚠ Avertissement
>
> Par défaut, pour les **variables continues**, `gtsummary::tbl_summary()` affiche la médiane tandis que `gtsummary::add_ci()` calcule l'intervalle de confiance d'une moyenne !
> Il faut donc :
>
> - soit afficher la moyenne dans `gtsummary::tbl_summary()` à l'aide du paramètre `statistic` ;
> - soit calculer les intervalles de confiance d'une médiane (méthode `"wilcox.text"`) via le paramètre `method` de `gtsummary::add_ci()`.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv, trav.imp),
    statistic = age ~ "{mean} ({sd})"
  ) |>
  add_ci(
    method = heures.tv ~ "wilcox.test"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.24: ajouter les intervalles de confiance

Caractéristique	**N = 2 000**	**95% CI**
age	48 (17)	47, 49
heures.tv	2,00 (1,00 – 3,00)	2,5, 2,5
Manquant	5	
trav.imp		
Le plus important	29 (2,8%)	1,9%, 4,0%
Aussi important que le reste	259 (25%)	22%, 27%
Moins important que le reste	708 (68%)	65%, 70%
Peu important	52 (5,0%)	3,8%, 6,5%
Manquant	952	

L'argument `statistic` permet de personnaliser la présentation de l'intervalle ; `conf.level` de changer le niveau de confiance et `style_fun` de modifier la mise en forme des nombres de l'intervalle.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    statistic = ~ "{mean}"
  ) |>
  add_ci(
    statistic = ~ "entre {conf.low} et {conf.high}",
    conf.level = .9,
    style_fun = ~ purrr::partial(style_number, digits = 1)
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 18.25: des intervalles de confiance personnalisés

Caractéristique	**N = 2 000**	**90% CI**
age	48	entre 47,5 et 48,8
heures.tv	2,25	entre 2,2 et 2,3

Caractéristique	N = 2 000	90% CI
Manquant	5	

18.4.2 Calcul manuel

Le calcul de l'intervalle de confiance d'une **moyenne** s'effectue avec la fonction `t.test()`.

```
hdv2003$age |> t.test()
```

```
    One Sample t-test

data:  hdv2003$age
t = 127.12, df = 1999, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 47.41406 48.89994
sample estimates:
mean of x
   48.157
```

Le résultat renvoyé est une liste contenant de multiples informations.

```
hdv2003$age |> t.test() |> str()
```

```
List of 10
 $ statistic  : Named num 127
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 1999
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 0
 $ conf.int   : num [1:2] 47.4 48.9
  ..- attr(*, "conf.level")= num 0.95
 $ estimate   : Named num 48.2
  ..- attr(*, "names")= chr "mean of x"
```

```
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "mean"
 $ stderr     : num 0.379
 $ alternative: chr "two.sided"
 $ method     : chr "One Sample t-test"
 $ data.name  : chr "hdv2003$age"
 - attr(*, "class")= chr "htest"
```

Si l'on a besoin d'accéder spécifiquement à l'intervalle de confiance calculé :

```
hdv2003$age |> t.test() |> purrr::pluck("conf.int")
```

```
[1] 47.41406 48.89994
attr(,"conf.level")
[1] 0.95
```

Pour celui d'une **médiane**, on utilisera `wilcox.test()` en précisant `conf.int = TRUE`.

```
hdv2003$age |> wilcox.test(conf.int = TRUE)
```

```

    Wilcoxon signed rank test with continuity correction

data:  hdv2003$age
V = 2001000, p-value < 2.2e-16
alternative hypothesis: true location is not equal to 0
95 percent confidence interval:
 47.00001 48.50007
sample estimates:
(pseudo)median
      47.99996
```

```
hdv2003$age |>
  wilcox.test(conf.int = TRUE) |>
  purrr::pluck("conf.int")
```

```
[1] 47.00001 48.50007
attr(,"conf.level")
[1] 0.95
```

Pour une **proportion**, on utilisera `prop.test()` en lui transmettant le nombre de succès et le nombre d'observations, qu'il faudra donc avoir calculé au préalable. On peut également passer une table à deux entrées avec le nombre de succès puis le nombre d'échecs.

Ainsi, pour obtenir l'intervalle de confiance de la proportion des enquêtés qui considèrent leur travail comme *peu important*, en tenant compte des valeurs manquantes, le plus simple est d'effectuer le code suivant[25] :

```
xtabs(~ I(hdv2003$trav.imp == "Peu important"), data = hdv2003) |>
  rev() |>
  prop.test()
```

[25] Notez l'utilisation de `rev()` pour inverser le tableau créé avec `xtabs()` afin que le nombre de succès (`TRUE`) soit indiqués avant le nombre d'échecs (`FALSE`).

```
    1-sample proportions test with continuity correction

data:  rev(xtabs(~I(hdv2003$trav.imp == "Peu important"), data = hdv2003)), null probability 0
X-squared = 848.52, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.03762112 0.06502346
sample estimates:
         p
0.04961832
```

Par défaut, `prop.test()` produit un intervalle de confiance bilatéral en utilisant la méthode de Wilson avec correction de continuité. Pour plus d'information sur les différentes manières de calculer l'intervalle de confiance d'une proportion, on pourra se référer à ce billet de blog.

Astuce

Comme on le voit, il n'est pas aisé, avec les fonctions de **R base** de calculer les intervalles de confiance pour toutes les modalités d'une variable catégorielle.
On pourra éventuellement avoir recours à la petite fonction suivante qui réalise le tri à plat d'une variable catégorielle, calcule les proprotions et leurs intervalles de confiance.

```
prop_ci <- function(x, conf.level = .95, correct = TRUE) {
  tbl <- as.data.frame(table(x), responseName = "n")
  tbl$N <- sum(tbl$n)
  tbl$prop <- tbl$n / tbl$N
  tbl$conf.low <- NA_real_
  tbl$conf.high <- NA_real_
  for (i in 1:nrow(tbl)) {
    test <- prop.test(
      x = tbl$n[i],
      n = tbl$N[i],
      conf.level = conf.level,
      correct = correct
    )
    tbl$conf.low[i] <- test$conf.int[1]
    tbl$conf.high[i] <- test$conf.int[2]
  }
  tbl
}
prop_ci(hdv2003$trav.imp)
```

```
                            x   n    N       prop   conf.low  conf.high
1           Le plus important  29 1048 0.02767176 0.01894147 0.04001505
2 Aussi important que le reste 259 1048 0.24713740 0.22151849 0.27463695
3 Moins important que le reste 708 1048 0.67557252 0.64614566 0.70369541
4               Peu important  52 1048 0.04961832 0.03762112 0.06502346
```

18.5 webin-R

La statistique univariée est présentée dans le webin-R #03 (*statistiques descriptives avec gtsummary et esquisse*) sur YouTube.

https://youtu.be/oEF_8GXyP5c

19 Statistique bivariée & Tests de comparaison

19.1 Deux variables catégorielles

19.1.1 Tableau croisé avec `gtsummary`

Pour regarder le lien entre deux variables catégorielles, l'approche la plus fréquente consiste à réaliser un *tableau croisé*, ce qui s'obtient très facilement avec l'argument `by` de la fonction `gtsummary::tbl_summary()` que nous avons déjà abordée dans le chapitre sur la statistique univariée (cf. Section **??**).

Prenons pour exemple le jeu de données `gtsummary::trial` et croisons les variables *stage* et *grade*. On indique à `by` la variable à représenter en colonnes et à `include` celle à représenter en lignes.

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ',')
```

```
Setting theme `language: fr`
```

```
trial |>
  tbl_summary(
    include = stage,
    by = grade
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.1: un tableau croisé avec des pourcentages en colonne

Caractéristique	**I**, N = 68	**II**, N = 68	**III**, N = 64
T Stage			
T1	17 (25%)	23 (34%)	13 (20%)
T2	18 (26%)	17 (25%)	19 (30%)
T3	18 (26%)	11 (16%)	14 (22%)
T4	15 (22%)	17 (25%)	18 (28%)

Par défaut, les pourcentages affichés correspondent à des pourcentages en colonne. On peut demander des pourcentages en ligne avec `percent = "row"` ou des pourcentages du total avec `percent = "cell"`.

Il est possible de passer plusieurs variables à `include` mais une seule variable peut être transmise à `by`. La fonction `gtsummary::add_overall()` permet d'ajouter une colonne totale. Comme pour un tri à plat, on peut personnaliser les statistiques affichées avec `statistic`.

```
library(gtsummary)
trial |>
  tbl_summary(
    include = c(stage, trt),
    by = grade,
    statistic = ~ "{p}% ({n}/{N})",
    percent = "row"
  ) |>
  add_overall(last = TRUE)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.2: un tableau croisé avec des pourcentages en ligne

Caractéristique	**I**, N = 68	**II**, N = 68	**III**, N = 64	**Total**, N = 200
T Stage				
T1	32% (17/53)	43% (23/53)	25% (13/53)	100% (53/53)
T2	33% (18/54)	31% (17/54)	35% (19/54)	100% (54/54)
T3	42% (18/43)	26% (11/43)	33% (14/43)	100% (43/43)
T4	30% (15/50)	34% (17/50)	36% (18/50)	100% (50/50)
Chemotherapy Treatment				
Drug A	36% (35/98)	33% (32/98)	32% (31/98)	100% (98/98)
Drug B	32% (33/102)	35% (36/102)	32% (33/102)	100% (102/102)

> **!** Important
>
> Choisissez bien votre type de pourcentages (en lignes ou en colonnes). Si d'un point de vue purement statistique, ils permettent tous deux de décrire la relation entre les deux variables, ils ne correspondent au même *story telling*. Tout dépend donc du message que vous souhaitez faire passer, de l'histoire que vous souhaitez raconter.

`gtsummary::tbl_summary()` est bien adaptée dans le cadre d'une analyse de facteurs afin de représenter un *outcome* donné avec `by` et une liste de facteurs avec `include`.

Lorsque l'on ne croise que deux variables et que l'on souhaite un affichage un peu plus traditionnel d'un tableau croisé, on peut utiliser `gtsummary::tbl_cross()` à laquelle on transmettra une et une seule variable à `row` et une et une seule variable à `col`. Pour afficher des pourcentages, il faudra indiquer le type de pourcentages voulus avec `percent`.

```
trial |>
  tbl_cross(
    row = stage,
    col = grade,
    percent = "row"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.3: un tableau croisé avec tbl_cross()

	I	II	III	Total
T Stage				
T1	17 (32%)	23 (43%)	13 (25%)	53 (100%)
T2	18 (33%)	17 (31%)	19 (35%)	54 (100%)
T3	18 (42%)	11 (26%)	14 (33%)	43 (100%)
T4	15 (30%)	17 (34%)	18 (36%)	50 (100%)
Total	68 (34%)	68 (34%)	64 (32%)	200 (100%)

19.1.2 Représentations graphiques

La représentation graphique la plus commune pour le croisement de deux variables catégorielles est le diagramme en barres, que l'on réalise avec la géométrie `ggplot2::geom_bar()` et en utilisant les esthétiques x et *fill* pour représenter les deux variables.

```
library(ggplot2)
ggplot(trial) +
  aes(x = stage, fill = grade) +
  geom_bar() +
  labs(x = "T Stage", fill = "Grade", y = "Effectifs")
```

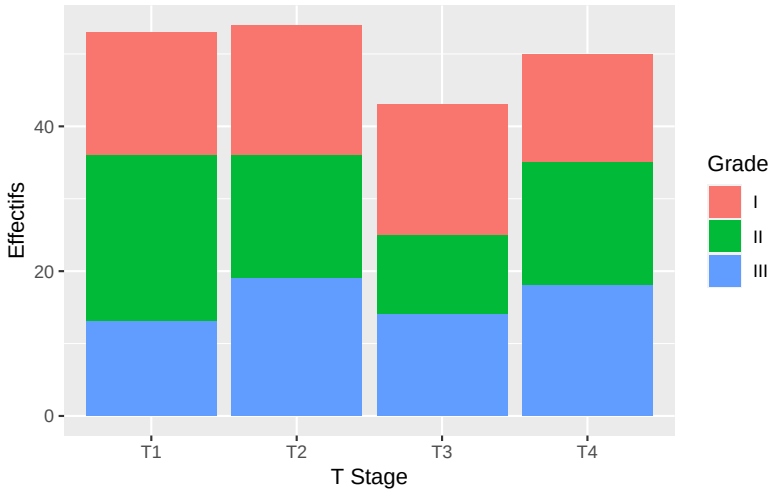


Figure 19.1: un graphique en barres croisant deux variables

On peut modifier la position des barres avec le paramètre `position`.

```
library(ggplot2)
ggplot(trial) +
  aes(x = stage, fill = grade) +
  geom_bar(position = "dodge") +
  labs(x = "T Stage", fill = "Grade", y = "Effectifs")
```

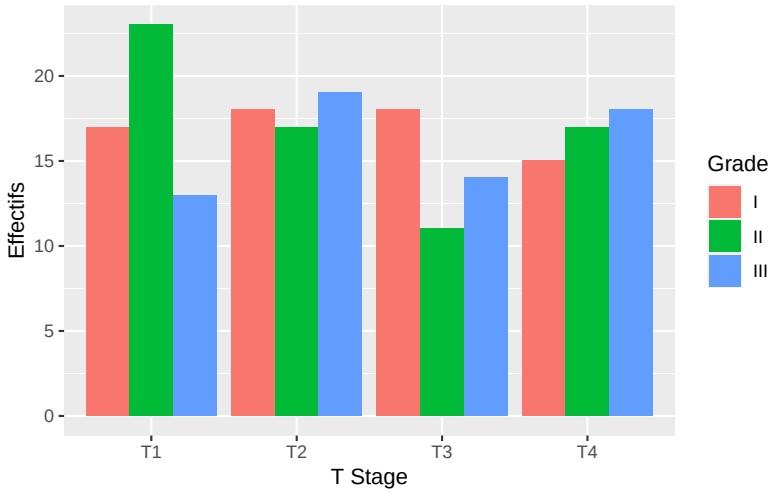


Figure 19.2: un graphique avec des barres côte à côte

Pour des barres cumulées, on aura recours à `position = "fill"`. Pour que les étiquettes de l'axe des `y` soient représentées sous forme de pourcentages (i.e. `25%` au lieu de `0.25`), on aura recours à la fonction `scales::percent()` qui sera transmise à `ggplot2::scale_y_continuous()`.

```
library(ggplot2)
ggplot(trial) +
  aes(x = stage, fill = grade) +
  geom_bar(position = "fill") +
  labs(x = "T Stage", fill = "Grade", y = "Proportion") +
  scale_y_continuous(labels = scales::percent)
```

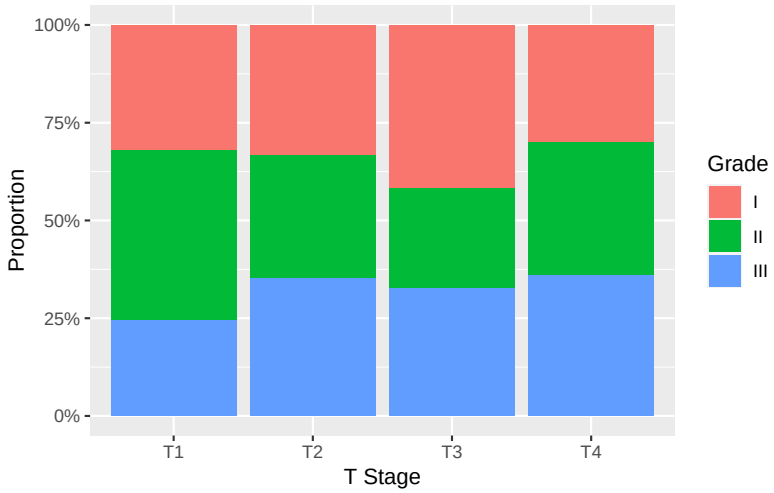
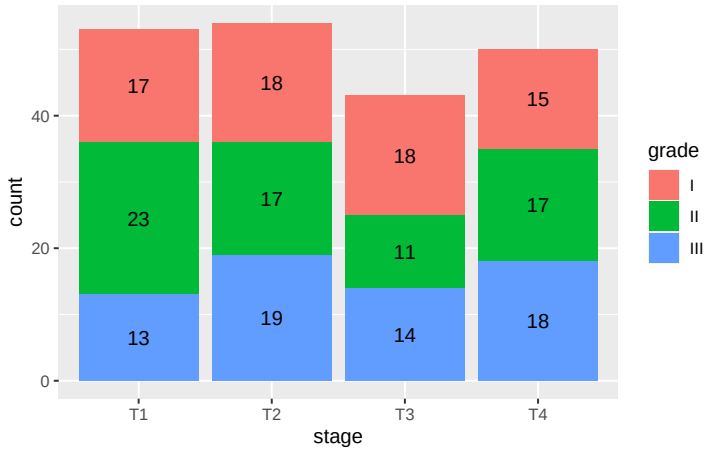


Figure 19.3: un graphique en barres cumulées

Ajouter des étiquettes sur un diagramme en barres

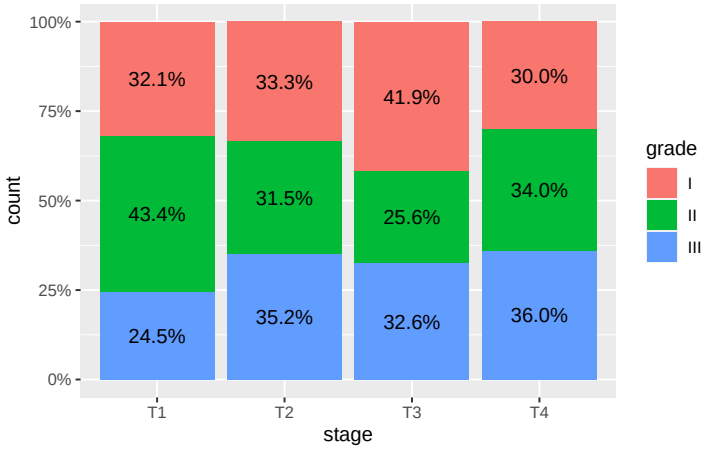
Il est facile d'ajouter des étiquettes en ayant recours à `ggplot2::geom_text()`, à condition de lui passer les bons paramètres.
Tout d'abord, il faudra préciser `stat = "count"` pour indiquer que l'on souhaite utiliser la statistique `ggplot2::stat_count()` qui est celle utilisé par défaut par `ggplot2::geom_bar()`. C'est elle qui permets de compter le nombre d'observations.
Il faut ensuite utiliser l'esthétique *label* pour indiquer ce que l'on souhaite afficher comme étiquettes. La fonction `after_stat(count)` permet d'accéder à la variable *count* calculée par `ggplot2::stat_count()`.
Enfin, il faut indiquer la position verticale avec `ggplot2::position_stack()`. En précisant un ajustement de vertical de `0.5`, on indique que l'on souhaite positionner l'étiquette au milieu.

```r
ggplot(trial) +
  aes(
    x = stage, fill = grade,
    label = after_stat(count)
  ) +
  geom_bar() +
  geom_text(
    stat = "count",
    position = position_stack(.5)
  )
```



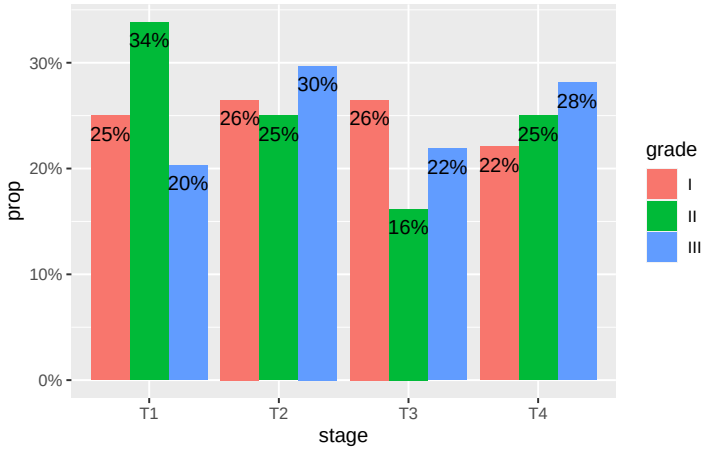
Pour un graphique en barres cumulées, on peut utiliser de manière similaire `ggplot2::position_fill()`. On ne peut afficher directement les proportions avec `ggplot2::stat_count()`. Cependant, nous pouvons avoir recours à `ggstats::stat_prop()`, déjà évoquée dans le chapitre sur la statistique univariée (cf. Section **??**) et dont le dénominateur doit être précisé via l'esthétique *by*.

```r
library(ggstats)
ggplot(trial) +
  aes(
    x = stage,
    fill = grade,
    by = stage,
    label = scales::percent(after_stat(prop), accuracy = .1)
  ) +
  geom_bar(position = "fill") +
  geom_text(
    stat = "prop",
    position = position_fill(.5)
  ) +
  scale_y_continuous(labels = scales::percent)
```



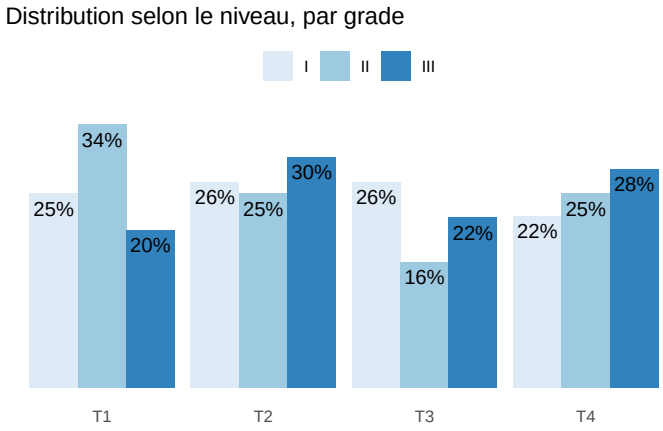
On peut aussi comparer facilement deux distributions, ici la proportion de chaque niveau de qualification au sein chaque sexe.

```r
p <- ggplot(trial) +
  aes(
    x = stage,
    y = after_stat(prop),
    fill = grade,
    by = grade,
    label = scales::percent(after_stat(prop), accuracy = 1)
  ) +
  geom_bar(
    stat = "prop",
    position = position_dodge(.9)
  ) +
  geom_text(
    aes(y = after_stat(prop) - 0.01),
    stat = "prop",
    position = position_dodge(.9),
    vjust = "top"
  ) +
  scale_y_continuous(labels = scales::percent)
p
```



Il est possible d'alléger le graphique en retirant des éléments superflus.

```r
p +
  theme_light() +
  xlab("") +
  ylab("") +
  labs(fill = "") +
  ggtitle("Distribution selon le niveau, par grade") +
  theme(
    panel.grid = element_blank(),
    panel.border = element_blank(),
    axis.text.y = element_blank(),
    axis.ticks = element_blank(),
    legend.position = "top"
  ) +
  scale_fill_brewer()
```



Pour visualiser chaque étape du code, vous pouvez consulter le diaporama suivant : https://larmarange.github.io/guide-R/analyses/ressources/flipbook-geom_bar-dodge.html

19.1.3 Calcul manuel

Les deux fonctions de base permettant le calcul d'un tri à plat sont `table()` et `xtabs()` (cf. Section **??**). Ces mêmes fonctions permettent le calcul du tri croisé de deux variables (ou plus).

Pour `table()`, on passera les deux vecteurs à croisés, tandis que pour `xtabs()` on décrira le tableau attendu à l'aide d'une formule.

```
table(trial$stage, trial$grade)
```

```
    I II III
T1 17 23  13
T2 18 17  19
T3 18 11  14
T4 15 17  18
```

```
tab <- xtabs(~ stage + grade, data = trial)
tab
```

```
     grade
stage  I II III
   T1 17 23  13
   T2 18 17  19
   T3 18 11  14
   T4 15 17  18
```

Le tableau obtenu est basique et ne contient que les effectifs. La fonction `addmargins()` permet d'ajouter les totaux par ligne et par colonne.

```
tab |> addmargins()
```

```
     grade
stage   I  II III Sum
  T1   17  23  13  53
  T2   18  17  19  54
  T3   18  11  14  43
  T4   15  17  18  50
  Sum  68  68  64 200
```

Pour le calcul des pourcentages, le plus simple est d'avoir recours au package {questionr} qui fournit les fonctions questionr::cprop(), questionr::rprop() et questionr::prop() qui permettent de calculer, respectivement, les pourcentages en colonne, en ligne et totaux.

```
questionr::cprop(tab)
```

```
       grade
stage   I     II    III   Ensemble
  T1     25.0  33.8  20.3  26.5
  T2     26.5  25.0  29.7  27.0
  T3     26.5  16.2  21.9  21.5
  T4     22.1  25.0  28.1  25.0
  Total 100.0 100.0 100.0 100.0
```

```
questionr::rprop(tab)
```

```
          grade
stage      I     II    III   Total
  T1        32.1  43.4  24.5 100.0
  T2        33.3  31.5  35.2 100.0
  T3        41.9  25.6  32.6 100.0
  T4        30.0  34.0  36.0 100.0
  Ensemble  34.0  34.0  32.0 100.0
```

```
questionr::prop(tab)
```

```
       grade
stage   I     II    III   Total
  T1     8.5  11.5   6.5  26.5
  T2     9.0   8.5   9.5  27.0
  T3     9.0   5.5   7.0  21.5
  T4     7.5   8.5   9.0  25.0
  Total 34.0  34.0  32.0 100.0
```

19.1.4 Test du Chi² et dérivés

Dans le cadre d'un tableau croisé, on peut tester l'existence d'un lien entre les modalités de deux variables, avec le très classique test du Chi² (parfois écrit ^{2} ou Chi²). Pour une présentation plus détaillée du test, on pourra se référer à ce cours de Julien Barnier.

Le test du Chi² peut se calculer très facilement avec la fonction `chisq.test()` appliquée au tableau obtenu avec `table()` ou `xtabs()`.

```
tab <- xtabs(~ stage + grade, data = trial)
tab
```

```
     grade
stage  I II III
   T1 17 23  13
   T2 18 17  19
   T3 18 11  14
   T4 15 17  18
```

```
chisq.test(tab)
```

```

    Pearson's Chi-squared test

data:  tab
X-squared = 4.8049, df = 6, p-value = 0.5691
```

Si l'on est adepte de `{gtsummary}`, il suffit d'appliquer `gtsummary::add_p()` au tableau produit avec `gtsummary::tbl_summary()`.

```
trial |>
  tbl_summary(
    include = stage,
    by = grade
  ) |>
  add_p()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.4: un tableau croisé avec test du khi²

Caractéristique	**I**, N = 68	**II**, N = 68	**III**, N = 64	**p-valeur**
T Stage				0,6
T1	17 (25%)	23 (34%)	13 (20%)	
T2	18 (26%)	17 (25%)	19 (30%)	
T3	18 (26%)	11 (16%)	14 (22%)	
T4	15 (22%)	17 (25%)	18 (28%)	

Dans notre exemple, les deux variables *stage* et *grade* ne sont clairement pas corrélées.

Un test alternatif est le test exact de Fisher. Il s'obtient aisément avec `fisher.test()` ou bien en le spécifiant via l'argument `test` de `gtsummary::add_p()`.

```
tab <- xtabs(~ stage + grade, data = trial)
fisher.test(tab)
```

```
    Fisher's Exact Test for Count Data

data:  tab
p-value = 0.5801
alternative hypothesis: two.sided
```

```
trial |>
  tbl_summary(
    include = stage,
    by = grade
  ) |>
  add_p(test = all_categorical() ~ "fisher.test")
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.5: un tableau croisé avec test exact de Fisher

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
T Stage				0,6
T1	17 (25%)	23 (34%)	13 (20%)	
T2	18 (26%)	17 (25%)	19 (30%)	
T3	18 (26%)	11 (16%)	14 (22%)	
T4	15 (22%)	17 (25%)	18 (28%)	

> **Note**
>
> Formellement, le test de Fisher suppose que les marges du tableau (totaux lignes et colonnes) sont fixées, puisqu'il repose sur une loi hypergéométrique, et donc celui-ci se prête plus au cas des situations expérimentales (plans d'expérience, essais cliniques) qu'au cas des données tirées d'études observationnelles.
> En pratique, le test du Chi² étant assez robuste quant aux déviations par rapport aux hypothèses d'applications du test (effectifs théoriques supérieurs ou égaux à 5), le test de Fisher présente en général peu d'intérêt dans le cas de l'analyse des tableaux de contingence.

19.1.5 Comparaison de deux proportions

Pour comparer deux proportions, la fonction de base est `prop.test()` à laquelle on passera un tableau à 2×2 dimensions.

```
tab <- xtabs(~ I(stage == "T1") + trt, data = trial)
tab |> questionr::cprop()
```

```
                trt
I(stage == "T1") Drug A Drug B Ensemble
           FALSE  71.4   75.5   73.5
           TRUE   28.6   24.5   26.5
           Total 100.0  100.0  100.0
```

```
tab |> prop.test()
```

```
    2-sample test for equality of proportions with continuity correction

data:  tab
X-squared = 0.24047, df = 1, p-value = 0.6239
alternative hypothesis: two.sided
95 percent confidence interval:
 -0.2217278  0.1175050
sample estimates:
   prop 1    prop 2
0.4761905 0.5283019
```

Il est également envisageable d'avoir recours à un test exact de Fisher. Dans le cas d'un tableau à 2×2 dimensions, le test exact de Fisher ne teste pas si les deux proportions sont différents, mais plutôt si leur *odds ratio* (qui est d'ailleurs renvoyé par la fonction) est différent de 1.

```
fisher.test(tab)
```

```
    Fisher's Exact Test for Count Data

data:  tab
p-value = 0.5263
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.4115109 1.5973635
sample estimates:
odds ratio
 0.8125409
```

Mais le plus simple reste encore d'avoir recours à `{gtsummary}` et à sa fonction `gtsummary::add_difference()` que l'on peut appliquer à un tableau où le paramètre `by` n'a que deux modalités. Pour la différence de proportions, il faut que les variables transmises à `include` soit dichotomiques.

```
trial |>
  tbl_summary(
    by = trt,
    include = response
  ) |>
  add_difference()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.6: différence entre deux proportions

Caractéristique	Drug A, N = 98	Drug B, N = 102	Difference	95% IC	p-valeur
Tumor Response	28 (29%)	33 (34%)	-4,2%	-18% – 9,9%	0,6
Manquant	3	4			

Attention : si l'on passe une variable catégorielle à trois modalités ou plus, c'est la différence des moyennes standardisées (globale pour la variable) qui sera calculée et non la différence des proportions dans chaque groupe.

```
trial |>
  tbl_summary(
    by = trt,
    include = grade
  ) |>
  add_difference()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
```

```
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.7: différence moyenne standardisée

Caractéristiques	**Drug A**, N = 98	**Drug B**, N = 102	**Difference**	**95% IC**
Grade			0,07	-0,20 – 0,35
I	35 (36%)	33 (32%)		
II	32 (33%)	36 (35%)		
III	31 (32%)	33 (32%)		

Pour calculer la différence des proportions pour chaque modalité de *grade*, il est nécessaire de transformer, en amont, la variable catégorielle *grade* en trois variables dichotomiques (de type oui/non, une par modalité), ce qui peut se faire facilement avec la fonction `fastDummies::dummy_cols()` de l'extension `{fastDummies}`.

```
trial |>
  fastDummies::dummy_cols("grade") |>
  tbl_summary(
    by = trt,
    include = starts_with("grade_"),
    digits = ~ c(0, 1)
  ) |>
  add_difference()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.8: différence entre proportions avec création de variables dichotomiques

Caractéristique	Drug A, N = 98	Drug B, N = 102	Difference	95% IC	p-valeur
grade_I	35 (35,7%)	33 (32,4%)	3,4%	-11% – 17%	0,7
grade_II	32 (32,7%)	36 (35,3%)	-2,6%	-17% – 11%	0,8
grade_III	31 (31,6%)	33 (32,4%)	-0,72%	-14% – 13%	>0,9

19.2 Une variable continue selon une variable catégorielle

19.2.1 Tableau comparatif avec `gtsummary`

Dans le chapitre sur la statitstique univariée (cf. Section **??**), nous avons abordé comment afficher les statistiques descriptives d'une variable continue avec `gtsummary::tbl_summary()`. Pour comparer une variable continue selon plusieurs groupes définis par une variable catégorielle, il suffit d'utiliser le paramètre `by` :

```
trial |>
  tbl_summary(
    include = age,
    by = grade
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.9: âge médian et intervalle interquartile selon le grade

Caractéristique	**I**, N = 68	**II**, N = 68	**III**, N = 64
Age	47 (37 – 56)	48 (37 – 57)	47 (38 – 58)
Manquant	2	6	3

La fonction `gtsummary::add_overall()` permet d'ajouter une colonne total et `gtsummary::modify_spanning_header()` peut-être utilisé pour ajouter un en-tête de colonne.

```
trial |>
  tbl_summary(
    include = age,
    by = grade
  ) |>
  add_overall(last = TRUE) |>
  modify_spanning_header(
    all_stat_cols(stat_0 = FALSE) ~ "**Grade**"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.10: âge médian et intervalle interquartile selon le grade

Caractéristique	**I**, N = 68	**II**, N = 68	**III**, N = 64	**Total**, N = 200
Age	47 (37 – 56)	48 (37 – 57)	47 (38 – 58)	47 (38 – 57)
Manquant	2	6	3	11

Comme pour un tri à plat, on peut personnaliser les statistiques à afficher avec `statistic`.

```
trial |>
  tbl_summary(
    include = age,
    by = grade,
    statistic = all_continuous() ~ "{mean} ({sd})",
    digits = all_continuous() ~ c(1, 1)
  ) |>
  add_overall(last = TRUE)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.11: âge moyen et écart-type selon le grade

Caractéristique	**I**, N = 68	**II**, N = 68	**III**, N = 64	**Total**, N = 200
Age	46,2 (15,2)	47,5 (13,7)	48,1 (14,1)	47,2 (14,3)
Manquant	2	6	3	11

19.2.2 Représentations graphiques

La moyenne ou la médiane sont des indicateurs centraux et ne suffisent pas à rendre compte des différences de distribution d'une variable continue entre plusieurs sous-groupes.

Une représentation usuelle pour comparer deux distributions consiste à avoir recours à des boîtes à moustaches que l'on obtient avec `ggplot2::geom_boxplot()`.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_boxplot(fill = "lightblue") +
  theme_light()
```

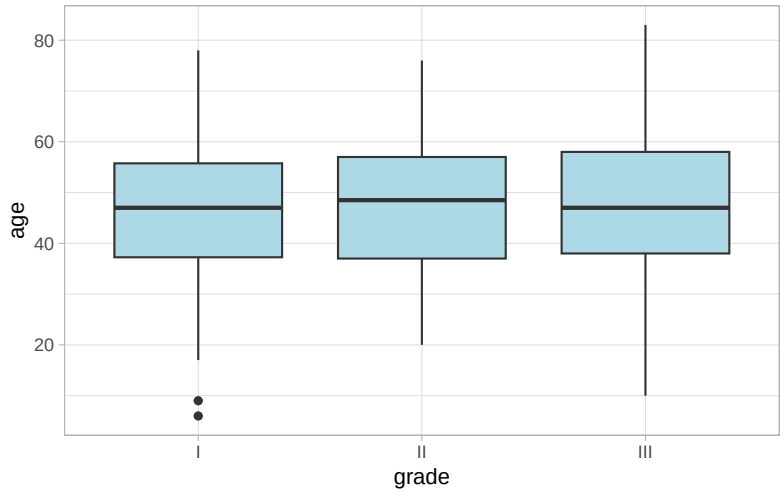


Figure 19.4: boîtes à moustache

> **Astuce**
>
> Le trait central représente la médiane, le rectangle est délimité par le premier et le troisième quartiles (i.e. le 25e et le 75e percentiles). Les traits verticaux vont jusqu'aux extrêmes (minimum et maximum) ou jusqu'à 1,5 fois l'intervalle interquartile. Si des points sont situés à plus d'1,5 fois l'intervalle interquartile au-dessus du 3e quartile ou en-dessous du 1er quartile, ils sont considérés comme des valeurs atypiques et représentés par un point. Dans l'exemple précédent, c'est le cas des deux plus petites valeurs observées pour le grade I.

Alternativement, on peut utiliser un graphique en violons qui représentent des courbes de densité dessinées en mirroir.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_violin(fill = "lightblue") +
  theme_light()
```

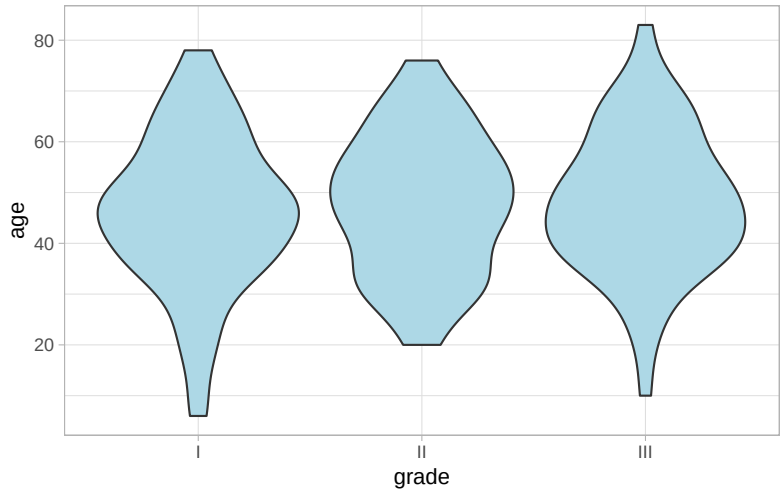


Figure 19.5: graphique en violons

Il est toujours possible de représenter les observations invidu-elles sous la forme d'un nuage de points. Le paramètre `alpha` permet de rendre les points transparents afin de mieux visualiser les supperpositions de points.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_point(alpha = .25, colour = "blue") +
  theme_light()
```

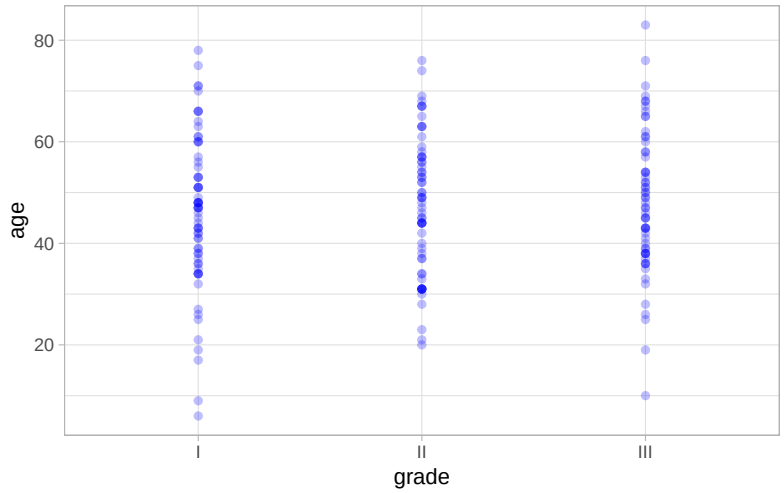


Figure 19.6: un nuage de points avec une variable continue et une variable catégorielle

Comme la variable *grade* est catégorielle, tous les points d'une meme modalité sont représentées sur une même ligne. La représentation peut être améliorée en ajoutant un décalage aléatoire sur l'axe horizontal. Cela s'obtient avec `ggplot2::position_jitter()` en précisant `height = 0` pour ne pas ajouter de décalage vertical et `width = .2` pour décaler horizontalement les points entre -20% et +20%.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_point(
    alpha = .25,
    colour = "blue",
    position = position_jitter(height = 0, width = .2)
  ) +
  theme_light()
```

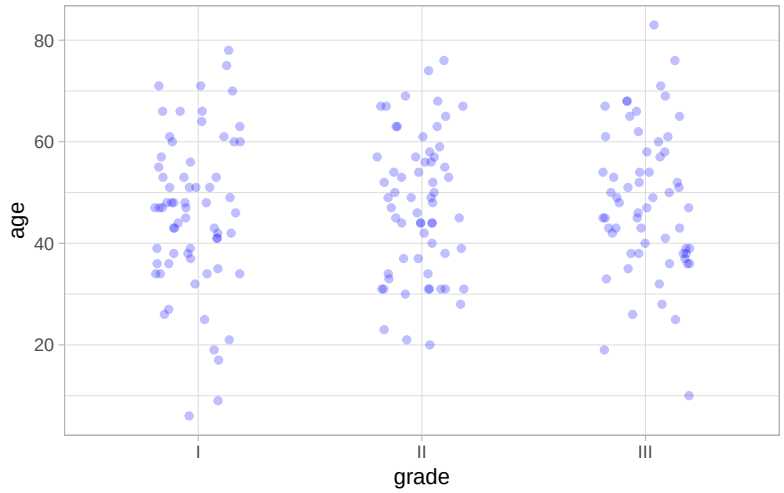


Figure 19.7: un nuage de points avec une variable continue et une variable catégorielle et avec un décalage horizontal aléatoire

La statistique `ggstats::stat_weighted_mean()` de `{ggstats}` permets de calculer à la volée la moyenne du nuage de points.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_point(stat = "weighted_mean", colour = "blue") +
  theme_light()
```

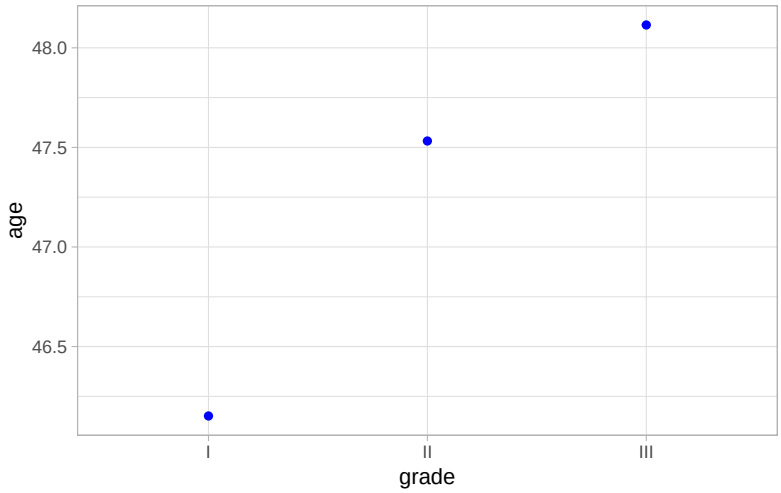


Figure 19.8: âge moyen selon le grade

Cela peut être utile pour effectuer des comparaisons multiples.

```
ggplot(trial) +
  aes(x = grade, y = age, colour = stage, group = stage) +
  geom_line(stat = "weighted_mean") +
  geom_point(stat = "weighted_mean") +
  facet_grid(cols = vars(trt)) +
  theme_light()
```

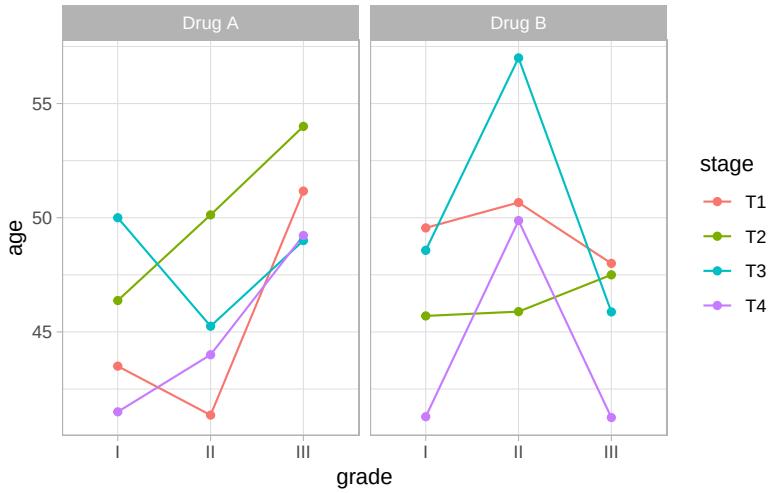


Figure 19.9: âge moyen selon le grade, par traitement et état d'avancement de la maladie

19.2.3 Calcul manuel

Le plus simple pour calculer des indicateurs par sous-groupe est d'avoir recours à `dplyr::summarise()` avec `dplyr::group_by()`.

```
library(dplyr)
trial |>
  group_by(grade) |>
  summarise(
    age_moy = mean(age, na.rm = TRUE),
    age_med = median(age, na.rm = TRUE)
  )
```

```
# A tibble: 3 x 3
  grade age_moy age_med
  <fct>   <dbl>   <dbl>
1 I        46.2    47
2 II       47.5    48.5
3 III      48.1    47
```

En base **R**, on peut avoir recours à `tapply()`. On lui indique d'abord le vecteur sur lequel on souhaite réaliser le calcul, puis un facteur qui indiquera les sous-groupes, puis une fonction qui sera appliquée à chaque sous-groupe et enfin, optionnellement, des arguments additionnels qui seront transmis à cette fonction.

```
tapply(trial$age, trial$grade, mean, na.rm = TRUE)
```

```
       I       II      III
46.15152 47.53226 48.11475
```

19.2.4 Tests de comparaison

Pour comparer des moyennes ou des médianes, le plus facile est encore d'avoir recours à `{gtsummary}` et sa fonction `gtsummary::add_p()`.

```
trial |>
  tbl_summary(
    include = age,
    by = grade
  ) |>
  add_p()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.12: test de comparaison sur la somme des rangs

Caractéristique	**I**, N = 68	**II**, N = 68	**III**, N = 64	**p-valeur**
Age	47 (37 – 56)	48 (37 – 57)	47 (38 – 58)	0,8
Manquant	2	6	3	

Par défaut, pour les **variables continues**, un test de Kruskal-Wallis calculé avec la fonction `stats::kruskal.test()` est utilisé lorsqu'il y a trois groupes ou plus, et un test de Wilcoxon-Mann-Whitney calculé avec `stats::wilcox.test()` (test de comparaison des rangs) lorsqu'il n'y a que deux groupes. Au sens strict, il ne s'agit pas de tests de comparaison des médianes mais de tests sur la somme des rangs[26]. En pratique, ces tests sont appropriés lorsque l'on présente les médianes et les intervalles interquartiles.

Si l'on affiche des moyennes, il serait plus juste d'utiliser un test *t de Student* (test de comparaison des moyennes) calculé avec `stats::t.test()`, valable seulement si l'on compare deux moyennes. Pour tester si trois moyennes ou plus sont égales, on aura plutôt recours à `stats::oneway.test()`.

On peut indiquer à `gtsummary::add_p()` le test à utiliser avec le paramètre `test`.

[26] Si l'on a besoin spécifiquement d'un test de comparaison des médianes, il existe le **test de Brown-Mood** disponible dans le package `{coin}` avec la fonction `coin::median_test()`. Attention, il ne faut pas confondre ce test avec le **test de dispersion de Mood** implémenté dans la fonction `stats::mood.test()`.

```
trial |>
  tbl_summary(
    include = age,
    by = grade,
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
  add_p(
    test = all_continuous() ~ "oneway.test"
  )
```

```
Multiple parameters; naming those columns num.df, den.df
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.13: test de comparaison des moyennes

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
Age	46 (15)	48 (14)	48 (14)	0,7
Manquant	2	6	3	

! Précision statistique

Classiquement, le test t de Student présuppose l'égalité des variances entre les deux sous-groupes, ce qui permet de former une estimation commune de la variance des deux échantillons (on parle de pooled variance), qui revient à une moyenne pondérée des variances estimées à partir des deux échantillons. Pour tester l'égalité des variances de deux échantillons, on peut utiliser `stats::var.test()`.
Dans le cas où l'on souhaite relaxer cette hypothèse d'égalité des variances, le test de Welch ou la correction de Satterthwaite reposent sur l'idée que l'on utilise les deux estimations de variance séparément, suivie d'une approximation des degrés de liberté pour la somme de ces deux variances.
Par défaut, la fonction `stats::t.test()` réalise un test de Welch. Pour un test classique de Student, il faut lui préciser `var.equal = TRUE`.
De manière similaire, `stats::oneway.test()` ne présuppose pas, par défaut, l'égalité des variances et généralise donc le test de Welch au cas à trois modalités ou plus. Cependant, on peut là encore indiquer `var.equal = TRUE`, auquel cas une analyse de variance (ANOVA) classique sera réalisée, que l'on peut aussi obtenir avec `stats::aov()`.
Il est possible d'indiquer à `gtsummary::add_p()` des arguments additionnels à passer à la fonction utilisée pour réaliser le test :

```
trial |>
  tbl_summary(
    include = age,
    by = trt,
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
  add_p(
    test = all_continuous() ~ "t.test",
    test.args = all_continuous() ~ list(var.equal = TRUE)
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Caractéristique	Drug A, N = 98	Drug B, N = 102	p-valeur
Age	47 (15)	47 (14)	0,8
Manquant	7	4	

19.2.5 Différence de deux moyennes

La fonctions `gtsummary::add_difference()` permet, pour une variable continue et si la variable catégorielle spécifiée via `by` n'a que deux modalités, de calculer la différence des deux moyennes, l'intervalle de confiance de cette différence et test si cette différence est significativement différente de 0 avec `stats::t.test()`.

```
trial |>
  tbl_summary(
    include = age,
    by = trt,
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
  add_difference()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 19.15: différence de deux moyennes

Caractéristique	Drug A, N = 98	Drug B, N = 102	Difference	95% IC	p-valeur
Age	47 (15)	47 (14)	-0,44	-4,6 – 3,7	0,8
Manquant	7	4			

19.3 Deux variables continues

19.3.1 Représentations graphiques

La comparaison de deux variables continues se fait en premier lieu graphique, en représentant, via un nuage de points, l'ensemble des couples de valeurs. Notez ici l'application d'un niveau de transparence (`alpha`) afin de faciliter la lecture des points superposés.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  theme_light()
```

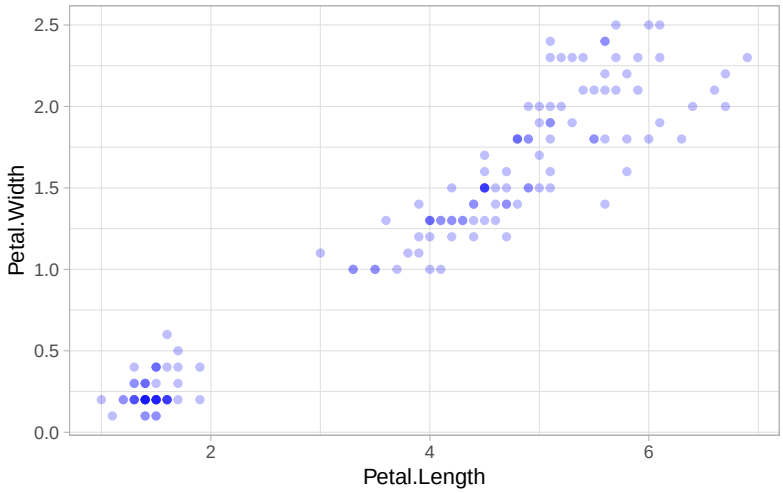


Figure 19.10: nuage de points

La géométrie `ggplot2::geom_smooth()` permets d'ajouter une courbe de tendance au graphique, avec son intervalle de confiance. Par défaut, il s'agit d'une régression polynomiale locale obtenue avec `stats::loess()`.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth() +
  geom_point(colour = "blue", alpha = .25) +
  theme_light()
```

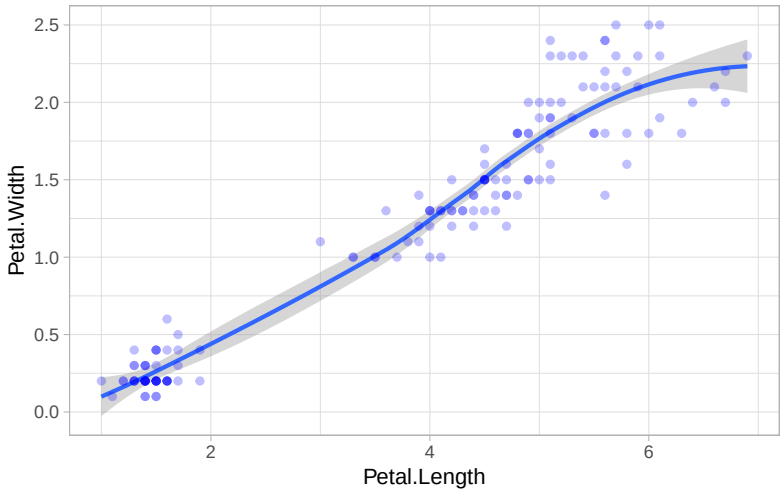


Figure 19.11: nuage de points avec une courbe de tendance

Pour afficher plutôt la droite de régression linéaire entre les deux variables, on précisera `method = "lm"`.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth(method = "lm") +
  geom_point(colour = "blue", alpha = .25) +
  theme_light()
```

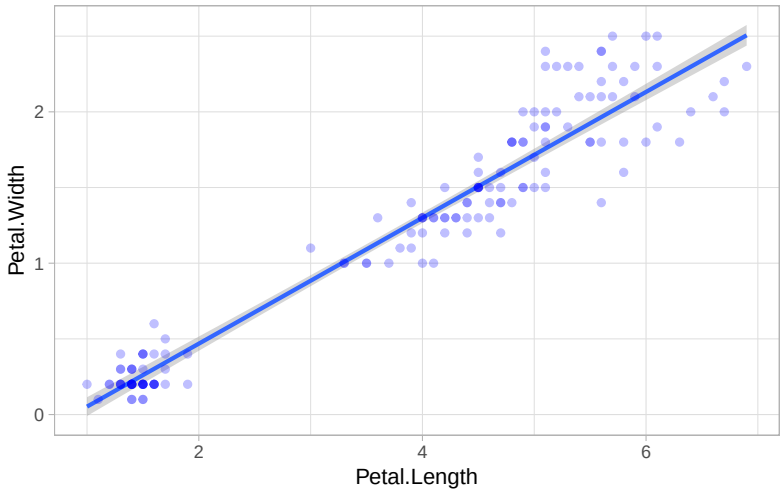


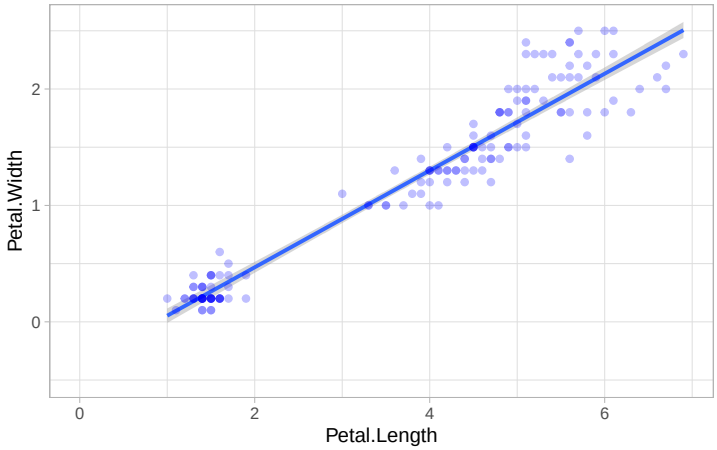
Figure 19.12: nuage de points avec droite de régression linéaire

Astuce pour afficher l'intercept

Supposons que nous souhaitions montrer l'endroit où la droite de régression coupe l'axe des ordonnées (soit le point sur l'axe y pour $x = 0$).
Nous pouvons étendre la surface du graphique avec `ggplot2::expand_limits()`. Cependant, cela n'étend pas pour autant la droite de régression.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth(method = "lm") +
  geom_point(colour = "blue", alpha = .25) +
  theme_light() +
  expand_limits(x = 0, y = -0.5)
```

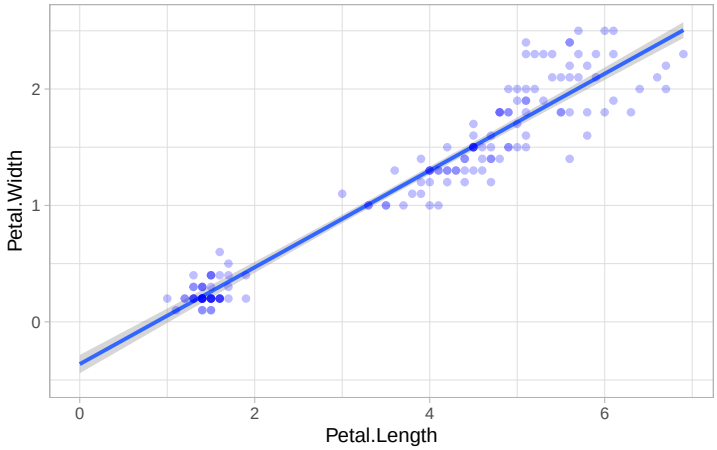
```
`geom_smooth()` using formula = 'y ~ x'
```



Une solution simple consiste à utiliser l'option `fullrange = TRUE` dans `ggplot2::geom_smooth()` pour étendre la droite de régression à l'ensemble du graphique.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth(method = "lm", fullrange = TRUE) +
  geom_point(colour = "blue", alpha = .25) +
  theme_light() +
  expand_limits(x = 0, y = -0.5)
```

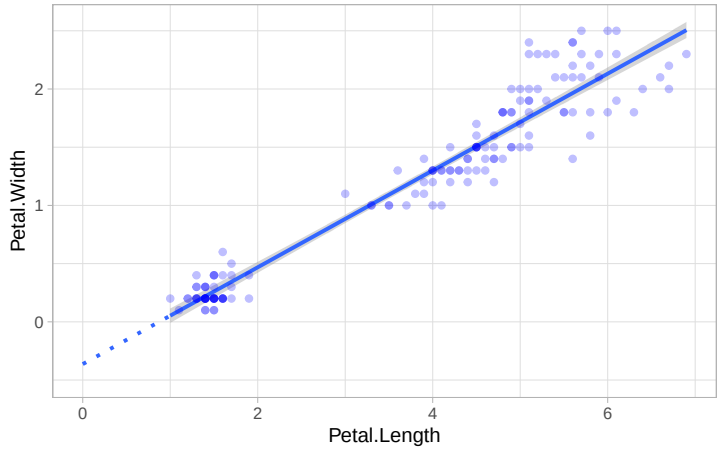
```
`geom_smooth()` using formula = 'y ~ x'
```



On peut contrôler plus finement la partie de droite à afficher avec l'argument `xseq` (liste des valeurs pour lesquelles on prédit et affiche le lissage). On peut coupler deux appels à `ggplot2::geom_smooth()` pour afficher l'extension de la droite vers la gauche en pointillés. L'option `se = FALSE` permet de ne pas calculer d'intervalles de confiance pour ce second appel.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth(
    method = "lm",
    xseq = seq(0, 1, by = .1),
    linetype = "dotted",
    se = FALSE
  ) +
  geom_smooth(method = "lm") +
  geom_point(colour = "blue", alpha = .25) +
  theme_light() +
  expand_limits(x = 0, y = -0.5)
```

```
`geom_smooth()` using formula = 'y ~ x'
`geom_smooth()` using formula = 'y ~ x'
```



La géométrie `ggplot2::geom_rug()` permet d'afficher une représentation synthétique de la densité de chaque variable sur les deux axes.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth(method = "lm") +
  geom_point(colour = "blue", alpha = .25) +
  geom_rug() +
  theme_light()
```

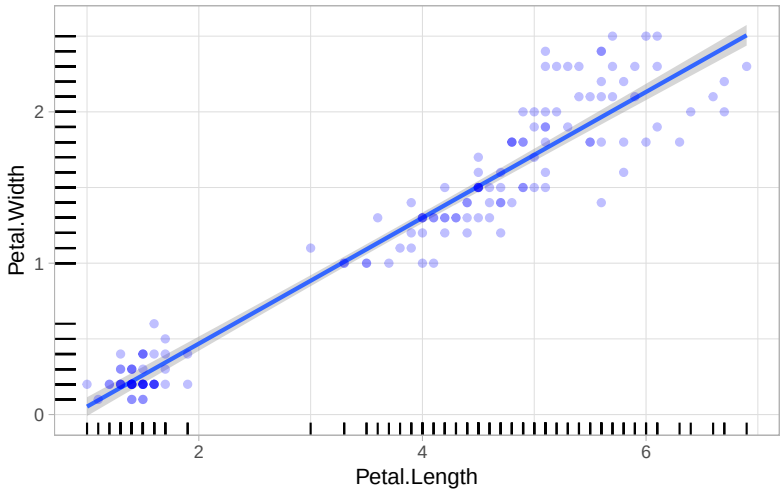


Figure 19.13: nuage de points avec représentation synthétique des densités marginales

19.3.2 Tester la relation entre les deux variables

Si l'on a besoin de calculer le coefficient de corrélation de Pearson entre deux variables, on aura recours à `stats::cor()`.

```
cor(iris$Petal.Length, iris$Petal.Width)
```

```
[1] 0.9628654
```

Pour aller plus loin, on peut calculer une régression linéaire entre les deux variables avec `stats::lm()`.

```
m <- lm(Petal.Length ~ Petal.Width, data = iris)
summary(m)
```

```
Call:
lm(formula = Petal.Length ~ Petal.Width, data = iris)

Residuals:
```

```
     Min       1Q   Median       3Q      Max
-1.33542 -0.30347 -0.02955  0.25776  1.39453

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.08356    0.07297   14.85   <2e-16 ***
Petal.Width  2.22994    0.05140   43.39   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4782 on 148 degrees of freedom
Multiple R-squared:  0.9271,    Adjusted R-squared:  0.9266
F-statistic:  1882 on 1 and 148 DF,  p-value: < 2.2e-16
```

Les résultats montrent une corrélation positive et significative entre les deux variables.

Pour une présentation propre des résultats de la régression linéaire, on utilisera `gtsummary::tbl_regression()`. La fonction `gtsummary::add_glance_source_note()` permet d'ajouter différentes statistiques en notes du tableau de résultats.

```
m |>
  tbl_regression() |>
  add_glance_source_note()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Caractéristique	**Beta**	**95% IC**	**p-valeur**
Petal.Width	2,2	2,1 – 2,3	<0,001

19.4 Matrice de corrélations

Le package `{GGally}` et sa fonction `GGally::ggpairs()` permettent de représenter facilement une matrice de corrélation

entre plusieurs variables, tant quantitatives que qualitatives.

```
library(GGally)
ggpairs(iris)
```

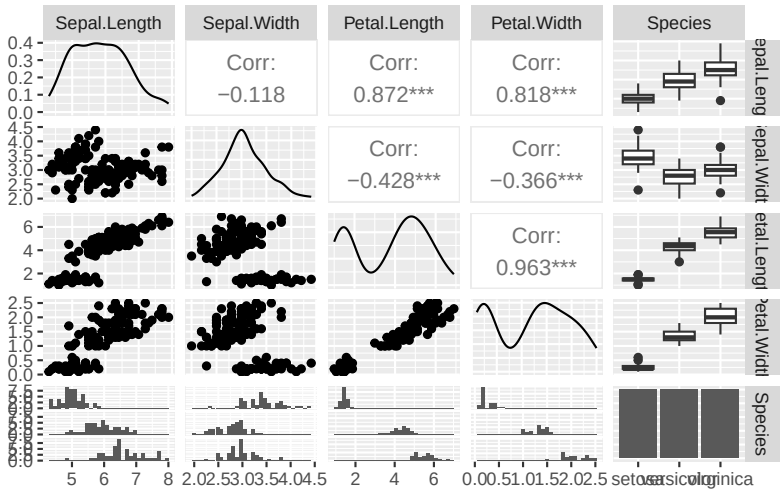


Figure 19.14: une matrice de corrélation avec ggpairs()

`GGally::ggpairs()` et sa petite sœur `GGally::ggduo()` offrent de nombreuses options de personnalisation qui sont détaillées sur le site dédié du package.

```
ggpairs(trial, mapping = aes(colour = trt))
```

Figure 19.15: un second example de matrice de corrélation

19.5 webin-R

La statistique univariée est présentée dans le webin-R #03 (*statistiques descriptives avec gtsummary et esquisse*) sur YouTube.

https://youtu.be/oEF_8GXyP5c

20 Échelles de Likert

Les échelles de Likert tirent leur nom du psychologue américain Rensis Likert qui les a développées. Elles sont le plus souvent utilisées pour des variables d'opinion. Elles sont codées sous forme de variable catégorielle et chaque item est codé selone une graduation comprenant en général cinq ou sept choix de réponse, par exemple : Tout à fait d'accord, D'accord, Ni en désaccord ni d'accord, Pas d'accord, Pas du tout d'accord.

Pour les échelles à nombre impair de choix, le niveau central permet d'exprimer une absence d'avis, ce qui rend inutile une modalité « Ne sait pas ». Les échelles à nombre pair de modalités voient l'omission de la modalité neutre et sont dites « à choix forcé ».

20.1 Exemple de données

Générons un jeu de données qui nous servira pour les différents exemples.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
library(labelled)
niveaux <- c(
  "Pas du tout d'accord",
  "Plutôt pas d'accord",
  "Ni d'accord, ni pas d'accord",
  "Plutôt d'accord",
  "Tout à fait d'accord"
)
set.seed(42)
df <-
  tibble(
    groupe = sample(c("A", "B"), 150, replace = TRUE),
    q1 = sample(niveaux, 150, replace = TRUE),
    q2 = sample(niveaux, 150, replace = TRUE, prob = 5:1),
    q3 = sample(niveaux, 150, replace = TRUE, prob = 1:5),
    q4 = sample(niveaux, 150, replace = TRUE, prob = 1:5),
    q5 = sample(c(niveaux, NA), 150, replace = TRUE),
    q6 = sample(niveaux, 150, replace = TRUE, prob = c(1, 0, 1, 1, 0))
  ) |>
  mutate(across(q1:q6, ~ factor(.x, levels = niveaux))) |>
  set_variable_labels(
    q1 = "Première question",
    q2 = "Seconde question",
    q3 = "Troisième question",
    q4 = "Quatrième question",
    q5 = "Cinquième question",
    q6 = "Sixième question"
  )
```

20.2 Tableau de fréquence

On peut tout à fait réaliser un tableau de fréquence classique avec `gtsummary::tbl_summary()`.

```
library(gtsummary)
df |>
  tbl_summary(include = q1:q6)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	**N = 150**
Première question	
Pas du tout d'accord	39 (26%)
Plutôt pas d'accord	32 (21%)
Ni d'accord, ni pas d'accord	25 (17%)
Plutôt d'accord	30 (20%)
Tout à fait d'accord	24 (16%)
Seconde question	
Pas du tout d'accord	56 (37%)
Plutôt pas d'accord	44 (29%)
Ni d'accord, ni pas d'accord	19 (13%)
Plutôt d'accord	26 (17%)
Tout à fait d'accord	5 (3.3%)
Troisième question	
Pas du tout d'accord	8 (5.3%)
Plutôt pas d'accord	17 (11%)
Ni d'accord, ni pas d'accord	29 (19%)
Plutôt d'accord	43 (29%)
Tout à fait d'accord	53 (35%)
Quatrième question	
Pas du tout d'accord	11 (7.3%)
Plutôt pas d'accord	19 (13%)
Ni d'accord, ni pas d'accord	31 (21%)
Plutôt d'accord	40 (27%)
Tout à fait d'accord	49 (33%)
Cinquième question	
Pas du tout d'accord	33 (26%)
Plutôt pas d'accord	25 (20%)
Ni d'accord, ni pas d'accord	28 (22%)
Plutôt d'accord	25 (20%)
Tout à fait d'accord	16 (13%)
Unknown	23
Sixième question	
Pas du tout d'accord	50 (33%)
Plutôt pas d'accord	0 (0%)

Characteristic	N = 150
Ni d'accord, ni pas d'accord	50 (33%)
Plutôt d'accord	50 (33%)
Tout à fait d'accord	0 (0%)

Cependant, cela produit un tableau inutilement long, d'autant plus que les variables *q1* à *q6* ont les mêmes modalités de réponse. Le package `{bstfun}` propose une fonction expérimentale `bstfun::tbl_likert()` offrant un affichage plus compact.

> **!** Important
>
> Ce package n'est pas disponible sur **CRAN** : il est donc nécessaire de l'installer depuis **GitHub** avec la commande suivante : `remotes::install_github("MSKCC-Epi-Bio/bstfun")`. Si vous êtes sous **Windows**, il vous faudra probablement installer au préalable **RTools** qui peut être téléchargé à l'adresse https://cran.r-project.org/bin/windows/Rtools/.

```
library(bstfun)
```

```
Attachement du package : 'bstfun'
```

```
L'objet suivant est masqué depuis 'package:gtsummary':
```

```
    trial
```

```
df |>
  tbl_likert(
    include = q1:q6
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	Pas du tout d'accord	Plutôt pas d'accord	Ni d'accord, ni pas d'accord	Plutôt d'accord	Tout à fait d'accord
Première question	39 (26%)	32 (21%)	25 (17%)	30 (20%)	24 (16%)
Seconde question	56 (37%)	44 (29%)	19 (13%)	26 (17%)	5 (3.3%)
Troisième question	8 (5.3%)	17 (11%)	29 (19%)	43 (29%)	53 (35%)
Quatrième question	11 (7.3%)	19 (13%)	31 (21%)	40 (27%)	49 (33%)
Cinquième question	33 (26%)	25 (20%)	28 (22%)	25 (20%)	16 (13%)
Sixième question	50 (33%)	0 (0%)	50 (33%)	50 (33%)	0 (0%)

On peut utiliser `add_n()` pour ajouter les effectifs totaux et `add_continuous_stat()` pour ajouter une statistique continue en traitant la variable comme un score (ici nous allons attribuer les valeurs -2, -1, 0, +1 et +2).

```
df |>
  tbl_likert(
    include = q1:q6,
    statistic = ~ "{p}%"
  ) |>
  add_n() |>
  add_continuous_stat(score_values = -2:2)
```

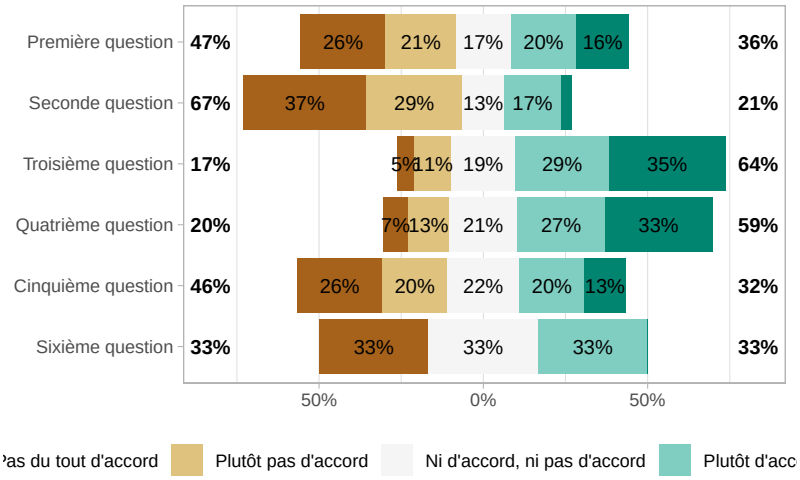
```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	N	Pas du tout d'accord	Plutôt pas d'accord	Ni d'accord, ni pas d'accord	Plutôt d'accord	Tout à fait d'accord	Mean
Première question	150	26%	21%	17%	20%	16%	-0.21
Seconde question	150	37%	29%	13%	17%	3.3%	-0.80
Troisième question	150	5.3%	11%	19%	29%	35%	0.77
Quatrième question	150	7.3%	13%	21%	27%	33%	0.65
Cinquième question	127	26%	20%	22%	20%	13%	-0.27
Sixième question	150	33%	0%	33%	33%	0%	-0.33

20.3 Représentations graphiques

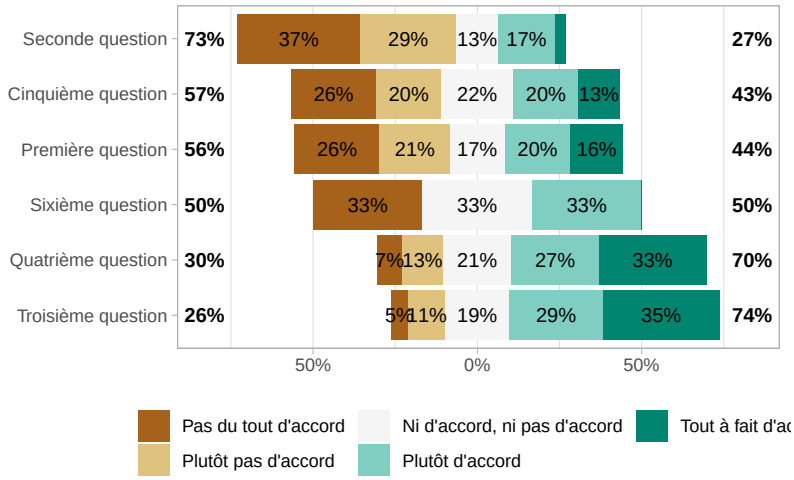
À partir de sa version 0.3.0, le package `{ggstats}` propose une fonction `ggstats::gglikert()` pour représenter des données de Likert sous forme d'un diagramme en barre centré sur la modalité centrale.

```
library(ggstats)
gglikert(df, include = q1:q6)
```



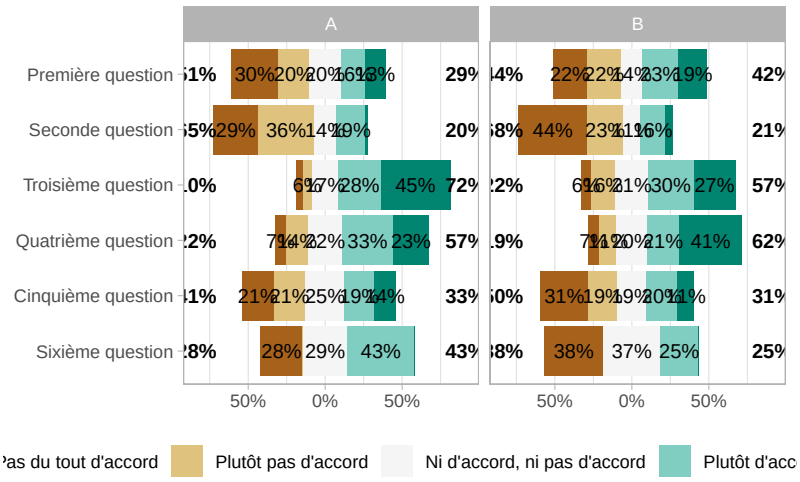
Par défaut, les pourcentages totaux ne prennent pas en compte la modalité centrale (lorsque le nombre de modalité est impair). On peut inclure la modalité centrale avec `totals_include_center = TRUE`, auquel cas la modalité centrale seront comptabilisée pour moitié de chaque côté. Le paramètre `sort` permet de trier les modalités (voir l'aide de `ggstats::gglikert()` pour plus de détails sur les différentes méthodes de tri).

```
df |>
  gglikert(
    include = q1:q6,
    totals_include_center = TRUE,
    sort = "ascending"
  ) +
  guides(
    fill = guide_legend(nrow = 2)
  )
```

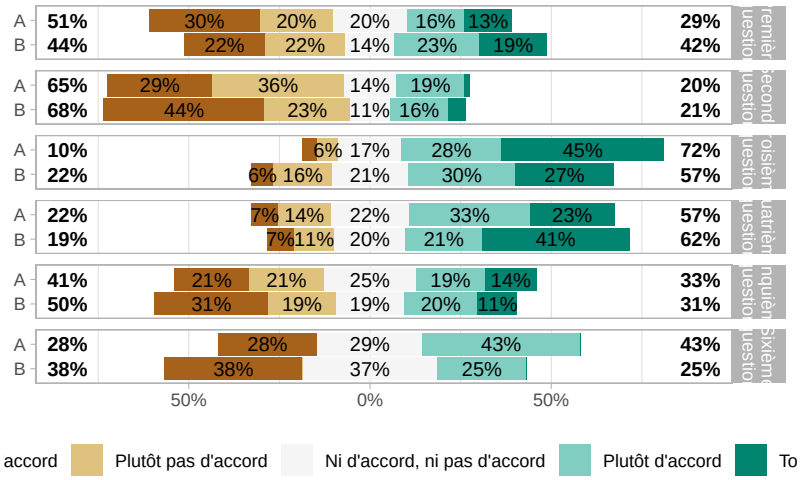


Il est possible de séparer les résultats par sous-groupe avec des facettes.

```
df |>
  gglikert(
    include = q1:q6,
    facet_cols = vars(groupe)
  )
```

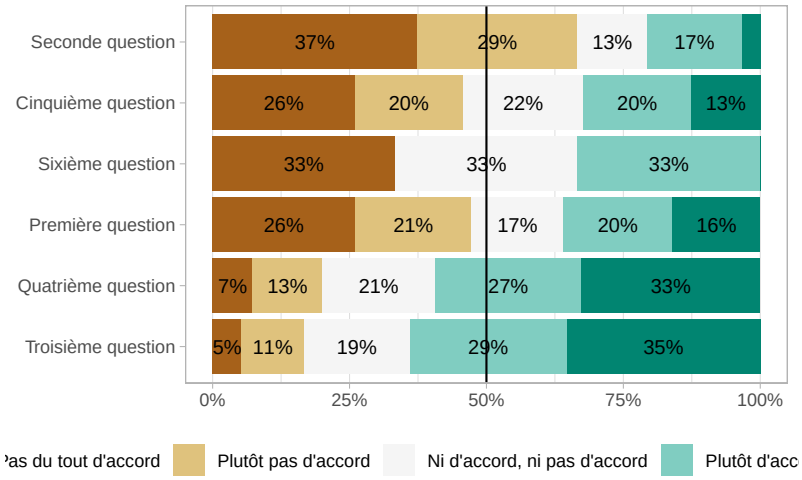


```
df |>
  gglikert(
    include = q1:q6,
    y = "groupe",
    facet_rows = vars(.question),
    facet_label_wrap = 15
  )
```



Une représentation alternative consiste à réaliser un graphique en barres classiques, ce que l'on peut aisément obtenir avec `ggstats::gglikert_stacked()`.

```
df |>
  gglikert_stacked(
    include = q1:q6,
    sort = "ascending",
    add_median_line = TRUE
  )
```


Seconde question
Cinquième question
Sixième question
Première question
Quatrième question
Troisième question
37%
29%
13%
17%
26%
20%
22%
20%
13%
33%
33%
33%
26%
21%
17%
20%
16%
7%
13%
21%
27%
33%
5%
11%
19%
29%
35%
0%
25%
50%
75%
100%
Plutôt pas d'accord
Ni d'accord, ni pas d'accord
Plutôt d'acc

21 Régression linéaire

Un modèle de régression linéaire est un modèle de régression qui cherche à établir une relation linéaire entre une **variable continue**, dite expliquée, et une ou plusieurs variables, dites explicatives.

21.1 Modèle à une seule variable explicative continue

Nous avons déjà abordé très rapidement la régression linéaire dans le chapitre sur la *statistique bivariée* (cf. Section **??**).

Reprenons le même exemple à partir du jeu de données `iris` qui comporte les caractéristiques de 150 fleurs de trois espèces différentes d'iris. Nous cherchons dans un premier temps à explorer la relation entre la largeur (*Petal.Width*) et la longueur des pétales (*Petal.Length*). Représentons cette relation sous la forme d'un nuage de points.

```
library(tidyverse)
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  labs(x = "Longueur", y = "Largeur") +
  theme_light()
```

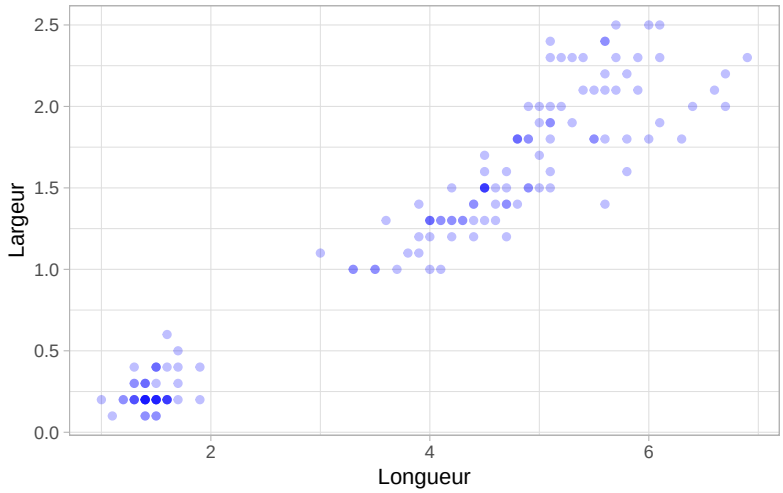


Figure 21.1: Relation entre la largeur et la longueur des pétales (nuage de points)

Il semble bien qu'il y a une **relation linéaire** entre ces deux variables, c'est-à-dire que la relation entre ces deux variables peut être représentée sous la forme d'une droite. Pour cela, on va rechercher la droite telle que la distance entre les points observés et la droite soit la plus petite possible. Cette droite peut être représentée graphique avec `ggplot2::geom_smooth()` et l'option `method = "lm"` :

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  geom_smooth(method = "lm") +
  labs(x = "Longueur", y = "Largeur") +
  theme_light()
```

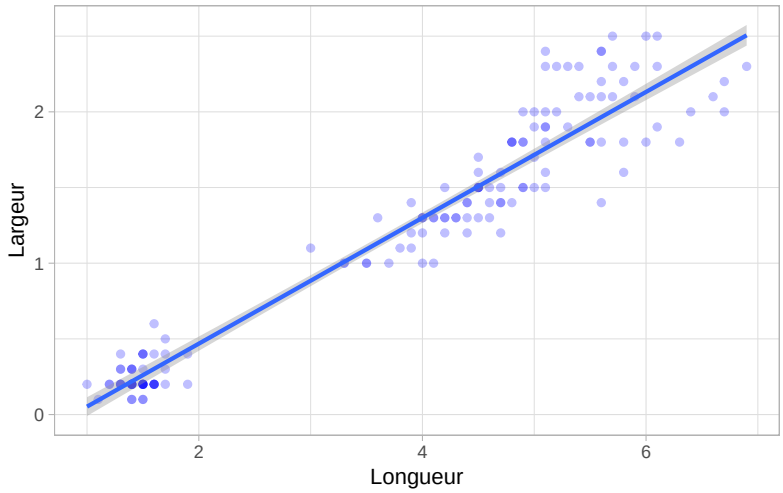


Figure 21.2: Relation linéaire entre la largeur et la longueur des pétales

La fonction de base pour calculer une régression linéaire est la fonction `stats::m()`. On doit en premier lieu spécifier le modèle à l'aide d'une **formule** : on indique la variable à expliquer dans la partie gauche de la formule et la variable explicative dans la partie droite, les deux parties étant séparées par un tilde[27] (`~`).

Dans le cas présent, la variable *Petal.Width* fait office de variable à expliquer et *Petal.Length* de variable explicative. Le modèle s'écrit donc `Petal.Width ~ Petal.Length`.

[27] Avec un clavier français, sous WIndows, le caracère tilde s'obtient en pressant simulténament les touches Alt Gr et 7.

```
mod <- lm(Petal.Width ~ Petal.Length, data = iris)
mod
```

```
Call:
lm(formula = Petal.Width ~ Petal.Length, data = iris)

Coefficients:
 (Intercept)  Petal.Length
     -0.3631        0.4158
```

Le résultat comporte deux coefficients. Le premier, d'une valeur de $0,4158$, est associé à la variable *Petal.Length* et indique la pente de la courbe (on parle de *slope* en anglais). Le second, d'une valeur de $-0,3631$, représente l'ordonnée à l'origine (*intercept* en anglais), c'est-à-dire la valeur estimée de *Petal.Width* lorsque *Petal.Length* vaut 0. Nous pouvons rendre cela plus visible en élargissant notre graphique.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  geom_abline(
    intercept = mod$coefficients[1],
    slope = mod$coefficients[2],
    linewidth = 1,
    colour = "red"
  ) +
  geom_vline(xintercept = 0, linewidth = 1, linetype = "dotted") +
  labs(x = "Longueur", y = "Largeur") +
  expand_limits(x = 0, y = -1) +
  theme_light()
```

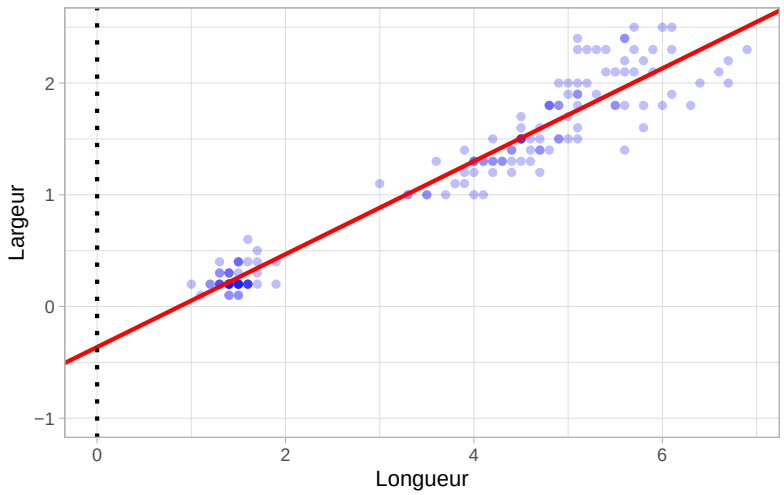


Figure 21.3: Relation linéaire entre la largeur et la longueur des pétales (représentation graphique de l'intercept)

Le modèle linéaire calculé estime donc que le relation entre nos deux variables peut s'écrire sous la forme suivante :

$$Petal.Width = 0,4158 \cdot Petal.Length - 0,3631$$

Le package `{gtsummary}` fournit `gtsummary::tbl_regression()`, une fonction bien pratique pour produire un tableau propre avec les coefficients du modèle, leur intervalle de confiance à 95% et leur p-valeurs[28]. On précisera `intercept = TRUE` pour forcer l'affichage de l'*intercept* qui est masqué par défaut.

[28] Si l'on a besoin de ces informations sous la forme d'un tableau de données classique, on pourra se référer à `broom.helpers::tidy_plus_plus()`, utilisée de manière sous-jacente par `gtsummary::tbl_regression()`, ainsi qu'à la méthode `broom::tidy()`. Ces fonctions sont génériques et peut être utilisées avec une très grande variété de modèles.

```
library(gtsummary)
```

#StandWithUkraine

```
mod %>%
  tbl_regression(intercept = TRUE)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 21.1: un tableau mis en forme des coefficients du modèle

Characteristic	**Beta**	**95% CI**	**p-value**
(Intercept)	-0.36	-0.44, -0.28	<0.001
Petal.Length	0.42	0.40, 0.43	<0.001

Les p-valeurs calculées nous indique si le coefficient est statistiquement différent de 0. En effet, pour la variable explicative, cela nous indique si la relation est statistiquement significative. Le signe du coefficient (positif ou négatif) nous indique le sens de la relation.

Astuce

Dans certains cas, si l'on suppose que la relation entre les deux variables est proportionnelle, on peut souhaiter calculer un modèle sans *intercept*. Par défaut, **R** ajoute un *intercept* à ses modèles. Pour forcer le calcul d'un modèle sans *intercept*, on ajoutera `- 1` à la formule définissant le modèle.

```
lm(Petal.Width ~ Petal.Length - 1, data = iris)
```

```
Call:
lm(formula = Petal.Width ~ Petal.Length - 1, data = iris)

Coefficients:
Petal.Length
      0.3365
```

21.2 Modèle à une seule variable explicative catégorielle

Si dans un modèle linéaire la variable à expliquer est nécessairement continue, il est possible de définir une variable explicative catégorielle. Prenons la variable *Species*.

```
library(labelled)
iris %>% look_for("Species")
```

```
pos variable label col_type missing values
5   Species  -     fct      0       setosa
                                    versicolor
                                    virginica
```

Il s'agit d'un facteur à trois modalités. Par défaut, la première valeur du facteur (ici *setosa*) va servir de modalité de référence.

```
mod <- lm(Petal.Width ~ Species, data = iris)
mod
```

```
Call:
lm(formula = Petal.Width ~ Species, data = iris)

Coefficients:
      (Intercept)  Speciesversicolor   Speciesvirginica
            0.246              1.080              1.780
```

```
mod %>%
  tbl_regression(intercept = TRUE)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 21.2: régression linaire avec une variable explicative catégorielle

Characteristic	**Beta**	**95% CI**	**p-value**
(Intercept)	0.25	0.19, 0.30	<0.001
Species			
setosa	—	—	
versicolor	1.1	1.0, 1.2	<0.001
virginica	1.8	1.7, 1.9	<0.001

Dans ce cas de figure, l'*intercept* représente la situation à la référence, donc pour l'espèce *setosa*.

Calculons les moyennes par espèce :

```
iris %>%
  group_by(Species) %>%
  summarise(mean(Petal.Width))
```

```
# A tibble: 3 x 2
  Species    `mean(Petal.Width)`
  <fct>                    <dbl>
1 setosa                   0.246
2 versicolor               1.33
3 virginica                2.03
```

Comme on le voit, l'*intercept* nous indique donc la moyenne observée pour l'espèce de référence $(0,246)$.

Le coefficient associé à *versicolor* correspond à la différence par rapport à la référence (ici $+1,080$). Comme vous pouvez le constater, il s'agit de la différence entre la moyenne observée pour versicolor $(1,326)$ et celle de la référence setosa $(0,246)$: $1,326 - 0,246 = 1,080$.

Ce coefficient est significativement différent de 0 (p<0,001), indiquant que la largeur des pétales diffère significativement entre les deux espèces.

> **Astuce**
>
> Lorsque l'on calcule le même modèle sans *intercept*, les coefficients s'interprètent un différement :
>
> ```
> lm(Petal.Width ~ Species - 1, data = iris)
> ```
>
> ```
> Call:
> lm(formula = Petal.Width ~ Species - 1, data = iris)
>
> Coefficients:
>     Speciessetosa  Speciesversicolor   Speciesvirginica
>             0.246              1.326              2.026
> ```
>
> En l'absence d'*intercept*, trois coefficients sont calculés et il n'y a plus ici de modalité de référence. Chaque coefficient représente donc la moyenne observée pour chaque modalité.
> On appelle **contrastes** les différents manières de coder des variables catégorielles dans un modèle. Nous y re-

viendrons plus en détail dans un chapitre dédié (cf. Chapitre **??**).

21.3 Modèle à plusieurs variables explicatives

Un des intérêts de la régression linéaire est de pouvoir estimer un modèle multivarié, c'est-à-dire avec plusieurs variables explicatives. Pour cela, on listera les différentes variables explicatives dans la partioe droite de la formule, séparées par le symbole `+`.

```
mod <- lm(
  Petal.Width ~ Petal.Length + Sepal.Width + Sepal.Length + Species,
  data = iris
)
mod
```

```
Call:
lm(formula = Petal.Width ~ Petal.Length + Sepal.Width + Sepal.Length +
    Species, data = iris)

Coefficients:
      (Intercept)       Petal.Length        Sepal.Width       Sepal.Length
         -0.47314            0.24220            0.24220           -0.09293
Speciesversicolor   Speciesvirginica
          0.64811            1.04637
```

```
mod %>%
  tbl_regression(intercept = TRUE)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 21.3: régression linaire avec plusieurs variables explicatives

Characteristic	**Beta**	**95% CI**	**p-value**
(Intercept)	-0.47	-0.82, -0.12	0.008
Petal.Length	0.24	0.15, 0.34	<0.001
Sepal.Width	0.24	0.15, 0.34	<0.001
Sepal.Length	-0.09	-0.18, 0.00	0.039
Species			
setosa	—	—	
versicolor	0.65	0.40, 0.89	<0.001
virginica	1.0	0.72, 1.4	<0.001

Ce type de modèle permet d'estimer l'effet de chaque variable explicative, toutes choses égales par ailleurs. Dans le cas présent, on s'aperçoit que la largeur des pétales diffère significativement selon les espèces, est fortement corrélée positivement à la longueur du pétale et la largeur du sépale et qu'il y a, lorsque l'on ajuste sur l'ensemble des autres variables, une relation négative (faiblement significative) avec la longueur du sépale.

Lorsque le nombre de coefficients est élevé, une représentation graphique est souvent plus facile à lire qu'un tableau. On parle alors de graphique en forêt ou *forest plot* en anglais. Rien de plus facile ! Il suffit d'avoir recours à `ggstats::ggcoef_model()`.

```
library(ggstats)
ggcoef_model(mod)
```

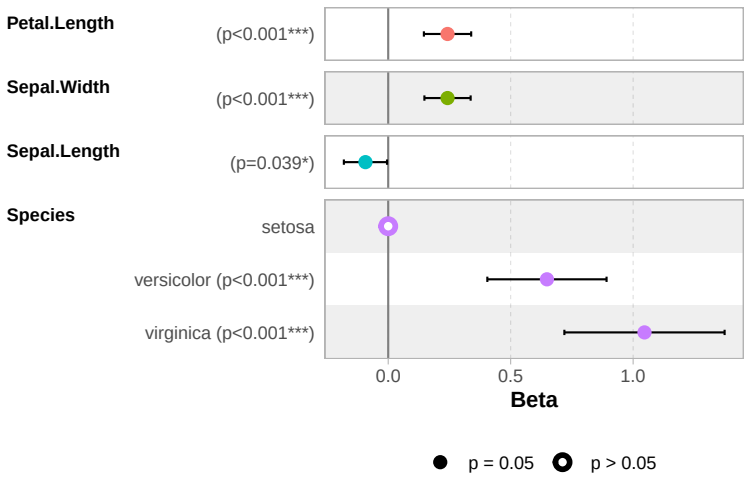


Figure 21.4: un graphique en forêt des coefficients du modèle

22 Régression logistique binaire

Dans le chapitre précédent (Chapitre **??**), nous avons abordé la régression linéaire pour les variables continues. Pour analyser une variable catégorielle, il est nécessaire d'avoir recours à d'autres types de modèles. Les modèles linéaires généralisés (*generalized linear models* ou *GLM* en anglais) sont une généralisation de la régression linéaire grâce à l'utilisation d'une fonction de **lien** utilisée pour transformer la variable à expliquer et pouvoir ainsi retomber sur un modèle linéaire classique. Il existe de multiples fonctions de lien correspondant à tout autant de modèles. Par exemple, on pourra utiliser un modèle de Poisson pour une variable entière positive ou un modèle d'incidence.

Pour une variable binaire (c'est-à-dire du type oui / non ou vrai / faux), les modèles les plus courants utilisent les fonctions de lien *probit* ou *logit*, cette dernière fonction correspondent à la **régression logistique binaire** (modèle *logit*). Comme pour la régression linéaire, les variables explicatives peuvent être continues et/ou catégorielles.

22.1 Préparation des données

Dans ce chapitre, nous allons encore une fois utiliser les données de l'enquête *Histoire de vie*, fournies avec l'extension `{questionr}`.

```
data(hdv2003, package = "questionr")
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 sur la probabilité de pratiquer un sport.

En premier lieu, il importe de vérifier, par exemple avec `labelled::look_for()`, que notre variable d'intérêt (ici *sport*) est correctement codée, c'est-à-dire que la première modalité correspondent à la référence (soit ne pas avoir vécu l'évènement d'intérêt) et que la seconde modalité corresponde au fait d'avoir vécu l'évènement.

```
library(labelled)
d |> look_for("sport")
```

```
pos variable label col_type missing values
19  sport    -     fct      0       Non
                                    Oui
```

Dans notre exemple, la modalité Non est déjà la première modalité. Il n'y a donc pas besoin de modifier notre variable.

Il faut également la présence éventuelle de données manquantes (`NA`)[29]. Les observations concernées seront tout simplement exclues du modèle lors de son calcul. Ce n'est pas notre cas ici.

[29] Pour visualiser le nombre de données manquantes (`NA`) de l'ensemble des variables d'un tableau, on pourra avoir recours à `questionr::freq.na()`.

> **Astuce**
>
> Alternativement, on pourra aussi coder notre variable à expliquer sous forme booléenne (`FALSE` / `TRUE`) ou numériquement en `0`/`1`.
> 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, de temps en temps, Oui, régulièrement. 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 catégorielles. En effet, dans un modèle,

tous les coefficients sont calculés par rapport à la modalité de référence (cf. Section **??**). Il importe donc de choisir une modalité de référence qui fasse sens afin de faciliter l'interprétation. Par ailleurs, ce choix doit dépendre de la manière dont on souhaite présenter les résultats (le *data storytelling* est essentiel). 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 ?

```
d |> look_for("sexe")
```

```
pos variable label col_type missing values
3   sexe     -     fct      0       Homme
                                    Femme
```

La modalité Femme s'avère ne pas être la première modalité. Nous devons appliquer la fonction `forcats::fct_relevel()` ou la fonction `stats::relevel()` :

```
library(tidyverse)
d <- d |>
  mutate(sexe = sexe |> fct_relevel("Femme"))
```

```
d$sexe |> questionr::freq()
```

```
         n  % val%
Femme 1101 55   55
Homme  899 45   45
```

Données labellisées

Si l'on utilise des données labellisées (voir Chapitre **??**), nos variables catégorielles seront stockées sous la forme d'un

vecteur numérique avec des étiquettes. Il sera donc nécessaire de convertir ces variables en facteurs, tout simplement avec `labelled::to_factor()` ou `labelled::unlabelled()`.

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.

```
d |> look_for("age", "heures")
```

```
pos variable  label col_type missing values
2   age       -     int      0
20  heures.tv -     dbl      5
```

Nos deux variables sont bien renseignées sous forme numérique (respectivement des entiers et des nombres décimaux).

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 (et donc en variable catégorielle) avec la fonction `cut()` (cf. Section **??**) :

```
d <- d |>
  mutate(
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                   "45-64 ans", "65 ans et plus")
      )
  )
d$groupe_ages |> questionr::freq()
```

```
                n    % val%
18-24 ans     169  8.5  8.5
```

```
25-44 ans      706 35.3 35.3
45-64 ans      745 37.2 37.2
65 ans et plus 380 19.0 19.0
```

Jetons maintenant un oeil à la variable *nivetud* :

```
d$nivetud |> questionr::freq()
```

```
                                                                  n    % val%
N'a jamais fait d'etudes                                         39  2.0  2.1
A arrete ses etudes, avant la derniere annee d'etudes primaires  86  4.3  4.6
Derniere annee d'etudes primaires                               341 17.0 18.1
1er cycle                                                       204 10.2 10.8
2eme cycle                                                      183  9.2  9.7
Enseignement technique ou professionnel court                   463 23.2 24.5
Enseignement technique ou professionnel long                    131  6.6  6.9
Enseignement superieur y compris technique superieur            441 22.0 23.4
NA                                                              112  5.6   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'améliorier notre modèle logistique, il peut être pertinent de regrouper certaines modalités (cf. Section **??**) :

```
d <- d |>
  mutate(
    etudes = nivetud |>
      fct_recode(
      "Primaire" = "N'a jamais fait d'etudes",
      "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
      "Primaire" = "Derniere annee d'etudes primaires",
      "Secondaire" = "1er cycle",
      "Secondaire" = "2eme cycle",
      "Technique / Professionnel" = "Enseignement technique ou professionnel court",
      "Technique / Professionnel" = "Enseignement technique ou professionnel long",
      "Supérieur" = "Enseignement superieur y compris technique superieur"
    )
  )
```

```
d$etudes |> questionr::freq()
```

```
                            n    % val%
Primaire                  466 23.3 24.7
Secondaire                387 19.4 20.5
Technique / Professionnel 594 29.7 31.5
Supérieur                 441 22.0 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 `forcats::fct_na_value_to_level()` :

```
d$etudes <- d$etudes |>
  fct_na_value_to_level("Non documenté")
```

Enfin, pour améliorer les différentes sorties (tableaux et figures), nous allons ajouter des étiquettes de variables (cf. Chapitre **??**) avec `labelled::set_variable_labels()`.

```
d <- d |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    relig = "Rapport à la religion",
    heures.tv = "Heures de télévision / jour"
  )
```

Code récapitulatif (préparation des données)

```
data(hdv2003, package = "questionr")
d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                   "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
    ) |>
    fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    relig = "Rapport à la religion",
    heures.tv = "Heures de télévision / jour"
  )
```

22.2 Statistiques descriptives

Avant toute analyse multivariée, il est toujours bon de procéder à une analyse descriptive bivariée simple, tout simplement avec `gtsummary::tbl_summary()`. Ajoutons quelques tests de comparaison avec `gtsummary::add_p()`. Petite astuce : `gtsummary::modify_spanning_header()` permet de rajouter un en-tête sur plusieurs colonnes.

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",", big.mark = " ")
```

```
d |>
  tbl_summary(
    by = sport,
    include = c(sexe, groupe_ages, etudes, relig, heures.tv)
  ) |>
  add_overall(last = TRUE) |>
  add_p() |>
  bold_labels() |>
  modify_spanning_header(
    update = all_stat_cols() ~ "**Pratique un sport ?**"
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 22.1: Pratique d'un sport selon différentes variables explicatives (analyse bivariée)

Caractéristique	**Non**, N = 1 277	**Oui**, N = 723	**Total**, N = 2 000	**p-valeur**
Sexe				<0,001
Femme	747 (58%)	354 (49%)	1 101 (55%)	
Homme	530 (42%)	369 (51%)	899 (45%)	

Caractéristique	**Non**, N = 1 277	**Oui**, N = 723	**Total**, N = 2 000	**p-valeur**
Groupe d'âges				<0,001
18-24 ans	58 (4,5%)	111 (15%)	169 (8,5%)	
25-44 ans	359 (28%)	347 (48%)	706 (35%)	
45-64 ans	541 (42%)	204 (28%)	745 (37%)	
65 ans et plus	319 (25%)	61 (8,4%)	380 (19%)	
Niveau d'études				<0,001
Primaire	416 (33%)	50 (6,9%)	466 (23%)	
Secondaire	270 (21%)	117 (16%)	387 (19%)	
Technique / Professionnel	378 (30%)	216 (30%)	594 (30%)	
Supérieur	186 (15%)	255 (35%)	441 (22%)	
Non documenté	27 (2,1%)	85 (12%)	112 (5,6%)	
Rapport à la religion				0,14
Pratiquant regulier	182 (14%)	84 (12%)	266 (13%)	
Pratiquant occasionnel	295 (23%)	147 (20%)	442 (22%)	
Appartenance sans pratique	473 (37%)	287 (40%)	760 (38%)	
Ni croyance ni appartenance	239 (19%)	160 (22%)	399 (20%)	
Rejet	60 (4,7%)	33 (4,6%)	93 (4,7%)	
NSP ou NVPR	28 (2,2%)	12 (1,7%)	40 (2,0%)	
Heures de télévision / jour	2,00 (1,00 – 3,00)	2,00 (1,00 – 3,00)	2,00 (1,00 – 3,00)	<0,001
Manquant	2	3	5	

22.3 Calcul de la régression logistique binaire

La spécification d'une régression logistique se fait avec `stats::glm()` et est très similaire à celle d'une régression linéaire simple (cf. Section **??**) : on indique la variable à expliquer suivie d'un tilde (`~`) puis des variables explicatives séparées par un plus (`+`)[30]. Il faut indiquer à `glm()` la famille du modèle souhaité : on indiquera simplement `family = binomial` pour un modèle *logit*[31].

```
mod <- glm(
  sport ~ sexe + groupe_ages + etudes + relig + heures_tv,
  family = binomial,
  data = d
)
```

Pour afficher les résultats du modèle, le plus simple est d'avoir recours à `gtsummary::tbl_regression()`.

```
mod |>
  tbl_regression(intercept = TRUE) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 22.2: Facteurs associés à la pratique d'un sport (régression logistique binaire)

Caractéristique	log(OR)	95% IC	p-valeur
(Intercept)	-0,80	-1,4 – -0,17	0,014
Sexe			
Femme	—	—	
Homme	0,44	0,23 – 0,65	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	-0,42	-0,87 – 0,03	0,065

[30] Il est possible de spécifier des modèles plus complexes, notamment avec des effets d'interaction, qui seront aborder plus loin (cf. Chapitre **??**).

[31] Pour un modèle *probit*, on indiquera `family = binomial("probit")`.

Caractéristique	log(OR)	95% IC	p-valeur
45-64 ans	-1,1	-1,6 – -0,62	<0,001
65 ans et plus	-1,4	-1,9 – -0,85	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	0,95	0,57 – 1,3	<0,001
Technique / Professionnel	1,0	0,68 – 1,4	<0,001
Supérieur	1,9	1,5 – 2,3	<0,001
Non documenté	2,2	1,5 – 2,8	<0,001
Rapport à la religion			
Pratiquant regulier	—	—	
Pratiquant occasionnel	-0,02	-0,39 – 0,35	>0,9
Appartenance sans pratique	-0,01	-0,35 – 0,34	>0,9
Ni croyance ni appartenance	-0,22	-0,59 – 0,16	0,3
Rejet	-0,38	-0,95 – 0,17	0,2
NSP ou NVPR	-0,08	-0,92 – 0,70	0,8
Heures de télévision / jour	-0,12	-0,19 – -0,06	<0,001

22.4 Interpréter les coefficients

L'*intercept* traduit la situation à la référence (i.e. toutes les variables catégorielles à leur modalité de référence et les variables continues à 0), après transformation selon la fonction de lien (i.e. après la transformation *logit*).

Illustrons cela. Supposons donc une personne de sexe féminin, âgée entre 18 et 24 ans, de niveau d'étude primaire, pratiquante régulière et ne regardant pas la télévision (situation de référence). Seul l'*intercept* s'applique dans le modèle, et donc le modèle prédit que sa probabilité de faire du sport est de $-0,80$

selon l'échelle *logit*. Retraduisons cela en probabilité classique avec la fonction *logit inverse*.

```
logit_inverse <- binomial("logit") |> purrr::pluck("linkinv")
logit_inverse(-0.80)
```

```
[1] 0.3100255
```

Selon le modèle, la probabilité que cette personne fasse du sport est donc de 31%.

Prenons maintenant une personne identique mais de sexe masculin. Nous devons donc considérer, en plus de l'*intercept*, le coefficient associé à la modalité Homme. Sa probabilité de faire du sport est donc :

```
logit_inverse(-0.80 + 0.44)
```

```
[1] 0.4109596
```

Le coefficient associé à Homme est donc un modificateur par rapport à la situation de référence.

Enfin, considérons que cette dernière personne regarde également la télévision 2 heures en moyenne par jour. Nous devons alors considérer le coefficient associé à la variable *heures.tv* et, comme il s'agit d'une variable continue, multiplier ce coefficient par 2, car le coefficient représente le changement pour un incrément de 1 unité.

```
logit_inverse(-0.80 + 0.44 + 2 * -0.12)
```

```
[1] 0.3543437
```

Il est crucial de bien comprendre comment dans quels cas et comment chaque coefficient est utilisé par le modèle.

Le package `{breakdown}` permet de mieux visualiser notre dernier exemple.

```r
individu3 <- d[1, ]
individu3$sexe[1] <- "Homme"
individu3$groupe_ages[1] <- "18-24 ans"
individu3$etudes[1] <- "Primaire"
individu3$relig[1] <- "Pratiquant regulier"
individu3$heures.tv[1] <- 2
```

```r
library(breakDown)
logit <- function(x) exp(x) / (1 + exp(x))
plot(
  broken(mod, individu3, predict.function = betas),
  trans = logit
) +
  scale_y_continuous(
    labels = scales::label_percent(),
    breaks = 0:5/5,
    limits = c(0, 1)
  )
```

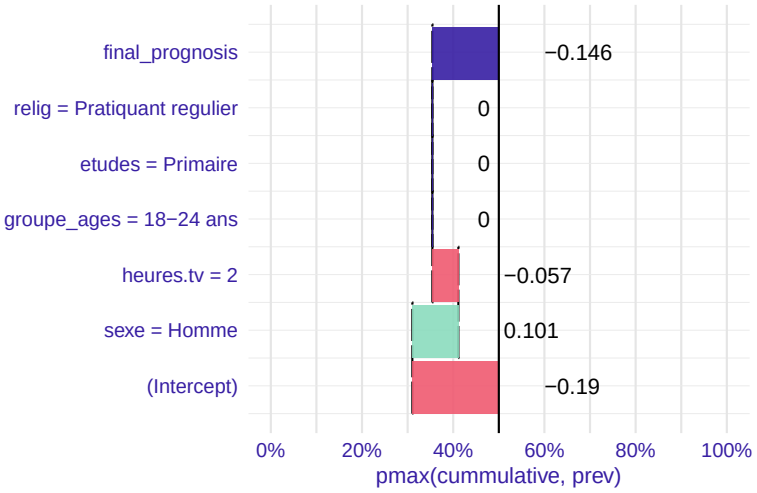


Figure 22.1: Décomposition de la probabilité de faire du sport de l'individu 3

22.5 La notion d'*odds ratio*

L'un des intérêts de la régression logistique *logit* réside dans le fait que l'exponentiel des coefficients correspond à des *odds ratio* ou rapport des côtes en français.

Astuce

Pour comprendre la notion de côte (*odd* en anglais), on peut se référer aux paris sportifs. Par exemple, lorsque les trois quarts des parieurs parient que le cheval A va remporter la course, on dit alors que ce cheval à une côte de trois contre un (trois personnes parient qu'il va gagner contre une personne qu'il va perdre). Prenon un autre cheval : si les deux tiers pensent que le cheval B va perdre (donc un tiers pense qu'il va gagner), on dira alors que sa côte est de un contre deux (une personne pense qu'il va gagner contre deux qu'il va perdre).
Si l'on connait la proportion ou probabilité p d'avoir vécu ou de vivre un événenement donné (ici gagner la course), la côte (l'*odd*) s'obtient avec la formule suivante : $p/(1-p)$. La côte du cheval A est bien $0,75/(1-0,75) = 0,75/0,25 = 3$ est celle du cheval B $(1/3)/(2/3) = 1/2 = 0,5$.
Pour comparer deux côtes (par exemple pour savoir si le cheval A a une probabilité plus élevée de remporter la course que le cheval B, selon les parieurs), on calculera tout simplement le rapport des côtes ou *odds ratio* (OR) : $OR_{A/B} = Odds_A/Odds_B = 3/0,5 = 6$.
Ce calcul peut se faire facilement dans **R** avec la fonction `questionr::odds.ratio()`.

```
questionr::odds.ratio(.75, 1/3)
```

```
[1] 6
```

L'*odds ratio* est donc égal à 1 si les deux côtes sont identiques, est supérieur à 1 si le cheval A une probabilité supérieure à celle du cheval B, et inférieur à 1 si c'est probabilité est inférieure.

On le voit, par construction, l'*odds ratio* de B par rapport à A est l'inverse de celui de A par rapport à B : $OR_{B/A} = 1/OR_{A/B}$.

Pour afficher les *odds ratio* il suffit d'indiquer `exponentiate = TRUE` à `gtsummary::tbl_regression()`.

```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 22.3: Facteurs associés à la pratique d'un sport (*odds ratios*)

Caractéristique	OR	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	1,55	1,26 – 1,91	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	0,66	0,42 – 1,03	0,065
45-64 ans	0,34	0,21 – 0,54	<0,001
65 ans et plus	0,25	0,15 – 0,43	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	2,59	1,77 – 3,83	<0,001
Technique / Professionnel	2,86	1,98 – 4,17	<0,001
Supérieur	6,63	4,55 – 9,80	<0,001
Non documenté	8,59	4,53 – 16,6	<0,001
Rapport à la religion			
Pratiquant regulier	—	—	
Pratiquant occasionnel	0,98	0,68 – 1,42	>0,9
Appartenance sans pratique	0,99	0,71 – 1,40	>0,9
Ni croyance ni appartenance	0,81	0,55 – 1,18	0,3

Caractéristique	OR	95% IC	p-valeur
Rejet	0,68	0,39 – 1,19	0,2
NSP ou NVPR	0,92	0,40 – 2,02	0,8
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

Pour une représentation visuelle, graphique en forêt ou *forest plot* en anglais, on aura tout simplement recours à `ggstats::ggcoef_model()`.

```
mod |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

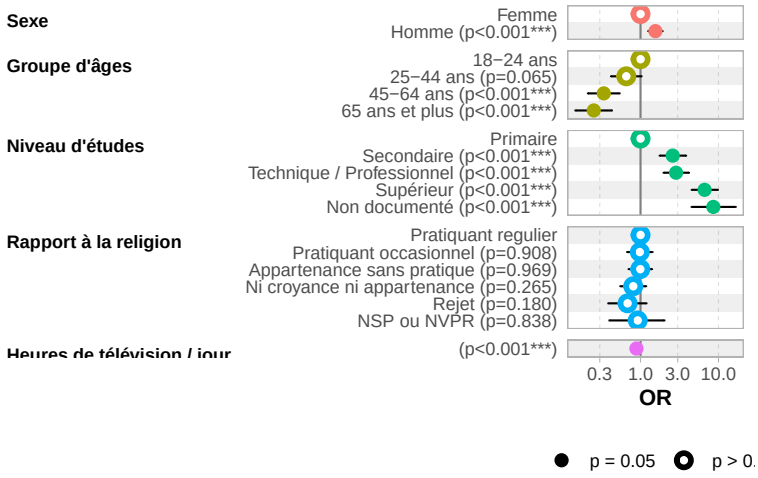


Figure 22.2: Facteurs associés à la pratique d'un sport (*forest plot*)

> **Note**
>
> Lorsque l'on réalise un *forest plot* de coefficients exponentialisés tels que des *odds ratios*, une bonne pratique consiste à utiliser une échelle logarithmique. En effet, l'inverse d'un *odds ratio* de 2 est 0,5. Avec une échelle logarithmique, la distance entre 0,5 et 1 est égale à celle entre 1 et 2. Sur la figure précédente, vous pourrez no-

ter que `ggstats::ggcoef_model()` applique automatiquement une échelle logarithmique lorsque `exponentiate = TRUE`.
Quelques références : Forest Plots: Linear or Logarithmic Scale? ou encore Graphing Ratio Measures on Forest Plot.

Mise en garde

En rédigeant les résultats de la régression, il faudra être vigilant à ne pas confondre les *odds ratios* avec des *prevalence ratios*. Avec un *odds ratio* de 1,55, il serait tentant d'écrire que les hommes ont une probabilité 55% supérieure de pratique un sport que les femmes (toutes choses égales par ailleurs). Une telle formulation correspond à un *prevalence ratio* (rapport des prévalences en français) ou *risk ratio* (rapport des risques), à savoir diviser la probabilité de faire du sport des hommes par celle des femmes, p_{hommes}/p_{femmes}. Or, cela ne correspond pas à la formule de l'*odds ratio*, à savoir $(p_{hommes}/(1-p_{hommes}))/(p_{femmes}/(1-p_{femmes}))$.
Lorsque le phénomène étudié est rare et donc que les probabilités sont faibles (inférieures à quelques pour-cents), alors il est vrai que les *odds ratio* sont approximativement égaux aux *prevalence ratios*. Mais ceci n'est plus du tout vrai pour des phénomènes plus fréquents.

22.6 Afficher les écarts-types plutôt que les intervalles de confiance

La manière de présenter les résultats d'un modèle de régression varie selon les disciplines, les champs thématiques et les revues. Si en sociologie et épidémiologie on aurait plutôt tendance à afficher les *odds ratio* avec leur intervalle de confiance, il est fréquent en économétrie de présenter plutôt les coefficients brust et leurs écarts-types (*standard deviation* ou *sd* en anglais). De même, plutôt que d'ajouter une colonne avec les p valeurs, un usage consiste à afficher des étoiles de significativités à la droite des coefficients significatifs.

Pour cela, on pourra personnaliser le tableau produit avec `gtsummary::tbl_regression()`, notamment avec `gtsummary::add_significance_stars()` pour l'ajout des étoiles de significativité, ainsi que `gtsummary::modify_column_hide()` et `gtsummary::modify_column_unhide()` pour afficher / masquer les colonnes du tableau produit[32].

[32] La liste des colonnes disponibles peut être obtenues avec `mod |> tbl_regression() |> purrr::pluck("table_body") |> colnames()`.

```
mod |>
  tbl_regression() |>
  add_significance_stars() |>
  modify_column_hide(c("ci", "p.value")) |>
  modify_column_unhide("std.error") |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 22.4: Présentation économétrique des facteurs associés à la pratique d'un sport

Caractéristique	log(OR)	ET
Sexe		
Femme	—	—
Homme	0,44***	0,106
Groupe d'âges		
18-24 ans	—	—
25-44 ans	-0,42	0,228
45-64 ans	-1,1***	0,238
65 ans et plus	-1,4***	0,274
Niveau d'études		
Primaire	—	—
Secondaire	0,95***	0,197
Technique / Professionnel	1,0***	0,190
Supérieur	1,9***	0,195
Non documenté	2,2***	0,330
Rapport à la religion		
Pratiquant regulier	—	—
Pratiquant occasionnel	-0,02	0,189

Caractéristique	log(OR)	ET
Appartenance sans pratique	-0,01	0,175
Ni croyance ni appartenance	-0,22	0,193
Rejet	-0,38	0,286
NSP ou NVPR	-0,08	0,411
Heures de télévision / jour	-0,12***	0,034

Les économistes pourraient préférer le package `{modelsummary}` à `{gtsummary}`. Ces deux packages ont un objectif similaire (la production de tableaux statistiques) mais abordent cet objectif avec des approches différentes. Il faut noter que `modelsummary::modelsummary()` n'affiche pas les modalités de référence, ni les étiquettes de variable.

```
mod |> modelsummary::modelsummary(stars = TRUE)
```

La fonction `modelsummary::modelplot()` permet d'afficher un graphique des coefficients.

```
mod |> modelsummary::modelplot()
```

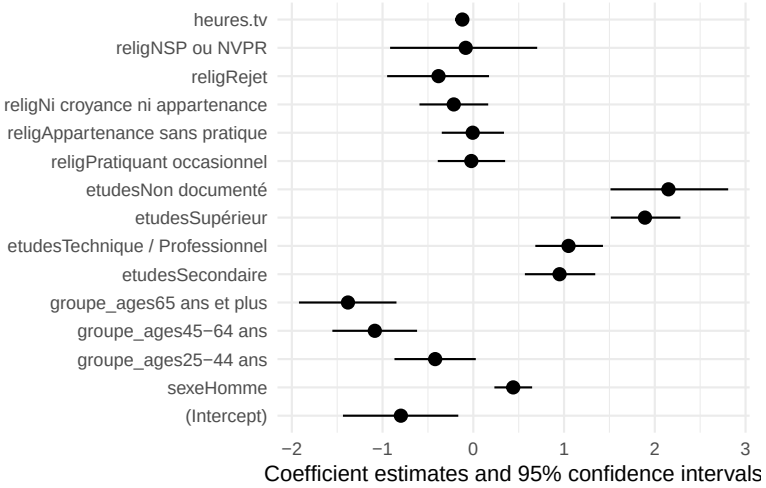


Figure 22.3: Facteurs associés à la pratique d'un sport avec modelplot()

Table 22.5: Présentation des facteurs associés à la pratique d'un sport avec modelsummary()

	(1)
(Intercept)	−0.798*
	(0.324)
sexeHomme	0.440***
	(0.106)
groupe_ages25-44 ans	−0.420+
	(0.228)
groupe_ages45-64 ans	−1.085***
	(0.238)
groupe_ages65 ans et plus	−1.381***
	(0.274)
etudesSecondaire	0.951***
	(0.197)
etudesTechnique / Professionnel	1.049***
	(0.190)
etudesSupérieur	1.892***
	(0.195)
etudesNon documenté	2.150***
	(0.330)
religPratiquant occasionnel	−0.022
	(0.189)
religAppartenance sans pratique	−0.007
	(0.175)
religNi croyance ni appartenance	−0.215
	(0.193)
religRejet	−0.384
	(0.286)
religNSP ou NVPR	−0.084
	(0.411)
heures.tv	−0.121***
	(0.034)
Num.Obs.	1995
AIC	2236.2
BIC	2320.1
Log.Lik.	−1103.086
F	21.691
RMSE	0.43

+ p < 0.1, * p < 0.05, ** p < 0.01, *** p < 0.001

ATTENTION : si l'on affiche les *odds ratio* avec `exponentiate = TRUE`, `modelsummary::modelplot()` conserve par défaut une échelle linéaire. On sera donc vigilant à appliquer `ggplot2::scale_x_log10()` manuellement pour utiliser une échelle logarithmique.

```
mod |>
  modelsummary::modelplot(exponentiate = TRUE) +
  ggplot2::scale_x_log10()
```

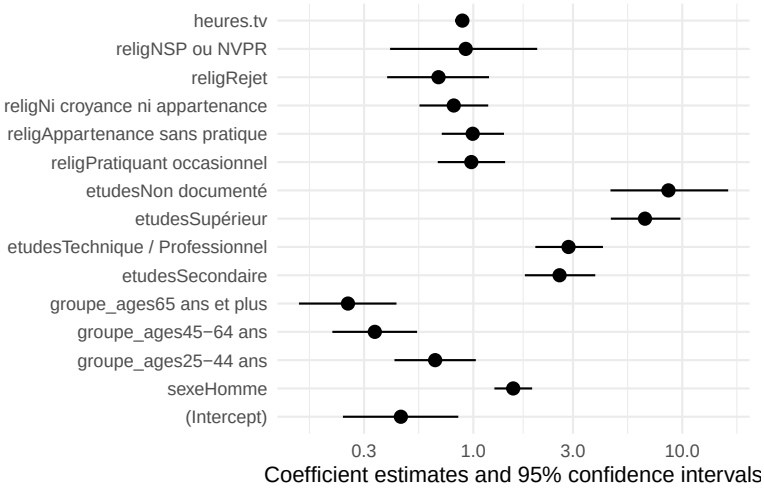


Figure 22.4: Odds Ratios associés à la pratique d'un sport avec modelplot()

22.7 Afficher toutes les comparaisons (*pairwise contrasts*)

Dans le tableau des résultats (Table **??**), pour les variables catégorielles, il importe de bien garder en mémoire que chaque *odds ratio* doit se comparer à la valeur de référence. Ainsi, les *odds ratios* affichés pour chaque classe d'âges correspondent à une comparaison avec la classe d'âges de références, les 18-24 ans. La p-valeur associée nous indique quant à elle si cet *odds ratio* est significativement de 1, donc si cette classe d'âges données se comporte différemment de celle de référence.

Mais cela ne nous dit nullement si les 65 ans et plus diffèrent des 45-64 ans. Il est tout à fait possible de recalculer l'*odds ratio* correspondant en rapport les *odds ratio* à la référence : $OR_{65+/45-64} = OR_{65+/18-24}/OR_{45-64/18-24}$.

Le package `{emmeans}` et sa fonction `emmeans::emmeans()` permettent de recalculer toutes les combinaisons d'*odds ratio* (on parle alors de *pairwise contrasts*) ainsi que leur intervalle de confiance et la p-valeur correspondante.

On peut ajouter facilement[33] cela au tableau produit avec `gtsummary::tbl_regression()` en ajoutant l'option `add_pairwise_contrasts = TRUE`.

[33] Cela nécessite néanmoins au minimum la version 1.11.0 du package `{broom.helpers}` et la version 1.6.3 de `{gtsummary}`.

```
mod |>
  tbl_regression(
    exponentiate = TRUE,
    add_pairwise_contrasts = TRUE
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

De même, on peur visualiser les coefficients avec la même option dans `ggstats::ggcoef_model()`[34]. On peut d'ailleurs choisir les variables concernées avec l'argument `pairwise_variables`.

[34] Cela nécessite néanmoins au minimum la version 1.11.0 du package `{broom.helpers}` et la version 0.2.0 de `{ggstats}`.

```
mod |>
  ggstats::ggcoef_model(
    exponentiate = TRUE,
    add_pairwise_contrasts = TRUE,
    pairwise_variables = c("groupe_ages", "etudes")
  )
```

Table 22.6: Facteurs associés à la pratique d'un sport (pairwise contrasts)

Caractéristique	**OR**	**95% IC**	**p-valeur**
___Sexe___			
Homme / Femme	1,55	1,26 – 1,91	<0,001
___Groupe d'âges___			
(25-44 ans) / (18-24 ans)	0,66	0,37 – 1,18	0,3
(45-64 ans) / (18-24 ans)	0,34	0,18 – 0,62	<0,001
(45-64 ans) / (25-44 ans)	0,51	0,38 – 0,70	<0,001
65 ans et plus / (18-24 ans)	0,25	0,12 – 0,51	<0,001
65 ans et plus / (25-44 ans)	0,38	0,24 – 0,61	<0,001
65 ans et plus / (45-64 ans)	0,74	0,47 – 1,17	0,3
___Niveau d'études___			
Secondaire / Primaire	2,59	1,51 – 4,43	<0,001
(Technique / Professionnel) / Primaire	2,86	1,70 – 4,79	<0,001
(Technique / Professionnel) / Secondaire	1,10	0,74 – 1,64	>0,9
Supérieur / Primaire	6,63	3,89 – 11,3	<0,001
Supérieur / Secondaire	2,56	1,69 – 3,88	<0,001
Supérieur / (Technique / Professionnel)	2,32	1,61 – 3,36	<0,001
Non documenté / Primaire	8,59	3,49 – 21,1	<0,001
Non documenté / Secondaire	3,32	1,46 – 7,53	<0,001
Non documenté / (Technique / Professionnel)	3,01	1,38 – 6,56	0,001
Non documenté / Supérieur	1,30	0,58 – 2,90	>0,9
___Rapport à la religion___			
Pratiquant occasionnel / Pratiquant regulier	0,98	0,57 – 1,68	>0,9
Appartenance sans pratique / Pratiquant regulier	0,99	0,60 – 1,63	>0,9
Appartenance sans pratique / Pratiquant occasionnel	1,02	0,68 – 1,52	>0,9
Ni croyance ni appartenance / Pratiquant regulier	0,81	0,47 – 1,40	0,9
Ni croyance ni appartenance / Pratiquant occasionnel	0,82	0,52 – 1,31	0,8
Ni croyance ni appartenance / Appartenance sans pratique	0,81	0,54 – 1,21	0,7
Rejet / Pratiquant regulier	0,68	0,30 – 1,54	0,8
Rejet / Pratiquant occasionnel	0,70	0,33 – 1,49	0,8
Rejet / Appartenance sans pratique	0,69	0,33 – 1,41	0,7
Rejet / Ni croyance ni appartenance	0,85	0,40 – 1,79	>0,9
NSP ou NVPR / Pratiquant regulier	0,92	0,29 – 2,97	>0,9
NSP ou NVPR / Pratiquant occasionnel	0,94	0,30 – 2,92	>0,9
NSP ou NVPR / Appartenance sans pratique	0,93	0,30 – 2,82	>0,9
NSP ou NVPR / Ni croyance ni appartenance	1,14	0,37 – 3,55	>0,9
NSP ou NVPR / Rejet	1,35	0,37 – 4,88	>0,9
___Heures de télévision / jour___	0,89	0,83 – 0,95	<0,001

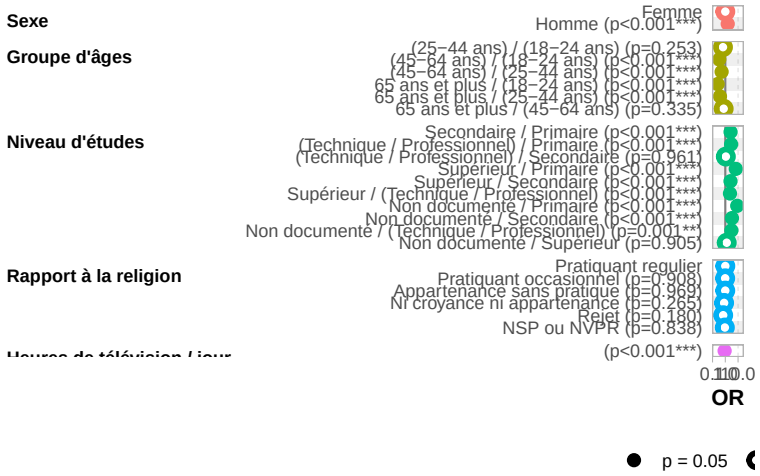


Figure 22.5: Facteurs associés à la pratique d'un sport (*pairwise contrasts*)

22.8 Identifier les variables ayant un effet significatif

Pour les variables catégorielles à trois modalités ou plus, les p-values associées aux *odds ratios* nous indique si un *odd ratio* est significativement différent de 1, par rapport à la modalité de référence. Mais cela n'indique pas si globalement une variable a un effet significatif sur le modèle. Pour tester l'effet global d'une variable, on peut avoir recours à la fonction `car::Anova()`. Cette dernière va tour à tour supprimer chaque variable du modèle et réaliser une analyse de variance (ANOVA) pour voir si la variance change significativement.

```
car::Anova(mod)
```

```
Analysis of Deviance Table (Type II tests)

Response: sport
           LR Chisq Df Pr(>Chisq)
sexe         17.309  1  3.176e-05 ***
```

```
groupe_ages   52.803  3  2.020e-11 ***
etudes       123.826  4  < 2.2e-16 ***
relig          4.232  5  0.5165401
heures.tv     13.438  1  0.0002465 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Ainsi, dans le cas présent, la suppression de la variable *relig* ne modifie significativement pas le modèle, indiquant l'absence d'effet de cette variable.

Si l'on a recours à `gtsummary::tbl_regression()`, on peut facilement ajouter les p-valeurs globales avec `gtsummary::add_global_p()`[35].

[35] Si l'on veut conserver les p-values individuelles associées à chaque *odds ratio*, on ajoutera l'option `keep = TRUE`.

```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels() |>
  add_global_p()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Note

Concernant le test réalisé dans le cadre d'une Anova, il existe trois tests différents que l'on présente comme le type 1, le type 2 et le type 3 (ou I, II et III). Pour une explication sur ces différents types, on pourra se référer (en anglais) à https://mcfromnz.wordpress.com/2011/03/02/anova-type-iiiiii-ss-explained/ ou encore http://md.psych.bio.uni-goettingen.de/mv/unit/lm_cat/lm_cat_unbal_ss_explained.html.
Le type I n'est pas recommandé dans le cas présent car il dépend de l'ordre dans lequel les différentes variables sont testées.
Lorsqu'il n'y a pas d'interaction dans un modèle, le type II serait à privilégier car plus puissant (nous aborderons les

Table 22.7: Ajout des p-valeurs globales

Caractéristique	**OR**	**95% IC**	**p-valeur**
___Sexe___			<0,001
Femme	—	—	
Homme	1,55	1,26 – 1,91	
___Groupe d'âges___			<0,001
18-24 ans	—	—	
25-44 ans	0,66	0,42 – 1,03	
45-64 ans	0,34	0,21 – 0,54	
65 ans et plus	0,25	0,15 – 0,43	
___Niveau d'études___			<0,001
Primaire	—	—	
Secondaire	2,59	1,77 – 3,83	
Technique / Professionnel	2,86	1,98 – 4,17	
Supérieur	6,63	4,55 – 9,80	
Non documenté	8,59	4,53 – 16,6	
___Rapport à la religion___			0,5
Pratiquant regulier	—	—	
Pratiquant occasionnel	0,98	0,68 – 1,42	
Appartenance sans pratique	0,99	0,71 – 1,40	
Ni croyance ni appartenance	0,81	0,55 – 1,18	
Rejet	0,68	0,39 – 1,19	
NSP ou NVPR	0,92	0,40 – 2,02	
___Heures de télévision / jour___	0,89	0,83 – 0,95	<0,001

interactions dans un prochain chapitre, cf. Chapitre **??**).
En présence d'interactions, il est conseillé d'avoir plutôt recours au type III. Cependant, en toute rigueur, pour utiliser le type III, il faut que les variables catégorielles soient codées en utilisant un contrastes dont la somme est nulle (un contrast de type somme ou polynomial). Or, par défaut, les variables catégorielles sont codées avec un contraste de type traitement (nous aborderons les différents types de contrastes plus tard, cf. Chapitre **??**).
Par défaut, `car::Anova()` utilise le type II et `gtsummary::add_global_p()` le type III. Dans les deux cas, il est possible de préciser le type de test avec `type = "II"` ou `type = "III"`.
Dans le cas de notre exemple, un modèle simple sans interaction, le type de test ne change pas les résultats.

22.9 Sélection pas à pas d'un meilleur modèle

Il est toujours tentant lorsque l'on recherche les facteurs associés à un phénomène d'inclure un nombre important de variables explicatives potentielles dans son modèle 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.

Pour une présentation didactique du cadre théorique de la sélection de modèle, vous pouvez consulter en ligne le cours de L. Rouvière sur la sélection/validation de modèles.

La technique de sélection descendante pas à pas est une approche visant à améliorer son modèle explicatif[36]. 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

[36] Il existe également des méthodes de *sélection ascendante pas à pas*, mais nous les aborderons pas ici.

peut encore améliorer le modèle et ainsi de suite. Lorsque le modèle ne peut plus être amélioré 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. Il s'agit d'un compromis entre le nombre de degrés de liberté (e.g. le nombre de coefficients dans le modèle) que l'on cherche à minimiser et la variance expliquée que l'on cherche à maximiser.

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.

```
mod2 <- step(mod)
```

```
Start:  AIC=2236.17
sport ~ sexe + groupe_ages + etudes + relig + heures.tv

              Df Deviance    AIC
- relig        5   2210.4 2230.4
<none>             2206.2 2236.2
- heures.tv    1   2219.6 2247.6
- sexe         1   2223.5 2251.5
- groupe_ages  3   2259.0 2283.0
- etudes       4   2330.0 2352.0

Step:  AIC=2230.4
sport ~ sexe + groupe_ages + etudes + heures.tv

              Df Deviance    AIC
<none>             2210.4 2230.4
- heures.tv    1   2224.0 2242.0
- sexe         1   2226.4 2244.4
- groupe_ages  3   2260.6 2274.6
- etudes       4   2334.3 2346.3
```

Le modèle initial a un AIC de 2236,2. À la première étape, il

apparait que la suppression de la variable religion permet diminuer l'AIC à 2210,4. Lors de la seconde étape, toute suppression d'une autre variable ferait augmenter l'AIC. La procédure s'arrête donc.

Pour obtenir directement l'AIC d'un modèle donné, on peut utiliser la fonction `AIC()`.

```
AIC(mod)
```

```
[1] 2236.173
```

```
AIC(mod2)
```

```
[1] 2230.404
```

On peut effectuer une analyse de variance ou ANOVA pour comparer les deux modèles avec la fonction `anova()`.

```
anova(mod, mod2, test = "Chisq")
```

```
Analysis of Deviance Table

Model 1: sport ~ sexe + groupe_ages + etudes + relig + heures.tv
Model 2: sport ~ sexe + groupe_ages + etudes + heures.tv
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1      1980     2206.2
2      1985     2210.4 -5  -4.2319   0.5165
```

Il n'y a pas de différences significatives entre nos deux modèles (p=0,52). Autrement dit, notre second modèle explique tout autant de variance que notre premier modèle, tout en étant plus parcimonieux.

> **Astuce**
>
> Une alternative à la fonction `stats::step()` est la fonction `MASS::stepAIC()` du package `{MASS}` qui fonctionne

de la même manière. Si cela ne change rien aux régressions logistiques classiques, il arrive que pour certains types de modèle la méthode `stats::step()` ne soit pas disponible, mais que `MASS::stepAIC()` puisse être utilisée à la place.

```
library(MASS)
```

```
Attachement du package : 'MASS'

L'objet suivant est masqué depuis 'package:gtsummary':

    select

L'objet suivant est masqué depuis 'package:dplyr':

    select
```

```
mod2bis <- stepAIC(mod)
```

```
Start:  AIC=2236.17
sport ~ sexe + groupe_ages + etudes + relig + heures.tv

              Df Deviance    AIC
- relig        5   2210.4 2230.4
<none>             2206.2 2236.2
- heures.tv    1   2219.6 2247.6
- sexe         1   2223.5 2251.5
- groupe_ages  3   2259.0 2283.0
- etudes       4   2330.0 2352.0

Step:  AIC=2230.4
sport ~ sexe + groupe_ages + etudes + heures.tv

              Df Deviance    AIC
<none>             2210.4 2230.4
- heures.tv    1   2224.0 2242.0
- sexe         1   2226.4 2244.4
- groupe_ages  3   2260.6 2274.6
- etudes       4   2334.3 2346.3
```

Un critère similaire à l'AIC est le critère BIC (*Bayesian Information Criterion*) appelé aussi SBC (*Schwarz information criterion*). Il s'obtient avec `stats::step()` en ajoutant l'argument `k = log(n)` où `n` est le nombre d'observations inclues dans le modèle que l'on peut obtenir avec `nrow(model.matrix(reg))` (pour tenir compte des éventuelles observations manquantes retirées des données pour le calcul du modèle).

```
mod2_bic <- step(mod, k = log(nrow(model.matrix(mod))))
```

```
Start:  AIC=2320.15
sport ~ sexe + groupe_ages + etudes + relig + heures.tv

              Df Deviance    AIC
- relig        5   2210.4 2286.4
<none>             2206.2 2320.2
- heures.tv    1   2219.6 2326.0
- sexe         1   2223.5 2329.9
- groupe_ages  3   2259.0 2350.2
- etudes       4   2330.0 2413.6

Step:  AIC=2286.39
sport ~ sexe + groupe_ages + etudes + heures.tv

              Df Deviance    AIC
<none>             2210.4 2286.4
- heures.tv    1   2224.0 2292.4
- sexe         1   2226.4 2294.8
- groupe_ages  3   2260.6 2313.8
- etudes       4   2334.3 2379.8
```

On peut facilement comparer visuellement deux modèles avec `ggstats::ggcoef_compare()` de `{ggstats}`.

```
library(ggstats)
ggcoef_compare(
  list("modèle complet" = mod, "modèle réduit" = mod2),
  exponentiate = TRUE
```

```
)
```

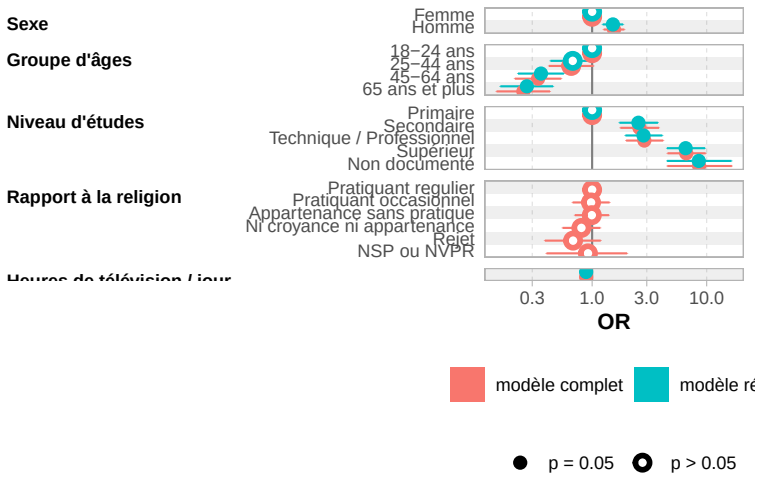


Figure 22.6: Comparaison visuelle des deux modèles (*dodge*)

```
ggcoef_compare(
  list("modèle complet" = mod, "modèle réduit" = mod2),
  type = "faceted",
  exponentiate = TRUE
)
```

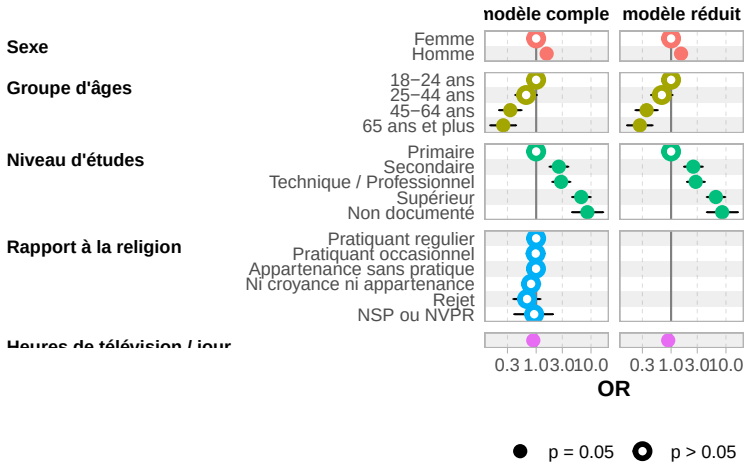


Figure 22.7: Comparaison visuelle des deux modèles (*faceted*)

Si l'on souhaite afficher l'AIC (ainsi que d'autres statistiques globales du modèle) en note du tableau des coefficients, on pourra utiliser `gtsummary::add_glance_source_note()`.

```
mod2 |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels() |>
  add_glance_source_note()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Sélection pas à pas et valeurs manquantes

Si certaines de nos variables explications contiennent des valeurs manquantes `NA`, cela peut entraîner des erreurs au moment d'avoir recours à `step()`, car le nombre d'observations dans le modèle va changer si on retire du modèle une variable explicative avec des valeurs manquantes.
Prenons un exemple, en ajoutant des valeurs manquantes

Table 22.8: Modèle obtenu après réduction du nombre de variables

Caractéristique	**OR**	**95% IC**	**p-valeur**
___Sexe___			
Femme	—	—	
Homme	1,52	1,24 – 1,87	<0,001
___Groupe d'âges___			
18-24 ans	—	—	
25-44 ans	0,68	0,43 – 1,06	0,084
45-64 ans	0,36	0,23 – 0,57	<0,001
65 ans et plus	0,27	0,16 – 0,46	<0,001
___Niveau d'études___			
Primaire	—	—	
Secondaire	2,54	1,73 – 3,75	<0,001
Technique / Professionnel	2,81	1,95 – 4,10	<0,001
Supérieur	6,55	4,50 – 9,66	<0,001
Non documenté	8,54	4,51 – 16,5	<0,001
___Heures de télévision / jour___	0,89	0,83 – 0,95	<0,001

à la variable *relig* (pour cela nous allons recoder les refus et les ne sait pas en `NA`).

```
d$relig_na <-
  d$relig |>
  fct_recode(
    NULL = "Rejet",
    NULL = "NSP ou NVPR"
  )

mod_na <- glm(
  sport ~ sexe + groupe_ages + etudes + relig_na + heures.tv,
  family = binomial,
  data = d
)
```

Au moment d'exécuter `step()` nous obtenons l'erreur mentionnée précédemment.

```
step(mod_na)
```

```
Start:  AIC=2077.34
sport ~ sexe + groupe_ages + etudes + relig_na + heures.tv

              Df Deviance    AIC
- relig_na     3   2053.8 2073.8
<none>             2051.3 2077.3
- heures.tv    1   2064.7 2088.7
- sexe         1   2068.1 2092.1
- groupe_ages  3   2098.8 2118.8
- etudes       4   2173.5 2191.5

Error in step(mod_na): le nombre de lignes utilisées a changé : supprimer les valeurs manquan
```

Pas d'inquiétude ! Il y a moyen de s'en sortir en adoptant la stratégie suivante :

1. créer une copie du jeu de données avec uniquement des observations sans valeur manquante pour nos variables explicatives ;
2. calculer notre modèle complet à partir de ce jeu de données ;
3. appliquer `step()` ;
4. recalculer le modèle réduit en repartant du jeu de données complet.

Première étape, ne garder que les observations complètes à l'aide de `tidyr::drop_na()`, en lui indiquant la liste des variables dans lesquelles vérifier la présence ou non de `NA`.

```
d_complet <- d |>
  drop_na(sexe, groupe_ages, etudes, relig_na, heures.tv)
```

Deuxième étape, calculons le modèle complet avec ce jeu données.

```r
mod_na_alt <- glm(
  sport ~ sexe + groupe_ages + etudes + relig_na + heures.tv,
  family = binomial,
  data = d_complet
)
```

Le modèle `mod_na_alt` est tout à fait identique au modèle `mod_na`, car `glm()` supprime de lui-même les valeurs manquantes quand elles existent. Nous pouvons maintenant utiliser `step()`.

```r
mod_na_reduit <- step(mod_na_alt)
```

```
Start:  AIC=2077.34
sport ~ sexe + groupe_ages + etudes + relig_na + heures.tv

              Df Deviance    AIC
- relig_na     3   2053.8 2073.8
<none>             2051.3 2077.3
- heures.tv    1   2064.7 2088.7
- sexe         1   2068.1 2092.1
- groupe_ages  3   2098.8 2118.8
- etudes       4   2173.5 2191.5

Step:  AIC=2073.8
sport ~ sexe + groupe_ages + etudes + heures.tv

              Df Deviance    AIC
<none>             2053.8 2073.8
- heures.tv    1   2067.0 2085.0
- sexe         1   2069.8 2087.8
- groupe_ages  3   2099.5 2113.5
- etudes       4   2176.7 2188.7
```

Cela s'exécute sans problème car tous les sous-modèles sont calculés à partir de `d_complet` et donc ont bien le même nombre d'observations. Cependant, dans notre modèle réduit, on a retiré 137 observations en raison d'une valeur manquante sur la variable *relig_na*, variable qui

n'est plus présente dans notre modèle réduit. Il serait donc pertinent de réintégrer ces observations.
Nous allons donc recalculer le modèle réduit mais à partir de `d`. Inutile de recopier à la main la formule du modèle réduit, car nous pouvons l'obtenir directement avec `mod_na_reduit$formula`.

```
mod_na_reduit2 <- glm(
  mod_na_reduit$formula,
  family = binomial,
  data = d
)
```

Attention : `mod_na_reduit` et `mod_na_reduit2` ne sont pas identiques puisque le second a été calculé sur un plus grand nombre d'observations, ce qui change très légèrement les valeurs des coefficients.
Pour automatiser l'ensemble de ce processus, on peut copier/coller le code de la fonction générique suivante :

```
step_with_na <- function(model, ...) {
  # refit the model without NAs
  model_no_na <- update(model, data = model.frame(model))
  # apply step()
  model_simplified <- step(model_no_na, ...)
  # recompute simplified model using full data
  update(model, formula = terms(model_simplified))
}
```

Elle réalise l'ensemble des opérations décrites plus haut en profitant de la flexibilité offerte par la fonction `update()`.
Attention : il s'agit d'une fonction expérimentale et elle n'est peut-être pas compatible avec tous les types de modèles. Elle a été testée avec les modèles `lm()`, `glm()` et `nnet::multinom()`.

```
mod_na_reduit_direct <- step_with_na(mod_na, trace = 0)
```

Le résultat obtenu est strictement identique.

```r
anova(mod_na_reduit2, mod_na_reduit_direct)
```

```
Analysis of Deviance Table

Model 1: sport ~ sexe + groupe_ages + etudes + heures.tv
Model 2: sport ~ sexe + groupe_ages + etudes + heures.tv
  Resid. Df Resid. Dev Df Deviance
1      1985     2210.4
2      1985     2210.4  0        0
```

Forcer certaines variables dans le modèle réduit

Même si l'on a recourt à `step()`, on peut vouloir forcer la présence de certaines variables dans le modèle, même si leur suppression minimiserait l'AIC, en général parce que l'on a des hypothèses spécifiques pour ces variables et que l'on peut avoir intérêt à montrer qu'elles n'ont pas d'effet dans le modèle multivarié.
Prenons un exemple :

```r
autre_modele <- glm(
  sport ~ sexe + relig + heures.tv + cinema + lecture.bd,
  family = binomial,
  data = d
)
step(autre_modele)
```

```
Start:  AIC=2350.99
sport ~ sexe + relig + heures.tv + cinema + lecture.bd

             Df Deviance    AIC
- relig       5   2333.5 2343.5
- lecture.bd  1   2332.6 2350.6
<none>            2331.0 2351.0
- sexe        1   2352.4 2370.4
- heures.tv   1   2364.1 2382.1
- cinema      1   2516.7 2534.7
```

```
Step:  AIC=2343.52
sport ~ sexe + heures.tv + cinema + lecture.bd

             Df Deviance    AIC
- lecture.bd  1   2335.1 2343.1
<none>            2333.5 2343.5
- sexe        1   2355.9 2363.9
- heures.tv   1   2367.2 2375.2
- cinema      1   2522.3 2530.3

Step:  AIC=2343.1
sport ~ sexe + heures.tv + cinema

            Df Deviance    AIC
<none>           2335.1 2343.1
- sexe       1   2357.1 2363.1
- heures.tv  1   2369.3 2375.3
- cinema     1   2526.2 2532.2


Call:  glm(formula = sport ~ sexe + heures.tv + cinema, family = binomial,
    data = d)

Coefficients:
(Intercept)    sexeHomme    heures.tv    cinemaOui
    -1.0296       0.4708      -0.1807       1.3658

Degrees of Freedom: 1994 Total (i.e. Null);  1991 Residual
  (5 observations effacées parce que manquantes)
Null Deviance:      2609
Residual Deviance: 2335     AIC: 2343
```

Ici, deux variables sont retirées du modèle par `step()` : *relig* et *lecture.bd*. Nous souhaiterions néanmoins que *lecture.bd* ne soit jamais retirée. Nous allons donc indiquer à `step()` un modèle minimum, via l'élément `lower` passé dans une liste nommée à l'argument `scope`s

```r
step(
  autre_modele,
  scope = list(lower = ~ lecture.bd)
)
```

```
Start:  AIC=2350.99
sport ~ sexe + relig + heures.tv + cinema + lecture.bd

            Df Deviance    AIC
- relig      5   2333.5 2343.5
<none>           2331.0 2351.0
- sexe       1   2352.4 2370.4
- heures.tv  1   2364.1 2382.1
- cinema     1   2516.7 2534.7

Step:  AIC=2343.52
sport ~ sexe + heures.tv + cinema + lecture.bd

            Df Deviance    AIC
<none>           2333.5 2343.5
- sexe       1   2355.9 2363.9
- heures.tv  1   2367.2 2375.2
- cinema     1   2522.3 2530.3


Call:  glm(formula = sport ~ sexe + heures.tv + cinema + lecture.bd,
    family = binomial, data = d)

Coefficients:
  (Intercept)      sexeHomme      heures.tv      cinemaOui  lecture.bdOui
      -1.0418         0.4766        -0.1795         1.3598         0.4017

Degrees of Freedom: 1994 Total (i.e. Null);  1990 Residual
  (5 observations effacées parce que manquantes)
Null Deviance:      2609
Residual Deviance: 2334     AIC: 2344
```

22.10 Régressions logistiques univariées

Les usages varient selon les disciplines et les revues scientifiques, mais il n'est pas rare de présenter, avant le modèle logistique multivarié, une succession de modèles logistiques univariés (i.e. avec une seule variable explicative à la fois) afin de présenter les *odds ratios* et leur intervalle de confiance et p-valeur associés avant l'ajustement multiniveau.

Afin d'éviter le code fastidieux consistant à réaliser chaque modèle un par un (par exemple `glm(sport ~ sexe, family = binomial, data = d)`) puis à en fusionner les résultats, on pourra tirer partie de `gtsummary::tbl_uvregression()` qui permet de réaliser toutes ces régressions individuelles en une fois et de les présenter dans un tableau synthétique.

```
d |>
  tbl_uvregression(
    y = sport,
    include = c(sexe, groupe_ages, etudes, relig, heures.tv),
    method = glm,
    method.args = list(family = binomial),
    exponentiate = TRUE
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

22.11 Présenter l'ensemble des résultats dans un même tableau

La fonction `gtsummary::tbl_merge()` permet de fusionner plusieurs tableaux (en tenant compte du nom des variables) et donc de présenter les différents résultats de l'analyse descriptive, univariée et multivariée dans un seul et même tableau.

Table 22.9: Régressions logistiques univariées

Caractéristique	**N**	**OR**	**95% IC**	**p-valeur**
___Sexe___	2 000			
Femme		—	—	
Homme		1,47	1,22 – 1,77	<0,001
___Rapport à la religion___	2 000			
Pratiquant regulier		—	—	
Pratiquant occasionnel		1,08	0,78 – 1,50	0,6
Appartenance sans pratique		1,31	0,98 – 1,78	0,071
Ni croyance ni appartenance		1,45	1,05 – 2,02	0,026
Rejet		1,19	0,72 – 1,95	0,5
NSP ou NVPR		0,93	0,44 – 1,88	0,8
___Heures de télévision / jour___	1 995	0,79	0,74 – 0,84	<0,001
___Groupe d'âges___	2 000			
18-24 ans		—	—	
25-44 ans		0,51	0,35 – 0,71	<0,001
45-64 ans		0,20	0,14 – 0,28	<0,001
65 ans et plus		0,10	0,07 – 0,15	<0,001
___Niveau d'études___	2 000			
Primaire		—	—	
Secondaire		3,61	2,52 – 5,23	<0,001
Technique / Professionnel		4,75	3,42 – 6,72	<0,001
Supérieur		11,4	8,11 – 16,3	<0,001
Non documenté		26,2	15,7 – 44,9	<0,001

```r
tbl_desc <-
  d |>
  tbl_summary(
    by = sport,
    include = c(sexe, groupe_ages, etudes, relig, heures.tv),
    statistic = all_categorical() ~ "{p}% ({n}/{N})",
    percent = "row",
    digits = all_categorical() ~ c(1, 0, 0)
  ) |>
  modify_column_hide("stat_1") |>
  modify_header("stat_2" ~ "**Pratique d'un sport**")

tbl_uni <-
  d |>
  tbl_uvregression(
    y = sport,
    include = c(sexe, groupe_ages, etudes, relig, heures.tv),
    method = glm,
    method.args = list(family = binomial),
    exponentiate = TRUE
  ) |>
  modify_column_hide("stat_n")

tbl_multi <-
  mod2 |>
  tbl_regression(exponentiate = TRUE)

list(tbl_desc, tbl_uni, tbl_multi) |>
  tbl_merge(
    tab_spanner = c(
      NA,
      "**Régressions univariées**",
      "**Régression multivariée**"
    )
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 22.10: tableau synthétique de l'analyse

Caractéristique	**Pratique d'un sport**	**OR**	**95% IC**	**p-valeur**	**OR**
___Sexe___					
Femme	32,2% (354/1 101)	—	—		—
Homme	41,0% (369/899)	1,47	1,22 – 1,77	<0,001	1,52
___Groupe d'âges___					
18-24 ans	65,7% (111/169)	—	—		—
25-44 ans	49,2% (347/706)	0,51	0,35 – 0,71	<0,001	0,68
45-64 ans	27,4% (204/745)	0,20	0,14 – 0,28	<0,001	0,36
65 ans et plus	16,1% (61/380)	0,10	0,07 – 0,15	<0,001	0,27
___Niveau d'études___					
Primaire	10,7% (50/466)	—	—		—
Secondaire	30,2% (117/387)	3,61	2,52 – 5,23	<0,001	2,54
Technique / Professionnel	36,4% (216/594)	4,75	3,42 – 6,72	<0,001	2,81
Supérieur	57,8% (255/441)	11,4	8,11 – 16,3	<0,001	6,55
Non documenté	75,9% (85/112)	26,2	15,7 – 44,9	<0,001	8,54
___Rapport à la religion___					
Pratiquant regulier	31,6% (84/266)	—	—		
Pratiquant occasionnel	33,3% (147/442)	1,08	0,78 – 1,50	0,6	
Appartenance sans pratique	37,8% (287/760)	1,31	0,98 – 1,78	0,071	
Ni croyance ni appartenance	40,1% (160/399)	1,45	1,05 – 2,02	0,026	
Rejet	35,5% (33/93)	1,19	0,72 – 1,95	0,5	
NSP ou NVPR	30,0% (12/40)	0,93	0,44 – 1,88	0,8	
___Heures de télévision / jour___	2,00 (1,00 – 3,00)	0,79	0,74 – 0,84	<0,001	0,89
Manquant	3				

Pour visualiser chaque étape du code, vous pouvez consulter le diaporama suivant : https://larmarange.github.io/guide-R/analyses/ressources/flipbook-regression-logistique.html

22.12 webin-R

La régression logistique est présentée sur YouTube dans le webin-R #06 (*régression logistique (partie 1)* et le le webin-R #07 (*régression logistique (partie 2)*.

https://youtu.be/-bdMv2aAqUY

https://youtu.be/BUo9i7XTLYQ

23 Prédictions marginales, contrastes marginaux & effets marginaux

Avertissement

Ce chapitre nécessite une version récente de `{broom.helpers}` (version 1.12.0), de `{gtsummary}` (version 1.6.3), de `{ggstats}` (version 0.2.1) et de `{marginaleffects}` (version 0.10.0).

Les coefficients d'une régression multivariée ne sont pas toujours facile à interpréter car ils ne sont pas forcément exprimés dans la même dimension que la variable d'intérêt. C'est notamment le cas pour une régression logistique binaire (cf. Chapitre **??**). Comment traduire la valeur d'un *odds ratio* en écart de probabilité ?

Dans certaines disciplines, notamment en économétrie, on préfère souvent présenter les effets marginaux qui tentent de traduire les résultats du modèle dans la dimension de la variable d'intérêt. Plusieurs approches existent et l'on trouve dans la littérature des expressions telles que effets marginaux, effets statistiques, moyennes marginales, pentes marginales, effets marginaux à la moyenne, et autres expressions similaires.

Différents auteurs peuvent utiliser la même expression pour désigner des indicateurs différents, ou bien des manières différentes de les calculer.

> **Note**
>
> Si vous n'êtes pas familier des estimations marginales et souhaitez aller à l'essentiel, vous pouvez, en première lecture, vous concentrer sur les prédications marginales moyennes et les contrastes marginaux moyens, avant d'explorer les autres variantes.

23.1 Terminologie

Dans ce guide, nous avons décidé d'adopter une terminologie consistante avec celle du package `{broom.helpers}`, elle même basée sur celle du package `{marginaleffects}`, dont la première version a été publié en septembre 2021, et avec le billet d'Andrew Heiss intitulé *Marginalia* et publié en mai 2022.

Lorsque l'on utilise un modèle ajusté pour prédire l'*outcome* selon certaines combinaisons de valeurs des régresseurs / variables explicatives, par exemple leurs valeurs observées ou leur moyenne, on obtient des **prédictions ajustées**. Lorsque ces dernières sont moyennées selon un régresseur spécifique, nous parlerons alors de **prédictions marginales**.

Les **contrastes marginaux** correspondent au calcul d'une différence entre des prédictions marginales, que ce soit pour une variable catégorielle (e.g. différence entre deux modalités) ou pour une variable continue (différence observée au niveau de l'*outcome* pour un certain changement du prédicteur).

Les **pentes marginales** ou **effets marginaux** sont définis, pour des variables continues, comme la dérivée partielle (*slope*) de l'équation de régression pour certains valeurs de la variable explicative. Dit autrement, un effet marginal correspond à la pente locale de la fonction de régression pour certaines valeurs choisies d'un prédicteur continue. De manière pratique, les effets marginaux sont similaires aux contrastes marginaux.

L'ensemble de ces indicateurs marginaux se calculent pour certaines valeurs typiques des variables explicatives, avec plusieurs approches possibles pour définir des valeurs typiques : moyenne / mode, valeurs observées, valeurs personnalisées…

Nous présenterons ces différents concepts plus en détail dans la suite de ce chapitre.

Plusieurs packages proposent des fonctions pour le calcul d'estimations marginales, `{marginaleffects}`, `{emmeans}`, `{margins}`, `{effects}`, ou encore `{ggeffects}`, chacun avec des approches et un vocabulaire légèrement différent.

Le package `{broom.helpers}` fournit plusieurs *tidiers* qui permettent d'appeler les fonctions de ces autres packages et de renvoyer un tableau de données compatible avec la fonction `broom.helpers::tidy_plus_plus()` et dès lors de pouvoir générer un tableau mis en forme avec `gtsummary::tbl_regression()` ou un graphique avec `ggstats::ggcoef_model()`.

23.2 Données d'illustration

Pour illustrer ce chapitre, nous allons reprendre le modèle réduit utilisé dans le chapitre sur la régression logistique binaire (cf. Chapitre **??**).

```
library(tidyverse)
library(labelled)
library(gtsummary)
theme_gtsummary_language(
  "fr",
  decimal.mark = ",",
  big.mark = " "
)

data(hdv2003, package = "questionr")

d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
```

```
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                   "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
    ) |>
    fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    heures.tv = "Heures de télévision / jour"
  )

mod <- glm(
  sport ~ sexe + groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
```

```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.1: Odds Ratios du modèle logistique

Caractéristique	OR	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	1,52	1,24 – 1,87	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	0,68	0,43 – 1,06	0,084
45-64 ans	0,36	0,23 – 0,57	<0,001
65 ans et plus	0,27	0,16 – 0,46	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	2,54	1,73 – 3,75	<0,001
Technique / Professionnel	2,81	1,95 – 4,10	<0,001
Supérieur	6,55	4,50 – 9,66	<0,001
Non documenté	8,54	4,51 – 16,5	<0,001
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

Il faut se rappeler que pour calculer le modèle, les observations ayant au moins une valeur manquante ont été exclues. Le modèle n'a donc pas été calculé sur 2000 observations (nombre de lignes de `hdv2003`) mais sur 1995. On peut obtenir le tableau de données du modèle (*model frame*), qui ne contient que les variables et les observations utilisées, avec `broom.helpers::model_get_model_frame()`.

```
mf <- mod %>%
  broom.helpers::model_get_model_frame()
nrow(mf)
```

```
[1] 1995
```

```
colnames(mf)
```

```
[1] "sport"       "sexe"        "groupe_ages" "etudes"      "heures.tv"
```

23.3 Prédictions marginales

23.3.1 Prédictions marginales moyennes

Pour illustrer et mieux comprendre ce que représente la différence entre les femmes et les hommes, nous allons effectuer des prédictions avec notre modèle en ne faisant varier que la variable *sexe*.

Une première approche consiste à dupliquer nos données observées et à supposer que tous les individus sont des femmes, puis à supposer que tous les individus sont des hommes.

```
mf_femmes <- mf |> mutate(sexe = "Femme")
mf_hommes <- mf |> mutate(sexe = "Homme")
```

Nos deux jeux de données sont donc identiques pour toutes les autres variables et ne varient que pour le *sexe*. Nous pouvons maintenant prédire, à partir de notre modèle ajusté, la probabilité de faire du sport de chacun des individus de ces deux nouveaux jeux de données, puis à en calculer la moyenne.

```
mod |> predict(type = "response", newdata = mf_femmes) |> mean()
```

```
[1] 0.324814
```

```
mod |> predict(type = "response", newdata = mf_hommes) |> mean()
```

```
[1] 0.4036624
```

Nous obtenons ainsi des **prédictions marginales moyennes**, *average marginal predictions* en anglais, de respectivement 32% et 40% pour les femmes et pour les hommes.

Le même résultat, avec en plus un intervalle de confiance, peut s'obtenir avec `marginaleffects::predictions()`.

```r
library(marginaleffects)
mod |>
  predictions(variables = "sexe", by = "sexe", type = "response")
```

```
  sexe Estimate Std. Error    z Pr(>|z|) 2.5 % 97.5 %
 Femme    0.325     0.0130 25.0   <0.001 0.299  0.350
 Homme    0.404     0.0147 27.5   <0.001 0.375  0.432

Columns: sexe, estimate, std.error, statistic, p.value, conf.low, conf.high
```

Pour une variable continue, on peut procéder de la même manière en générant des prédictions marginales pour certaines valeurs de la variable. Par défaut, `marginaleffects::predictions()` réalise des prédictions selon les 5 nombres de Tukey (*Tukey's five numbers*, à savoir minimum, premier quartile, médiane, troisième quartile et maximum).

```r
mod |>
  predictions(variables = "heures.tv", by = "heures.tv", type = "response")
```

```
 heures.tv Estimate Std. Error     z Pr(>|z|)  2.5 % 97.5 %
         0    0.410    0.01711 23.96   <0.001 0.3764  0.443
         1    0.386    0.01220 31.64   <0.001 0.3621  0.410
         2    0.363    0.00991 36.58   <0.001 0.3432  0.382
         3    0.340    0.01145 29.66   <0.001 0.3173  0.362
        12    0.168    0.04220  3.99   <0.001 0.0855  0.251

Columns: heures.tv, estimate, std.error, statistic, p.value, conf.low, conf.high
```

Le package {broom.helpers} fournit la fonction `broom.helpers::tidy_marginal_predictions()` qui génèrent les prédictions marginales de chaque variable[37] avec `marginaleffects::predictions()` et renvoie les résultat dans un format directement utilisable avec `gtsummary::tbl_regression()`.

[37] La fonction `broom.helpers::tidy_marginal_predictions()` peut également gérer des combinaisons de variables ou interactions, voir Chapitre **??**).

> **Note**
>
> Il est à noter que `broom.helpers::tidy_marginal_predictions()` renvoie des p-valeurs qui, par défaut, teste si les valeurs prédites sont différentes de 0 (sur l'échelle de la fonction de lien, donc différentes de 50% dans le cas d'une régression logistique). Ce type de tests n'est pas vraiment pertinent dans le cas présent. On peut facilement masquer la colonne des p-valeurs avec `modify_column_hide("p.value")`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    type = "response",
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
  bold_labels() |>
  modify_column_hide("p.value")
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.2: Prédictions marginales moyennes

Caractéristique	Average Marginal Predictions	95% IC
Sexe		
Femme	32.5%	29.9% – 35.0%
Homme	40.4%	37.5% – 43.2%
Groupe d'âges		
18-24 ans	51.2%	42.2% – 60.1%
25-44 ans	42.7%	39.3% – 46.2%

Caractéristique	Average Marginal Predictions	95% IC
45-64 ans	29.9%	26.6% – 33.2%
65 ans et plus	24.9%	19.7% – 30.0%
Niveau d'études		
Primaire	16.1%	11.9% – 20.4%
Secondaire	31.8%	27.2% – 36.4%
Technique / Professionnel	34.0%	30.3% – 37.7%
Supérieur	53.2%	48.4% – 57.9%
Non documenté	59.2%	47.0% – 71.5%
Heures de télévision / jour		
0	41.0%	37.6% – 44.3%
1	38.6%	36.2% – 41.0%
2	36.3%	34.3% – 38.2%
3	34.0%	31.7% – 36.2%
12	16.8%	8.6% – 25.1%

La fonction `broom.helpers::plot_marginal_predictions()` permet de visualiser les prédictions marginales à la moyenne en réalisant une liste de graphiques, un par variable, que nous pouvons combiner avec `patchwork::wrap_plots()`. L'opérateur `&` permet d'appliquer une fonction de `{ggplot2}` à chaque sous-graphique. Ici, nous allons uniformiser l'axe des y.

```
p <- mod |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
```

```
  patchwork::wrap_plots() &
  scale_y_continuous(
    limits = c(0, .8),
    labels = scales::label_percent()
  )
```

```
p
```

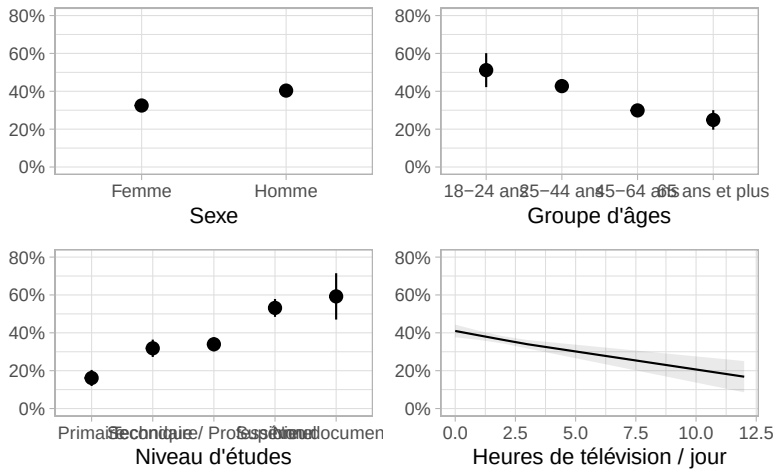


Figure 23.1: Prédictions marginales moyennes

Il est ici difficile de lire les étiquettes de la variable *etudes*. Nous pouvons éventuellement inverser l'axe des x et celui des y avec `ggplot2::coord_flip()`.

```
p & coord_flip()
```

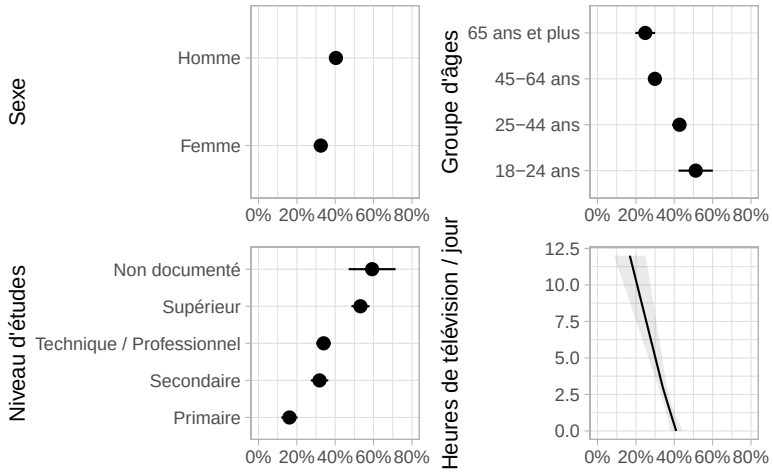


Figure 23.2: Prédictions marginales moyennes

Une alternative possible avec d'avoir recours à `ggtstats::ggcoef_model()`.

```
mod |>
  ggstats::ggcoef_model(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    tidy_args = list(type = "response"),
    show_p_values = FALSE,
    signif_stars = FALSE,
    significance = NULL,
    vline = FALSE
  ) +
  scale_x_continuous(labels = scales::label_percent())
```

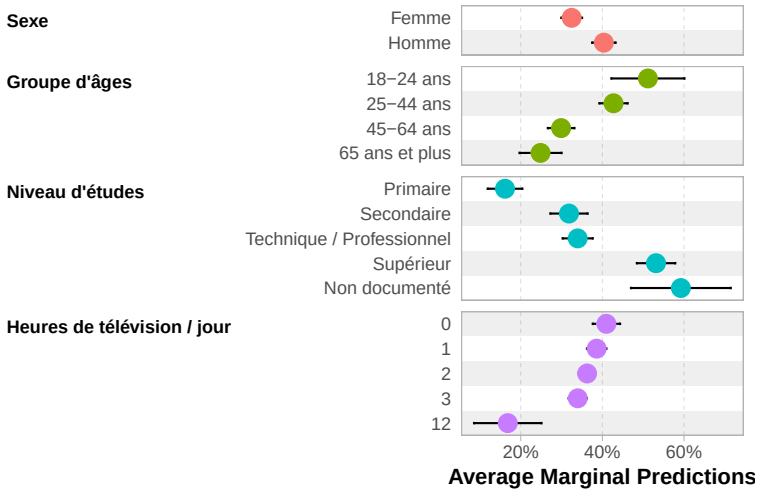


Figure 23.3: Prédictions marginales moyennes

! Importance de l'argument `type` pour les modèles `glm`

Lorsque l'on a recours à des modèles calculés avec `glm()`, il est possible de réaliser des prédictions selon deux échelles : l'échelle de notre *outcome* ou variable à expliquer (`type = "response"`), ici exprimée en probabilités ou proportions puisqu'il s'agit d'une régression logistique, ou bien selon l'échelle de la fonction de lien (`type = "link"`) du modèle, ici la fonction *logit* (voir Section **??**).
Avec l'option `type = "reponse"`, on indique à `{marginaleffects}` de calculer pour chaque individu une prédiction selon l'échelle de l'outcome puis de procéder à la moyenne, ce que nous avons fait dans les exemples précédents.
Si nous avions indiqué `type = "link"`, les prédictions auraient été faites selon l'échelle de la fonction de lien avant d'être moyennée.

```
mod |> predict(type = "link", newdata = mf_femmes) |> mean()
```

```
[1] -0.910525
```

```
  mod |> predict(type = "link", newdata = mf_hommes) |> mean()
```

```
[1] -0.4928844
```

```
  mod |>
    predictions(variables = "sexe", by = "sexe", type = "link")
```

```
  sexe Estimate Std. Error      z Pr(>|z|)  2.5 % 97.5 %
 Femme   -0.911     0.0751 -12.13   <0.001 -1.058 -0.763
 Homme   -0.493     0.0779  -6.33   <0.001 -0.646 -0.340

Columns: sexe, estimate, std.error, statistic, p.value, conf.low, conf.high
```

Depuis la version 0.10.0 de {marginaleffects}, si l'on ne précise pas le paramètre type (i.e. si type = NULL), la fonction marginaleffects::predictions() réalise les prédictions selon l'échelle de la fonction de lien, calcule les moyennes puis re-transforme ce résultat selon l'échelle de la variable à expliquer.

```
  logit_inverse <- binomial("logit") |> purrr::pluck("linkinv")
  mod |> predict(type = "link", newdata = mf_femmes) |> mean() |> logit_inverse()
```

```
[1] 0.2868924
```

```
  mod |> predict(type = "link", newdata = mf_hommes) |> mean() |> logit_inverse()
```

```
[1] 0.3792143
```

```
  mod |>
    predictions(variables = "sexe", by = "sexe")
```

```
  sexe Estimate Pr(>|z|) 2.5 % 97.5 %
 Femme    0.287   <0.001 0.258  0.318
 Homme    0.379   <0.001 0.344  0.416

Columns: sexe, estimate, p.value, conf.low, conf.high
```

Or, la plupart du temps, le logit inverse de la moyenne des prédictions est différent de la moyenne des logit inverse des prédictions !
Les résultats seront similaires et du même ordre de grandeur, mais pas identiques.

23.3.2 Prédictions marginales à la moyenne

Pour les prédictions marginales moyennes, nous avons réalisé des prédictions pour chaque observations du tableau d'origine, en faisant varier juste une variable à la fois, avant de calculer la moyenne des prédictions.

Une alternative consiste à générer une sorte d'individu moyen / typique puis à réaliser des prédictions pour cette unique individu, en faisant juste varier la variable explicative d'intérêt. On parle alors de **prédictions marginales à la moyenne**, *marginal predictions at the mean* en anglais.

23.3.2.1 avec `{marginaleffects}`

On peut réaliser cela avec `{marginaleffects}` en précisant `newdata = "mean"`. Prenons un exemple pour la variable *sexe* :

```
mod |> predictions(variables = "sexe", newdata = "mean")
```

```
 Estimate Pr(>|z|) 2.5 % 97.5 %
    0.239   <0.001 0.196  0.289
    0.323   <0.001 0.273  0.378

Columns: rowid, rowidcf, estimate, p.value, conf.low, conf.high, sport, groupe_ages, etudes, he
```

Dans ce cas de figure, `{marginaleffects}` considère pour chaque variable continue sa moyenne (ici `2.246` pour *heures.tv*) et pour chaque variable catégorielle son mode (la valeur observée la plus fréquente, ici `"Technique / Professionnel"`

pour la variable *etudes*). On fait juste varier les modalités de *sexe* puis on calculer la probabilité de faire du sport de ces individus moyens.

On peut également passer le paramètre `newdata = "mean"` à `broom.helpers::tidy_marginal_predictions()` ou même à `gtsummary::tbl_regression()`[38].

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    newdata = "mean",
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
  bold_labels() |>
  modify_column_hide("p.value")
```

[38] Les paramètres additionnels indiqués à `gtsummary::tbl_regression()` sont transmis en cascade à `broom.helpers::tidy_plus_plus()` puis à `broom.helpers::tidy_marginal_predictions()` et enfin à `marginaleffects::predictions()`.

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.3: Prédictions marginales à la moyenne

Caractéristique	Marginal Predictions at the Mean	95% IC
Sexe		
Femme	23.9%	19.6% – 28.9%
Homme	32.3%	27.3% – 37.8%
Groupe d'âges		
18-24 ans	46.8%	36.1% – 57.8%
25-44 ans	37.3%	32.1% – 42.8%
45-64 ans	23.9%	19.6% – 28.9%
65 ans et plus	19.2%	14.0% – 25.6%

Caractéristique	Marginal Predictions at the Mean	95% IC
Niveau d'études		
Primaire	10.1%	7.3% – 13.7%
Secondaire	22.1%	17.7% – 27.2%
Technique / Professionnel	23.9%	19.6% – 28.9%
Supérieur	42.3%	36.0% – 48.9%
Non documenté	48.9%	34.7% – 63.2%
Heures de télévision / jour		
0	29.2%	23.6% – 35.5%
1	26.8%	21.9% – 32.3%
2	24.5%	20.1% – 29.5%
3	22.3%	18.1% – 27.2%
12	8.8%	4.6% – 16.4%

De même, on peut générer une représentation graphique :

```
p <- mod |>
  broom.helpers::plot_marginal_predictions(newdata = "mean") |>
  patchwork::wrap_plots() &
  scale_y_continuous(
    limits = c(0, .8),
    labels = scales::label_percent()
  ) &
  coord_flip()
```

```
p
```

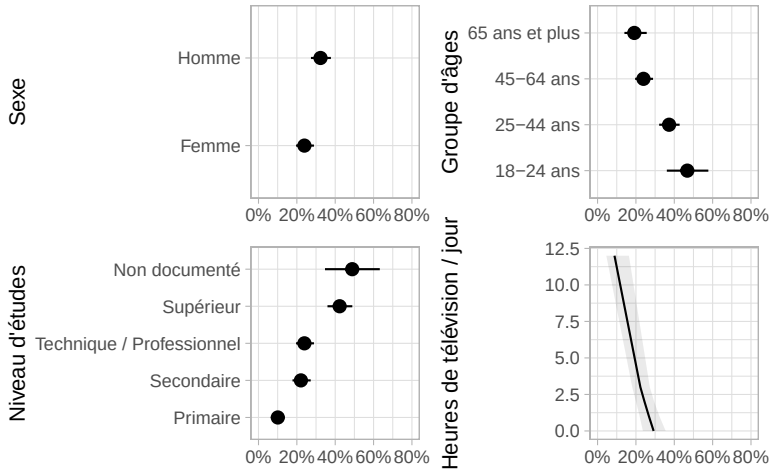


Figure 23.4: Prédictions marginales à la moyenne

Si l'on souhaite utiliser `ggstats::ggcoef_model()`, on peut directement indiquer `newdata = "mean"`. Il faudra passer cette option via `tidy_args` qui prend une liste d'arguments à transmettre à `tidy_fun`.

```
mod |>
  ggstats::ggcoef_model(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    tidy_args = list(newdata = "mean"),
    show_p_values = FALSE,
    signif_stars = FALSE,
    significance = NULL,
    vline = FALSE
  ) +
  scale_x_continuous(labels = scales::label_percent())
```

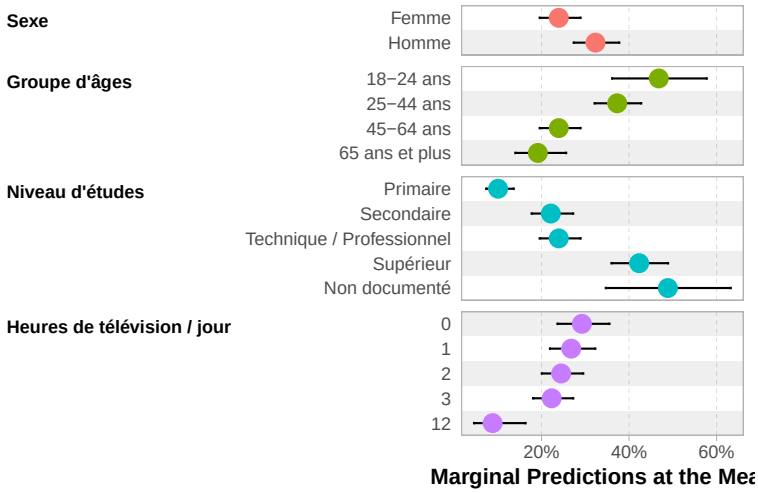


Figure 23.5: Prédictions marginales à la moyenne

23.3.2.2 avec `{effects}`

Le package `{effects}`[39] adopte une approche un peu différente pour définir un individu moyen.

Calculons les prédictions marginales à la moyenne avec la fonction `effects::Effect()`.

```
e <- effects::Effect("sexe", mod)
e
```

```
 sexe effect
sexe
    Femme     Homme
0.2868924 0.3792143
```

On le voit, les résultats sont là encore assez proches mais différents. Regardons de plus près les données utilisées pour les prédictions.

```
e$model.matrix
```

[39] Malgré son nom, le package `{effects}` ne calcule pas des effets marginaux mais des prédictions marginales, selon la terminologie retenue au début de ce document.

```
  (Intercept) sexeHomme groupe_ages25-44 ans groupe_ages45-64 ans
1           1         0            0.3533835            0.3719298
2           1         1            0.3533835            0.3719298
  groupe_ages65 ans et plus etudesSecondaire etudesTechnique / Professionnel
1                 0.1904762         0.193985                        0.2962406
2                 0.1904762         0.193985                        0.2962406
  etudesSupérieur etudesNon documenté heures.tv
1       0.2205514          0.05614035  2.246566
2       0.2205514          0.05614035  2.246566
attr(,"assign")
 [1] 0 1 2 2 2 3 3 3 3 4
attr(,"contrasts")
attr(,"contrasts")$sexe
[1] "contr.treatment"

attr(,"contrasts")$groupe_ages
[1] "contr.treatment"

attr(,"contrasts")$etudes
[1] "contr.treatment"
```

Pour les variables continues, `{effects}` utilise la moyenne observée de la variable, comme précédemment avec `{marginaleffects}`. Par contre, pour les variables catégorielles, ce n'est pas le mode qui est utilisé, mais l'ensemble des modalités, pondérées selon leur proportion observée dans l'échantillon. Cette approche a l'avantage de moyenniser également les variables catégorielles, même si les indvidus pour lesquels une prédiction est réalisée sont complètement fictifs.

On peut utiliser `broom.helpers::tidy_all_effects()` pour générer un tableau de prédictions marginales avec `{effects}`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_all_effects,
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.4: Prédictions marginales à la moyenne avec le package effects

Caractéristique	Marginal Predictions at the Mean	95% IC
Sexe		
Femme	28.7%	25.8% – 31.8%
Homme	37.9%	34.4% – 41.6%
Groupe d'âges		
18-24 ans	51.2%	41.0% – 61.3%
25-44 ans	41.5%	37.4% – 45.7%
45-64 ans	27.3%	23.9% – 30.9%
65 ans et plus	22.0%	17.4% – 27.5%
Niveau d'études		
Primaire	14.9%	11.3% – 19.3%
Secondaire	30.7%	26.2% – 35.7%
Technique / Professionnel	32.9%	29.1% – 37.0%
Supérieur	53.4%	48.3% – 58.4%
Non documenté	59.9%	46.6% – 71.8%
Heures de télévision / jour		
0	38.9%	34.8% – 43.2%
3	30.7%	28.1% – 33.4%

Caractéristique	Marginal Predictions at the Mean	95% IC
6	23.6%	18.9% – 28.9%
9	17.7%	11.9% – 25.4%
10	16.0%	10.1% – 24.4%

Pour une représentation graphique, nous pouvons utiliser les fonctions internes d'`{effects}` en appliquant `plot()` aux résultats de `effects::allEffects()` qui calcule les prédictions marginales de chaque variable.

```
mod |>
  effects::allEffects() |>
  plot()
```

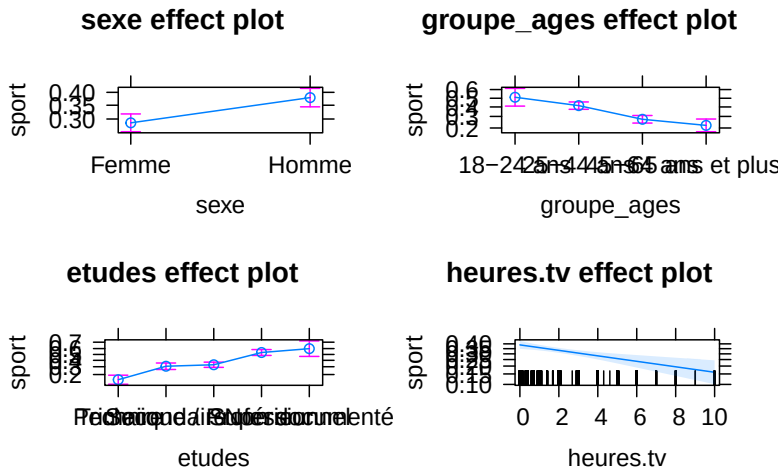


Figure 23.6: Prédictions marginales à la moyenne avec le package effects

On peut aussi utiliser `ggstats::ggcoef_model()`[40].

[40] De manière générale, `ggstats::ggcoef_model()` est compatible avec les mêmes `tidy_fun` que `gtsummary::tbl_regression()`, les deux fonctions utilisant en interne `broom.helpers::tidy_plus_plus()`.

```
mod |>
  ggstats::ggcoef_model(
    tidy_fun = broom.helpers::tidy_all_effects,
    vline = FALSE
  ) +
  scale_x_continuous(labels = scales::label_percent())
```

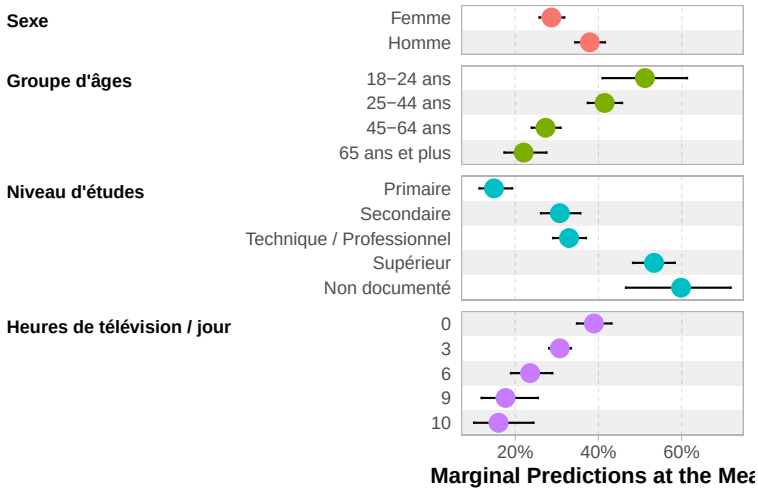


Figure 23.7: Prédictions marginales à la moyenne avec le package effects

23.3.3 Variantes

Le package {ggeffects} propose une fonction `ggeffects::ggpredict()` qui calcule des prédictions marginales à la moyenne des variables continues et à la première modalité (utilisée comme référence) des variables catégorielles. On ne peut donc plus, au sens strict, parler de prédictions à la moyenne. {broom.helpers} fournit une fonction `tidy_ggpredict()`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_ggpredict,
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
```

```
    bold_labels()
```

```
Data were 'prettified'. Consider using `terms="heures.tv [all]"` to get
  smooth plots.
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.5: Prédictions marginales avec ggpredict()

Caractéristique	Marginal Predictions	95% IC
Sexe		
Femme	23.8%	15.3% – 35.1%
Homme	32.2%	21.4% – 45.4%
Groupe d'âges		
18-24 ans	23.8%	15.3% – 35.1%
25-44 ans	17.5%	12.7% – 23.5%
45-64 ans	10.1%	7.3% – 13.7%
65 ans et plus	7.8%	5.5% – 10.9%
Niveau d'études		
Primaire	23.8%	15.3% – 35.1%
Secondaire	44.2%	33.0% – 56.1%
Technique / Professionnel	46.8%	36.1% – 57.8%
Supérieur	67.2%	56.2% – 76.6%
Non documenté	72.8%	62.8% – 80.9%

Caractéristique	Marginal Predictions	95% IC
Heures de télévision / jour		
0	29.1%	18.7% – 42.3%
1	26.7%	17.2% – 38.9%
2	24.4%	15.7% – 35.8%
3	22.2%	14.2% – 33.0%
4	20.2%	12.8% – 30.4%
5	18.3%	11.4% – 28.2%
6	16.6%	10.0% – 26.1%
7	15.0%	8.8% – 24.3%
8	13.5%	7.7% – 22.7%
9	12.1%	6.6% – 21.2%
10	10.9%	5.7% – 19.9%
11	9.8%	4.9% – 18.7%
12	8.8%	4.2% – 17.6%

Pour une représentation graphique, on peut utiliser les fonctionnalités natives inclues dans le package `{ggeffects}`.

```
mod |>
  ggeffects::ggpredict() |>
  plot() |>
  patchwork::wrap_plots()
```

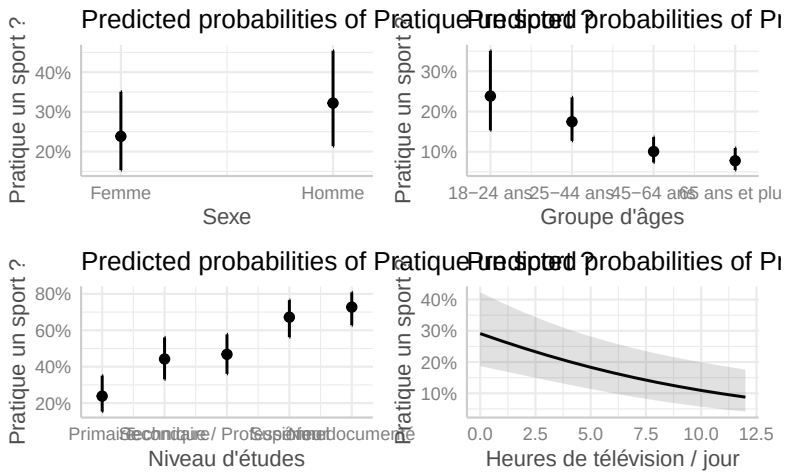


Figure 23.8: Prédictions marginales avec ggpredict()

23.4 Contrastes marginaux

Maintenant que nous savons estimer des prédictions marginales, nous pouvons facilement calculer des **contrastes marginaux**, à savoir des différences entre prédictions marginales.

23.4.1 Contrastes marginaux moyens

Considérons tout d'abord la variable catégorielle *sexe* et calculons les prédictions marginales moyennes avec `marginaleffects::predictions()`.

```
pred <- predictions(mod, variables = "sexe", by = "sexe", type = "response")
pred
```

```
  sexe Estimate Std. Error    z Pr(>|z|) 2.5 % 97.5 %
 Femme    0.325     0.0130 25.0   <0.001 0.299  0.350
 Homme    0.404     0.0147 27.5   <0.001 0.375  0.432

Columns: sexe, estimate, std.error, statistic, p.value, conf.low, conf.high
```

Le contraste entre les hommes et les femmes est tout simplement la différence et les deux prédictions marginales.

```
pred$estimate[2] - pred$estimate[1]
```

```
[1] 0.07884839
```

La fonction `marginaleffects::avg_comparisons()` permet de réaliser directement ce calcul.

```
avg_comparisons(mod, variables = "sexe")
```

```
 Term      Contrast Estimate Std. Error z Pr(>|z|)  2.5 % 97.5 %
 sexe Homme - Femme   0.0788     0.0197 4   <0.001 0.0402  0.117

Columns: term, contrast, estimate, std.error, statistic, p.value, conf.low, conf.high
```

> **Astuce**
>
> Dans les faits, `marginaleffects::avg_comparisons()` a calculé la différence entre les hommes et les femmes pour chaque observation d'origine puis a réalisé la moyenne des différences. Mathématiquement, la moyenne des différences est équivalente à la différence des moyennes.

Les contrastes calculés ici ont été moyennés sur l'ensemble des valeurs observées. On parle donc de **contrastes marginaux moyens** (*average marginal contrasts*).

Par défaut, chaque modalité est contrastée avec la première modalité prise comme référence (voir exemple ci-dessous avec la variable *groupe_ages*.

Regardons maintenant une variable continue.

```
avg_comparisons(mod, variables = "heures.tv")
```

```
     Term Contrast Estimate Std. Error     z Pr(>|z|)   2.5 %  97.5 %
 heures.tv       +1  -0.0227     0.0062 -3.66   <0.001 -0.0348 -0.0105

Columns: term, contrast, estimate, std.error, statistic, p.value, conf.low, conf.high
```

Par défaut, `marginaleffects::avg_comparisons()` calcule, pour chaque valeur observée de *heures.tv*, l'effet sur la probabilité de pratiquer un sport d'augmenter de 1 le nombre d'heures quotidiennes de télévision (plus précisément la différence des valeurs prédites pour la valeur observée plus 0,5 et la valeur observée moins 0,5).

On peut facilement obtenir la liste des contrastes marginaux pour l'ensemble des variables.

```
avg_comparisons(mod)
```

```
       Term                              Contrast Estimate Std. Error     z
 etudes      Non documenté - Primaire                0.4309     0.0691  6.23
 etudes      Secondaire - Primaire                   0.1568     0.0314  4.99
 etudes      Supérieur - Primaire                    0.3701     0.0337 10.98
 etudes      Technique / Professionnel - Primaire    0.1781     0.0295  6.04
 groupe_ages 25-44 ans - 18-24 ans                  -0.0844     0.0492 -1.71
 groupe_ages 45-64 ans - 18-24 ans                  -0.2127     0.0507 -4.20
 groupe_ages 65 ans et plus - 18-24 ans             -0.2631     0.0556 -4.73
 heures.tv   +1                                     -0.0227     0.0062 -3.66
 sexe        Homme - Femme                           0.0788     0.0197  4.00
 Pr(>|z|)   2.5 %  97.5 %
   <0.001  0.2954  0.5665
   <0.001  0.0952  0.2184
   <0.001  0.3040  0.4361
   <0.001  0.1203  0.2359
   0.0865 -0.1808  0.0121
   <0.001 -0.3121 -0.1133
   <0.001 -0.3720 -0.1541
   <0.001 -0.0348 -0.0105
   <0.001  0.0402  0.1175

Columns: term, contrast, estimate, std.error, statistic, p.value, conf.low, conf.high
```

Il est important de noter que le nom des colonnes n'est pas compatible avec les fonctions de `{broom.helpers}` et par extension avec `gtsummary::tbl_regression()` et `ggstats::ggcoef_model()`. On utilisera donc `broom.helpers::tidy_marginal_contrasts()` qui remets en forme le tableau de résultats dans un format compatible. On pourra ainsi produire un tableau propre des résultats[42].

[41] Il existe également la fonction `broom.helpers::tidy_avg_comparisons()` mais on lui préférera `broom.helpers::tidy_marginal_contrasts()`. Pour un modèle sans interaction, les résultats sont identiques. Mais `broom.helpers::tidy_marginal_contrasts()` peut gérer des termes d'interactions, ce qui sera utile dans un prochain chapitre (cf. Chapitre **??**).

[42] Notez l'utilisation de `style_positive = "plus"` dans l'appel de `scales::label_percent()` pour ajouter un signe `+` devant les valeurs positives, afin de bien indiquer que l'on représente le résultat d'une différence.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_contrasts,
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus"
    )
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.6: Contrastes marginaux moyens

Caractéristique	Average Marginal Contrasts	95% IC	p-valeur
Sexe			
Homme - Femme	+7.9%	+4.0% – +11.7%	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-8.4%	-18.1% – +1.2%	0,086
45-64 ans - 18-24 ans	-21.3%	-31.2% – -11.3%	<0,001
65 ans et plus - 18-24 ans	-26.3%	-37.2% – -15.4%	<0,001
Niveau d'études			

Caractéristique	Average Marginal Contrasts	95% IC	p-valeur
Non documenté - Primaire	+43.1%	+29.5% – +56.6%	<0,001
Secondaire - Primaire	+15.7%	+9.5% – +21.8%	<0,001
Supérieur - Primaire	+37.0%	+30.4% – +43.6%	<0,001
Technique / Professionnel - Primaire	+17.8%	+12.0% – +23.6%	<0,001
Heures de télévision / jour			
+1	-2.3%	-3.5% – -1.1%	<0,001

De même, on peut représenter les contrastes marginaux moyens avec `ggstats::ggcoef_model()`.

```
ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_marginal_contrasts
) +
  ggplot2::scale_x_continuous(
    labels = scales::label_percent(style_positive = "plus")
  )
```

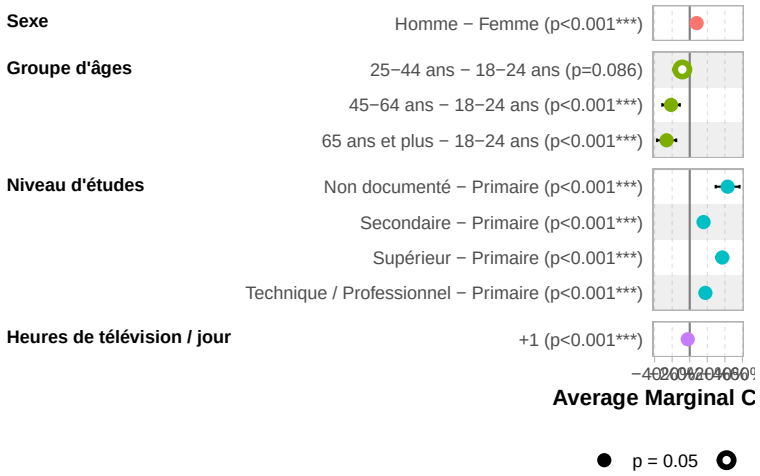


Figure 23.9: Contrastes marginaux moyens

Astuce

Il est possible de personnaliser le type de contrastes calculés, variable par variable, avec l'option `variables_list` de `broom.helpers::tidy_marginal_contrasts()`. La syntaxe est un peu particulière : il faut transmettre une liste de listes.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_contrasts,
    variables_list = list(
      list(heures.tv = 2),
      list(groupe_ages = "pairwise"),
      list(etudes = "sequential")
    ),
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus"
    )
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Caractéristique	Average Marginal Contrasts	95% IC	p-valeur
Heures de télévision / jour			
+2	-4.5%	-7.0% – -2.1%	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-8.4%	-18.1% – +1.2%	0,086
45-64 ans - 18-24 ans	-21.3%	-31.2% – -11.3%	<0,001
45-64 ans - 25-44 ans	-12.8%	-17.6% – -8.1%	<0,001
65 ans et plus - 18-24 ans	-26.3%	-37.2% – -15.4%	<0,001
65 ans et plus - 25-44 ans	-17.9%	-24.3% – -11.4%	<0,001
65 ans et plus - 45-64 ans	-5.0%	-11.0% – +0.9%	0,10
Niveau d'études			
Non documenté - Supérieur	+6.1%	-7.0% – +19.2%	0,4
Secondaire - Primaire	+15.7%	+9.5% – +21.8%	<0,001
Supérieur - Technique / Professionnel	+19.2%	+13.3% – +25.1%	<0,001
Technique / Professionnel - Secondaire	+2.1%	-3.8% – +8.0%	0,5

On peut obtenir le même résultat avec `broom.helpers::tidy_avg_comparison()` avec une syntaxe un peu plus simple (en passant une liste via `variables` au lieu d'une liste de listes via `variables_list`).

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_avg_comparisons,
    variables = list(
      heures.tv = 2,
      groupe_ages = "pairwise",
      etudes = "sequential"
    ),
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus"
    )
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Caractéristique	Average Marginal Contrasts	95% IC	p-valeur
Niveau d'études			
Non documenté - Supérieur	+6.1%	-7.0% – +19.2%	0,4
Secondaire - Primaire	+15.7%	+9.5% – +21.8%	<0,001
Supérieur - Technique / Professionnel	+19.2%	+13.3% – +25.1%	<0,001
Technique / Professionnel - Secondaire	+2.1%	-3.8% – +8.0%	0,5

Groupe d'âges			
25-44 ans - 18-24 ans	-8.4%	-18.1% – +1.2%	0,086
45-64 ans - 18-24 ans	-21.3%	-31.2% – -11.3%	<0,001
45-64 ans - 25-44 ans	-12.8%	-17.6% – -8.1%	<0,001
65 ans et plus - 18-24 ans	-26.3%	-37.2% – -15.4%	<0,001
65 ans et plus - 25-44 ans	-17.9%	-24.3% – -11.4%	<0,001
65 ans et plus - 45-64 ans	-5.0%	-11.0% – +0.9%	0,10
Heures de télévision / jour			
+2	-4.5%	-7.0% – -2.1%	<0,001

23.4.2 Contrastes marginaux à la moyenne

Comme précédemment, plutôt que de calculer les contrastes marginaux pour chaque individu observé avant de faire la moyenne des résultats, une approche alternative consiste à considérer un individu moyen / typique et à calculer les contrastes marginaux pour cet individu. On parle alors de **contrastes marginaux à la moyenne** (*marginal contrasts at the mean*).

Avec `{marginaleffects}`, il suffit de spécifier `newdata = "mean"`. Les variables continues seront fixées à leur moyenne et les variables catégorielles à leur mode (modalité la plus fréquente dans l'échantillon).

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_contrasts,
    newdata = "mean",
    estimate_fun = scales::label_percent(
```

```
      accuracy = 0.1,
      style_positive = "plus"
    )
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.9: Contrastes marginaux à la moyenne

Caractéristique	Marginal Contrasts at the Mean	95% IC	p-valeur
Sexe			
Homme - Femme	+8.4%	+4.3% – +12.5%	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-9.5%	-20.5% – +1.5%	0,090
45-64 ans - 18-24 ans	-22.9%	-33.8% – -11.9%	<0,001
65 ans et plus - 18-24 ans	-27.6%	-39.2% – -16.1%	<0,001
Niveau d'études			
Non documenté - Primaire	+38.8%	+24.1% – +53.5%	<0,001
Secondaire - Primaire	+12.0%	+7.0% – +17.1%	<0,001
Supérieur - Primaire	+32.2%	+25.7% – +38.7%	<0,001
Technique / Professionnel - Primaire	+13.9%	+9.0% – +18.7%	<0,001
Heures de télévision / jour			

Caractéristique	Marginal Contrasts at the Mean	95% IC	p-valeur
+1	-2.2%	-3.4% – -1.0%	<0,001

Pour la fonction `ggstats::ggcoef_model()`, on utilisera l'argument `tidy_args` pour transmettre l'option `newdata = "mean"`.

```
ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_marginal_contrasts,
  tidy_args = list(newdata = "mean")
) +
  ggplot2::scale_x_continuous(
    labels = scales::label_percent(style_positive = "plus")
  )
```

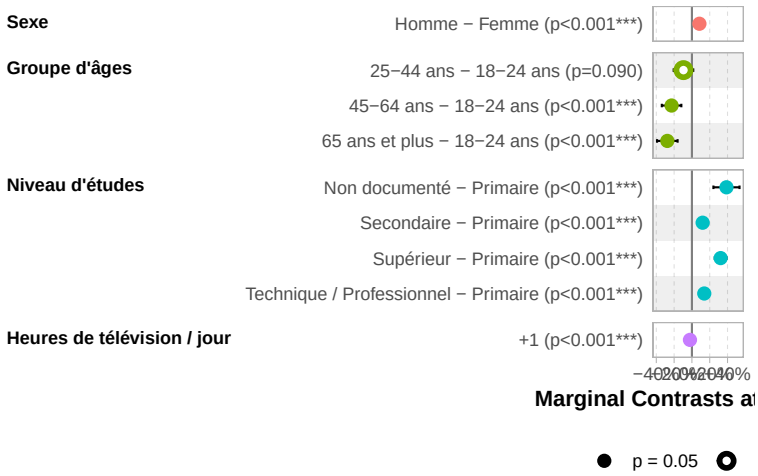


Figure 23.10: Contrastes marginaux à la moyenne

23.5 Pentes marginales / Effets marginaux

Les effets marginaux, ou plus précisément les pentes marginales, sont similaires aux contrastes marginaux, avec un différence subtile. Pour une variable continue, les contrastes marginaux sont une différence entre deux prédictions tandis que les **effets marginaux** (*marginal effects*) ou **pentes marginales** (*marginal slopes*). Dis autrement, l'effect marginal d'un régresseur continu x est la pente / dérivée $\partial y/\partial x$ la fonction de prédiction y, mesurée à des valeurs spécifiques de x.

Les effets marginaux sont le plus souvent calculés selon l'échelle de l'*outcome* et représentent le changement attendu de l'*outcome* pour une augmentation du régresseur d'une unité.

Par définition, les effets marginaux ne sont pas définis pour les variables catégorielles. La plupart des fonctions rapportent, à la place, les contrastes marginaux pour ces variables catégorielles.

Comme pour les prédictions marginales et les contrastes marginaux, plusieurs approches existent (voir par exemple la vignette dédiée du package `{marginaleffects}`).

23.5.1 Pentes marginales moyennes / Effets marginaux moyens

Les **effets marginaux moyens** (*average marginal effects*) sont calculés en deux temps : (1) un effet marginal est calculé pour chaque individu observé dans le modèle ; (ii) puis la moyenne de ces effets individuels est calculée.

On aura tout simplement recours à la fonction `marginaleffects::avg_slopes()`.

```
avg_slopes(mod)
```

```
      Term                          Contrast Estimate Std. Error     z
etudes     Non documenté - Primaire              0.4309     0.0691  6.23
etudes     Secondaire - Primaire                 0.1568     0.0314  4.99
etudes     Supérieur - Primaire                  0.3701     0.0337 10.98
```

```
etudes      Technique / Professionnel - Primaire   0.1781     0.0295  6.04
groupe_ages 25-44 ans - 18-24 ans                 -0.0844     0.0492 -1.71
groupe_ages 45-64 ans - 18-24 ans                 -0.2127     0.0507 -4.20
groupe_ages 65 ans et plus - 18-24 ans            -0.2631     0.0556 -4.73
heures.tv   dY/dX                                 -0.0227     0.0062 -3.66
sexe        Homme - Femme                          0.0788     0.0197  4.00
Pr(>|z|)   2.5 %  97.5 %
  <0.001  0.2954  0.5665
  <0.001  0.0952  0.2184
  <0.001  0.3040  0.4361
  <0.001  0.1203  0.2359
  0.0865 -0.1808  0.0121
  <0.001 -0.3121 -0.1133
  <0.001 -0.3720 -0.1541
  <0.001 -0.0348 -0.0105
  <0.001  0.0402  0.1175

Columns: term, contrast, estimate, std.error, statistic, p.value, conf.low, conf.high
```

Pour un usage avec `broom.helpers::tidy_plus_plus()`, `gtsummary::tbl_regression()` ou `ggstats::ggcoef_model()`, on utilisera `broom.helpers::tidy_avg_slopes()`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_avg_slopes,
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus"
    )
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.10: Effets marginaux moyens

Caractéristique	Average Marginal Effects	95% IC	p-valeur
Niveau d'études			
Non documenté - Primaire	+43.1%	+29.5% – +56.6%	<0,001
Secondaire - Primaire	+15.7%	+9.5% – +21.8%	<0,001
Supérieur - Primaire	+37.0%	+30.4% – +43.6%	<0,001
Technique / Professionnel - Primaire	+17.8%	+12.0% – +23.6%	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-8.4%	-18.1% – +1.2%	0,086
45-64 ans - 18-24 ans	-21.3%	-31.2% – -11.3%	<0,001
65 ans et plus - 18-24 ans	-26.3%	-37.2% – -15.4%	<0,001
Heures de télévision / jour			
dY/dX	-2.3%	-3.5% – -1.1%	<0,001
Sexe			
Homme - Femme	+7.9%	+4.0% – +11.7%	<0,001

```
ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_avg_slopes
) +
  ggplot2::scale_x_continuous(
    labels = scales::label_percent(style_positive = "plus")
  )
```

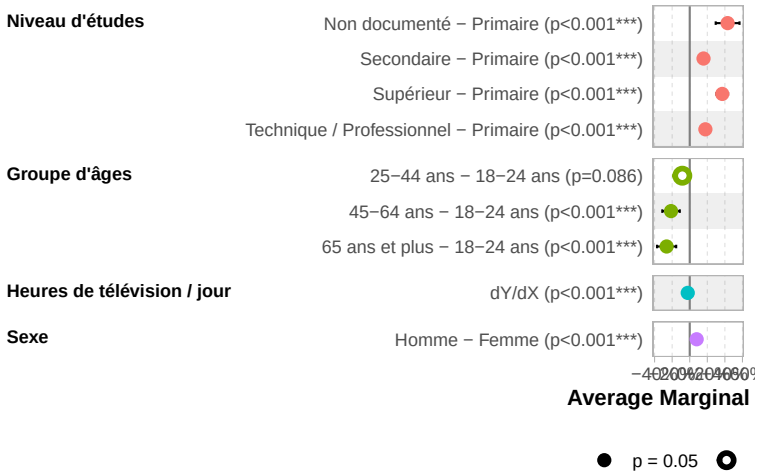


Figure 23.11: Effets marginaux moyens

Un résultat similaire peut être obtenu avec `margins::margins()`, le package `{margins}` s'inspirant de la commande **Stata** `margins`.

```
margins::margins(mod) %>% tidy()
```

```
# A tibble: 9 x 5
  term                          estimate std.error statistic  p.value
  <chr>                            <dbl>     <dbl>     <dbl>    <dbl>
1 etudesNon documenté              0.431    0.0691       6.23 4.60e-10
2 etudesSecondaire                 0.157    0.0314       4.99 6.10e- 7
3 etudesSupérieur                  0.370    0.0337      11.0  4.69e-28
4 etudesTechnique / Professionnel  0.178    0.0295       6.04 1.53e- 9
5 groupe_ages25-44 ans            -0.0844   0.0492      -1.71 8.65e- 2
6 groupe_ages45-64 ans            -0.213    0.0507      -4.20 2.73e- 5
7 groupe_ages65 ans et plus       -0.263    0.0556      -4.73 2.21e- 6
8 heures.tv                       -0.0227   0.00620     -3.66 2.56e- 4
9 sexeHomme                        0.0788   0.0197       4.00 6.26e- 5
```

For `{broom.helpers}`, `{gtsummary}` or `{ggstats}`, use `broom.helpers::tidy_margins()`.

```r
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_margins,
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus"
    )
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 23.11: Effets marginaux moyens avec margins

Caractéristique	Average Marginal Effects	95% IC	p-valeur
Niveau d'études			
Primaire	—	—	
Non documenté	+43.1%	+29.5% – +56.6%	<0,001
Secondaire	+15.7%	+9.5% – +21.8%	<0,001
Supérieur	+37.0%	+30.4% – +43.6%	<0,001
Technique / Professionnel	+17.8%	+12.0% – +23.6%	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	-8.4%	-18.1% – +1.2%	0,086
45-64 ans	-21.3%	-31.2% – -11.3%	<0,001
65 ans et plus	-26.3%	-37.2% – -15.4%	<0,001
Heures de télévision / jour	-2.3%	-3.5% – -1.1%	<0,001

Caractéristique	Average Marginal Effects	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	+7.9%	+4.0% – +11.7%	<0,001

```
ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_margins
) +
  ggplot2::scale_x_continuous(
    labels = scales::label_percent(style_positive = "plus")
  )
```

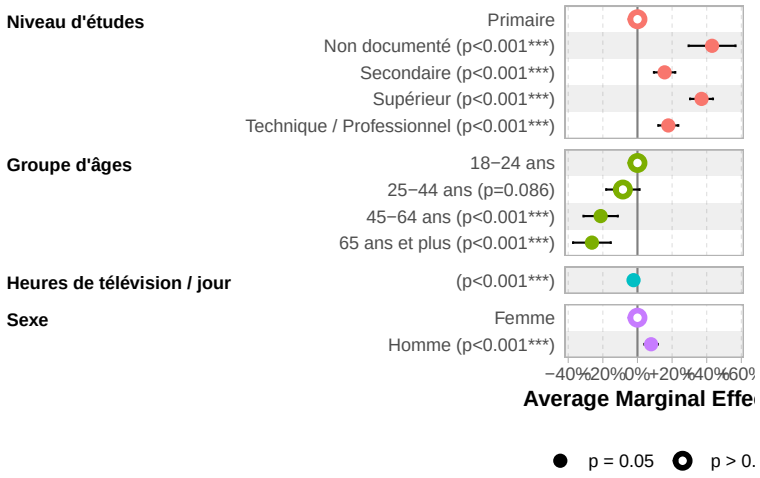


Figure 23.12: Effets marginaux moyens avec margins

23.5.2 Pentes marginales à la moyenne / Effets marginaux à la moyenne

Pour les **effets marginaux à la moyenne** (*marginal effect at the mean*), simplement indiquer `newdata = "mean"` à `broom.helpers::tidy_marginaleffects()`.

23.6 Lectures complémenaires (en anglais)

- Documentation of the `marginaleffects` package par Vincent Arel-Bundock
- Marginalia: A guide to figuring out what the heck marginal effects, marginal slopes, average marginal effects, marginal effects at the mean, and all these other marginal things are par Andrew Heiss
- Introduction to Adjusted Predictions and Marginal Effects in R par Daniel Lüdecke
- An Introduction to `margins`
- Marginal effects / slopes, contrasts, means and predictions with broom.helpers

24 Contrastes (variables catégorielles)

Dans les modèles de régression (comme les modèles linéaires, cf. Chapitre **??**, ou les modèles linéaires généralisés comme la régression logistique binaire, cf. Chapitre **??**), une transformation des variables catégorielles est nécessaire pour qu'elles puissent être prises en compte dans le modèle. On va dès lors définir des **contrastes**.

De manière générale, une variable catégorielle à n modalités va être transformée en *n-1* variables quantitatives. Il existe cependant plusieurs manières de faire (i.e. plusieurs types de contrastes). Et, selon les contrastes choisis, les coefficients du modèles ne s'interpréteront pas de la même manière.

24.1 Contrastes de type traitement

Par défaut, **R** applique des contrastes de type traitement pour un facteur non ordonné. Il s'agit notamment des contrastes utilisés par défaut dans les chapitres précédents.

24.1.1 Exemple 1 : un modèle linéaire avec une variable catégorielle

Commençons avec un premier exemple que nous allons calculer avec le jeu de données *trial* chargé en mémoire lorsque l'on appelle l'extension `{gtsummary}`. Ce jeu de données contient les observations de 200 patients. Nous nous intéressons à deux variables en particulier : *marker* une variable numérique correspondant à un marqueur biologique et *grade* un facteur à trois modalités correspondant à différent groupes de patients.

Regardons la moyenne de *marker* pour chaque valeur de *grade*.

```
library(tidyverse)
library(gtsummary)
trial |>
  select(marker, grade) |>
  tbl_summary(
    by = grade,
    statistic = marker ~ "{mean}",
    digits = marker ~ 4
  ) |>
  add_overall(last = TRUE)
```

Characteristic	**I**, N = 68	**II**, N = 68	**III**, N = 64	**Overall**, N = 200
Marker Level (ng/mL)	1.0669	0.6805	0.9958	0.9160
Unknown	2	5	3	10

Utilisons maintenant une régression linaire pour modéliser la valeur de *marker* en fonction de *grade*.

```
mod1_trt <- lm(marker ~ grade, data = trial)
mod1_trt
```

```
Call:
lm(formula = marker ~ grade, data = trial)

Coefficients:
(Intercept)      gradeII     gradeIII
     1.0669      -0.3864       -0.0711
```

Le modèle obtenu contient trois coefficients ou termes : un intercept et deux termes associés à la variable *grade*.

Pour bien interpréter ces coefficients, il faut comprendre comment la variable *grade* a été transformée avant d'être inclue dans le modèle. Nous pouvons voir cela avec la fonction `contrasts()`.

```
contrasts(trial$grade)
```

```
    II III
I    0   0
II   1   0
III  0   1
```

Ce que nous montre cette matrice, c'est que la variable catégorielle *grade* à 3 modalités a été transformée en 2 variables binaires que l'on retrouve sous les noms de *gradeII* et *gradeIII* dans le modèle : *gradeII* vaut 1 si *grade* est égal à *II* et 0 sinon; *gradeIII* vaut 1 si *grade* est égal à *III* et 0 sinon. Si *grade* est égal à *I*, alors *gradeII* et *gradeIII* valent 0.

Il s'agit ici d'un contraste dit de traitement ou la première modalité joue ici le rôle de **modalité de référence**.

Dans ce modèle linéaire, la valeur de l'intercept correspond à la moyenne de *marker* lorsque nous nous trouvons à la référence, donc quand *grade* est égal à *I* dans cet exemple. Et nous pouvons le constater dans notre tableau précédent des moyennes, `1.0669` correspond bien à la moyenne de *marker* pour la modalité *I*.

La valeur du coefficient associé à *markerII* correspond à l'écart par rapport à la référence lorsque *marker* est égal à *II*. Autrement dit, la moyenne de *marker* pour la modalité *II* correspond à la somme de l'intercept et du coefficient *markerII*. Et nous retrouvons bien la relation suivante : `0.6805 = 1.0669 + -0.3864`. De même, la moyenne de *marker* lorsque *grade* vaut *III* est égale à la somme de l'intercept et du terme *markerIII*.

Lorsqu'on utilise des contrastes de type traitement, chaque terme du modèle peut être associé à une et une seule modalité d'origine de la variable catégorielle. Dès lors, il est possible de rajouter la modalité de référence lorsque l'on présente les résultats et on peut même lui associer la valeurs 0, ce qui peut

être fait avec `gtsummary::tbl_regression()` avec l'option `add_estimate_to_reference_rows = TRUE`.

```
mod1_trt |>
  tbl_regression(
    intercept = TRUE,
    add_estimate_to_reference_rows = TRUE
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	Beta	95% CI	p-value
(Intercept)	1.1	0.86, 1.3	<0.001
Grade			
I	0.00	—	
II	-0.39	-0.68, -0.09	0.010
III	-0.07	-0.37, 0.23	0.6

24.1.2 Exemple 2 : une régression logistique avec deux variables catégorielles

Pour ce deuxième exemple, nous allons utiliser le jeu de données *hdv2003* fourni par l'extension `{questionr}` et recoder la variable *age* en groupes d'âges à 4 modalités.

```
library(questionr)
data("hdv2003")

library(tidyverse)
```

```
hdv2003 <- hdv2003 |>
  mutate(
    groupe_ages = cut(
      age,
      c(16, 25, 45, 65, 99),
```

```
      right = FALSE,
      include.lowest = TRUE
    ) |>
      fct_recode(
        "16-24" = "[16,25)",
        "25-44" = "[25,45)",
        "45-64" = "[45,65)",
        "65+" = "[65,99]"
      )
  ) |>
  labelled::set_variable_labels(
    groupe_ages = "Groupe d'âges",
    sexe = "Sexe"
  )
```

Nous allons faire une régression logistique binaire pour investiguer l'effet du *sexe* (variable à 2 modalités) et du *groupe d'âges* (variable à 4 modalités) sur la pratique du *sport*.

```
mod2_trt <- glm(
  sport ~ sexe + groupe_ages,
  family = binomial,
  data = hdv2003
)
mod2_trt
```

```
Call:  glm(formula = sport ~ sexe + groupe_ages, family = binomial,
    data = hdv2003)

Coefficients:
     (Intercept)         sexeFemme  groupe_ages25-44  groupe_ages45-64
          0.9021           -0.4455           -0.6845           -1.6535
  groupe_ages65+
         -2.3198

Degrees of Freedom: 1999 Total (i.e. Null);  1995 Residual
Null Deviance:      2617
Residual Deviance: 2385     AIC: 2395
```

Le modèle contient 5 termes : 1 intercept, 1 coefficient pour la variable *sexe* et 3 coefficients pour la variable *groupe_ages*. Comme précédemment, nous pouvons constater que les variables à *n* modalités sont remplacées par défaut (contrastes de type traitement) par *n-1* variables binaires, la première modalité jouant à chaque fois le rôle de modalité de référence.

```
contrasts(hdv2003$sexe)
```

```
       Femme
Homme      0
Femme      1
```

```
contrasts(hdv2003$groupe_ages)
```

```
      25-44 45-64 65+
16-24     0     0   0
25-44     1     0   0
45-64     0     1   0
65+       0     0   1
```

L'intercept correspond donc à la situation à la référence, c'est-à-dire à la prédiction du modèle pour les hommes (référence de *sexe*) âgés de 16 à 24 ans (référence de *groupe_ages*).

Il est possible d'exprimer cela en termes de probabilité en utilisant l'inverse de la fonction *logit* (puisque nous avons utilisé un modèle *logit*).

```
inv_logit <- binomial("logit")$linkinv
inv_logit(0.9021)
```

```
[1] 0.7113809
```

Selon le modèle, les hommes âgés de 16 à 24 ans ont donc 71% de chance de pratiquer du sport.

Regardons maintenant le coefficient associé à *sexeFemme* (-0.4455) : il représente (pour la modalité de référence des autres

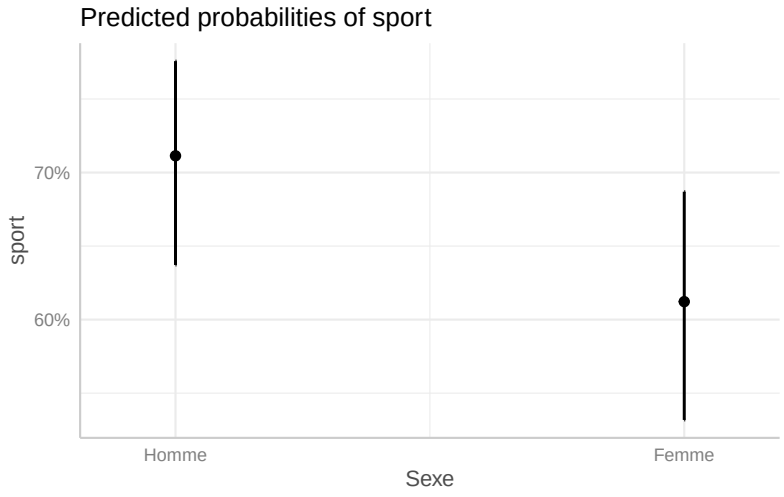
variables, soit pour les 16-24 ans ici) la correction à appliquer à l'intercept pour obtenir la probabilité de faire du sport. Il s'agit donc de la différence entre les femmes et les hommes pour le groupe des 16-24 ans.

```
inv_logit(0.9021 - 0.4455)
```

```
[1] 0.6122073
```

Autrement dit, selon le modèle, la probabilité de faire du sport pour une femme âgée de 16 à 24 ans est de 61%. On peut représenter cela avec la fonction `ggeffects::ggpredict()` de `{ggeffects}`, qui représente les prédictions d'une variable toutes les autres variables étant à la référence.

```
library(ggeffects)
ggpredict(mod2_trt, "sexe") |> plot()
```



Bien souvent, pour une régression logistique, on préfère représenter les exponentielles des coefficients qui correspondent à des *odds ratios*.

```
mod2_trt |>
  tbl_regression(
    exponentiate = TRUE,
    intercept = TRUE,
    add_estimate_to_reference_rows = TRUE
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	OR	95% CI	p-value
(Intercept)	2.46	1.76, 3.48	<0.001
Sexe			
Homme	1.00	—	
Femme	0.64	0.53, 0.78	<0.001
Groupe d'âges			
16-24	1.00	—	
25-44	0.50	0.35, 0.71	<0.001
45-64	0.19	0.13, 0.27	<0.001
65+	0.10	0.06, 0.15	<0.001

Or, 0,64 correspond bien à l'odds ratio entre 61% et 71% (que l'on peut calculer avec `questionr::odds.ratio()`).

```
questionr::odds.ratio(0.6122, 0.7114)
```

```
[1] 0.6404246
```

De la même manière, les différents coefficients associés à *groupe_ages* correspondent à la différence entre chaque groupe d'âges et sa modalité de référence (ici 16-24 ans), quand les autres variables (ici le *sexe*) sont à leur référence (ici les hommes).

Pour prédire la probabilité de faire du sport pour un profil particulier, il faut prendre en compte toutes les termes qui

s'appliquent et qui s'ajoutent à l'intercept. Par exemple, pour une femme de 50 ans il faut considérer l'intercept (0.9021), le coefficient *sexeFemme* (-0.4455) et le coefficient *groupe_ages45-64* (-1.6535). Sa probabilité de faire du sport est donc de 23%.

```
inv_logit(0.9021 - 0.4455 - 1.6535)
```

```
[1] 0.2320271
```

24.1.3 Changer la modalité de référence

Il est possible de personnaliser les contrastes à utiliser et avoir un recours à un contraste de type traitement mais en utilisant une autre modalité que la première comme référence, avec la fonction `contr.treatment()`. Le premier argument de la fonction corresponds au nombre de modalités de la variable et le paramètre `base` permets de spécifier la modalité de référence (1 par défaut).

```
contr.treatment(4, base = 2)
```

```
  1 3 4
1 1 0 0
2 0 0 0
3 0 1 0
4 0 0 1
```

`contr.SAS()` permets de spécifier un contraste de type traitement dont la modalité de référence est la dernière.

```
contr.SAS(4)
```

```
  1 2 3
1 1 0 0
2 0 1 0
3 0 0 1
4 0 0 0
```

Les contrastes peuvent être modifiés de deux manières : au moment de la construction du modèle (via l'option `contrasts`) ou comme attribut des variables (via la fonction `contrasts()`).

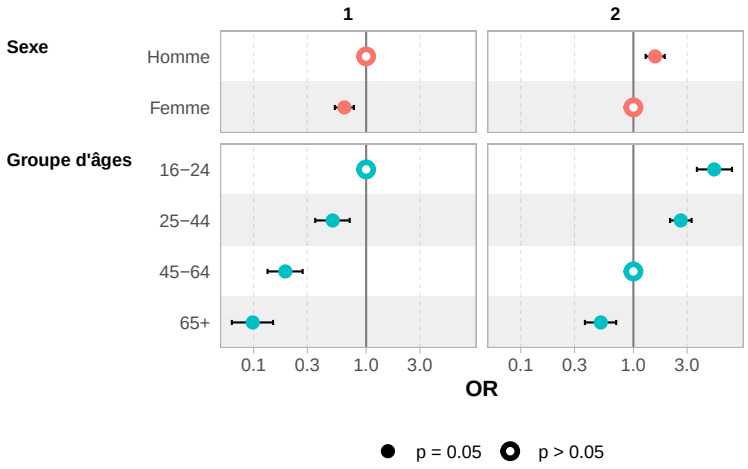
```
contrasts(hdv2003$sexe) <- contr.SAS(2)
mod2_trt_bis <- glm(
  sport ~ sexe + groupe_ages,
  family = binomial,
  data = hdv2003,
  contrasts = list(groupe_ages = contr.treatment(4, 3))
)
mod2_trt_bis |>
  tbl_regression(exponentiate = TRUE, intercept = TRUE)
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	OR	95% CI	p-value
(Intercept)	0.30	0.25, 0.36	<0.001
Sexe			
Homme	1.56	1.29, 1.90	<0.001
Femme	—	—	
Groupe d'âges			
16-24	5.23	3.67, 7.52	<0.001
25-44	2.64	2.12, 3.29	<0.001
45-64	—	—	
65+	0.51	0.37, 0.70	<0.001

Comme les modalités de référence ont changé, l'intercept et les différents termes ont également changé (puisque l'on ne compare plus à la même référence).

```
ggstats::ggcoef_compare(
  list(mod2_trt, mod2_trt_bis),
  exponentiate = TRUE,
  type = "faceted"
)
```



Cependant, du point de vue explicatif et prédictif, les deux modèles sont rigoureusement identiques.

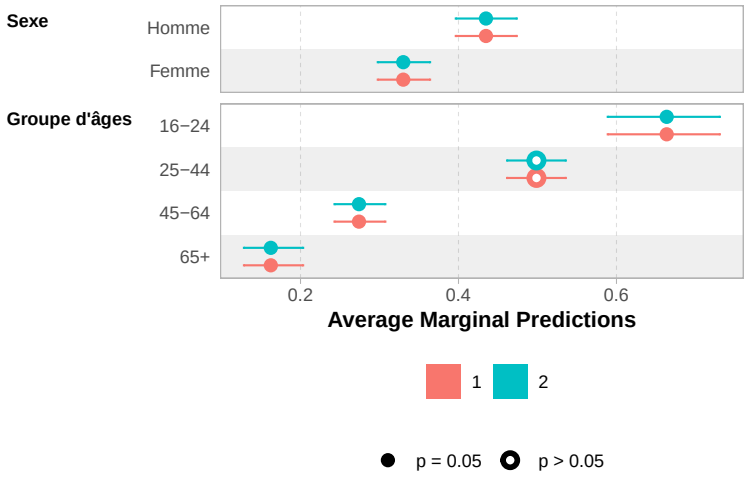
```
anova(mod2_trt, mod2_trt_bis, test = "Chisq")
```

```
Analysis of Deviance Table

Model 1: sport ~ sexe + groupe_ages
Model 2: sport ~ sexe + groupe_ages
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1      1995     2385.2
2      1995     2385.2  0        0
```

De même, leurs prédictions marginales (cf. Chapitre **??**) sont identiques.

```
ggstats::ggcoef_compare(
  list(mod2_trt, mod2_trt_bis),
  tidy_fun = broom.helpers::tidy_marginal_predictions,
  type = "dodge",
  vline = FALSE
)
```



24.2 Contrastes de type somme

Nous l'avons vu, les contrastes de type traitement nécessitent de définir une modalité de référence et toutes les autres modalités seront comparées à cette modalité de référence. Une alternative consiste à comparer toutes les modalités à la grande moyenne, ce qui s'obtient avec un contraste de type somme que l'on obtient avec `contr.sum()`.

24.2.1 Exemple 1 : un modèle linéaire avec une variable catégorielle

Reprenons notre premier exemple de tout à l'heure et modifions seulement le contraste.

```
contrasts(trial$grade) <- contr.sum
mod1_sum <- lm(
  marker ~ grade,
  data = trial
)
mod1_sum
```

```
Call:
lm(formula = marker ~ grade, data = trial)

Coefficients:
(Intercept)       grade1       grade2
     0.9144       0.1525      -0.2339
```

L'*intercept* correspond à ce qu'on appelle parfois la grande moyenne (ou *great average* en anglais). Il ne s'agit pas de la moyenne observée de *marker* mais de la moyenne des moyennes de chaque sous-groupe. Cela va constituer la situation de référence de notre modèle, en quelque sorte indépendante des effets de la variable *grade*.

```r
mean(trial$marker, na.rm = TRUE)
```

```
[1] 0.9159895
```

```r
moy_groupe <-
  trial |>
  dplyr::group_by(grade) |>
  dplyr::summarise(moyenne_marker = mean(marker, na.rm = TRUE))
moy_groupe
```

```
# A tibble: 3 x 2
  grade moyenne_marker
  <fct>          <dbl>
1 I              1.07
2 II             0.681
3 III            0.996
```

```r
mean(moy_groupe$moyenne_marker)
```

```
[1] 0.9144384
```

Le terme *grade1* correspond quant à lui au modificateur associé à la première modalité de la variable *grade* à savoir *I*. C'est l'écart, pour cette modalité, à la grande moyenne : `1.0669 - 0.9144 = 0.1525`.

De même, le terme *grade2* correspond à l'écart pour la modalité *II* par rapport à la grande moyenne : `0.6805 - 0.9144 = -0.2339`.

Qu'en est-il de l'écart à la grande moyenne pour la modalité *III* ? Pour cela, voyons tout d'abord comment la variable *grade* a été codée :

```
contrasts(trial$grade)
```

```
    [,1] [,2]
I      1    0
II     0    1
III   -1   -1
```

Comme précédemment, cette variable à trois modalités a été codée avec deux termes. Les deux premiers termes correspondent aux écarts à la grande moyenne des deux premières modalités. La troisième modalité est, quant à elle, codée systématiquement `-1`. C'est ce qui assure que la somme des contributions soit nulle et donc que l'*intercept* capture la grande moyenne.

L'écart à la grande moyenne pour la troisième modalité s'obtient donc en faisant la somme des autres termes et en l'inversant : `(0.1525 - 0.2339) * -1 = 0.0814 = 0.9958 - 0.9144`.

On peut calculer / afficher la valeur associée à la dernière modalité en précisant `add_estimate_to_reference_rows = TRUE` lorsque l'on appelle `gtsummary::tbl_regression()`.

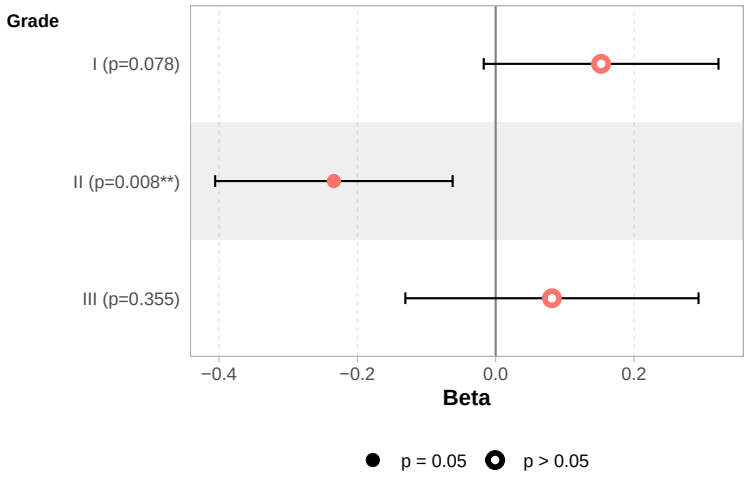
```
mod1_sum |>
  tbl_regression(
    intercept = TRUE,
    add_estimate_to_reference_rows = TRUE
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	Beta	95% CI	p-value
(Intercept)	0.91	0.79, 1.0	<0.001
Grade			
I	0.15	-0.02, 0.32	0.078
II	-0.23	-0.41, -0.06	0.008
III	0.08	-0.13, 0.29	0.4

De même, cette valeur est correctement affichée par `ggstats::ggcoef_model()`.

```
ggstats::ggcoef_model(mod1_sum)
```



Le fait d'utiliser des contrastes de type traitement ou somme n'a aucun impact sur la valeur prédictive du modèle. La quantité de variance expliquée, la somme des résidus ou encore l'AIC sont identiques. En un sens, il s'agit du même modèle. C'est seulement la manière d'interpréter les coefficients du modèle qui change.

```r
anova(mod1_trt, mod1_sum, test = "Chisq")
```

```
Analysis of Variance Table

Model 1: marker ~ grade
Model 2: marker ~ grade
  Res.Df    RSS Df Sum of Sq Pr(>Chi)
1    187 134.17
2    187 134.17  0         0
```

24.2.2 Exemple 2 : une régression logistique avec deux variables catégorielles

Reprenons notre second exemple et codons les variables catégorielles avec un traitement de type somme.

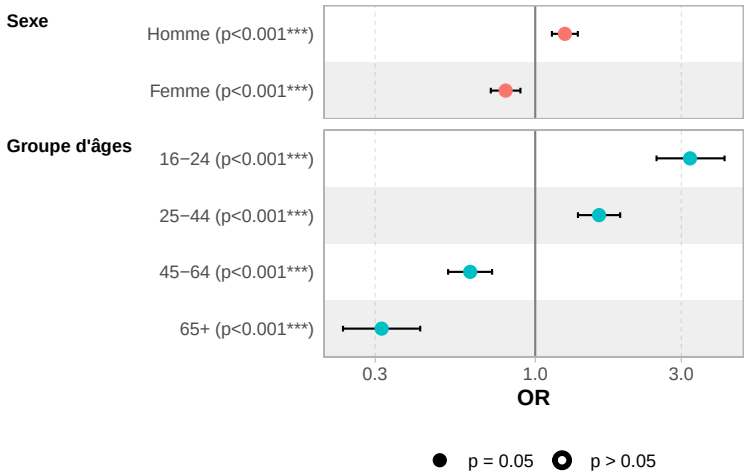
```r
mod2_sum <- glm(
  sport ~ sexe + groupe_ages,
  family = binomial,
  data = hdv2003,
  contrasts = list(sexe = contr.sum, groupe_ages = contr.sum)
)
mod2_sum |>
  tbl_regression(
    exponentiate = TRUE,
    intercept = TRUE,
    add_estimate_to_reference_rows = TRUE
  )
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	OR	95% CI	p-value
(Intercept)	0.62	0.55, 0.69	<0.001
Sexe			
Homme	1.25	1.13, 1.38	<0.001

Characteristic	OR	95% CI	p-value
Femme	0.80	0.72, 0.89	<0.001
Groupe d'âges			
16-24	3.20	2.49, 4.15	<0.001
25-44	1.62	1.38, 1.89	<0.001
45-64	0.61	0.52, 0.72	<0.001
65+	0.31	0.24, 0.42	<0.001

```
ggstats::ggcoef_model(mod2_sum, exponentiate = TRUE)
```



Cette fois-ci, l'*intercept* capture la situation à la grande moyenne à la fois du sexe et du groupe d'âges, et les coefficients s'interprètent donc comme modificateurs de chaque modalité par rapport à cette grande moyenne. En ce sens, les contrastes de type somme permettent donc de capturer l'effet de chaque modalité.

Du point de vue explicatif et prédictif, le fait d'avoir recours à des contrastes de type somme ou traitement n'a aucun impact : les deux modèles sont rigoureusement identiques. Il n'y a que la manière d'interpréter les coeffcients qui chagnge.

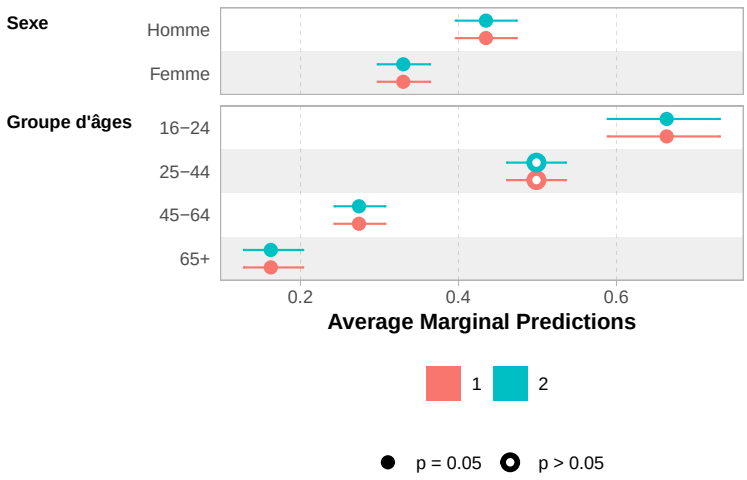
```
anova(mod2_trt, mod2_sum, test = "Chisq")
```

```
Analysis of Deviance Table

Model 1: sport ~ sexe + groupe_ages
Model 2: sport ~ sexe + groupe_ages
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1      1995     2385.2
2      1995     2385.2  0        0
```

Les prédictions marginales (cf. Chapitre **??**) sont identiques.

```
ggstats::ggcoef_compare(
  list(mod2_trt, mod2_sum),
  tidy_fun = broom.helpers::tidy_marginal_predictions,
  type = "dodge",
  vline = FALSE
)
```



24.3 Autres types de contrastes

24.3.1 Contrastes de type Helmert

Les contrastes de Helmert sont un peu plus complexes : ils visent à comparer la seconde modalité à la première, la troisième

à la moyenne des deux premières, la quatrième à la moyenne des trois premières, etc.

Prenon un exemple avec une variable catégorielle à quatre modalités.

```
contrasts(trial$stage) <- contr.helmert
contrasts(trial$stage)
```

```
   [,1] [,2] [,3]
T1   -1   -1   -1
T2    1   -1   -1
T3    0    2   -1
T4    0    0    3
```

```
mod_helmert <- lm(
  marker ~ stage,
  data = trial
)
mod_helmert
```

```
Call:
lm(formula = marker ~ stage, data = trial)

Coefficients:
(Intercept)       stage1       stage2       stage3
    0.91661      0.19956      0.03294     -0.02085
```

Pour bien comprendre comment interpréter ces coefficients, calculons déjà la grande moyenne.

```
m <- trial |>
  dplyr::group_by(stage) |>
  dplyr::summarise(moy = mean(marker, na.rm = TRUE))
mean(m$moy)
```

```
[1] 0.9166073
```

On le voit, l'*intercept* (`0.9166`) capture ici cette grande moyenne, à savoir la moyenne des moyennes de chaque sous-groupe.

Maintenant, pour interpréter les coefficients, regardons comment évolue la moyenne à chaque fois que l'on ajoute une modalité. La fonction `dplyr::cummean()` nous permet de calculer la moyenne cumulée, c'est-à-dire la moyenne de la valeur actuelle et des valeurs des lignes précédentes. Avec `dplyr::lag()` nous pouvons obtenir la moyenne cumulée de la ligne précédente. Il nous est alors possible de calculer l'écart entre les deux, et donc de voir comment la moyenne a changé avec l'ajout d'une modalité.

```
m <- m |>
  dplyr::mutate(
    moy_cum = dplyr::cummean(moy),
    moy_cum_prec = dplyr::lag(moy_cum),
    ecart = moy_cum - moy_cum_prec
  )
m
```

```
# A tibble: 4 x 5
  stage   moy moy_cum moy_cum_prec   ecart
  <fct> <dbl>   <dbl>        <dbl>   <dbl>
1 T1    0.705   0.705       NA     NA
2 T2    1.10    0.905        0.705  0.200
3 T3    1.00    0.937        0.905  0.0329
4 T4    0.854   0.917        0.937 -0.0208
```

On le voit, les valeurs de la colonne *ecart* correspondent aux coefficients du modèle.

Le premier terme *stage1* compare la deuxième modalité (*T2*) à la première (*T1*) et indique l'écart entre la moyenne des moyennes de *T1* et *T2* et la moyenne de *T1*.

Le second terme *stage2* compare la troisième modalité (*T3*) aux deux premières (*T1* et *T2*) et indique l'écart entre la moyenne des moyennes de *T1*, *T2* et *T3* par rapport à la moyenne des moyennes de *T1* et *T2*.

Le troisième terme *stage3* compare la quatrième modalité (*T4*) aux trois premières (*T1, T2* et *T3*) et indique l'écart entre la moyenne des moyennes de *T1*, *T2*, *T3* et *T4* par rapport à la moyenne des moyennes de *T1*, *T2* et *T3*.

Les contrastes de Helmert sont ainsi un peu plus complexes à interpréter et à réserver à des cas particuliers où ils prennent tout leur sens.

24.3.2 Contrastes polynomiaux

Les contrastes polynomiaux, définis avec `contr.poly()`, sont utilisés par défaut pour les variables catégorielles ordonnées. Ils permettent de décomposer les effets selon une composante linéaire, une composante quadratique, une composante cubique, voire des composantes de degrés supérieurs.

```
contrasts(trial$stage) <- contr.poly
contrasts(trial$stage)
```

```
           .L   .Q         .C
T1 -0.6708204  0.5 -0.2236068
T2 -0.2236068 -0.5  0.6708204
T3  0.2236068 -0.5 -0.6708204
T4  0.6708204  0.5  0.2236068
```

```
mod_poly <- lm(
  marker ~ stage,
  data = trial
)
mod_poly
```

```
Call:
lm(formula = marker ~ stage, data = trial)

Coefficients:
(Intercept)      stage.L      stage.Q      stage.C
    0.91661      0.07749     -0.27419      0.10092
```

Ici aussi, l'*intercept* correspond à la grande moyenne des moyennes. Il est par contre plus difficile de donner un sens interprétatif / sociologique aux différents coefficients.

24.4 Lectures additionnelles

- *A (sort of) Complete Guide to Contrasts in R* par Rose Maier
- *An introductory explanation of contrast coding in R linear models* par Athanassios Protopapas
- *Understanding Sum Contrasts for Regression Models: A Demonstration* par Mona Zhu

25 Interactions

Dans un modèle statistique classique, on fait l'hypothèse implicite que chaque variable explicative est indépendante des autres. Cependant, cela ne se vérifie pas toujours. Par exemple, l'effet de l'âge peut varier en fonction du sexe.

Nous pourrons dès lors ajouter à notre modèle des **interactions** entre variables.

25.1 Données d'illustration

Reprenons le modèle que nous avons utilisé dans le chapitre sur la régression logistique binaire (cf. Chapitre **??**).

```
library(tidyverse)
library(labelled)

data(hdv2003, package = "questionr")

d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                   "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
```

```
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
    ) |>
    fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    heures.tv = "Heures de télévision / jour"
  )
```

25.2 Modèle sans interaction

Nous avions alors exploré les facteurs associés au fait de pratiquer du sport.

```
mod <- glm(
  sport ~ sexe + groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
library(gtsummary)
theme_gtsummary_language(
  "fr",
  decimal.mark = ",",
  big.mark = " "
)
```

```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 25.1: Odds Ratios du modèle logistique simple

Caractéristique	OR	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	1,52	1,24 – 1,87	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	0,68	0,43 – 1,06	0,084
45-64 ans	0,36	0,23 – 0,57	<0,001
65 ans et plus	0,27	0,16 – 0,46	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	2,54	1,73 – 3,75	<0,001
Technique / Professionnel	2,81	1,95 – 4,10	<0,001
Supérieur	6,55	4,50 – 9,66	<0,001
Non documenté	8,54	4,51 – 16,5	<0,001
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

Selon les résultats de notre modèle, les hommes pratiquent plus un sport que les femmes et la pratique du sport diminue avec l'âge.

Dans le chapitre sur les estimations marginales, cf. Chapitre **??**, nous avons présenté la fonction `broom.helpers::plot_marginal_predictions()` qui permet de représenter les prédictions marginales moyennes du modèle.

```
mod |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
```

```
  patchwork::wrap_plots() &
  scale_y_continuous(
    limits = c(0, .8),
    labels = scales::label_percent()
  )
```

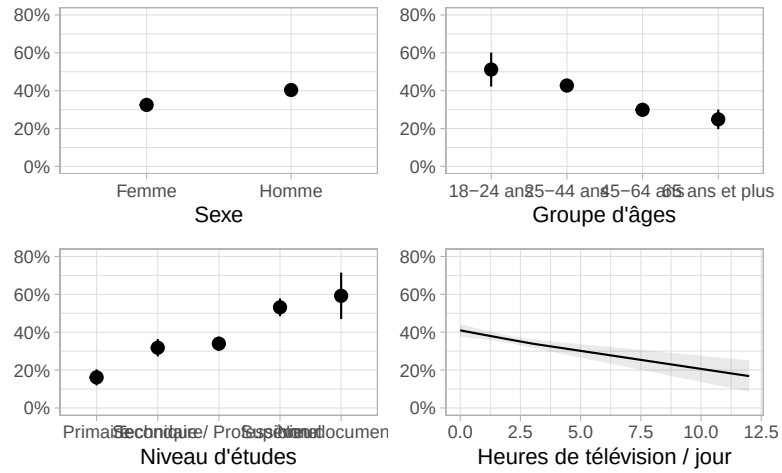


Figure 25.1: Prédictions marginales moyennes du modèle simple

25.3 Définition d'une interaction

Cependant, l'effet de l'âge est-il le même selon le sexe ? Nous allons donc introduire une interaction entre l'âge et le sexe dans notre modèle, ce qui sera représenté par `sexe * groupe_ages`dans l'équation du modèle.

```
mod2 <- glm(
  sport ~ sexe * groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
```

Commençons par regarder les prédictions marginales du modèle avec interaction.

```
mod2 |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
  patchwork::wrap_plots(ncol = 1) &
  scale_y_continuous(
    labels = scales::label_percent()
  )
```

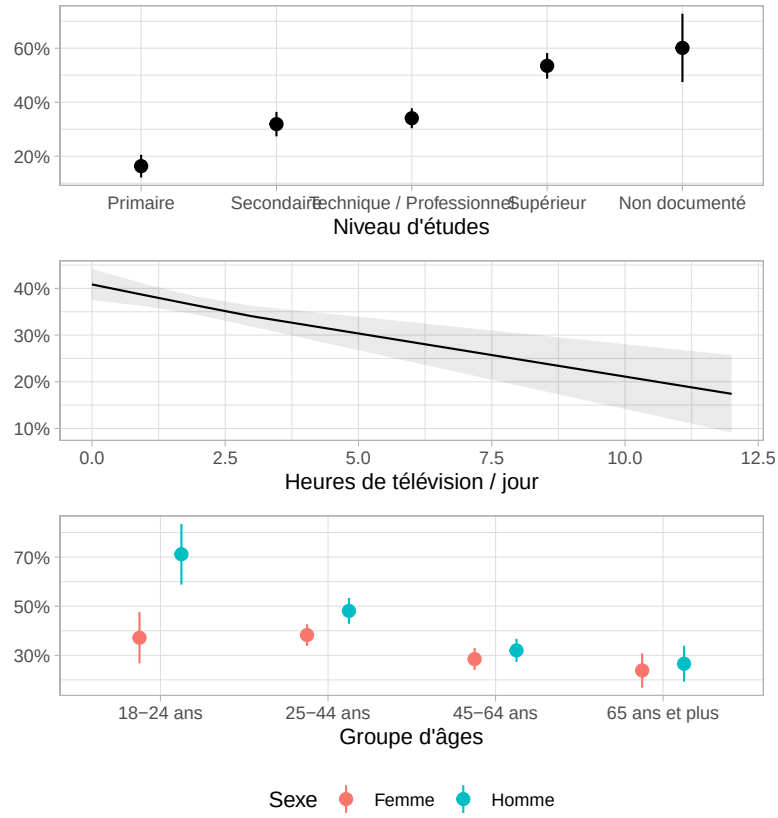


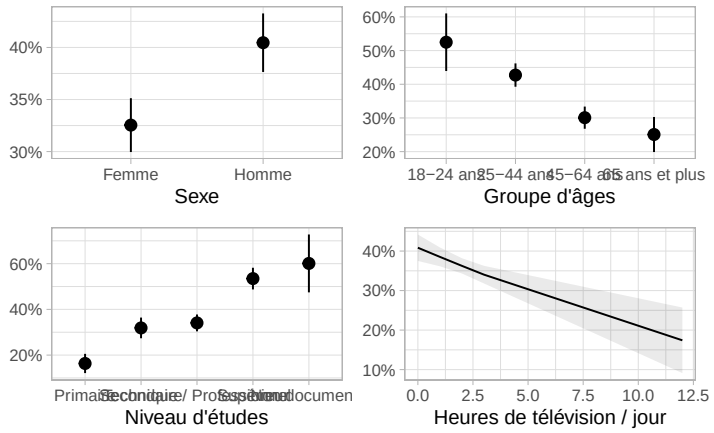
Figure 25.2: Prédictions marginales moyennes du modèle avec interaction

Sur ce graphique, on voit que la pratique d'un sport diminue fortement avec l'âge chez les hommes, tandis que cette diminution est bien plus modérée chez les femmes.

Astuce

Par défaut, `broom.helpers::plot_marginal_predictions()` détecte la présence d'interactions dans le modèle et calcule les prédictions marginales pour chaque combinaison de variables incluent dans une interaction. Il reste possible de calculer des prédictions marginales individuellement pour chaque variable du modèle. Pour cela, il suffit d'indiquer `variables_list = "no_interaction"`.

```
mod2 |>
  broom.helpers::plot_marginal_predictions(
    variables_list = "no_interaction",
    type = "response"
  ) |>
  patchwork::wrap_plots() &
  scale_y_continuous(
    labels = scales::label_percent()
  )
```



25.4 Significativité de l'interaction

L'ajout d'une interaction au modèle augmente la capacité prédictive du modèle mais, dans le même temps, augmente le

nombre de coefficients (et donc de degrés de liberté). La question se pose donc de savoir si l'ajout d'un terme d'interaction améliore notre modèle.

En premier lieu, nous pouvons comparer les AIC des modèles avec et sans interaction.

```
AIC(mod)
```

```
[1] 2230.404
```

```
AIC(mod2)
```

```
[1] 2223.382
```

L'AIC du modèle avec interaction est plus faible que celui sans interaction, nous indiquant un gain : notre modèle avec interaction est donc meilleur.

On peut tester avec `car::Anova()` si l'interaction est statistiquement significative[43].

```
car::Anova(mod2, type = "III")
```

```
Analysis of Deviance Table (Type III tests)

Response: sport
                 LR Chisq Df Pr(>Chisq)
sexe               19.349  1  1.089e-05 ***
groupe_ages        15.125  3  0.0017131 **
etudes            125.575  4  < 2.2e-16 ***
heures.tv          12.847  1  0.0003381 ***
sexe:groupe_ages   13.023  3  0.0045881 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La p-valeur associée au terme d'interaction (`sexe:groupe_ages`) est inférieure à 1% : l'interaction a donc bien un effet significatif.

[43] Lorsqu'il y a une interaction, il est préférable d'utiliser le type III, cf. Section **??**. En toute rigueur, il serait préférable de coder nos variables catégorielles avec un contraste de type somme (cf. Chapitre **??**). En pratique, nous pouvons nous en passer ici.

Nous pouvons également utiliser `gtsummary::add_global_p()`.

```
mod2 |>
  tbl_regression(exponentiate = TRUE) |>
  add_global_p() |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 25.2: Odds Ratios du modèle logistique avec interaction

Caractéristique	OR	95% IC	p-valeur
Sexe			<0,001
Femme	—	—	
Homme	5,10	2,41 – 11,4	
Groupe d'âges			0,002
18-24 ans	—	—	
25-44 ans	1,06	0,61 – 1,85	
45-64 ans	0,64	0,36 – 1,15	
65 ans et plus	0,49	0,25 – 0,97	
Niveau d'études			<0,001
Primaire	—	—	
Secondaire	2,55	1,74 – 3,78	
Technique / Professionnel	2,84	1,97 – 4,14	
Supérieur	6,69	4,60 – 9,89	
Non documenté	8,94	4,64 – 17,6	
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001
Sexe * Groupe d'âges			0,005
Homme * 25-44 ans	0,31	0,13 – 0,70	
Homme * 45-64 ans	0,24	0,10 – 0,54	
Homme * 65 ans et plus	0,23	0,09 – 0,60	

25.5 Interprétation des coefficients

Jetons maintenant un œil aux coefficients du modèle. Pour rendre les choses plus visuelles, nous aurons recours à

ggtstats::ggcoef_model().

```
mod2 |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

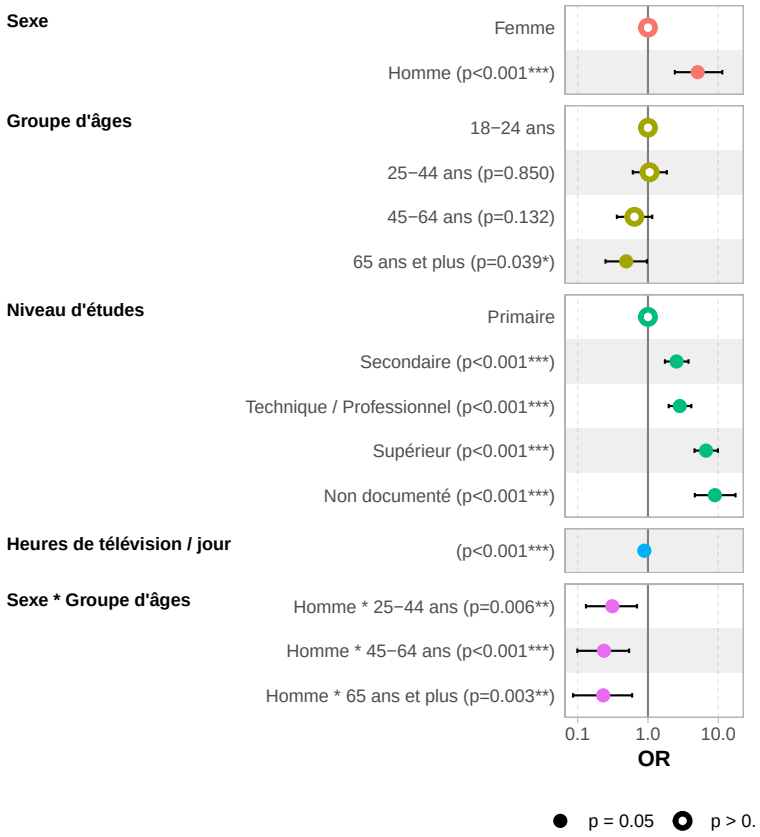


Figure 25.3: Coefficients (odds ratio) du modèle avec interaction

Concernant les variables *sexe* et *groupe_ages*, nous avons trois séries de coefficients : une série pour le sexe, une pour le groupe d'âges et enfin des coefficients pour l'interaction entre le sexe et le groupe d'âges.

Pour bien interpréter ces coefficients, il faut toujours avoir en tête les modalités choisies comme référence pour chaque variable.

Supposons une femme de 60 ans, dont toutes les autres variables correspondent aux modalités de référence (i.e. de niveau primaire, qui ne regarde pas la télévision). Regardons ce que prédit le modèle quant à sa probabilité de faire du sport au travers d'une représentation graphique, grâce au package `{breakDown}`.

```
library(breakDown)
logit <- function(x) exp(x)/(1+exp(x))
nouvelle_observation <- d[1, ]
nouvelle_observation$sexe[1] = "Femme"
nouvelle_observation$groupe_ages[1] = "45-64 ans"
nouvelle_observation$etud[1] = "Primaire"
nouvelle_observation$heures.tv[1] = 0
plot(
  broken(mod2, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1) +
  ylab("Probabilité de faire du sport")
```

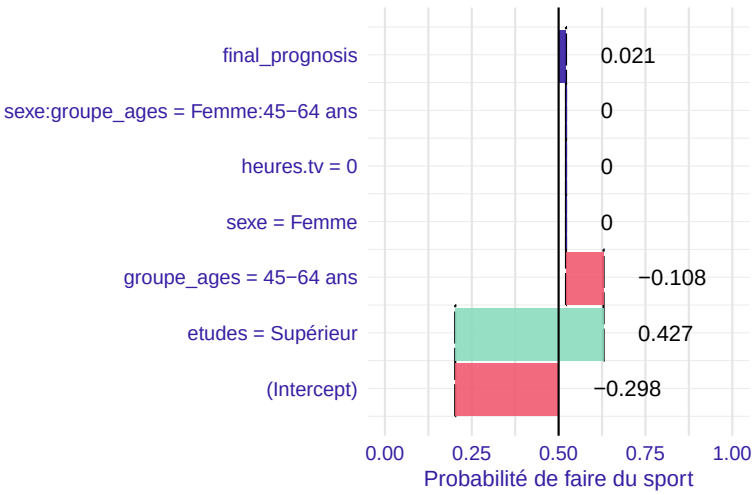


Figure 25.4: Représentation graphique de l'estimation de la probabilité de faire du sport pour une femme de 60 ans

En premier lieu, l'*intercept* s'applique et permet de déterminer la probabilité de base de faire du sport à la référence. Femme étant la modalité de référence pour la variable *sexe*, cela ne modifie pas le calcul de la probabilité de faire du sport. Par contre, il y a une modification induite par la modalité 45-64 ans de la variable *groupe_ages*.

Regardons maintenant la situation d'un homme de 20 ans.

```
nouvelle_observation$sexe[1] = "Homme"
nouvelle_observation$groupe_ages[1] = "18-24 ans"
plot(
  broken(mod2, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1.2) +
  ylab("Probabilité de faire du sport")
```

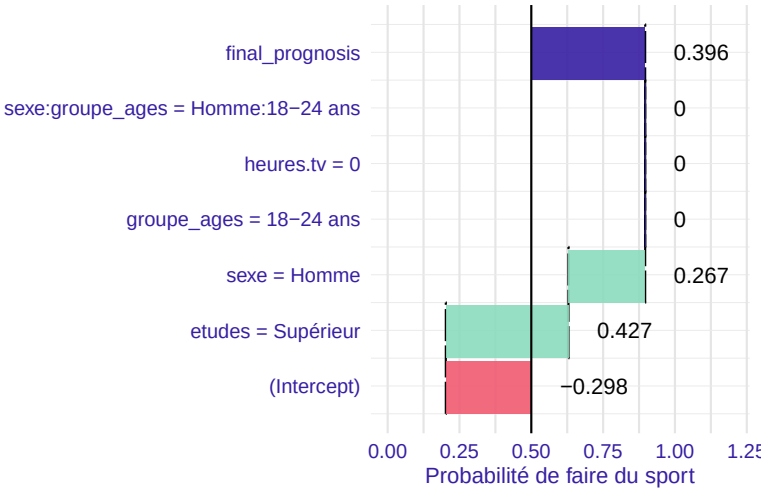


Figure 25.5: Représentation graphique de l'estimation de la probabilité de faire du sport pour un homme de 20 ans

Nous sommes à la modalité de référence pour l'âge par contre il y a un effet important du sexe. Le coefficient associé globalement à la variable sexe correspond donc à l'effet du sexe à la modalité de référence du groupe d'âges.

Regardons enfin la situation d'un homme de 60 ans.

```
nouvelle_observation$groupe_ages[1] = "45-64 ans"
plot(
  broken(mod2, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1.2) +
  ylab("Probabilité de faire du sport")
```

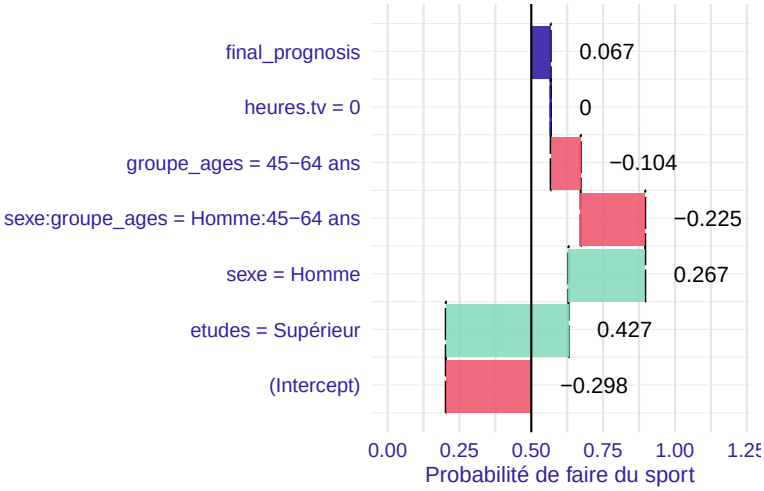


Figure 25.6: Représentation graphique de l'estimation de la probabilité de faire du sport pour un homme de 60 ans

Cette fois, plusieurs coefficients s'appliquent : à la fois le coefficient sexe = Homme (effet du sexe pour les 18-24 ans), le coefficient groupe_ages = 45-64 ans qui est l'effet de l'âge pour les femmes de 45-64 ans par rapport aux 18-24 ans et le coefficient sexe:groupe_ages = Homme:45-64 ans qui indique l'effet spécifique qui s'applique aux hommes de 45-64 ans, d'une part par rapport aux femmes du même âge et d'autre part par rapport aux hommes de 18-24 ans. L'effet des coefficients d'interaction doivent donc être interprétés par rapport aux autres coefficients du modèle qui s'appliquent, en tenant compte des modalités de référence.

25.6 Définition alternative de l'interaction

Il est cependant possible d'écrire le même modèle différemment. En effet, `sexe * groupe_ages` dans la formule du modèle est équivalent à l'écriture `sexe + groupe_ages + sexe:groupe_ages`, c'est-à-dire que l'on demande des coefficients pour la variable *sexe* à la référence de *groupe_ages*, des coefficients pour *groupe_ages* à la référence de *sexe* et enfin des coefficients pour tenir compte de l'interaction.

On peut se contenter d'une série de coefficients uniques pour l'interaction en indiquant seulement `sexe : groupe_ages`.

```
mod3 <- glm(
  sport ~ sexe : groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
```

Au sens strict, ce modèle explique tout autant le phénomène étudié que le modèle précédent. On peut le vérifier facilement avec `stats::anova()`.

```
anova(mod2, mod3, test = "Chisq")
```

```
Analysis of Deviance Table

Model 1: sport ~ sexe * groupe_ages + etudes + heures.tv
Model 2: sport ~ sexe:groupe_ages + etudes + heures.tv
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1      1982     2197.4
2      1982     2197.4  0        0
```

De même, les prédictions marginales sont les mêmes, comme nous pouvons le constater avec `ggstats::ggcoef_compare()`.

```
ggstats::ggcoef_compare(
  list("sexe * groupe_ages" = mod2, "sexe : groupe_ages" = mod3),
  tidy_fun = broom.helpers::tidy_marginal_predictions,
  significance = NULL,
```

```
    vline = FALSE
  ) +
    scale_x_continuous(labels = scales::label_percent())
```

```
Warning: Model matrix is rank deficient. Some variance-covariance parameters are
  missing.
```

```
Warning: Model matrix is rank deficient. Some variance-covariance parameters are
  missing.
```

```
Warning: Model matrix is rank deficient. Some variance-covariance parameters are
  missing.
```

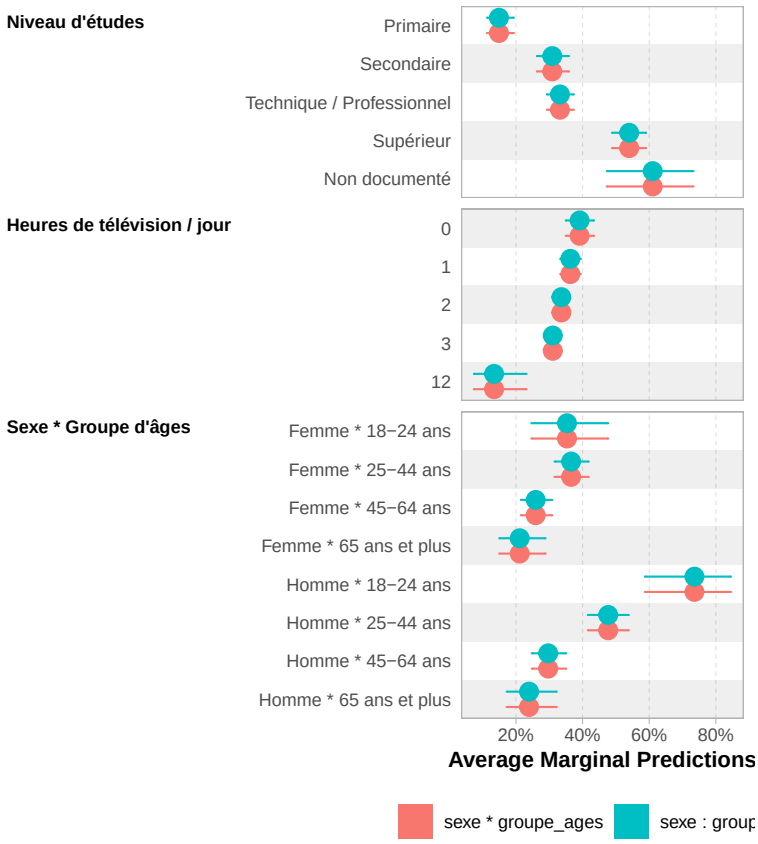


Figure 25.7: Comparaison des prédictions marginales moyennes des deux modèles avec interaction

Par contre, regardons d'un peu plus près les coefficients de ce nouveau modèle. Nous allons voir que leur interprétation est légèrement différente.

```
mod3 |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

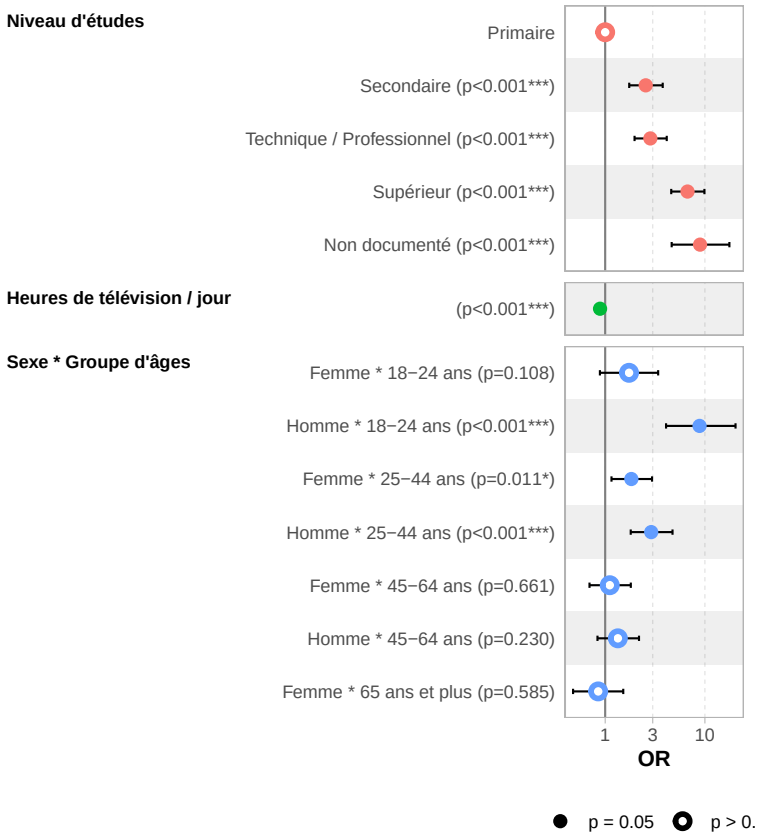


Figure 25.8: Coefficients (odds ratio) du modèle avec interaction simple entre le sexe et le groupe d'âges

Cette fois-ci, il n'y a plus de coefficients globaux pour la variable *sexe* ni pour *groupe_ages* mais des coefficients pour chaque combinaison de ces deux variables. Reprenons l'exemple de notre homme de 60 ans.

```
plot(
  broken(mod3, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1.2) +
  ylab("Probabilité de faire du sport")
```

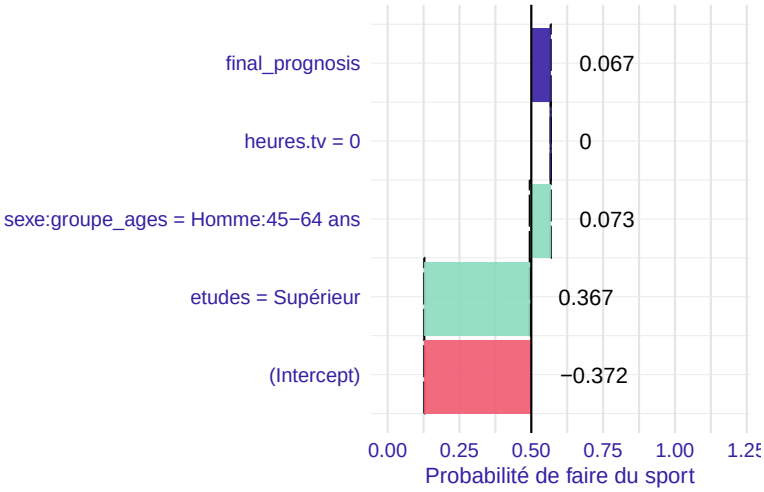


Figure 25.9: Représentation graphique de l'estimation de la probabilité de faire du sport pour un homme de 60 ans (interaction simple)

Cette fois-ci, le coefficient d'interaction fournit indique l'effet combiné du sexe et du groupe d'âges par rapport à la situation de référence (femme de 18-24 ans).

Que l'on définisse une interaction simple (`sexe:groupe_ages`) ou complète (`sexe*groupe_ages`), les deux modèles calculés sont donc identiques en termes prédictifs et explicatifs, mais l'interprétation de leurs coefficients diffèrent.

Astuce

Il peut y avoir de multiples interactions dans un modèle, d'ordre 2 (entre deux variables) ou plus (entre trois variables ou plus). Il est dès lors tentant de tester les multiples interactions possibles de manière itératives afin d'identifier celles à retenir. C'est justement le but de la fonction `ggmulti::glmulti()` `glmulti::glmulti()` permets de tester toutes les combinaisons d'interactions d'ordre 2 dans un modèle, en retenant le meilleur modèle à partir d'un critère spécifié (par défaut l'AIC). ATTENTION : le temps de calcul de `glmulti::glmulti()` peut-être long.

25.7 Pour aller plus loin

Il y a d'autres extensions dédiées à l'analyse des interactions d'un modèle, de même que de nombreux supports de cours en ligne dédiés à cette question.

- Les effets d'interaction par Jean-François Bickel
- Analysing interactions of fitted models par Helios De Rosario Martínez

25.8 webin-R

Les interactions sont abordées dans le webin-R #07 (*régression logistique partie 2*) sur YouTube.

https://youtu.be/BUo9i7XTLYQ

26 Multicolinéarité

Dans une régression, la multicolinéarité est un problème qui survient lorsque certaines variables de prévision du modèle mesurent le même phénomène. Une multicolinéarité prononcée s'avère problématique, car elle peut augmenter la variance des coefficients de régression et les rendre instables et difficiles à interpréter. Les conséquences de coefficients instables peuvent être les suivantes :

- les coefficients peuvent sembler non significatifs, même lorsqu'une relation significative existe entre le prédicteur et la réponse ;
- les coefficients de prédicteurs fortement corrélés varieront considérablement d'un échantillon à un autre ;
- lorsque des termes d'un modèle sont fortement corrélés, la suppression de l'un de ces termes aura une incidence considérable sur les coefficients estimés des autres. Les coefficients des termes fortement corrélés peuvent même présenter le mauvais signe.

La multicolinéarité n'a aucune incidence sur l'adéquation de l'ajustement, ni sur la qualité de la prévision. Cependant, les coefficients individuels associés à chaque variable explicative ne peuvent pas être interprétés de façon fiable.

26.1 Définition

Au sens strict, on parle de multicolinéarité parfaite lorsqu'une des variables explicatives d'un modèle est une combinaison linéaire d'une ou plusieurs autres variables explicatives introduites dans le même modèle. L'absence de multicolinéarité parfaite est une des conditions requises pour pouvoir estimer un

modèle linéaire et, par extension, un modèle linéaire généralisé (dont les modèles de régression logistique).

Dans les faits, une multicolinéarité parfaite n'est quasiment jamais observée. Mais une forte multicolinéarité entre plusieurs variables peut poser problème dans l'estimation et l'interprétation d'un modèle.

Une erreur fréquente est de confondre multicolinéarité et corrélation. Si des variables colinéaires sont *de facto* fortement corrélées entre elles, deux variables corrélées ne sont pas forcément colinéaires. En termes non statistiques, il y a colinéarité lorsque deux ou plusieurs variables mesurent la même chose.

Prenons un exemple. Nous étudions les complications après l'accouchement dans différentes maternités d'un pays en développement. On souhaite mettre dans le modèle, à la fois le milieu de résidence (urbain ou rural) et le fait qu'il y ait ou non un médecin dans la clinique. Or, dans la zone d'enquête, les maternités rurales sont dirigées seulement par des sage-femmes tandis que l'on trouve un médecin dans toutes les maternités urbaines sauf une. Dès lors, dans ce contexte précis, le milieu de résidence prédit presque totalement la présence d'un médecin et on se retrouve face à une multicolinéarité (qui serait même parfaite s'il n'y avait pas une clinique urbaine sans médecin). On ne peut donc distinguer l'effet de la présence d'un médecin de celui du milieu de résidence et il ne faut mettre qu'une seule de ces deux variables dans le modèle, sachant que du point de vue de l'interprétation elle capturera à la fois l'effet de la présence d'un médecin et celui du milieu de résidence.

Par contre, si dans notre région d'étude, seule la moitié des maternités urbaines disposait d'un médecin, alors le milieu de résidence n'aurait pas été suffisant pour prédire la présence d'un médecin. Certes, les deux variables seraient corrélées mais pas colinéaires. Un autre exemple de corrélation sans colinéarité, c'est la relation entre milieu de résidence et niveau d'instruction. Il y a une corrélation entre ces deux variables, les personnes résidant en ville étant généralement plus instruites. Cependant, il existe également des personnes non instruites en ville et des personnes instruites en milieu rural. Le milieu de résidence n'est donc pas suffisant pour prédire le niveau d'instruction.

26.2 Mesure de la colinéarité

Il existe différentes mesures de la multicolinéarité. L'extension `{mctest}` en fournie plusieurs, mais elle n'est utilisable que si l'ensemble des variables explicatives sont de type numérique.

L'approche la plus classique consiste à examiner les facteurs d'inflation de la variance (FIV) ou variance inflation factor (VIF) en anglais. Les FIV estiment de combien la variance d'un coefficient est augmentée en raison d'une relation linéaire avec d'autres prédicteurs. Ainsi, un FIV de 1,8 nous dit que la variance de ce coefficient particulier est supérieure de 80 % à la variance que l'on aurait dû observer si ce facteur n'est absolument pas corrélé aux autres prédicteurs.

Si tous les FIV sont égaux à 1, il n'existe pas de multicolinéarité, mais si certains FIV sont supérieurs à 1, les prédicteurs sont corrélés. Il n'y a pas de consensus sur la valeur au-delà de laquelle on doit considérer qu'il y a multicolinéarité. Certains auteurs, comme Paul Allison[44], disent de regarder plus en détail les variables avec un FIV supérieur à 2,5. D'autres ne s'inquiètent qu'à partir de 5. Il n'existe pas de test statistique qui permettrait de dire s'il y a colinéarité ou non[45].

L'extension `{car}` fournit une fonction `car::vif()` permettant de calculer les FIV à partir d'un modèle. Elle implémente même une version généralisée permettant de considérer des facteurs catégoriels et des modèles linéaires généralisés comme la régression logistique.

Reprenons, pour exemple, un modèle logistique que nous avons déjà abordé dans d'autres chapitres.

```
library(tidyverse)
library(labelled)

data(hdv2003, package = "questionr")

d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
```

[44] When Can You Safely Ignore Multicollinearity?

[45] Pour plus de détails, voir ce post de Davig Giles, Can You Actually TEST for Multicollinearity?, qui explique pourquoi ce n'est pas possible.

```
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                   "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
    ) |>
    fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    heures.tv = "Heures de télévision / jour"
  )

mod <- glm(
  sport ~ sexe + groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
```

Le calcul des FIV se fait simplement en passant le modèle à la fonction `car::vif()`.

```
mod |> car::vif()
```

```
                GVIF Df GVIF^(1/(2*Df))
sexe        1.024640  1        1.012245
groupe_ages 1.745492  3        1.097285
etudes      1.811370  4        1.077087
heures.tv   1.057819  1        1.028503
```

Dans notre exemple, tous les FIV sont proches de 1. Il n'y a donc pas de problème potentiel de colinéarité à explorer.

Pour un tableau propre, nous pouvons aussi utiliser `gtsummary::add_vif()`.

```
library(gtsummary)
theme_gtsummary_language(
  "fr",
  decimal.mark = ",",
  big.mark = " "
)
```

```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels() |>
  add_vif()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 26.1: Résumé du modèle logistique simple avec affichage des VIF généralisés

Caractéristique	**OR**	**95% IC**	**p-valeur**	**GVIF**	**Adjusted GVIF**
Sexe				1,0	1,0
Femme	—	—			
Homme	1,52	1,24 – 1,87	<0,001		
Groupe d'âges				1,7	1,1
18-24 ans	—	—			

Caractéristique	OR	95% IC	p-valeur	GVIF	Adjusted GVIF
25-44 ans	0,68	0,43 – 1,06	0,084		
45-64 ans	0,36	0,23 – 0,57	<0,001		
65 ans et plus	0,27	0,16 – 0,46	<0,001		
Niveau d'études				1,8	1,1
Primaire	—	—			
Secondaire	2,54	1,73 – 3,75	<0,001		
Technique / Professionnel	2,81	1,95 – 4,10	<0,001		
Supérieur	6,55	4,50 – 9,66	<0,001		
Non documenté	8,54	4,51 – 16,5	<0,001		
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001	1,1	1,0

26.3 La multicolinéarité est-elle toujours un problème ?

Là encore, il n'y a pas de consensus sur cette question. Certains analystes considèrent que tout modèle où certains prédicteurs seraient colinéaires n'est pas valable. Dans le billet When Can You Safely Ignore Multicollinearity?, Paul Allison évoque quant à lui des situations où la multicolinéarité peut être ignorée en toute sécurité. Le texte ci-dessous est une traduction de ce billet.

1. Les variables avec des FIV élevés sont des variables de contrôle, et les variables d'intérêt n'ont pas de FIV élevés.

Voici le problème de la multicolinéarité : ce n'est un problème que pour les variables qui sont colinéaires. Il augmente les erreurs-types de leurs coefficients et peut rendre ces coefficients instables de plusieurs façons. Mais tant que les variables colinéaires ne sont utilisées que comme variables de contrôle, et qu'elles ne sont pas colinéaires avec vos variables d'intérêt, il n'y a pas de problème. Les coefficients des variables d'intérêt ne sont pas affectés et la performance des variables de contrôle n'est pas altérée.

Voici un exemple tiré de ces propres travaux : l'échantillon est constitué de collèges américains, la variable dépendante est le taux d'obtention de diplôme et la variable d'intérêt est un indicateur (factice) pour les secteurs public et privé. Deux variables de contrôle sont les scores moyens au SAT et les scores moyens à l'ACT pour l'entrée en première année. Ces deux variables ont une corrélation supérieure à ,9, ce qui correspond à des FIV d'au moins 5,26 pour chacune d'entre elles. Mais le FIV pour l'indicateur public/privé n'est que de 1,04. Il n'y a donc pas de problème à se préoccuper et il n'est pas nécessaire de supprimer l'un ou l'autre des deux contrôles, à condition que l'on ne cherche pas à interpréter ou comparer l'un par rapport à l'autre les coefficients de ces deux variables de contrôle.

2. Les FIV élevés sont causés par l'inclusion de puissances ou de produits d'autres variables.

Si vous spécifiez un modèle de régression avec x et x^2, il y a de bonnes chances que ces deux variables soient fortement corrélées. De même, si votre modèle a x, z et xz, x et z sont susceptibles d'être fortement corrélés avec leur produit. Il n'y a pas de quoi s'inquiéter, car la valeur p de xz n'est pas affectée par la multicolinéarité. Ceci est facile à démontrer : vous pouvez réduire considérablement les corrélations en centrant les variables (c'est-à-dire en soustrayant leurs moyennes) avant de créer les puissances ou les produits. Mais la valeur p pour x^2 ou pour xz sera exactement la même, que l'on centre ou non. Et tous les résultats pour les autres variables (y compris le R^2 mais sans les termes d'ordre inférieur) seront les mêmes dans les deux cas. La multicolinéarité n'a donc pas de conséquences négatives.

3. Les variables avec des FIV élevés sont des variables

indicatrices (factices) qui représentent une variable catégorielle avec trois catégories ou plus.

Si la proportion de cas dans la catégorie de référence est faible, les variables indicatrices auront nécessairement des FIV élevés, même si la variable catégorielle n'est pas associée à d'autres variables dans le modèle de régression.

Supposons, par exemple, qu'une variable de l'état matrimonial comporte trois catégories : actuellement marié, jamais marié et anciennement marié. Vous choisissez anciennement marié comme catégorie de référence, avec des variables d'indicateur pour les deux autres. Ce qui se passe, c'est que la corrélation entre ces deux indicateurs devient plus négative à mesure que la fraction de personnes dans la catégorie de référence diminue. Par exemple, si 45 % des personnes ne sont jamais mariées, 45 % sont mariées et 10 % sont anciennement mariées, les valeurs du FIV pour les personnes mariées et les personnes jamais mariées seront d'au moins 3,0.

Est-ce un problème ? Eh bien, cela signifie que les valeurs p des variables indicatrices peuvent être élevées. Mais le test global selon lequel tous les indicateurs ont des coefficients de zéro n'est pas affecté par des FIV élevés. Et rien d'autre dans la régression n'est affecté. Si vous voulez vraiment éviter des FIV élevés, il suffit de choisir une catégorie de référence avec une plus grande fraction des cas. Cela peut être souhaitable pour éviter les situations où aucun des indicateurs individuels n'est statistiquement significatif, même si l'ensemble des indicateurs est significatif.

26.4 webin-R

La multicolinéarité est abordée dans le webin-R #07 (*régression logistique partie 2*) sur YouTube.

https://youtu.be/BUo9i7XTLYQ

partie IV

Données pondérées avec `survey`

27 Définir un plan d'échantillonnage

Lorsque l'on travaille avec des données d'enquêtes, il est fréquent que les données soient **pondérées**. Cette pondération est nécessaire pour assurer la représentativité des données lorsque les participants ont été sélectionnés avec des probabilités variables. C'est par exemple le cas lorsqu'on réalise une enquête en grappes et/ou stratifiée.

De nombreuses fonctions acceptent une variable de pondération. Cependant, la simple prise en compte de la pondération est souvent insuffisante si l'on ne tient pas compte du plan d'échantillonnage de l'enquête. 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.

Diverses fonctions de **R** peuvent prendre en compte une variable de pondération. Mais, en règle générale, elles sont incapables de tenir compte du plan d'échantillonnage. Il est donc préférable de privilégier le package `{survey}` qui est spécialement dédié au traitement d'enquêtes ayant des techniques d'échantillonnage et de pondération potentiellement très complexes. `{survey}` peut également être utilisée pour des pondérations simples.

27.1 Différents types d'échantillonnage

L'**échantillonnage aléatoire simple** ou **échantillonnage équiprobable** est une méthode pour laquelle tous les échan-

tillons 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 une 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 ou clusters en anglais). 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*[46] (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 aléatoire simple.

[46] Vaste programme d'enquêtes réalisées à intervalles réguliers dans les pays à faible et moyen revenu, disponibles sur https://dhsprogram.com/.

27.2 Avec `survey::svydesign()`

La fonction `survey::svydesign()` accepte plusieurs arguments décrits en détail 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 /grappe 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.

On peut tout à fait définir un **échantillonnage aléatoire simple** (on considère donc que toutes les observations ont

le même poids, égal à 1). Pour rappel, en l'absence de clusters/grappes, il faut préciser `ids = ~ 1`, ce paramètre n'ayant pas de valeur par défaut.

```
p_iris <- survey::svydesign(
  ids = ~ 1,
  data = iris
)
```

```
Warning in svydesign.default(ids = ~1, data = iris): No weights or
probabilities supplied, assuming equal probability
```

```
p_iris
```

```
Independent Sampling design (with replacement)
survey::svydesign(ids = ~1, data = iris)
```

Pour un jeu de données **simplement pondéré** (chaque ligne représente plusieurs observations) :

```
titanic <- dplyr::as_tibble(Titanic)
titanic |> labelled::look_for()
```

```
pos variable label col_type missing values
1   Class    -     chr      0
2   Sex      -     chr      0
3   Age      -     chr      0
4   Survived -     chr      0
5   n        -     dbl      0
```

```
p_titanic <- survey::svydesign(
  ids = ~ 1,
  data = titanic,
  weights = ~ n
)
p_titanic
```

```
Independent Sampling design (with replacement)
survey::svydesign(ids = ~1, data = titanic, weights = ~n)
```

Pour un **échantillon stratifié** pour lequel les strates sont indiquées dans la variable *stype* et les poids indiquées dans la variable *pw*.

```r
data("api", package = "survey")
p_strates <- survey::svydesign(
  id = ~ 1,
  strata = ~ stype,
  weights = ~ pw,
  data = apistrat
)
p_strates
```

```
Stratified Independent Sampling design (with replacement)
survey::svydesign(id = ~1, strata = ~stype, weights = ~pw, data = apistrat)
```

Pour une **enquête en grappes à 1 degré**, pour laquelle l'identifiant des grappes (*clusters*) est indiqué par la variable *dnum*.

```r
data("api", package = "survey")
p_grappes <- survey::svydesign(
  id = ~ dnum,
  weights = ~ pw,
  data = apiclus1
)
p_grappes
```

```
1 - level Cluster Sampling design (with replacement)
With (15) clusters.
survey::svydesign(id = ~dnum, weights = ~pw, data = apiclus1)
```

Voici un exemple un peu plus complexe d'une **enquête en grappes à deux degrés** (les deux niveaux étant donnés par les variables *dnum* et *snum*). Les poids ne sont pas fournis mais

la taille des grappes est connue et renseignée dans les variables *fpc1* et *fpc2* que nous pourrons donc transmettre via l'argument `fpc`.

```
data("api", package = "survey")
p_grappes2 <- survey::svydesign(
  id = ~ dnum + snum,
  fpc = ~ fpc1 + fpc2,
  data = apiclus2
)
p_grappes2
```

```
2 - level Cluster Sampling design
With (40, 126) clusters.
survey::svydesign(id = ~dnum + snum, fpc = ~fpc1 + fpc2, data = apiclus2)
```

Dans le cas présent, `{survey}` a calculé les poids s'appliquant à chaque individu. On peut les obtenir avec la fonction `weights()`, en l'occurrence avec `p_grappes2 |> weights()`.

Enfin, 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 utiliser 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 milieu 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 :

```
eds$poids <- eds$V005/1000000
p_eds <- survey::svydesign(
  ids = ~ V021,
  data = eds,
```

```
  strata = ~ V022,
  weights = ~ poids
)
```

27.3 Avec `srvyr::as_survey_design()`

Dans le prochain chapitre (cf. Chapitre **??**), nous aborderons le package `{srvyr}` qui est aux objets `{survey}` ce que `{dplyr}` est aux tableaux de données : plus précisément, ce package étend les verbes de `{dplyr}` aux plans d'échantillonnage complexe.

La fonction `srvyr::as_survey_design()` est équivalente à `survey::svydesign()` mais avec quelques différences :

- le paramètre `ids` dispose d'une valeur par défaut et on peut l'ignorer en l'absence de grappes ;
- les variables ne sont pas spécifiées avec une formule mais avec les mêmes sélecteurs que `dplyr::select()` ;
- l'objet renvoyé est à la fois du type `"survey.design"` et du type `"tbl_svy"`, une sorte de *tibble* pour les objets `{survey}`.

Reprenons nos exemples précédents en commençant par un **échantillonnage aléatoire simple**.

```
t_iris <- iris |>
  srvyr::as_survey_design()
t_iris
```

```
Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
 - ids: `1`
Data variables: Sepal.Length (dbl), Sepal.Width (dbl), Petal.Length (dbl),
  Petal.Width (dbl), Species (fct)
```

```
class(t_iris)
```

```
[1] "tbl_svy"        "survey.design2" "survey.design"
```

Pour un jeu de données **simplement pondéré** (chaque ligne représente plusieurs observations) :

```
titanic <- dplyr::as_tibble(Titanic)
t_titanic <- titanic |>
  srvyr::as_survey_design(weights = n)
t_titanic
```

```
Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
 - ids: `1`
 - weights: n
Data variables: Class (chr), Sex (chr), Age (chr), Survived (chr), n (dbl)
```

Pour un **échantillon stratifié** pour lequel les strates sont indiquées dans la variable *stype* et les poids indiquées dans la variable *pw*.

```
data("api", package = "survey")
t_strates <- apistrat |>
  srvyr::as_survey_design(strata = stype, weights = pw)
t_strates
```

```
Stratified Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
 - ids: `1`
 - strata: stype
 - weights: pw
Data variables: cds (chr), stype (fct), name (chr), sname (chr), snum (dbl),
  dname (chr), dnum (int), cname (chr), cnum (int), flag (int), pcttest (int),
  api00 (int), api99 (int), target (int), growth (int), sch.wide (fct),
  comp.imp (fct), both (fct), awards (fct), meals (int), ell (int), yr.rnd
  (fct), mobility (int), acs.k3 (int), acs.46 (int), acs.core (int), pct.resp
  (int), not.hsg (int), hsg (int), some.col (int), col.grad (int), grad.sch
  (int), avg.ed (dbl), full (int), emer (int), enroll (int), api.stu (int), pw
  (dbl), fpc (dbl)
```

Pour une **enquête en grappes à 1 degré**, pour laquelle l'identifiant des grappes (*clusters*) est indiqué par la variable *dnum*.

```
data("api", package = "survey")
t_grappes <- apiclus1 |>
    srvyr::as_survey_design(id = dnum, weights = pw)
t_grappes
```

```
1 - level Cluster Sampling design (with replacement)
With (15) clusters.
Called via srvyr
Sampling variables:
 - ids: dnum
 - weights: pw
Data variables: cds (chr), stype (fct), name (chr), sname (chr), snum (dbl),
  dname (chr), dnum (int), cname (chr), cnum (int), flag (int), pcttest (int),
  api00 (int), api99 (int), target (int), growth (int), sch.wide (fct),
  comp.imp (fct), both (fct), awards (fct), meals (int), ell (int), yr.rnd
  (fct), mobility (int), acs.k3 (int), acs.46 (int), acs.core (int), pct.resp
  (int), not.hsg (int), hsg (int), some.col (int), col.grad (int), grad.sch
  (int), avg.ed (dbl), full (int), emer (int), enroll (int), api.stu (int), fpc
  (dbl), pw (dbl)
```

Voici un exemple un peu plus complexe d'une **enquête en grappes à deux degrés** (les deux niveaux étant donnés par les variables *dnum* et *snum*). Les poids ne sont pas fournis mais la taille des grappes est connue et renseignée dans les variables *fpc1* et *fpc2* que nous pourrons donc transmettre via l'argument `fpc`.

```
data("api", package = "survey")
data("api", package = "survey")
t_grappes2 <- apiclus2 |>
    srvyr::as_survey_design(id = c(dnum, snum), fpc = c(fpc1, fpc2))
t_grappes2
```

```
2 - level Cluster Sampling design
With (40, 126) clusters.
```

```
Called via srvyr
Sampling variables:
 - ids: `dnum + snum`
 - fpc: `fpc1 + fpc2`
Data variables: cds (chr), stype (fct), name (chr), sname (chr), snum (dbl),
  dname (chr), dnum (int), cname (chr), cnum (int), flag (int), pcttest (int),
  api00 (int), api99 (int), target (int), growth (int), sch.wide (fct),
  comp.imp (fct), both (fct), awards (fct), meals (int), ell (int), yr.rnd
  (fct), mobility (int), acs.k3 (int), acs.46 (int), acs.core (int), pct.resp
  (int), not.hsg (int), hsg (int), some.col (int), col.grad (int), grad.sch
  (int), avg.ed (dbl), full (int), emer (int), enroll (int), api.stu (int), pw
  (dbl), fpc1 (dbl), fpc2 (int[1d])
```

Enfin, 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 utiliser 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 milieu 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$poids <- eds$V005/1000000
t_eds <- eds |>
  srvyr::as_survey_design(
    ids = V021,
    strata = V022,
    weights = poids
  )
```

27.4 webin-R

La statistique univariée est présentée dans le webin-R #10 (*données pondérées, plan d'échantillonnage complexe & survey*) sur YouTube.

https://youtu.be/aXCn9SyhcTE

28 Manipulation de données pondérées

L'objet créé avec `survey::svydesign()` ou `srvyr::as_survey_design()` n'est plus un tableau de données, mais plutôt un tableau de données auquel est attaché un plan d'échantillonnage. Les colonnes du tableau d'origine ne sont plus directement accessibles avec l'opérateur `$`. En fait, elles sont stockées dans un sous-objet `$variables`.

```
titanic <- dplyr::as_tibble(Titanic)
t_titanic <- titanic |>
  srvyr::as_survey_design(weights = n)
t_titanic$variables |> dplyr::glimpse()
```

```
Rows: 32
Columns: 5
$ Class    <chr> "1st", "2nd", "3rd", "Crew", "1st", "2nd", "3rd", "Crew", "1s~
$ Sex      <chr> "Male", "Male", "Male", "Male", "Female", "Female", "Female",~
$ Age      <chr> "Child", "Child", "Child", "Child", "Child", "Child", "Child"~
$ Survived <chr> "No", "No", "No", "No", "No", "No", "No", "No", "No", "No", "~
$ n        <dbl> 0, 0, 35, 0, 0, 0, 17, 0, 118, 154, 387, 670, 4, 13, 89, 3, 5~
```

Il n'est pas aisé de modifier des variables dans un objet de ce type. Il est donc préférable de procéder à l'ensemble des nettoyages, recodages de variables (et au besoin transformation des vecteurs labellisés en facteur), avant de définir le plan d'échantillonnage et de procéder aux analyses.

Si l'on souhaite manipuler les données, le plus simple est d'avoir recours au package `{srvyr}` qui étend les verbes de `{dplyr}` (cf. Chapitre **??**) aux objets `{survey}`.

28.1 Utilisation de {srvyr}

{srvyr} fournit les verbes `srvyr::select()` et `srvyr::filter()` pour sélectionner respectivement des colonnes et des lignes.

```
library(srvyr)
```

```
Attachement du package : 'srvyr'
```

```
L'objet suivant est masqué depuis 'package:stats':
```

```
    filter
```

```
t_titanic |> select(Sex, Age)
```

```
Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
 - ids: `1`
 - weights: n
Data variables: Sex (chr), Age (chr)
```

```
t_titanic |> filter(Sex == "Female")
```

```
Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
 - ids: `1`
 - weights: n
Data variables: Class (chr), Sex (chr), Age (chr), Survived (chr), n (dbl)
```

On peut aussi utiliser `srvyr::pull()` pour extraire le contenu d'une colonne ou `srvyr::drop_na()` pour supprimer les observations contenant des valeurs manquantes.

> **Avertissement**
>
> Par contre, le verbe `arrange()` (tri du tableau) ou encore les fonctions de jointures (telles que `left_join()`) ne sont pas implémentées car ce type d'opération entraînerait des modifications du plan d'échantillonnage. Il est donc préférable de réaliser ce type d'opérations avant la déclaration du plan d'échantillonnage (quand les données sont donc encore stockées dans un tableau de données classiques).

`srvyr` fournit également le verbe `srvyr::summarize()` permettant de calculer des statistiques sur l'ensemble du fichier ou par sous-groupe (en combinant `summarize()` avec `group_by()`). Afin de prendre en compte correctement la pondération et le plan d'échantillonnage, `srvyr` fournit des fonctions adaptées pour un usage au sein de `summarize()` : `srvyr::survey_mean()`, `srvyr::survey_total()`, `srvyr::survey_prop()`, `srvyr::survey_ratio()`, `srvyr::survey_quantile()` ou encore `srvyr::survey_median()`.

```
t_titanic |>
  group_by(Sex, Class, Survived) |>
  summarise(taux_survie = survey_prop()) |>
  filter(Survived == "Yes")
```

```
When `proportion` is unspecified, `survey_prop()` now defaults to `proportion = TRUE`.
i This should improve confidence interval coverage.
This message is displayed once per session.
```

```
Warning: There were 24 warnings in `dplyr::summarise()`.
The first warning was:
i In argument: `taux_survie = survey_prop()`.
i In group 1: `Sex = "Female"`, `Class = "1st"`, `Survived = "No"`.
Caused by warning in `summary.glm()`:
! les observations de poids nul n'ont pas été utilisées pour le calcul de la dispersion
i Run `dplyr::last_dplyr_warnings()` to see the 23 remaining warnings.
```

```
# A tibble: 8 x 5
# Groups:   Sex, Class [8]
```

```
  Sex    Class Survived taux_survie taux_survie_se
  <chr>  <chr> <chr>          <dbl>          <dbl>
1 Female 1st   Yes            0.972         0.0384
2 Female 2nd   Yes            0.877         0.145
3 Female 3rd   Yes            0.459         0.306
4 Female Crew  Yes            0.870         0.163
5 Male   1st   Yes            0.344         0.312
6 Male   2nd   Yes            0.140         0.150
7 Male   3rd   Yes            0.173         0.183
8 Male   Crew  Yes            0.223         0.249
```

28.2 Lister / Rechercher des variables

La fonction `labelled::look_for()`, que nous avons déjà abordée (cf. Section **??**), est compatible avec les objets `{survey}` et peut donc être utilisée pour lister ou rechercher des variables.

```
t_titanic <- titanic |>
  labelled::set_variable_labels(
    Class = "Class du passager",
    Sex = "Sexe du passager",
    Age = "Enfant ou adulte ?",
    Survived = "A survécu au naufrage ?",
    n = "Nombre d'observations"
  ) |>
  srvyr::as_survey_design(weights = n)
t_titanic |> labelled::look_for()
```

```
pos variable label                  col_type missing values
1   Class    Class du passager      chr      0
2   Sex      Sexe du passager       chr      0
3   Age      Enfant ou adulte ?     chr      0
4   Survived A survécu au naufrage ? chr      0
5   n        Nombre d'observations  dbl      0
```

```
t_titanic |> labelled::look_for("nau")
```

```
pos variable label                    col_type missing values
4   Survived A survécu au naufrage ? chr      0
```

28.3 Extraire un sous-échantillon

Si l'on souhaite travailler sur un sous-échantillon de l'enquête, il importe de définir le plan d'échantillonnage sur l'ensemble du jeu de données **avant** de procéder à la sélection des observations.

La fonction classique pour sélectionner des lignes est `subset()`. Cependant, elle a un inconvénient lorsque nos données comportent des étiquettes de variables (cf. Chapitre **??**) ou de valeurs (Chapitre **??**), car les étiquettes ne sont pas conservées après l'opération.

On préférera donc avoir recours à `srvyr::filter()` qui conservent les attributs associés aux colonnes du tableau de données.

```
t_subset <- t_titanic |> subset(Sex == "Female")
t_subset |> labelled::look_for()
```

```
pos variable label col_type missing values
1   Class    -     chr      0
2   Sex      -     chr      0
3   Age      -     chr      0
4   Survived -     chr      0
5   n        -     dbl      0
```

```
t_filter <- t_titanic |> filter(Sex == "Female")
t_filter |> labelled::look_for()
```

```
pos variable label                    col_type missing values
1   Class    Class du passager        chr      0
2   Sex      Sexe du passager         chr      0
3   Age      Enfant ou adulte ?       chr      0
4   Survived A survécu au naufrage ? chr      0
5   n        Nombre d'observations    dbl      0
```

29 Analyses uni- et bivariées pondérées

29.1 La fonction `tbl_svysummary()`

Dans les chapitres sur la statistique univariée (cf. Chapitre **??**) et la statistique bivariée (cf. Chapitre **??**), nous avons abordé la fonction `gtsummary::tbl_summary()` qui permet de générer des tris à plats et des tableaux croisés prêts à être publiés.

Son équivalent pour les objets `{survey}` existe : il s'agit de la fonction `gtsummary::tbl_svysummary()` qui fonctionne de manière similaire.

Pour illustrer son fonctionnement, nous allons utiliser le jeu de données *fecondite* fournit dans le package `{questionr}`. Ce jeu de données fournit un tableau de données `femmes` comportant une variable *poids* de pondération que nous allons utiliser. Les données catégorielles étant stockées sous forme de vecteurs numériques avec étiquettes de valeurs (cf. Chapitre **??**), nous allons les convertir en facteurs avec `labelled::unlabelled()`. De même, certaines valeurs manquantes sont indiquées sous formes de *user NAs* (cf. Section **??**) : nous allons les convertir en valeurs manquantes classiques (*regular NAs*) avec `labelled::user_na_to_na()`.

```
data("fecondite", package = "questionr")
library(srvyr)
```

```
Attachement du package : 'srvyr'
```

```
L'objet suivant est masqué depuis 'package:stats':
```

```
    filter
```

```
dp <- femmes |>
  labelled::user_na_to_na() |>
  labelled::unlabelled() |>
  as_survey_design(weights = poids)
dp
```

```
Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
 - ids: `1`
 - weights: poids
Data variables: id_femme (dbl), id_menage (dbl), poids (dbl), date_entretien
  (date), date_naissance (date), age (dbl), milieu (fct), region (fct), educ
  (fct), travail (fct), matri (fct), religion (fct), journal (fct), radio
  (fct), tv (fct), nb_enf_ideal (dbl), test (fct)
```

Chargeons `{gtsummary}` et définissons le français comme langue de rendu des tableaux.

```
library(gtsummary)
theme_gtsummary_language(
  language = "fr",
  decimal.mark = ",",
  big.mark = " "
)
```

```
Setting theme `language: fr`
```

Pour réaliser un tableau croisé, il nous suffit d'appeler `gtsummary::tbl_svysummary()` de la même manière que l'on aurait procédé avec `gtsummary::tbl_summary()`. En arrière plan, `gtsummary::tbl_svysummary()` appellera les différentes fonctions statistiques de `{survey}` : la pondération ainsi que les spécificités du plan d'échantillonnage seront donc correctement prises en compte.

```r
dp |>
  tbl_svysummary(
    by = milieu,
    include = c(age, educ, travail)
  ) |>
  add_overall(last = TRUE) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 29.1: Tableau croisé sur des données pondérées

Caractéristique	**urbain**, N = 1 026	**rural**, N = 1 002	**Total**, N = 2 027
Âge révolu (en années) à la date de passation du questionnaire	26 (20 – 33)	28 (22 – 36)	27 (21 – 35)
Niveau d'éducation			
aucun	414 (40%)	681 (68%)	1 095 (54%)
primaire	251 (24%)	257 (26%)	507 (25%)
secondaire	303 (30%)	61 (6,1%)	364 (18%)
supérieur	58 (5,7%)	3 (0,3%)	61 (3,0%)
A un emploi ?			
non	401 (39%)	269 (27%)	670 (33%)
oui	621 (61%)	731 (73%)	1 351 (67%)
Manquant	5	1	6

! Important

Par défaut, les effectifs (ainsi que les pourcentages et autres statistiques) affichés sont pondérés. Il est important de bien comprendre ce que représentent ces effectifs pondérés pour les interpréter correctement. Pour cela, il faut savoir comment les poids de l'enquête ont été calculés.
Dans certains cas, lorsque la population totale est connue, la somme des poids est égale à cette population totale dans laquelle l'échantillon a été tiré au sort. Les effectifs pondérés représentent donc une estimation des effectifs dans la population totale et ne représentent en rien le nombre d'observations dans l'enquête.
Dans d'autres enquêtes, les poids sont générés de telle manière que la somme des poids correspondent au nombre total de personnes enquêtées. Dans ce genre de situation, on a souvent tendance, à tort, à interpréter les effectifs pondérés comme un nombre d'observations. Or, il peut y avoir un écart important entre le nombre d'observations dans l'enquête et les effectifs pondérés.
On pourra éventuellement présenter séparément le nombre d'observations (i.e. les effectifs non pondérés) et les proportions pondérées. `gtsummary::tbl_svysummary()` fournit justement à la fois ces données pondérées et non pondérées. Il est vrai que cela nécessite quand même quelques manipulations. Pour les cellules, on précisera le type d'effectifs à afficher avec l'argument `statistic`. Pour personnaliser l'affiche du nombre de valeurs manquantes, cela doit se faire à un niveau plus global via `gtsummary::set_gtsummary_theme()`. Enfin, on passera par `gtsummary::modify_header()` pour personnaliser les en-têtes de colonne.

```
set_gtsummary_theme(
  list("tbl_summary-str:missing_stat" = "{N_miss_unweighted} obs.")
)
dp |>
  tbl_svysummary(
    by = milieu,
    include = c(educ, travail),
    statistic = all_categorical() ~ "{p}% ({n_unweighted} obs.)",
    digits = all_categorical() ~ c(1, 0)
  ) |>
  modify_header(
    all_stat_cols() ~ "**{level}** ({n_unweighted} obs.)"
  ) |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Caractéristique	**urbain** (912 obs.)	**rural** (1088 obs.)
Niveau d'éducation		
aucun	40,3% (375 obs.)	68,0% (763 obs.)
primaire	24,4% (213 obs.)	25,6% (247 obs.)
secondaire	29,5% (275 obs.)	6,1% (73 obs.)
supérieur	5,7% (49 obs.)	0,3% (5 obs.)
A un emploi ?		
non	39,2% (370 obs.)	26,9% (296 obs.)
oui	60,8% (537 obs.)	73,1% (790 obs.)
Manquant	5 obs.	2 obs.

Il faut noter qu'une modification du thème impactera tous les tableaux suivants, jusqu'à ce que le thème soit à nouveau modifié ou bien que l'on fasse appel à `gtsummary::reset_gtsummary_theme()`.

29.2 Calcul manuel avec `{survey}`

Lorsque l'on travail avec un plan d'échantillonnage, on ne peut utiliser les fonctions statistiques classiques de **R**. On aura recours à leurs équivalents fournis par `{survey}` :

- `survey::svymean()`, `survey::svyvar()`, `survey::svytotal()`, `survey::svyquantile()` : moyenne, variance, total, quantiles
- `survey::svytable()` : tri à plat et tableau croisé
- `survey::svychisq()` : test du ^{2}
- `survey::svyby()` : statistiques selon un facteur
- `survey::svyttest()` : test t de Student de comparaison de moyennes
- `survey::svyciprop()` : intervalle de confiance d'une proportion
- `survey::svyratio()` : ratio de deux variables continues

Ces fonctions prennent leurs arguments sous forme de formules pour spécifier les variables d'intérêt.

```
survey::svymean(~ age, dp)
```

```
      mean     SE
age 28.468 0.2697
```

```
survey::svymean(~ age, dp) |> confint()
```

```
       2.5 %   97.5 %
age 27.93931 28.99653
```

```
survey::svyquantile(~age, dp, quantile = c(0.25, 0.5, 0.75), ci = TRUE)
```

```
$age
     quantile ci.2.5 ci.97.5        se
0.25       21     21      22 0.2549523
0.5        27     27      28 0.2549523
0.75       35     35      37 0.5099045

attr(,"hasci")
[1] TRUE
attr(,"class")
[1] "newsvyquantile"
```

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 (`~`)[47].

[47] Cette syntaxe est similaire à celle de `xtabs()`.

```
survey::svytable(~region, dp)
```

```
region
    Nord      Est      Sud    Ouest
611.0924 175.7404 329.2220 911.2197
```

```
survey::svytable(~milieu + educ, dp)
```

```
        educ
milieu        aucun   primaire secondaire  supérieur
  urbain 413.608780 250.665214 303.058978  58.412688
  rural  681.131096 256.694363  61.023980   2.679392
```

La fonction `questionr::freq()` peut être utilisée si on lui passe en argument non pas la variable elle-même, mais son tri à plat obtenu avec `survey::svytable()` :

```
survey::svytable(~region, dp) |>
  questionr::freq(total = TRUE)
```

```
           n     %  val%
Nord   611.1  30.1  30.1
Est    175.7   8.7   8.7
Sud    329.2  16.2  16.2
Ouest  911.2  44.9  44.9
Total 2027.3 100.0 100.0
```

Les fonctions `questionr::rprop()` et `questionr::cprop()` peuvent être utilisées pour calculer les pourcentages en ligne ou en colonne.

```
survey::svytable(~milieu + educ, dp) |>
  questionr::cprop()
```

```
        educ
milieu   aucun primaire secondaire supérieur Ensemble
  urbain  37.8  49.4     83.2       95.6      50.6
  rural   62.2  50.6     16.8        4.4      49.4
  Total  100.0 100.0    100.0      100.0     100.0
```

Le principe de la fonction `survey::svyby()` est similaire à celui de `tapply()` (cf. Section **??**). Elle permet de calculer des statistiques selon plusieurs sous-groupes définis par un facteur.

```
survey::svyby(~age, ~region, dp, survey::svymean)
```

```
      region      age        se
Nord    Nord 29.03299 0.4753268
Est      Est 27.54455 0.5261669
Sud      Sud 28.96830 0.6148223
Ouest  Ouest 28.08626 0.4458201
```

29.3 Intervalles de confiance et tests statistiques

La fonction `gtsummary::add_ci()` peut être appliquée à des tableaux produits avec `gtsummary::tbl_svysummary()`[48]. Les

[48] Cela requiert une version récente (1.7.0) de `{gtsummary}`.

méthodes utilisées sont adaptées à la prise en compte d'un plan d'échantillonnage. On se référera à la document de la fonction pour plus de détails sur les méthodes statistiques utilisées. **Rappel :** pour les variables continues, on sera vigilant à ce que la statistique affichée (médiane par défaut) corresponde au type d'intervalle de confiance calculé (moyenne par défaut).

```
dp |>
  tbl_svysummary(
    include = c(age, region),
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
  add_ci() |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 29.3: Intervalles de confiance avec prise en compte du plan d'échantillonnage

Caractéristique	**N = 2 027**	**95% CI**
Âge révolu (en années) à la date de passation du questionnaire	28 (9)	28, 29
Région de résidence		
Nord	611 (30%)	28%, 33%
Est	176 (8,7%)	7,7%, 9,8%
Sud	329 (16%)	14%, 18%
Ouest	911 (45%)	42%, 48%

De même, on peut aisément effectuer des tests de comparaison avec `gtsummary::add_p()`. Là aussi, les tests utilisés sont des

adaptations des tests classiques avec différentes corrections pour tenir compte à la fois de la pondération et du plan d'échantillonnage.

```
dp |>
  tbl_svysummary(
    include = c(age, region),
    by = milieu
  ) |>
  add_p() |>
  bold_labels()
```

```
Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 29.4: Tests de comparaison avec prise en compte du plan d'échantillonnage

Caractéristique	**urbain**, N = 1 026	**rural**, N = 1 002	**p-valeur**
Âge révolu (en années) à la date de passation du questionnaire	26 (20 – 33)	28 (22 – 36)	<0,001
Région de résidence			<0,001
Nord	265 (26%)	346 (35%)	
Est	48 (4,7%)	128 (13%)	
Sud	79 (7,7%)	250 (25%)	
Ouest	633 (62%)	278 (28%)	

29.4 Impact du plan d'échantillonnage

Lorsque l'on calcul des proportions, moyennes ou médianes pondérées, seuls les poids entrent en ligne de compte. Le plan d'échantillonnage (strates et/ou grappes) n'a de son côté pas d'effet. Par contre, le plan d'échantillonnage a un impact important sur le calcul des variances et, par extension, sur le calcul des intervalles de confiance et des tests de comparaison.

Pour illustrer cela, nous allons considérer un même jeu de données, avec la même variable de poids, mais en faisant varier la présence de strates et de grappes.

Commençons par regarder le jeu de données *apistrat* fourni par `{survey}`.

```
data("api", package = "survey")
nrow(apistrat)
```

```
[1] 200
```

```
summary(apistrat$pw)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  15.10   19.05   32.28   30.97   44.21   44.21 
```

```
sum(apistrat$pw)
```

```
[1] 6194
```

Nous avons ici un tableau de données de 200 lignes, avec des poids variant entre 15 et 44. Nous pouvons définir une pondération simple et croiser deux variables.

```
d_ponderation_simple <- apistrat |>
  as_survey_design(weights = pw)
tbl <- survey::svytable(~ awards + yr.rnd, design = d_ponderation_simple)
```

```
  tbl
```

```
      yr.rnd
awards      No     Yes
   No  2068.34  168.09
   Yes 3274.06  683.51
```

Réalisons un test du Chi² entre ces deux variables. Si nous appliquions la fonction classique `chisq.test()` sur ce tableau, cette fonction considérait que nous avons 6194 observations (somme des poids) et dès lors nous obtiendrions une p-valeur très faible.

```
  tbl |> chisq.test()
```

```
    Pearson's Chi-squared test with Yates' continuity correction

data:  tbl
X-squared = 113.84, df = 1, p-value < 2.2e-16
```

Le calcul précédent ne tient pas compte que nous n'avons que 200 observations dans notre échantillon. Refaisons le calcul `survey::svychisq()` qui est adaptée aux plans d'échantillonnage.

```
  survey::svychisq(~ awards + yr.rnd, design = d_ponderation_simple)
```

```
    Pearson's X^2: Rao & Scott adjustment

data:  NextMethod()
F = 2.9162, ndf = 1, ddf = 199, p-value = 0.08926
```

Le résultat est ici tout autre et notre test n'est plus significatif au seuil de 5% ! Ici, les corrections de Rao & Scott permettent justement de tenir compte que nous avons un échantillon de seulement 200 observations.

Regardons maintenant si, à poids égal, il y a une différence entre une enquête stratifiée et une enquête en grappes.

```
  # Pondération simple
  survey::svytable(~ awards + yr.rnd, design = d_ponderation_simple)
```

```
      yr.rnd
awards      No     Yes
   No  2068.34  168.09
   Yes 3274.06  683.51
```

```
  survey::svychisq(~ awards + yr.rnd, design = d_ponderation_simple)
```

```
    Pearson's X^2: Rao & Scott adjustment

data:  NextMethod()
F = 2.9162, ndf = 1, ddf = 199, p-value = 0.08926
```

```
  # Enquête stratifiée
  d_strates <- apistrat |>
    as_survey_design(weights = pw, strata = stype)
  survey::svytable(~ awards + yr.rnd, design = d_strates)
```

```
      yr.rnd
awards      No     Yes
   No  2068.34  168.09
   Yes 3274.06  683.51
```

```
  survey::svychisq(~ awards + yr.rnd, design = d_strates)
```

```
    Pearson's X^2: Rao & Scott adjustment

data:  NextMethod()
F = 2.9007, ndf = 1, ddf = 197, p-value = 0.09012
```

```r
# Enquête en grappes
d_grappes <- apistrat |>
  as_survey_design(weights = pw, ids = dnum)
survey::svytable(~ awards + yr.rnd, design = d_grappes)
```

```
      yr.rnd
awards      No     Yes
   No  2068.34  168.09
   Yes 3274.06  683.51
```

```r
survey::svychisq(~ awards + yr.rnd, design = d_grappes)
```

```
    Pearson's X^2: Rao & Scott adjustment

data:  NextMethod()
F = 3.1393, ndf = 1, ddf = 134, p-value = 0.0787
```

On le constate : dans les trois cas les tableaux croisés sont identiques, mais pour autant les trois p-valeurs diffèrent.

Dès lors qu'un calcul de variance est impliqué, la simple prise en compte des poids est insuffisante : il faut appliquer des corrections en fonction du plan d'échantillonnage !

Pas d'inquiétude, `{survey}` s'en occupe pour vous, dès lors que le plan d'échantillonnage a correctement été défini.

30 Graphiques pondérés

Le package `{ggplot2}` n'est compatible directement avec les objets `{survey}`. Cependant, il accepte une esthétique *weight* qui permet de définir une variable de pondération.

> ⚠ Avertissement
>
> **ATTENTION :** les graphiques obtenus ne sont corrects qu'à la condition que seuls les poids soient nécessaires pour les construire, ce qui est le cas d'un nuage de points ou d'un diagramme en barres.
> Par contre, si le calcul du graphique implique le calcul de variances, la représentation sera incorrecte. Par exemple, avec `ggplot2::geom_smooth()`, les intervalles de confiance affichés ne prendront pas correctement en compte le plan d'échantillonnage.

Reprenons le jeu de données *fecondite* que nous avons abordé dans le chapitre sur les analyses bivariées pondérées, cf. Chapitre **??**. Les poids d'enquête y sont indiqués dans la colonne *poids*. Pour rappel, les données catégorielles étant stockées sous forme de vecteurs numériques avec étiquettes de valeurs (cf. Chapitre **??**), nous allons les convertir en facteurs avec `labelled::unlabelled()`.

```
data("fecondite", package = "questionr")
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.1     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.1     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
```

```
v purrr     1.0.1
-- Conflicts ---------------------------------------- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
d <- labelled::unlabelled(femmes)
```

Pour réaliser un graphique, nous pouvons reprendre ce que nous avons vu dans notre chapitre introductif sur `{ggplot2}`, cf. Chapitre **??**, en spécifiant simplement l'esthétique *weight*.

```
ggplot(d) +
  aes(x = region, fill = test, weight = poids) +
  geom_bar(position = "fill")
```

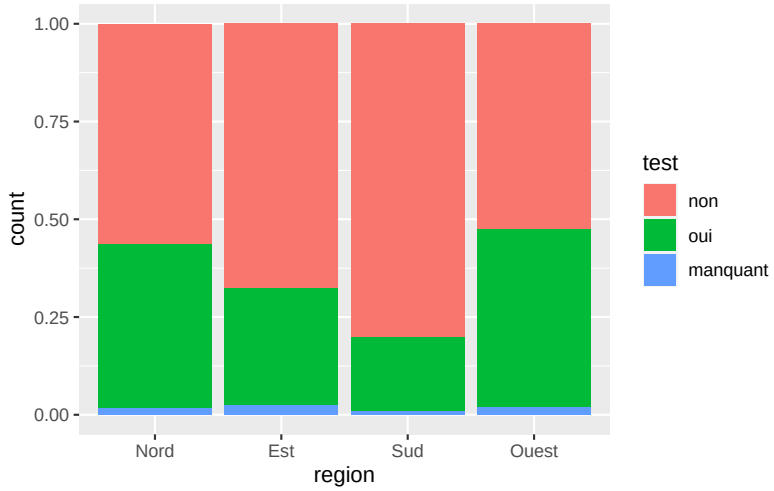


Figure 30.1: Un graphique en barres pondéré

Si l'on a déjà créé un objet `{survey}`, on peut obtenir les poids des observations avec la fonction `weights()`. Les données sont quant à elle accessibles via le sous-élément nommé *variables*.

```
library(srvyr)
```

```
Attachement du package : 'srvyr'

L'objet suivant est masqué depuis 'package:stats':

    filter
```

```
dp <- femmes |>
  labelled::unlabelled() |>
  as_survey_design(weights = poids)
```

```
ggplot(dp$variables) +
  aes(x = region, fill = test, weight = weights(dp)) +
  geom_bar(position = "fill")
```

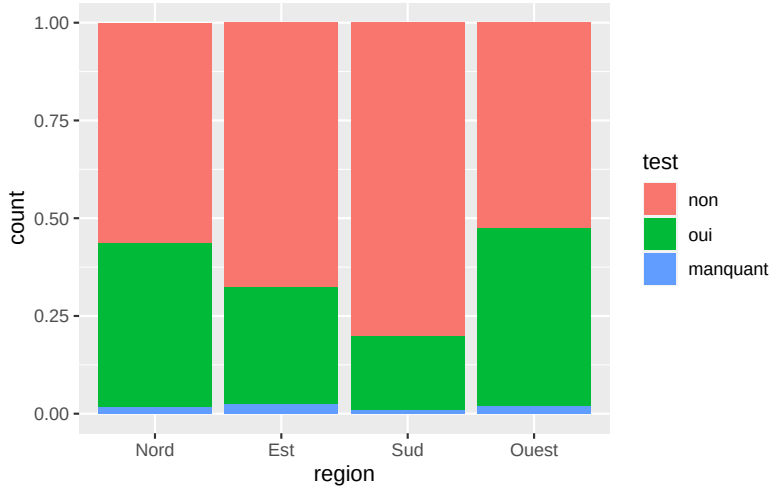


Figure 30.2: Un graphique en barres pondéré

Pour se faciliter les choses, on peut avoir directement recours à la fonction `ggstats::ggsurvey()`, que l'on utilisera à la place de `ggplot2::ggplot()`, et qui fait exactement la même chose que dans notre exemple précédent : on lui passe un objet de type `{survey}` et la fonction en extrait le sous-élément *variables* pour le passer à `ggplot2::ggplot()` et les poids qui sont automatiquement associés à l'esthétique *weight*.

Ainsi, le code de notre graphique précédent s'écrit tout simplement[49] :

```
ggstats::ggsurvey(dp) +
  aes(x = region, fill = test) +
  geom_bar(position = "fill")
```

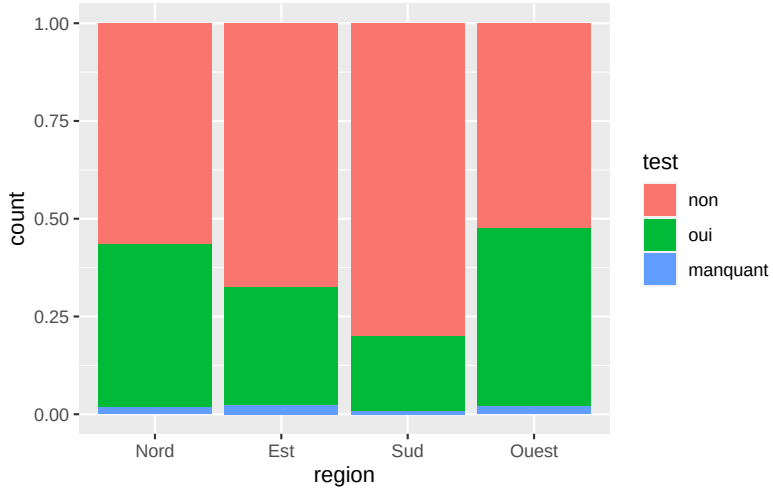


Figure 30.3: Un graphique en barres pondéré

[49] Notez que les poids ont déjà été associés à la bonne esthétique et qu'il n'est donc pas nécessaire de le refaire dans l'appel à `aes()`.

31 Régression logistique binaire pondérée

Nous avons abordé la régression logistique binaire non pondérée dans un chapitre dédié, cf. Chapitre **??**. Elle se réalise classiquement avec la fonction `glm()` en spécifiant `family = binomial`.

Lorsque l'on utilise des données d'enquêtes, l'approche est similaire sauf que l'on aura recours à la fonction `survey::svyglm()` qui sait gérer des objets `{survey}` : non seulement la pondération sera prise en compte, mais le calcul des intervalles de confiance et des p-valeurs sera ajusté en fonction du plan d'échantillonnage.

31.1 Données des exemples

Nous allons reprendre les même données issues de l'enquête *Histoire de vie 2003*, mais en tenant compte cette fois-ci des poids de pondération fourni dans la variable *poids*.

```
library(tidyverse)
library(labelled)
data(hdv2003, package = "questionr")
d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
```

```
        labels = c("18-24 ans", "25-44 ans",
                   "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
    ) |>
    fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    relig = "Rapport à la religion",
    heures.tv = "Heures de télévision / jour",
    poids = "Pondération de l'enquête"
  )
```

Il ne nous reste qu'à définir notre objet `{survey}` en spécifiant la pondération fournie avec l'enquête. La documentation ne mentionne ni strates ni grappes.

```
library(srvyr)
library(survey)
dp <- d |>
  as_survey_design(weights = poids)
```

31.2 Calcul de la régression logistique binaire

La syntaxe de `survey::svyglm()` est similaire à celle de `glm()` sauf qu'elle a un argument `design` au lieu de `data`.

La plupart du temps, les poids de pondération ne sont pas des nombres entiers, mais des nombres décimaux. Dès lors, on ne peut plus utiliser la famille de modèles binomiale (qui repose sur des nombres entiers de succès et d'échecs)[50]. On aura plutôt recours à la famille quasi-binomiale, que l'on spécifie avec `family = quasibinomial` et qui constitue une extension de la famille binomiale pouvant gérer des poids non entiers.

[50] Si l'on indique `family = binomial`, vous obtiendrez avec une version récente de **R** un message d'avertissement du type `Avis : nombre de succès non entier dans un glm binomial !`. Avec une version plus ancienne de **R**, vous devriez même avoir un message d'erreur.

```
mod <- svyglm(
  sport ~ sexe + groupe_ages + etudes + relig + heures.tv,
  family = quasibinomial,
  design = dp
)
```

Simple, non ?

31.3 Sélection de modèle

Comme précédemment, il est possible de procéder à une sélection de modèle pas à pas, par minimisation de l'AIC, avec `step()`.

```
mod2 <- step(mod)
```

```
Start:  AIC=2309.89
sport ~ sexe + groupe_ages + etudes + relig + heures.tv

              Df Deviance    AIC
- relig        5   2266.3 2302.2
<none>             2263.9 2309.9
- heures.tv    1   2276.2 2320.2
- sexe         1   2276.4 2320.4
- groupe_ages  3   2313.9 2353.8
- etudes       4   2383.5 2421.2

Step:  AIC=2296.28
sport ~ sexe + groupe_ages + etudes + heures.tv

              Df Deviance    AIC
```

```
<none>              2266.3 2296.3
- heures.tv     1   2278.4 2306.4
- sexe          1   2279.0 2307.0
- groupe_ages   3   2318.3 2342.1
- etudes        4   2387.2 2408.8
```

> **Sélection pas à pas et valeurs manquantes**
>
> Nous avons abordé dans le chapitre sur la régression logistique binaire la problématique des valeurs manquantes lors d'une sélection pas à pas descendante par minimisation de l'AIC (cf. encadré de la Section **??**). La même approche peut être appliquée avec des données pondérées. Cependant, la fonction `step_with_na()` que nous avons présenté n'est pas compatible avec les modèles `survey::svyglm()` puisqu'ils prennent en entrée un argument `design` et non `data`.

31.4 Affichage des résultats

Nous pouvons tout à fait utiliser `gtsumarry::tbl_regression()` avec ce type de modèles. De même, on peut utiliser `gtsummary::add_global_p()` pour calculer les p-valeurs globales des variables ou encore `gtsummary::add_vif()` pour vérifier la multicolinéarité (cf. Chapitre **??**).

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",", big.mark = " ")
```

```
mod2 |>
  tbl_regression(exponentiate = TRUE) |>
  add_global_p(keep = TRUE) |>
  add_vif() |>
  bold_labels()
```

```
Warning in printHypothesis(L, rhs, names(b)): one or more coefficients in the hypothesis inclu
     arithmetic operators in their names;
```

```
  the printed representation of the hypothesis will be omitted

Warning in printHypothesis(L, rhs, names(b)): one or more coefficients in the hypothesis inclu
     arithmetic operators in their names;
  the printed representation of the hypothesis will be omitted

Table printed with `knitr::kable()`, not {gt}. Learn why at
https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Table 31.1: Facteurs associés à la pratique d'un sport (régression logistique pondérée)

Caractéristique	OR	95% IC	p-valeur	GVIF	Adjusted GVIF
Sexe			0,005	1,0	1,0
Femme	—	—			
Homme	1,44	1,12 – 1,87	0,005		
Groupe d'âges			<0,001	2,1	1,1
18-24 ans	—	—			
25-44 ans	0,85	0,48 – 1,51	0,6		
45-64 ans	0,40	0,22 – 0,73	0,003		
65 ans et plus	0,37	0,19 – 0,72	0,004		
Niveau d'études			<0,001	2,2	1,1
Primaire	—	—			
Secondaire	2,66	1,62 – 4,38	<0,001		
Technique / Professionnel	3,09	1,90 – 5,00	<0,001		
Supérieur	6,54	3,99 – 10,7	<0,001		
Non documenté	10,3	4,60 – 23,0	<0,001		

Caractéristique	OR	95% IC	p-valeur	GVIF	Adjusted GVIF
Heures de télévision / jour	0,89	0,82 – 0,97	0,006	1,1	1,0

Pour un graphique des coefficients, nous pouvons utiliser `ggstats::ggcoef_model()`.

```
mod2 |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

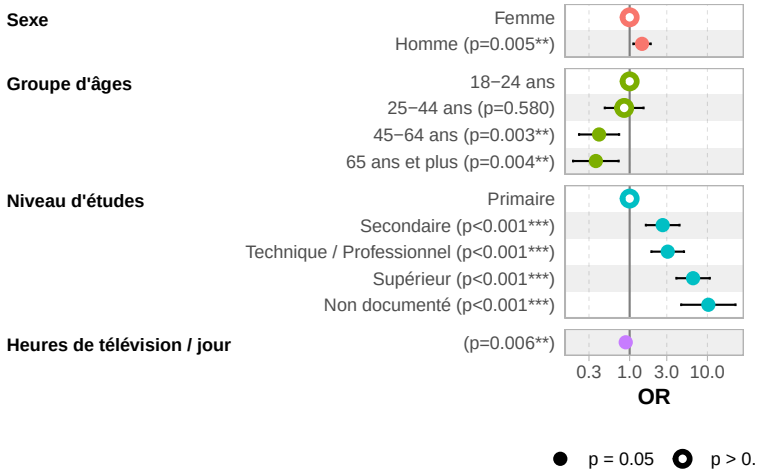


Figure 31.1: Facteurs associés à la pratique d'un sport (régression logistique pondérée)

31.5 Prédictions marginales

Pour visualiser les prédictions marginales moyennes du modèle (cf. Section **??**), nous pouvons utiliser `broom.helpers::plot_marginal_predictions()`.

```
mod2 |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
```

```
patchwork::wrap_plots() &
scale_y_continuous(
  limits = c(0, .8),
  labels = scales::label_percent()
)
```

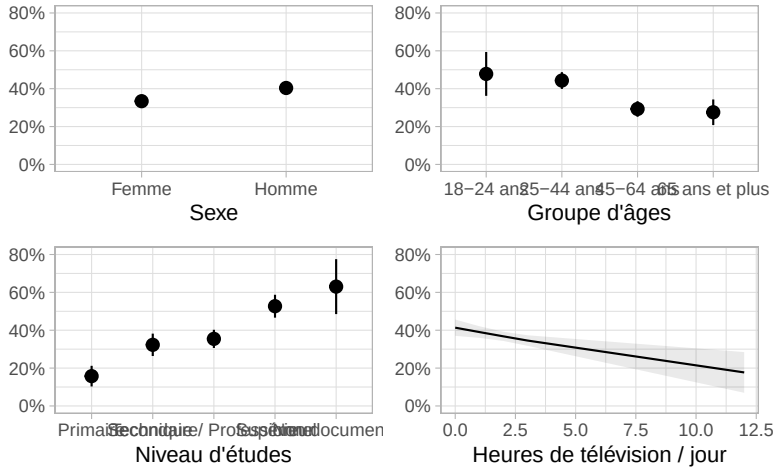


Figure 31.2: Prédictions marginales moyennes du modèle pondéré

partie V

Manipulation avancée

32 Fusion de tables

Il est fréquent d'avoir à gérer des données réparties dans plusieurs tables de données, notamment lorsque l'on a une enquêtes réalisée à différents niveaux (par exemple, un questionnaire ménage et un questionnaire individu) ou des données longitudinales.

On peut distinguer deux types d'actions :

- l'ajout de variables (jointure entre tables)
- l'ajout d'observations (concaténation de tables)

32.1 Jointures avec dplyr

Le jeu de données `{nycflights13}` est un exemple de données réparties en plusieurs tables. Ici on en a trois : les informations sur les vols, celles sur les aéroports et celles sur les compagnies aériennes sont dans trois tables distinctes.

`{dplyr}` propose différentes fonctions permettant de travailler avec des données structurées de cette manière.

```
library(tidyverse)
library(nycflights13)
data(flights)
data(airports)
data(airlines)
```

32.1.1 Clés implicites

Lorsque les données sont réparties dans plusieurs tables différentes, il est essentiel de repérer les identifiants permettant de naviguer d'une table à l'autre. Dans notre exemple, on peut voir

que la table `flights` contient le code de la compagnie aérienne du vol dans la variable *carrier* :

```
flights |> labelled::look_for()
```

```
pos variable       label col_type missing values
1   year           -     int      0
2   month          -     int      0
3   day            -     int      0
4   dep_time       -     int      8255
5   sched_dep_time -     int      0
6   dep_delay      -     dbl      8255
7   arr_time       -     int      8713
8   sched_arr_time -     int      0
9   arr_delay      -     dbl      9430
10  carrier        -     chr      0
11  flight         -     int      0
12  tailnum        -     chr      2512
13  origin         -     chr      0
14  dest           -     chr      0
15  air_time       -     dbl      9430
16  distance       -     dbl      0
17  hour           -     dbl      0
18  minute         -     dbl      0
19  time_hour      -     dttm     0
```

Et que par ailleurs la table `airlines` contient une information supplémentaire relative à ces compagnies, à savoir le nom complet.

```
airlines |> labelled::look_for()
```

```
pos variable label col_type missing values
1   carrier  -     chr      0
2   name     -     chr      0
```

Il est donc naturel de vouloir associer les deux, en l'occurrence pour ajouter les noms complets des compagnies à la table `flights`. Dans ce cas on va faire une *jointure* : les lignes d'une

table seront associées à une autre en se basant non pas sur leur position, mais sur les valeurs d'une ou plusieurs colonnes. Ces colonnes sont appelées des *clés*.

Pour faire une jointure de ce type, on va utiliser la fonction `dplyr::left_join()` :

```
fusion <- flights |> left_join(airlines)
```

```
Joining with `by = join_by(carrier)`
```

Pour faciliter la lecture, on va afficher seulement certaines colonnes du résultat et les premières lignes de la table :

```
fusion |>
  select(month, day, carrier, name) |>
  head(10)
```

```
# A tibble: 10 x 4
   month   day carrier name
   <int> <int> <chr>   <chr>
 1     1     1 UA      United Air Lines Inc.
 2     1     1 UA      United Air Lines Inc.
 3     1     1 AA      American Airlines Inc.
 4     1     1 B6      JetBlue Airways
 5     1     1 DL      Delta Air Lines Inc.
 6     1     1 UA      United Air Lines Inc.
 7     1     1 B6      JetBlue Airways
 8     1     1 EV      ExpressJet Airlines Inc.
 9     1     1 B6      JetBlue Airways
10     1     1 AA      American Airlines Inc.
```

On voit que la table obtenue est bien la fusion des deux tables d'origine selon les valeurs des deux colonnes clés *carrier*. On est parti de la table `flights`, et pour chaque ligne on a ajouté les colonnes de `airlines` pour lesquelles la valeur de *carrier* est la même. On a donc bien une nouvelle colonne `name` dans notre table résultat, avec le nom complet de la compagnie aérienne.

> **Note**
>
> Nous sommes ici dans le cas le plus simple concernant les clés de jointure : les deux clés sont uniques et portent le même nom dans les deux tables. Par défaut, si on ne lui spécifie pas explicitement les clés, `{dplyr}` fusionne en utilisant l'ensemble des colonnes communes aux deux tables. On peut d'ailleurs voir dans cet exemple qu'un message a été affiché précisant que la jointure s'est faite sur la variable *carrier*.

32.1.2 Clés explicites

La table `airports`, elle, contient des informations supplémentaires sur les aéroports : nom complet, altitude, position géographique, etc. Chaque aéroport est identifié par un code contenu dans la colonne *faa*.

Si on regarde la table `flights`, on voit que le code d'identification des aéroports apparaît à deux endroits différents : pour l'aéroport de départ dans la colonne *origin*, et pour celui d'arrivée dans la colonne *dest*. On a donc deux clés de jointures possibles, et qui portent un nom différent de la clé de `airports`.

On va commencer par fusionner les données concernant l'aéroport de départ. Pour simplifier l'affichage des résultats, on va se contenter d'un sous-ensemble des deux tables :

```
flights_ex <- flights |> select(month, day, origin, dest)
airports_ex <- airports |> select(faa, alt, name)
```

Si on se contente d'un `dplyr::left_join()` comme à l'étape précédente, on obtient un message d'erreur car aucune colonne commune ne peut être identifiée comme clé de jointure :

```
flights_ex |> left_join(airports_ex)
```

```
Error in `left_join()`:
! `by` must be supplied when `x` and `y` have no common variables.
```

```
i Use `cross_join()` to perform a cross-join.
```

On doit donc spécifier explicitement les clés avec l'argument `by` de `dplyr::left_join()`. Ici la clé est nommée *origin* dans la première table, et *faa* dans la seconde. La syntaxe est donc la suivante :

```
flights_ex |>
  left_join(airports_ex, by = c("origin" = "faa")) |>
  head(10)
```

```
# A tibble: 10 x 6
   month   day origin dest    alt name
   <int> <int> <chr>  <chr> <dbl> <chr>
 1     1     1 EWR    IAH      18 Newark Liberty Intl
 2     1     1 LGA    IAH      22 La Guardia
 3     1     1 JFK    MIA      13 John F Kennedy Intl
 4     1     1 JFK    BQN      13 John F Kennedy Intl
 5     1     1 LGA    ATL      22 La Guardia
 6     1     1 EWR    ORD      18 Newark Liberty Intl
 7     1     1 EWR    FLL      18 Newark Liberty Intl
 8     1     1 LGA    IAD      22 La Guardia
 9     1     1 JFK    MCO      13 John F Kennedy Intl
10     1     1 LGA    ORD      22 La Guardia
```

On constate que les deux nouvelles colonnes *name* et *alt* contiennent bien les données correspondant à l'aéroport de départ.

On va stocker le résultat de cette jointure dans `flights_ex` :

```
flights_ex <- flights_ex |>
  left_join(airports_ex, by = c("origin" = "faa"))
```

Supposons qu'on souhaite maintenant fusionner à nouveau les informations de la table `airports`, mais cette fois pour les aéroports d'arrivée de notre nouvelle table `flights_ex`. Les deux clés sont donc désormais *dest* dans la première table, et *faa* dans la deuxième. La syntaxe est donc la suivante :

```r
flights_ex |>
  left_join(airports_ex, by=c("dest" = "faa")) |>
  head(10)
```

```
# A tibble: 10 x 8
   month   day origin dest  alt.x name.x              alt.y name.y
   <int> <int> <chr>  <chr> <dbl> <chr>               <dbl> <chr>
 1     1     1 EWR    IAH      18 Newark Liberty Intl    97 George Bush Interco~
 2     1     1 LGA    IAH      22 La Guardia             97 George Bush Interco~
 3     1     1 JFK    MIA      13 John F Kennedy Intl     8 Miami Intl
 4     1     1 JFK    BQN      13 John F Kennedy Intl    NA <NA>
 5     1     1 LGA    ATL      22 La Guardia           1026 Hartsfield Jackson ~
 6     1     1 EWR    ORD      18 Newark Liberty Intl   668 Chicago Ohare Intl
 7     1     1 EWR    FLL      18 Newark Liberty Intl     9 Fort Lauderdale Hol~
 8     1     1 LGA    IAD      22 La Guardia            313 Washington Dulles I~
 9     1     1 JFK    MCO      13 John F Kennedy Intl    96 Orlando Intl
10     1     1 LGA    ORD      22 La Guardia            668 Chicago Ohare Intl
```

Cela fonctionne, les informations de l'aéroport d'arrivée ont bien été ajoutées, mais on constate que les colonnes ont été renommées. En effet, ici les deux tables fusionnées contenaient toutes les deux des colonnes *name* et *alt*. Comme on ne peut pas avoir deux colonnes avec le même nom dans un tableau, `{dplyr}` a renommé les colonnes de la première table en *name.x* et *alt.x*, et celles de la deuxième en *name.y* et *alt.y*.

C'est pratique, mais pas forcément très parlant. On pourrait renommer manuellement les colonnes pour avoir des intitulés plus explicites avec `dplyr::rename()`, mais on peut aussi utiliser l'argument `suffix` de `dplyr::left_join()`, qui permet d'indiquer les suffixes à ajouter aux colonnes. Ainsi, on peut faire :

```r
flights_ex |>
  left_join(
    airports_ex,
    by = c("dest" = "faa"),
    suffix = c("_depart", "_arrivee")
  ) |>
  head(10)
```

```
# A tibble: 10 x 8
   month   day origin dest  alt_depart name_depart      alt_arrivee name_arrivee
   <int> <int> <chr>  <chr>      <dbl> <chr>                  <dbl> <chr>
 1     1     1 EWR    IAH           18 Newark Liberty ~          97 George Bush~
 2     1     1 LGA    IAH           22 La Guardia                97 George Bush~
 3     1     1 JFK    MIA           13 John F Kennedy ~           8 Miami Intl
 4     1     1 JFK    BQN           13 John F Kennedy ~          NA <NA>
 5     1     1 LGA    ATL           22 La Guardia              1026 Hartsfield ~
 6     1     1 EWR    ORD           18 Newark Liberty ~         668 Chicago Oha~
 7     1     1 EWR    FLL           18 Newark Liberty ~           9 Fort Lauder~
 8     1     1 LGA    IAD           22 La Guardia               313 Washington ~
 9     1     1 JFK    MCO           13 John F Kennedy ~          96 Orlando Intl
10     1     1 LGA    ORD           22 La Guardia               668 Chicago Oha~
```

On obtient ainsi directement des noms de colonnes nettement plus clairs.

32.1.3 Types de jointures

Jusqu'à présent nous avons utilisé la fonction `dplyr::left_join()`, mais il existe plusieurs types de jointures.

Partons de deux tables d'exemple, `personnes` et `voitures` :

```
personnes <- tibble(
  nom = c("Sylvie", "Sylvie", "Monique", "Gunter", "Rayan", "Rayan"),
  voiture = c("Twingo", "Ferrari", "Scenic", "Lada", "Twingo", "Clio")
)
personnes
```

```
# A tibble: 6 x 2
  nom     voiture
  <chr>   <chr>
1 Sylvie  Twingo
2 Sylvie  Ferrari
3 Monique Scenic
4 Gunter  Lada
5 Rayan   Twingo
6 Rayan   Clio
```

```r
voitures <- tibble(
  voiture = c("Twingo", "Ferrari", "Clio", "Lada", "208"),
  vitesse = c("140", "280", "160", "85", "160")
)
voitures
```

```
# A tibble: 5 x 2
  voiture vitesse
  <chr>   <chr>
1 Twingo  140
2 Ferrari 280
3 Clio    160
4 Lada    85
5 208     160
```

32.1.3.1 `left_join()`

Si on fait un `dplyr::left_join()` de `voitures` sur `personnes` :

```r
personnes |> left_join(voitures, by = "voiture")
```

```
# A tibble: 6 x 3
  nom     voiture vitesse
  <chr>   <chr>   <chr>
1 Sylvie  Twingo  140
2 Sylvie  Ferrari 280
3 Monique Scenic  <NA>
4 Gunter  Lada    85
5 Rayan   Twingo  140
6 Rayan   Clio    160
```

On voit que chaque ligne de `personnes` est bien présente, et qu'on lui a ajouté une ligne de `voitures` correspondante si elle existe. Dans le cas du *Scenic*, il n'y a avait pas de ligne dans `voitures`, donc *vitesse* a été peuplée avec la valeur manquante `NA`. Dans le cas de la *208*, présente dans `voitures` mais pas dans `personnes`, la ligne n'apparaît pas.

La clé de fusion étant unique dans la table de droite, le nombre de lignes de la table de gauche est donc bien préservée.

```
personnes |> nrow()
```

```
[1] 6
```

```
personnes |> left_join(voitures, by = "voiture") |> nrow()
```

```
[1] 6
```

Si on fait un `dplyr::left_join()` cette fois de `personnes` sur `voitures`, c'est l'inverse :

```
voitures |> left_join(personnes, by = "voiture")
```

```
# A tibble: 6 x 3
  voiture vitesse nom
  <chr>   <chr>   <chr>
1 Twingo  140     Sylvie
2 Twingo  140     Rayan
3 Ferrari 280     Sylvie
4 Clio    160     Rayan
5 Lada    85      Gunter
6 208     160     <NA>
```

La ligne *208* est bien là avec la variable *nom* remplie avec une valeur manquante `NA`. Par contre *Monique* est absente.

> **!** Important
>
> On remarquera que la ligne *Twingo*, présente deux fois dans `personnes`, a été dupliquée pour être associée aux deux lignes de données de `Sylvie` et `Rayan`. Autrement dit, si la clé de fusion n'est pas unique dans la table de droite, certaines de lignes de la table de gauche seront dupliquées.

En résumé, quand on fait un `left_join(x, y)`, toutes les lignes de `x` sont présentes, et dupliquées si nécessaire quand elles apparaissent plusieurs fois dans `y`. Les lignes de `y` non présentes dans `x` disparaissent. Les lignes de `x` non présentes dans `y` se voient attribuer des valeurs manquantes `NA` pour les nouvelles colonnes.

Intuitivement, on pourrait considérer que `left_join(x, y)` signifie ramener l'information de la table `y` sur la table `x`.

En général, `dplyr::left_join()` sera le type de jointures le plus fréquemment utilisé.

32.1.3.2 `right_join()`

La jointure `dplyr::right_join()` est l'exacte symétrique de `dplyr::left_join()`, c'est-à dire que `x |> right_join(y)` est équivalent[51] à `y |> left_join(x)` :

[51] À l'exception de l'ordre des variables dans le tableau final.

```
personnes |> right_join(voitures, by = "voiture")
```

```
# A tibble: 6 x 3
  nom    voiture vitesse
  <chr>  <chr>   <chr>
1 Sylvie Twingo  140
2 Sylvie Ferrari 280
3 Gunter Lada    85
4 Rayan  Twingo  140
5 Rayan  Clio    160
6 <NA>   208     160
```

```
voitures |> left_join(personnes, by = "voiture")
```

```
# A tibble: 6 x 3
  voiture vitesse nom
  <chr>   <chr>   <chr>
1 Twingo  140     Sylvie
```

```
2 Twingo  140     Rayan
3 Ferrari 280     Sylvie
4 Clio    160     Rayan
5 Lada    85      Gunter
6 208     160     <NA>
```

32.1.3.3 `inner_join()`

Dans le cas de `dplyr::inner_join()`, seules les lignes présentes à la fois dans `x` et `y` sont présentes (et si nécessaire dupliquées) dans la table résultat :

```
personnes |> inner_join(voitures, by = "voiture")
```

```
# A tibble: 5 x 3
  nom    voiture vitesse
  <chr>  <chr>   <chr>
1 Sylvie Twingo  140
2 Sylvie Ferrari 280
3 Gunter Lada    85
4 Rayan  Twingo  140
5 Rayan  Clio    160
```

Ici la ligne *208* est absente, ainsi que la ligne *Monique*, qui dans le cas d'un `dplyr::left_join()` avait été conservée et s'était vue attribuer `NA` à *vitesse*.

32.1.3.4 `full_join()`

Dans le cas de `dplyr::full_join()`, toutes les lignes de `x` et toutes les lignes de `y` sont conservées (avec des `NA` ajoutés si nécessaire) même si elles sont absentes de l'autre table :

```
personnes |> full_join(voitures, by = "voiture")
```

```
# A tibble: 7 x 3
  nom     voiture vitesse
```

```
  <chr>   <chr>   <chr>
1 Sylvie  Twingo  140
2 Sylvie  Ferrari 280
3 Monique Scenic  <NA>
4 Gunter  Lada    85
5 Rayan   Twingo  140
6 Rayan   Clio    160
7 <NA>    208     160
```

32.1.3.5 `semi_join()` et `anti_join()`

`dplyr::semi_join()` et `dplyr::anti_join()` sont des jointures *filtrantes*, c'est-à-dire qu'elles sélectionnent les lignes de `x` sans ajouter les colonnes de `y`.

Ainsi, `dplyr::semi_join()` ne conservera que les lignes de `x` pour lesquelles une ligne de `y` existe également, et supprimera les autres. Dans notre exemple, la ligne `Monique` est donc supprimée :

```
personnes |> semi_join(voitures, by = "voiture")
```

```
# A tibble: 5 x 2
  nom    voiture
  <chr>  <chr>
1 Sylvie Twingo
2 Sylvie Ferrari
3 Gunter Lada
4 Rayan  Twingo
5 Rayan  Clio
```

Un `dplyr::anti_join()` fait l'inverse, il ne conserve que les lignes de `x` absentes de `y`. Dans notre exemple, on ne garde donc que la ligne *Monique* :

```
personnes |> anti_join(voitures, by = "voiture")
```

```
# A tibble: 1 x 2
  nom     voiture
```

```
  <chr>   <chr>
1 Monique Scenic
```

32.2 Jointures avec `merge()`

La fonction `merge()` est la fonction de **R base** pour fusionner des tables entre elles.

Par défaut, elle réalise un *inner join*, c'est-à-dire qu'elle ne garde que les observations dont la clé est retrouvée dans les deux tableaux fusionnés

```
merge(personnes, voitures, by = "voiture")
```

```
  voiture    nom vitesse
1    Clio  Rayan     160
2 Ferrari Sylvie     280
3    Lada Gunter      85
4  Twingo Sylvie     140
5  Twingo  Rayan     140
```

Les paramètres `all.x` et `all.y` permettent de réaliser fusions à gauche, à droite ou complète. L'équivalent de `dplyr::left_join()` sera obtenu avec `all.x = TRUE`, celui de `dplyr::right_join()` avec `all.y = TRUE` et celui de `dplyr::full_join()` avec `all.x = TRUE, all.y = TRUE`.

```
merge(personnes, voitures, by = "voiture", all.x = TRUE)
```

```
  voiture     nom vitesse
1    Clio   Rayan     160
2 Ferrari  Sylvie     280
3    Lada  Gunter      85
4  Scenic Monique    <NA>
5  Twingo  Sylvie     140
6  Twingo   Rayan     140
```

```
personnes |> left_join(voitures)
```

```
Joining with `by = join_by(voiture)`
```

```
# A tibble: 6 x 3
  nom     voiture vitesse
  <chr>   <chr>   <chr>
1 Sylvie  Twingo  140
2 Sylvie  Ferrari 280
3 Monique Scenic  <NA>
4 Gunter  Lada    85
5 Rayan   Twingo  140
6 Rayan   Clio    160
```

32.3 Ajouter des observations avec `bind_rows()`

La fonction `base::rbind()`, fournie nativement avec **R** pour ajouter des observations à un tableau, doit être évitée car elle générera des résultats non pertinents si les tableaux que l'on concatènent n'ont pas exactement les mêmes colonnes dans le même ordre.

La fonction `dplyr::bind_rows()` de `{dplyr}` permet d'ajouter des lignes à une table à partir d'une ou plusieurs autres tables.

L'exemple suivant (certes très artificiel) montre l'utilisation de `dplyr::bind_rows()`. On commence par créer trois tableaux `t1`, `t2` et `t3` :

```
t1 <- airports |>
  select(faa, name, lat, lon) |>
  slice(1:2)
t1
```

```
# A tibble: 2 x 4
  faa   name                              lat   lon
```

```
  <chr> <chr>                          <dbl> <dbl>
1 04G   Lansdowne Airport               41.1 -80.6
2 06A   Moton Field Municipal Airport   32.5 -85.7
```

```
t2 <- airports |>
  select(name, faa, lon, lat) |>
  slice(5:6)

t2
```

```
# A tibble: 2 x 4
  name                           faa     lon   lat
  <chr>                          <chr> <dbl> <dbl>
1 Jekyll Island Airport          09J   -81.4  31.1
2 Elizabethton Municipal Airport 0A9   -82.2  36.4
```

```
t3 <- airports |>
  select(faa, name) |>
  slice(100:101)
t3
```

```
# A tibble: 2 x 2
  faa   name
  <chr> <chr>
1 ADW   Andrews Afb
2 AET   Allakaket Airport
```

On concatène ensuite les trois tables avec `dplyr::bind_rows()` :

```
bind_rows(t1, t2, t3)
```

```
# A tibble: 6 x 4
  faa   name                             lat   lon
  <chr> <chr>                          <dbl> <dbl>
1 04G   Lansdowne Airport               41.1 -80.6
2 06A   Moton Field Municipal Airport   32.5 -85.7
3 09J   Jekyll Island Airport           31.1 -81.4
```

```
4 0A9   Elizabethton Municipal Airport  36.4 -82.2
5 ADW   Andrews Afb                     NA    NA
6 AET   Allakaket Airport               NA    NA
```

On remarquera que si des colonnes sont manquantes pour certaines tables, comme les colonnes *lat* et *lon* de `t3`, des valeurs manquantes `NA` sont automatiquement insérées.

De plus, peu importe l'ordre des variables entre les différentes tables, `dplyr::bind_rows()` les réassociera en considérant que deux colonnes ayant le même nom dans deux tableaux correspondent à la même variable.

Il peut être utile, quand on concatène des lignes, de garder une trace du tableau d'origine de chacune des lignes dans le tableau final. C'est possible grâce à l'argument `.id` de `dplyr::bind_rows()`. On passe à cet argument le nom d'une colonne qui contiendra l'indicateur d'origine des lignes :

```
bind_rows(t1, t2, t3, .id = "source")
```

```
# A tibble: 6 x 5
  source faa   name                           lat   lon
  <chr>  <chr> <chr>                        <dbl> <dbl>
1 1      04G   Lansdowne Airport             41.1 -80.6
2 1      06A   Moton Field Municipal Airport  32.5 -85.7
3 2      09J   Jekyll Island Airport         31.1 -81.4
4 2      0A9   Elizabethton Municipal Airport 36.4 -82.2
5 3      ADW   Andrews Afb                    NA    NA
6 3      AET   Allakaket Airport              NA    NA
```

Par défaut la colonne `.id` ne contient qu'un nombre, différent pour chaque tableau. On peut lui spécifier des valeurs plus explicites en "nommant" les tables dans `dplyr::bind_rows()` de la manière suivante :

```
bind_rows(table1 = t1, table2 = t2, table3 = t3, .id = "source")
```

```
# A tibble: 6 x 5
  source faa   name                           lat   lon
```

```
  <chr>  <chr> <chr>                           <dbl> <dbl>
1 table1 04G   Lansdowne Airport                41.1 -80.6
2 table1 06A   Moton Field Municipal Airport    32.5 -85.7
3 table2 09J   Jekyll Island Airport            31.1 -81.4
4 table2 0A9   Elizabethton Municipal Airport   36.4 -82.2
5 table3 ADW   Andrews Afb                      NA    NA
6 table3 AET   Allakaket Airport                NA    NA
```

Une alternative à `dplyr::bind_rows()` est la fonction `plyr::rbind.fill()` de l'extension `{plyr}` qui fonctionne de manière similaire.

33 Dates avec `lubridate`

Dans cette section, nous aborderons la gestion des dates des heures dans **R**. Si ce type de données peut paraître simple à première vue, la situation se complique dès lors que l'on souhaite effectuer des calculs entre dates. En effet, le nombre de jours varie d'un mois à un autre, voir d'une année à l'autre si l'on tient compte des années bissextiles. Quand on manipule des heures, il faut aussi pouvoir prendre en compte les différents fuseaux horaires ainsi que les changements liés à l'heure d'été.

Heureusement, le package `{lubridate}` permet de résoudre la plupart des problèmes posés par la manipulation de date. Il est chargé par défaut avec la commande `library(tidyverse)`. Nous allons également charger en mémoire le package `{nycflights13}` qui nous fournira les données utilisées dans nos exemples.

```
library(tidyverse)
library(nycflights13)
```

33.1 Création de dates / de dates-heures

Il existe trois types de variables pour représenter des dates et des heures :

- une **date**, de la classe `Date` et représentée dans un tibble avec `<date>`
- une **heure**, de la classe `hms` et représentée dans un tibble avec `<time>`
- une **date-heure**, de la classe `POSIXct` et représentée dans un tibble avec `<dttm>`

Les classes `Date` et `POSIXct` sont gérées nativement par **R** tandis que la classe `hms` est fournies par le package homonyme `{hms}`. Cette dernière classe est d'un usage plus spécifique. Dans cette section, nous allons nous concentrer sur les dates et les dates-heures.

Il est toujours préférable d'utiliser la classe la plus simple. Si vous gérez uniquement des dates, privilégiez la classe `Date`. La classe `POSIXct`, plus complexe, permet d'ajouter une heure associée à un fuseau horaire.

Pour obtenir la date ou la date-heure courante, vous pouvez appeler `today()` ou `now()` :

```
today()
```

```
[1] "2023-07-05"
```

```
now()
```

```
[1] "2023-07-05 16:03:28 CEST"
```

33.1.1 lors de l'import d'un fichier CSV

Si le fichier CSV contient des dates ou des dates-heures au format ISO8601, `readr::read_csv()` saura les reconnaître automatiquement :

```
csv <- "
  date,datetime
  2022-01-02,2022-01-02 05:12
"
read_csv(csv)
```

```
# A tibble: 1 x 2
  date       datetime
  <date>     <dttm>
1 2022-01-02 2022-01-02 05:12:00
```

Astuce

Le format **ISO8601** est un standard international pour l'écriture de dates[52] sous la forme `AAAA-MM-JJ` afin d'éviter la confusion entre les habitudes de différents pays, par exemple `JJ/MM/AAAA` en France ou `MM/JJ/AA` dans les pays anglosaxons.

Pour les autres formats, non standards, il sera nécessaire d'utiliser `col_types` avec `col_date()` pour spécifier comme lire et interpréter les chaînes de caractères. `{readr}` comprend la spécification **POSIX** qui permet de décrire une format de date[53]. Il s'agit de codes commençant par le symbole `%` et indiquant un composant d'une date. Par exemple, `%Y-%m-%d` correspond au format ISO8601, par exemple `2023-10-03` pour le 3 octobre 2023. Le tableau Table **??** liste les principales options.

[52] https://xkcd.com/1179/

Table 33.1: Les formats de dates compris par readr

Type	Code	Signification	Exemple
Année	`%Y`	année sur 4 chiffres	2021
	`%y`	année sur 2 chiffres	21
Mois	`%m`	numéro du mois	2
	`%b`	nom abrégé	Feb
	`%B`	nom complet	February
Jour	`%d`	jour sur 2 chiffres	02
	`%e`	jour sur 1 ou 2 chiffres	2
Heure	`%H`	heure sur 24 heures	13
	`%I`	heure sur 12 heures	1
	`%p`	AM ou PM	pm
Minute	`%M`	minutes	35
Seconde	`%S`	secondes	45
	`%OS`	secondes avec une composante décimale	45.35
Fuseau horaire	`%Z`	nom du fuseau	America/Chicago
	`%z`	décalage du fuseau par rapport au temps universel UTC	+0800

Type	Code	Signification	Exemple
Autre	%.	sauter un caractère (autre qu'un chiffre)	:
	%*	sauter un nombre quelconque de caractères (autres qu'un chiffre)	

Et voici un exemple de code induisant une lecture différente d'une date ambiguë.

```
csv <- "
  date
  01/02/15
"

read_csv(csv, col_types = cols(date = col_date("%m/%d/%y")))
```

```
# A tibble: 1 x 1
  date
  <date>
1 2015-01-02
```

```
read_csv(csv, col_types = cols(date = col_date("%d/%m/%y")))
```

```
# A tibble: 1 x 1
  date
  <date>
1 2015-02-01
```

```
read_csv(csv, col_types = cols(date = col_date("%y/%m/%d")))
```

```
# A tibble: 1 x 1
  date
  <date>
1 2001-02-15
```

Quel que soit le format original, les dates importées seront toujours affichées par **R** au format ISO.

Astuce

Si vous utilisez `%b` ou `%B`, il est essentiel de spécifier la langue utilisée avec le paramètre `local` de `col_date()`. Pour voir l'ensemble des langues couvertes, vous pouvez appelez `readr::date_names_langs()` et pour voir les chaînes de langues correspondantes `readr::date_names_lang()`. Si vos données n'utilise pas des noms standards, vous pouvez créer votre propre jeu de correspondance avec `readr::date_names()`.

```
date_names_langs()
```

```
  [1] "af"  "agq" "ak"  "am"  "ar"  "as"  "asa" "az"  "bas" "be"  "bem" "bez"
 [13] "bg"  "bm"  "bn"  "bo"  "br"  "brx" "bs"  "ca"  "cgg" "chr" "cs"  "cy"
 [25] "da"  "dav" "de"  "dje" "dsb" "dua" "dyo" "dz"  "ebu" "ee"  "el"  "en"
 [37] "eo"  "es"  "et"  "eu"  "ewo" "fa"  "ff"  "fi"  "fil" "fo"  "fr"  "fur"
 [49] "fy"  "ga"  "gd"  "gl"  "gsw" "gu"  "guz" "gv"  "ha"  "haw" "he"  "hi"
 [61] "hr"  "hsb" "hu"  "hy"  "id"  "ig"  "ii"  "is"  "it"  "ja"  "jgo" "jmc"
 [73] "ka"  "kab" "kam" "kde" "kea" "khq" "ki"  "kk"  "kkj" "kl"  "kln" "km"
 [85] "kn"  "ko"  "kok" "ks"  "ksb" "ksf" "ksh" "kw"  "ky"  "lag" "lb"  "lg"
 [97] "lkt" "ln"  "lo"  "lt"  "lu"  "luo" "luy" "lv"  "mas" "mer" "mfe" "mg"
[109] "mgh" "mgo" "mk"  "ml"  "mn"  "mr"  "ms"  "mt"  "mua" "my"  "naq" "nb"
[121] "nd"  "ne"  "nl"  "nmg" "nn"  "nnh" "nus" "nyn" "om"  "or"  "os"  "pa"
[133] "pl"  "ps"  "pt"  "qu"  "rm"  "rn"  "ro"  "rof" "ru"  "rw"  "rwk" "sah"
[145] "saq" "sbp" "se"  "seh" "ses" "sg"  "shi" "si"  "sk"  "sl"  "smn" "sn"
[157] "so"  "sq"  "sr"  "sv"  "sw"  "ta"  "te"  "teo" "th"  "ti"  "to"  "tr"
[169] "twq" "tzm" "ug"  "uk"  "ur"  "uz"  "vai" "vi"  "vun" "wae" "xog" "yav"
[181] "yi"  "yo"  "zgh" "zh"  "zu"
```

```
date_names_lang("fr")
```

```
<date_names>
Days:   dimanche (dim.), lundi (lun.), mardi (mar.), mercredi (mer.), jeudi
        (jeu.), vendredi (ven.), samedi (sam.)
Months: janvier (janv.), février (févr.), mars (mars), avril (avr.), mai (mai),
        juin (juin), juillet (juil.), août (août), septembre (sept.),
```

```
        octobre (oct.), novembre (nov.), décembre (déc.)
AM/PM:  AM/PM
```

```
date_names_lang("en")
```

```
<date_names>
Days:   Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed), Thursday
        (Thu), Friday (Fri), Saturday (Sat)
Months: January (Jan), February (Feb), March (Mar), April (Apr), May (May),
        June (Jun), July (Jul), August (Aug), September (Sep), October
        (Oct), November (Nov), December (Dec)
AM/PM:  AM/PM
```

```
csv <- "date
3 de febrero de 2001"

read_csv(
  csv,
  col_types = cols(date = col_date("%d de %B de %Y")),
  locale = locale("es")
)
```

```
# A tibble: 1 x 1
  date
  <date>
1 2001-02-03
```

33.1.2 à partir d'une chaîne de caractères

Le langage de spécification de la date et du temps est puissant, mais il nécessite une analyse minutieuse du format de la date. Une autre approche consiste à utiliser les fonctions de `{lubridate}` qui tentent de déterminer automatiquement le format une fois que vous avez spécifié l'ordre des composants. Pour les utiliser, identifiez l'ordre dans lequel l'année, le mois et le jour apparaissent dans vos dates, puis placez "y", "m" et "d" dans le même ordre. Cela vous donne le nom de la fonction `{lubridate}` qui analysera votre date. Par exemple :

```
ymd("2017-01-31")
```

```
[1] "2017-01-31"
```

```
mdy("January 31st, 2017")
```

```
[1] "2017-01-31"
```

```
dmy("31-Jan-2017")
```

```
[1] "2017-01-31"
```

`ymd()` et ses soeurs créeent des dates. Pour des dates-heures, ajoutez un tiret bas et les lettres “h”, “m” et/ou “s” :

```
ymd_hms("2017-01-31 20:11:59")
```

```
[1] "2017-01-31 20:11:59 UTC"
```

```
mdy_hm("01/31/2017 08:01")
```

```
[1] "2017-01-31 08:01:00 UTC"
```

33.1.3 à partir des composants

Parfois, les différentes composantes d’une date (jour, mois, année…) sont stockées dans des colonnes séparées. C’est le cas par exemple dans la table `flights` issue du package `{nycflights13}`.

```
flights |>
  select(year, month, day, hour, minute) |>
  head()
```

```
# A tibble: 6 x 5
   year month   day  hour minute
  <int> <int> <int> <dbl>  <dbl>
1  2013     1     1     5     15
2  2013     1     1     5     29
3  2013     1     1     5     40
4  2013     1     1     5     45
5  2013     1     1     6      0
6  2013     1     1     5     58
```

Pour créer une date ou une date-heure à partir de colonnes séparées, il suffit d'utiliser `lubridate::make_date()` pour les dates et `lubridate::make_datetime()` pour les dates-heures :

```
flights |>
  select(year, month, day, hour, minute) |>
  mutate(
    departure = make_datetime(year, month, day, hour, minute),
    departure_date = make_date(year, month, day)
  ) |>
  head()
```

```
# A tibble: 6 x 7
   year month   day  hour minute departure           departure_date
  <int> <int> <int> <dbl>  <dbl> <dttm>              <date>
1  2013     1     1     5     15 2013-01-01 05:15:00 2013-01-01
2  2013     1     1     5     29 2013-01-01 05:29:00 2013-01-01
3  2013     1     1     5     40 2013-01-01 05:40:00 2013-01-01
4  2013     1     1     5     45 2013-01-01 05:45:00 2013-01-01
5  2013     1     1     6      0 2013-01-01 06:00:00 2013-01-01
6  2013     1     1     5     58 2013-01-01 05:58:00 2013-01-01
```

33.1.4 conversion

Pour convertir une date en date-heure, ou l'inverse, utilisez `lubridate::as_datetime()` ou `lubridate::as_date()` :

```
as_datetime(today())
```

```
[1] "2023-07-05 UTC"
```

```
as_date(now())
```

```
[1] "2023-07-05"
```

33.2 Manipuler les composants d'une date/date-heure

33.2.1 Extraire un composant

Pour extraire un composant d'une date ou d'une date-heure, il suffit d'utiliser l'une des fonctions suivantes : `year()` (année), `month()` (mois), `mday()` (jour du mois), `yday()` (jours de l'année), `wday()` (jour de la semaine), `hour()` (heure), `minute()` (minute), ou `second()` (seconde).

```
datetime <- ymd_hms("2026-07-08 12:34:56")

year(datetime)
```

```
[1] 2026
```

```
month(datetime)
```

```
[1] 7
```

```
mday(datetime)
```

```
[1] 8
```

```
yday(datetime)
```

```
[1] 189
```

```r
wday(datetime)
```

```
[1] 4
```

Pour `month()` et `wday()`, vous pouvez indiquer `label = TRUE` pour récupérer le nom abrégé du mois ou du jours de la semaine. Ajoutez `abbr = FALSE` pour le nom complet.

```r
month(datetime, label = TRUE)
```

```
[1] juil
12 Levels: janv < févr < mars < avr < mai < juin < juil < août < ... < déc
```

```r
wday(datetime, label = TRUE, abbr = FALSE)
```

```
[1] mercredi
7 Levels: dimanche < lundi < mardi < mercredi < jeudi < ... < samedi
```

Les noms sont affichés dans la langue de votre ordinateur. On peut utiliser le paramètre `locale` pour changer la langue. Attention : le code peut varier selon votre système d'exploitation. Vous pouvez essayer déjà de simplement indiquer le code à 2 lettres de la langue visée, par exemple `"de"` pour l'allemand. Si cela ne fonctionne pas, essayez `"de_DE"` (allemand utilisé en Allemagne), `"de_DE.UTF-8"` (format utilisé par MacOS et plusieurs distributions Linux), la variante `"de_DE.utf8"` (utilisée par certaines distributions Linux) ou bien encore `"German.UTF-8"` (utilisé par Windows).

```r
month(datetime, label = TRUE, abbr = FALSE, locale = "en")
```

```
[1] July
12 Levels: January < February < March < April < May < June < ... < December
```

```r
month(datetime, label = TRUE, abbr = FALSE, locale = "es_ES.utf8")
```

```
[1] julio
12 Levels: enero < febrero < marzo < abril < mayo < junio < ... < diciembre
```

```
month(datetime, label = TRUE, abbr = FALSE, locale = "German.UTF-8")
```

```
[1] Juli
12 Levels: Januar < Februar < März < April < Mai < Juni < Juli < ... < Dezember
```

33.2.2 Arrondis

Les fonctions `lubridate::round_date()`, `lubridate::floor_date()` et `lubridate::ceiling_date()` permettent d'arrondir une date à l'unité la plus proche, inférieure ou supérieure. On devra préciser avec `unit` l'unité utilisée pour arrondir. Les valeurs acceptées sont `"second"`, `"minute"`, `"hour"`, `"day"`, `"week"`, `"month"`, `"bimonth"` (bimestre, i.e. période de 2 mois), `"quarter"` (trimestre), `season` (saison), `halfyear` (semestre) et `year`, ou un multiple de ces valeurs.

```
d <- ymd("2022-05-14")
floor_date(d, unit = "week")
```

```
[1] "2022-05-08"
```

```
floor_date(d, unit = "month")
```

```
[1] "2022-05-01"
```

```
floor_date(d, unit = "3 months")
```

```
[1] "2022-04-01"
```

```
floor_date(d, unit = "year")
```

```
[1] "2022-01-01"
```

33.2.3 Modifier un composant

Les mêmes fonctions peuvent être utilisées pour modifier un composant particulier d'une date-heure.

```
datetime <- ymd_hms("2026-07-08 12:34:56")

year(datetime) <- 2030
datetime
```

```
[1] "2030-07-08 12:34:56 UTC"
```

```
month(datetime) <- 01
datetime
```

```
[1] "2030-01-08 12:34:56 UTC"
```

```
hour(datetime) <- hour(datetime) + 1
datetime
```

```
[1] "2030-01-08 13:34:56 UTC"
```

Une alternative, plutôt que de modifier une date-heure, consiste à créer une copie modifiée avec `lubridate::update()`. Cela permet également de modifier plusieurs éléments à la fois :

```
update(datetime, year = 2030, month = 2, mday = 2, hour = 2)
```

```
[1] "2030-02-02 02:34:56 UTC"
```

Si les valeurs sont trop importantes (trop de jours par exemple), la fonction ajoutera les unités en trop pour générer une date valide :

```
update(ymd("2023-02-01"), mday = 30)
```

```
[1] "2023-03-02"
```

```
update(ymd("2023-02-01"), hour = 400)
```

```
[1] "2023-02-17 16:00:00 UTC"
```

33.3 Durées, périodes, intervalles & Arithmétique

Il existe plusieurs manières de représenter les intervalles de temps entre deux dates :

- les **durées** (*Duration*), qui représentent un nombre exact de secondes ;
- les **périodes** (*Periods*), qui représentent une durée sous la forme d'unités de temps telles que des semaines ou des mois ;
- les **intervalles** (*Intervals*), qui sont définis par une date-heure de début et une date-heure de fin.

33.3.1 Durées (Duration)

Avec **R**, lorsque l'on soustrait deux dates, on obtient un objet de la classe `difftime`.

```
diff <- ymd("2021-06-30") - ymd("1979-10-14")
diff
```

```
Time difference of 15235 days
```

Un objet `difftime` enregistre une durée sous la forme d'un nombre de secondes, de minutes, d'heures, de jours ou de semaines. Du fait de variations de l'unité d'un objet à l'autre, ils ne sont pas toujours faciles à manipuler. Pour lever toute ambiguïté, on préférera les objets de la class `Duration` qui stockent les durées sous la forme d'un nombre de secondes. La conversion peut se faire avec `lubridate::as.duration()`.

```
as.duration(diff)
```

```
[1] "1316304000s (~41.71 years)"
```

Il est possible de créer facilement des durées avec une série de fonctions dédiées dont le nom commence par `"d"`

```
dseconds(15)
```

```
[1] "15s"
```

```
dminutes(10)
```

```
[1] "600s (~10 minutes)"
```

```
dhours(c(12, 24))
```

```
[1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
ddays(0:5)
```

```
[1] "0s"                "86400s (~1 days)"  "172800s (~2 days)"
[4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
```

```
dweeks(3)
```

```
[1] "1814400s (~3 weeks)"
```

```
dyears(1)
```

```
[1] "31557600s (~1 years)"
```

Les durées sont toujours exprimées en secondes. Des unités plus grandes sont créées en convertissant les minutes, les heures, les jours, les semaines et les années en secondes : 60 secondes dans une minute, 60 minutes dans une heure, 24 heures dans un jour et 7 jours dans une semaine. Les unités de temps plus grandes posent davantage de problèmes. Une année utilise le nombre « moyen » de jours dans une année, c'est-à-dire 365,25. Il n'existe aucun moyen de convertir un mois en durée, car les variations sont trop importantes.

Il est possible d'additionner et de multiplier les durées :

```
2 * dyears(1)
```

```
[1] "63115200s (~2 years)"
```

```
dyears(1) + dweeks(12) + dhours(15)
```

```
[1] "38869200s (~1.23 years)"
```

On peut ajouter ou soustraire des durées à une date.

```
demain <- today() + ddays(1)
il_y_a_un_an <- today() - dyears(1)
```

Cependant, comme les durées représentent un nombre exact de secondes, vous pouvez parfois obtenir un résultat inattendu :

```
one_am <- ymd_hms("2026-03-08 01:00:00", tz = "America/New_York")

one_am
```

```
[1] "2026-03-08 01:00:00 EST"
```

```
one_am + ddays(1)
```

```
[1] "2026-03-09 02:00:00 EDT"
```

Pourquoi lorsqu'on ajoute un jour, on passe de 1 heure du matin à 2 heures du matin ? Si vous regardez attentivement la date, vous remarquerez que le fuseau a changé. Le 8 mars 2026 n'aura que 23 heures aux États-Unis en raison du passage à l'heure d'été. En ajoutant une durée de 1 jour, nous avons ajouté exactement 24 heures. Le même type de phénomène peut s'observer en ajoutant une durée d'une année, car on considère que cela représente en moyenne 365.25 jours.

33.3.2 Périodes (Period)

Pour résoudre ce problème, `{lubridate}` a introduit les périodes (de classe `Period`) qui représentent une durée en nombre de secondes, minutes, heures, jours, mois et années, sans préciser la durée exacte de chaque mois ou année. Cela permet de faire des calculs plus intuitifs :

```
one_am
```

```
[1] "2026-03-08 01:00:00 EST"
```

```
one_am + days(1)
```

```
[1] "2026-03-09 01:00:00 EDT"
```

Comme pour les durées, on peut créer facilement des périodes avec des fonctions dédiées (notez ici le pluriel des noms de fonction, alors que celles permettant d'extraire un composant d'une date étaient au singulier) :

```
hours(c(12, 24))
```

```
[1] "12H 0M 0S" "24H 0M 0S"
```

```
days(7)
```

```
[1] "7d 0H 0M 0S"
```

```
months(1:6)
```

```
[1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
[5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
```

On peut ajouter, soustraire et multiplier les périodes entre elles.

```
10 * (months(6) + days(1))
```

```
[1] "60m 10d 0H 0M 0S"
```

```
days(50) + hours(25) + minutes(2)
```

```
[1] "50d 25H 2M 0S"
```

Bien sûr, on peut ajouter ou soustraire une période à une date :

```
# Exemple avec une année bissextile
ymd("2024-01-01") + dyears(1)
```

```
[1] "2024-12-31 06:00:00 UTC"
```

```
ymd("2024-01-01") + years(1)
```

```
[1] "2025-01-01"
```

```
# Exemple avec un passage à l'heure d'été
one_am + ddays(1)
```

```
[1] "2026-03-09 02:00:00 EDT"
```

```
one_am + days(1)
```

```
[1] "2026-03-09 01:00:00 EDT"
```

Restent malgré tout quelques cas problématiques. Essayons d'ajouter 1 mois à la date du 31 janvier 2021.

```
ymd("2021-01-31") + months(1)
```

```
[1] NA
```

Ce calcul a jouté 1 au mois, sans toucher à l'année ni au jour, produisant la date du 31 février 2021 qui n'existe pas, produisant ainsi `NA`. Pour du calcul impliquant des dates et des périodes, il est préférable d'utiliser les opérateurs dédiés `%m+%` pour l'addition et `%m-%` pour la soustraction.

```
ymd("2021-01-31") %m+% months(1)
```

```
[1] "2021-02-28"
```

Lorsque le résultat produit une date inexistante, cela renvoie la dernière date correcte, ici le 28 février 2021, ce qui correspond bien à la fin du mois considéré.

33.3.3 Intervalles (Interval)

Quelle est la durée réelle d'une année ? En 2015, il s'agissait de 365 jours alors qu'en 2016 on en comptait 366. Quand on s'intéresse aux mois, la situation est encore plus compliquée car il y a une grande variation du nombre de jours d'un mois à l'autre.

Pour des calculs précis entre deux dates, les durées et les intervalles sont souvent insuffisants. On pourra alors avoir recours aux intervalles (de la classe `Interval`) qui sont définis avec une date de début et une date de fin.

On peut créer un intervalle avec la fonction `lubridate::interval()` :

```
interval(ymd("2022-05-13"), ymd("2022-08-15"))
```

```
[1] 2022-05-13 UTC--2022-08-15 UTC
```

On peut également utiliser l'opérateur `%--%` :

```
y2023 <- ymd("2023-01-01") %--% ymd("2024-01-01")
y2024 <- ymd("2024-01-01") %--% ymd("2025-01-01")

y2023
```

```
[1] 2023-01-01 UTC--2024-01-01 UTC
```

```
y2024
```

```
[1] 2024-01-01 UTC--2025-01-01 UTC
```

On peut tester si une date est située dans un intervalle donné avec l'opérateur `%within%`.

```
int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))
ymd("2001-05-03") %within% int
```

```
[1] TRUE
```

On peut même tester si un intervalle est situé à l'intérieur d'un intervalle :

```
int2 <- interval(ymd("2001-06-01"), ymd("2001-11-11"))
int2 %within% int
```

```
[1] TRUE
```

Cela n'est valable que si l'ensemble du premier intervalle est situé à l'intérieur du second intervalle.

```
int3 <- interval(ymd("2001-06-01"), ymd("2002-06-01"))
int3 %within% int
```

```
[1] FALSE
```

Pour tester si deux intervalles ont une partie en commun, on pourra utiliser `lubridate::int_overlaps()`. La fonction `intersect()` renvoie la partie partagée par les deux intervalles.

```
int_overlaps(int3, int)
```

```
[1] TRUE
```

```
intersect(int3, int)
```

```
[1] 2001-06-01 UTC--2002-01-01 UTC
```

`{lubridate}` fournie plusieurs fonctions, de la forme `int_*()`, pour manipuler les intervalles.

```
int
```

```
[1] 2001-01-01 UTC--2002-01-01 UTC
```

```
int_start(int)
```

```
[1] "2001-01-01 UTC"
```

```
int_end(int)
```

```
[1] "2002-01-01 UTC"
```

```
int_flip(int)
```

```
[1] 2002-01-01 UTC--2001-01-01 UTC
```

On peut calculer facilement la durée d'un intervalle avec la fonction `lubridate::time_length()` :

```
time_length(int) # en seconde par défaut
```

```
[1] 31536000
```

```
time_length(int, unit = "weeks")
```

```
[1] 52.14286
```

```
time_length(int, unit = "days")
```

```
[1] 365
```

La fonction `time_length()` permet notamment de calculer correctement un âge.

33.4 Calcul d'un âge

En tant que démographe, je suis toujours attentif au calcul des âges. Les démographes distinguent l'**âge exact**, exprimé en années avec une partie décimale, et qui correspond à la durée entre la date considérée et la date de naissance ; l'**âge révolu**, qui correspond à l'âge au dernier anniversaire et exprimé avec un nombre entier d'années (c'est l'âge que nous utilisons dans notre vie quotidienne) · ; et l'**âge atteint** ou **âge par différence de millésimes**, qui correspond à la différence entre l'année en cours et l'année de naissance (c'est l'âge que l'on aura cette année le jour de son anniversaire).

Pour calculer un âge exact en années, nous ne pouvons pendre la durée en jours entre les deux dates et diviser par 365 puisqu'il y a des années bissextiles. Une approche correcte est déjà de considérer l'âge au dernière anniversaire pour la partie entière, puis de calculer la partie décimale comme étant le ratio entre la durée depuis le dernière anniversaire et la durée entre le dernier et le prochain anniversaire. C'est exactement ce que fait `lubridate::time_length()`.

```
naiss <- ymd("1979-11-28")
evt <- ymd("2022-07-14")
age_exact <- time_length(naiss %--% evt, unit = "years")
age_exact
```

```
[1] 42.62466
```

Pour un âge révolu, il suffit de ne garder que la partie entière de l'âge exact avec `trunc()`.

```
age_revolu <- trunc(age_exact)
age_revolu
```

```
[1] 42
```

Enfin, pour un âge atteint ou un âge par différence de millésimes, nous extrairons les deux années avant d'en faire la soustraction.

```
age_atteint <- year(evt) - year(naiss)
age_atteint
```

```
[1] 43
```

> **Astuce**
>
> Le calcul d'un âge moyen s'effectue normalement à partir d'âges exacts. Il arrive fréquemment que l'on ne dispose

dans les données d'enquêtes que de l'âge révolu. Auquel cas, il faut bien penser à rajouter 0,5 au résultat obtenu. En effet, un âge révolu peut être vu comme une classe d'âges exacts : les individus ayant 20 ans révolus ont entre 20 et 21 ans exacts, soit en moyenne 20,5 ans !

33.5 Fuseaux horaires

Les fuseaux horaires sont un sujet extrêmement complexe en raison de leur interaction avec les entités géopolitiques. Heureusement, nous n'avons pas besoin d'entrer dans tous les détails, car ils ne sont pas tous importants pour l'analyse des données, mais il y a quelques défis que nous devrons relever.

Le premier défi est que les noms courants des fuseaux horaires ont tendance à être ambigus. Par exemple, si vous êtes américain, vous connaissez probablement l'EST (*Eastern Standard Time*). Cependant, l'Australie et le Canada ont également une heure normale de l'Est ! Pour éviter toute confusion, **R** utilise les fuseaux horaires standard internationaux de l'IANA. Ceux-ci utilisent un schéma de dénomination cohérent `{zone}/{lieu}`, généralement sous la forme **{continent}/{ville}** ou `{océan}/{ville}`. Parmi les exemples, citons `"America/New_York"`, **"Europe/Paris"** et **"Pacific/Auckland"**.

On peut se demander pourquoi le fuseau horaire utilise une ville, alors que l'on pense généralement que les fuseaux horaires sont associés à un pays ou à une région à l'intérieur d'un pays. La raison en est que la base de données de l'IANA doit enregistrer des dizaines d'années de règles relatives aux fuseaux horaires. Au fil des décennies, les pays changent de nom (ou se séparent) assez fréquemment, mais les noms de villes ont tendance à rester inchangés. Un autre problème réside dans le fait que le nom doit refléter non seulement le comportement actuel, mais aussi l'ensemble de l'histoire. Par exemple, il existe des fuseaux horaires pour `"America/New_York"` et `"America/Detroit"`. Cela vaut la peine de lire la base de données brute des fuseaux horaires (disponible à l'adresse https://www.iana.org/time-zones)

rien que pour lire certaines de ces histoires !

Vous pouvez découvrir ce que **R** pense être votre fuseau horaire actuel avec `Sys.timezone()` :

```
Sys.timezone()
```

```
[1] "Europe/Paris"
```

La liste complète des fuseaux horaires est disponible avec `OlsonNames()`:

```
length(OlsonNames())
```

```
[1] 596
```

```
head(OlsonNames())
```

```
[1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
[4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

Dans **R**, le fuseau horaire est un attribut de la date-heure qui ne contrôle que l'affichage. Par exemple, ces trois objets représentent le même instant dans le temps :

```
x1 <- ymd_hms("2024-06-01 12:00:00", tz = "America/New_York")
x1
```

```
[1] "2024-06-01 12:00:00 EDT"
```

```
x2 <- ymd_hms("2024-06-01 18:00:00", tz = "Europe/Copenhagen")
x2
```

```
[1] "2024-06-01 18:00:00 CEST"
```

```
x3 <- ymd_hms("2024-06-02 04:00:00", tz = "Pacific/Auckland")
x3
```

```
[1] "2024-06-02 04:00:00 NZST"
```

Sauf indication contraire, `{lubridate}` utilise toujours l'heure UTC. UTC (Temps universel coordonné, compromis entre l'anglais CUT *Coordinated universal time* et le français TUC *Temps universel coordonné*) est le fuseau horaire standard utilisé par la communauté scientifique et est à peu près équivalent à GMT (*Greenwich Mean Time*). Il n'y a pas d'heure d'été, ce qui en fait une représentation pratique pour les calculs. Les opérations qui combinent des dates-heure, comme `c()`, ne tiennent souvent pas compte du fuseau horaire. Dans ce cas, les dates-heure s'afficheront dans le fuseau horaire du premier élément :

```
x4 <- c(x1, x2, x3)
x4
```

```
[1] "2024-06-01 12:00:00 EDT" "2024-06-01 12:00:00 EDT"
[3] "2024-06-01 12:00:00 EDT"
```

Vous pouvez modifier le fuseau horaire de deux manières :

- Conserver le même instant dans le temps, mais modifier la façon dont il est affiché. Utilisez cette option lorsque l'instant est correct, mais que vous souhaitez un affichage plus naturel.

  ```
  x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
  x4a
  ```

  ```
  [1] "2024-06-02 02:30:00 +1030" "2024-06-02 02:30:00 +1030"
  [3] "2024-06-02 02:30:00 +1030"
  ```

  ```
  x4a - x4
  ```

```
Time differences in secs
[1] 0 0 0
```

- Modifier l'instant sous-jacent dans le temps. Utilisez cette option lorsqu'un instant a été étiqueté avec un fuseau horaire incorrect et que vous devez le corriger.

```
x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b
```

```
[1] "2024-06-01 12:00:00 +1030" "2024-06-01 12:00:00 +1030"
[3] "2024-06-01 12:00:00 +1030"
```

```
x4b - x4
```

```
Time differences in hours
[1] -14.5 -14.5 -14.5
```

33.6 Pour aller plus loin

- le chapitre Dates and times de l'ouvrage *R for Data Science* (2e édition)
- la documentation du package `{lubridate}` : https://lubridate.tidyverse.org/