

Introduction à R et au tidyverse

Julien Barnier

2019-09-19

Table des Matières

À propos de ce document	3
1 Présentation	5
1.1 À propos de R	5
1.2 À propos de RStudio	6
1.3 À propos du <i>tidyverse</i>	6
1.4 Structure du document	7
1.5 Prérequis	7
I Introduction à R	9
2 Prise en main	11
2.1 La console	11
2.2 Objets	14
2.3 Fonctions	20
2.4 Regrouper ses commandes dans des scripts	24
2.5 Installer et charger des extensions (<i>packages</i>)	25
2.6 Exercices	27
3 Premier travail avec des données	31
3.1 Jeu de données d'exemple	31
3.2 Tableau de données (<i>data frame</i>)	32
3.3 Analyse univariée	38
3.4 Exercices	50
4 Analyse bivariée	53
4.1 Croisement de deux variables qualitatives	53
4.2 Croisement d'une variable quantitative et d'une variable qualitative	59
4.3 Croisement de deux variables quantitatives	64
4.4 Exercices	72
5 Organiser ses scripts	73
5.1 Les projets dans RStudio	73
5.2 Créer des sections dans un script	75

5.3 Répartir son travail entre plusieurs scripts	76
5.4 Désactiver la sauvegarde de l'espace de travail	78
II Introduction au tidyverse	81
6 Le tidyverse	83
6.1 Extensions	83
6.2 Installation	83
6.3 tidy data	85
6.4 tibbles	85
7 Importer et exporter des données	89
7.1 Import de fichiers textes	89
7.2 Import depuis un fichier Excel	94
7.3 Import de fichiers SAS, SPSS et Stata	95
7.4 Import de fichiers dBase	96
7.5 Connexion à des bases de données	96
7.6 Export de données	100
8 Visualiser avec ggplot2	103
8.1 Préparation	103
8.2 Initialisation	104
8.3 Exemples de <code>geom</code>	108
8.4 Mappages	122
8.5 Représentation de plusieurs <code>geom</code>	132
8.6 Faceting	139
8.7 Scales	143
8.8 Thèmes	163
8.9 L'add-in <code>esquisse</code>	166
8.10 Ressources	169
8.11 Exercices	170
9 Recoder des variables	181
9.1 Rappel sur les variables et les vecteurs	181
9.2 Tests et comparaison	183
9.3 Recoder une variable qualitative	186
9.4 Combiner plusieurs variables	197
9.5 Découper une variable numérique en classes	200
9.6 Exercices	203
10 Manipuler les données avec dplyr	209
10.1 Préparation	209
10.2 Les verbes de <code>dplyr</code>	210
10.3 Enchaîner les opérations avec le <i>pipe</i>	221
10.4 Opérations groupées	223

10.5 Autres fonctions utiles	236
10.6 Tables multiples	242
10.7 Ressources	254
10.8 Exercices	254
11 Manipuler du texte avec <code>stringr</code>	269
11.1 Expressions régulières	270
11.2 Concaténer des chaînes	270
11.3 Convertir en majuscules / minuscules	272
11.4 Découper des chaînes	272
11.5 Extraire des sous-chaînes par position	274
11.6 Déetecter des motifs	274
11.7 Extraire des motifs	275
11.8 Remplacer des motifs	277
11.9 Modificateurs de motifs	277
11.10 Ressources	278
11.11 Exercices	278
12 Mettre en ordre avec <code>tidyverse</code>	281
12.1 Tidy data	281
12.2 Trois règles pour des données bien rangées	283
12.3 Les verbes de <code>tidyverse</code>	285
12.4 Ressources	296
13 Diffuser et publier avec <code>rmarkdown</code>	297
13.1 Créer un nouveau document	302
13.2 Éléments d'un document R Markdown	303
13.3 Personnaliser le document généré	308
13.4 Options des blocs de code R	310
13.5 Rendu des tableaux	314
13.6 Modèles de documents	317
13.7 Ressources	323
Appendix	323
A Ressources	325
A.1 Aide	325
A.2 Ouvrages, blogs, MOOCs...	329
A.3 Extensions	330

À propos de ce document

Ce document est une introduction à l'utilisation du logiciel libre de traitement de données et d'analyse statistique R. Il se veut le plus accessible possible, y compris pour ceux qui ne sont pas particulièrement familiers avec l'informatique. Il se

base à la fois sur les fonctionnalités de R “de base”, et sur une série d’extensions de plus en plus populaires regroupées sous l’appellation *tidyverse*.

Ce document *n'est pas* une introduction aux méthodes statistiques d'analyse de données.

Il est basé sur R version 3.6.1 (2019-07-05).

Ce document est régulièrement corrigé et mis à jour. La version de référence est disponible en ligne à l'adresse :

- <https://juba.github.io/tidyverse>

Le code source est disponible [sur GitHub](#).

Pour toute suggestion ou correction, il est possible de me contacter [par mail](#) ou [sur Twitter](#).

Remerciements

Ce document a bénéficié de la relecture et des suggestions de Karine Pietropaoli, Diane Rodet, Jimmy Raturat et Fabienne Marquant. La première partie, tirée d'un précédent document *Introduction à R*, a également profité des corrections et des enrichissements de Mayeul Kauffmann, Julien Biaudet, Frédérique Giraud, Joël Gombin, Milan Bouchet-Valat et Joseph Larmarange.

Ce document est généré par l'excellente extension [bookdown](#) de [Yihui Xie](#).

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](#).



Figure 1: Licence Creative Commons

Chapitre 1

Présentation

1.1 À propos de R

R est un langage orienté vers le traitement et l'analyse quantitative de données, dérivé du langage S. Il est développé depuis les années 90 par un groupe de volontaires de différents pays et par une large communauté d'utilisateurs. C'est un logiciel libre, publié sous licence GNU GPL.

L'utilisation de R présente plusieurs avantages :

- c'est un logiciel multiplateforme, qui fonctionne aussi bien sur des systèmes Linux, Mac OS X ou Windows.
- c'est un logiciel libre, développé par ses utilisateurs, diffusable et modifiable par tout un chacun.
- c'est un logiciel gratuit.
- c'est un logiciel très puissant, dont les fonctionnalités de base peuvent être étendues à l'aide d'extensions développées par la communauté. Il en existe plusieurs milliers.
- c'est un logiciel dont le développement est très actif et dont la communauté d'utilisateurs et l'usage ne cessent de grandir.
- c'est un logiciel avec d'excellentes capacités graphiques.

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

- le logiciel, la documentation de référence et les principales ressources sont en anglais. Il est toutefois parfaitement possible d'utiliser R sans spécialement maîtriser cette langue et il existe de plus en plus de ressources francophones.
- R n'est pas un logiciel au sens classique du terme, mais plutôt un langage de programmation. Il fonctionne à l'aide de scripts (des petits programmes) édités et exécutés au fur et à mesure de l'analyse.

- en tant que langage de programmation, R a la réputation d'être difficile d'accès, notamment pour ceux n'ayant jamais programmé auparavant.

Ce document ne demande aucun prérequis en informatique ou en programmation. Juste un peu de motivation pour l'apprentissage du langage et, si possible, des données intéressantes sur lesquelles appliquer les connaissances acquises.

L'aspect langage de programmation et la difficulté qui en découle peuvent sembler des inconvénients importants. Le fait de structurer ses analyses sous forme de scripts (suite d'instructions effectuant les différentes opérations d'une analyse) présente cependant de nombreux avantages :

- le script garde par ordre chronologique l'ensemble des étapes d'une analyse, de l'importation des données à leur analyse en passant par les manipulations et les recodages.
- on peut à tout moment revenir en arrière et modifier ce qui a été fait.
- il est très rapide de réexécuter une suite d'opérations complexes.
- on peut très facilement mettre à jour les résultats en cas de modification des données sources.
- le script garantit, sous certaines conditions, la reproductibilité des résultats obtenus.

1.2 À propos de RStudio

RStudio n'est pas à proprement parler une interface graphique pour R, il s'agit plutôt d'un *environnement de développement intégré*, qui propose des outils et facilite l'écriture de scripts et l'usage de R au quotidien. C'est une interface bien supérieure à celles fournies par défaut lorsqu'on installe R sous Windows ou sous Mac¹.

Pour paraphraser [Hadrien Commenges](#), il n'y a pas d'obligation à utiliser RStudio, mais il y a une obligation à ne pas utiliser les interfaces de R par défaut.

RStudio est également un logiciel libre et gratuit. Une version payante existe, mais elle ne propose pas de fonctionnalités indispensables.

1.3 À propos du *tidyverse*

Le *tidyverse* est un ensemble d'extensions pour R (code développé par la communauté permettant de rajouter des fonctionnalités à R) construites autour d'une philosophie commune et conçues pour fonctionner ensemble. Elles facilitent l'utilisation de R dans les domaines les plus courants : manipulation des données, recodages, production de graphiques, etc.

¹Sous Linux R n'est fourni que comme un outil en ligne de commande.

La deuxième partie de ce document est entièrement basée sur les extensions du *tidyverse*, qui est présenté plus en détail chapitre 6.

1.4 Structure du document

Ce document est composé de deux grandes parties :

- Une *Introduction à R*, qui présente les bases du langage R et de l'interface RStudio
- Une *Introduction au tidyverse* qui présente cet ensemble d'extensions pour la visualisation, la manipulation des données et l'export de résultats

Les personnes déjà familières avec R “de base” peuvent sauter toute la partie *Introduction à R* et passer directement à l'*Introduction au tidyverse*.

1.5 Prérequis

Le seul prérequis pour suivre ce document est d'avoir installé R et RStudio sur votre ordinateur. Il s'agit de deux logiciels libres, gratuits, téléchargeables en ligne et fonctionnant sous PC, Mac et Linux.

Pour installer R, il suffit de se rendre sur une des pages suivantes ² :

- [Installer R sous Windows](#)
- [Installer R sous Mac](#)

Pour installer RStudio, rendez-vous sur la page suivante et téléchargez la version adaptée à votre système :

- <https://www.rstudio.com/products/rstudio/download/#download>

²Sous Linux, utilisez votre gestionnaire de packages habituel.

Partie I

Introduction à R

Chapitre 2

Prise en main

Une fois R et RStudio installés sur votre machine, nous n'allons pas lancer R mais plutôt RStudio.

RStudio n'est pas à proprement parler une interface graphique qui permettrait d'utiliser R de manière "classique" via la souris, des menus et des boîtes de dialogue. Il s'agit plutôt de ce qu'on appelle un *Environnement de développement intégré* (IDE) qui facilite l'utilisation de R et le développement de scripts (voir section 1.2).

2.1 La console

2.1.1 L'invite de commandes

Au premier lancement de RStudio, l'écran principal est découpé en trois grandes zones :

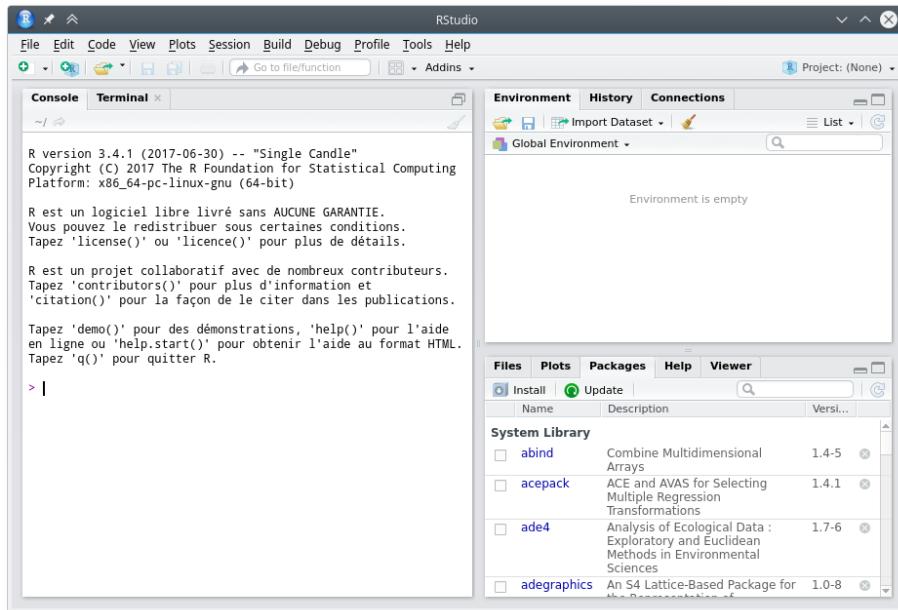


Figure 2.1: Interface de Rstudio

La zone de gauche se nomme *Console*. À son démarrage, RStudio a lancé une nouvelle session de R et c'est dans cette fenêtre que nous allons pouvoir interagir avec lui.

La *Console* doit normalement afficher un texte de bienvenue ressemblant à ceci :

```
R version 3.5.2 (2018-12-20) -- "Eggshell Igloo"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.
```

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

Nous pouvons tout de suite lui fournir une première commande, en saisissant le texte suivant puis en appuyant sur Entrée :

```
2 + 2
```

```
[1] 4
```

R nous répond immédiatement, et nous pouvons constater avec soulagement qu'il sait faire des additions à un chiffre¹. On peut donc continuer avec d'autres opérations :

```
5 - 7
```

```
[1] -2
```

```
4 * 12
```

```
[1] 48
```

```
-10 / 3
```

```
[1] -3.333333
```

```
5^2
```

```
[1] 25
```

Cette dernière opération utilise le symbole \wedge qui représente l'opération *puissance*. 5^2 signifie donc “5 au carré”, soit 25.

¹On peut ignorer pour le moment la présence du [1] en début de ligne.

2.1.2 Précisions concernant la saisie des commandes

Lorsqu'on saisit une commande, les espaces autour des opérateurs n'ont pas d'importance. Les trois commandes suivantes sont donc équivalentes, mais on privilégie en général la deuxième pour des raisons de lisibilité du code.

```
10+2
10 + 2
10      +      2
```

Quand vous êtes dans la console, vous pouvez utiliser les flèches vers le haut et vers le bas pour naviguer dans l'historique des commandes que vous avez tapées précédemment. Vous pouvez à tout moment modifier la commande affichée, et l'exécuter en appuyant sur **Entrée**.

Enfin, il peut arriver qu'on saisisse une commande de manière incomplète : oubli d'une parenthèse, faute de frappe, etc. Dans ce cas, R remplace l'invite de commande habituel par un signe + :

```
4 *
+
```

Cela signifie qu'il “attend la suite”. On peut alors soit compléter la commande sur cette nouvelle ligne et appuyer sur **Entrée**, soit, si on est perdu, tout annuler et revenir à l'invite de commandes normal en appuyant sur **Esc** ou **Échap**.

2.2 Objets

2.2.1 Objets simples

Faire des calculs c'est bien, mais il serait intéressant de pouvoir stocker un résultat quelque part pour pouvoir le réutiliser ultérieurement sans avoir à faire du copier/coller.

Pour conserver le résultat d'une opération, on peut le stocker dans un *objet* à l'aide de l'opérateur d'assignation `<-`. Cette “flèche” stocke ce qu'il y a à sa droite dans un objet dont le nom est indiqué à sa gauche.

Prenons tout de suite un exemple :

```
x <- 2
```

Cette commande peut se lire “prend la valeur 2 et mets la dans un objet qui s’appelle `x`”.

Si on exécute une commande comportant juste le nom d’un objet, R affiche son contenu :

```
x
```

```
[1] 2
```

On voit donc que notre objet `x` contient bien la valeur 2.

On peut évidemment réutiliser cet objet dans d’autres opérations. R le remplacera alors par sa valeur :

```
x + 4
```

```
[1] 6
```

On peut créer autant d’objets qu’on le souhaite.

```
x <- 2  
y <- 5  
resultat <- x + y  
resultat
```

```
[1] 7
```



Les noms d’objets peuvent contenir des lettres, des chiffres, les symboles `.` et `_`. Ils ne peuvent pas commencer par un chiffre. Attention, R fait la différence entre minuscules et majuscules dans les noms d’objets, ce qui signifie que `x` et `X` seront deux objets différents, tout comme `resultat` et `Resultat`.

De manière générale, il est préférable d’éviter les majuscules (pour les risques d’erreur) et les caractères accentués (pour des questions d’encodage) dans les noms d’objets.

De même, il faut essayer de trouver un équilibre entre clarté du nom (comprendre à quoi sert l’objet, ce qu’il contient) et sa longueur. Par exemple, on préférera comme nom d’objet `taille_conj1` à

```
taille_du_conjoint_numero_1 (trop long) ou à t1 (pas assez explicite).
```

Quand on assigne une nouvelle valeur à un objet déjà existant, la valeur précédente est perdue. Les objets n'ont pas de mémoire.

```
x <- 2
x <- 5
x
```

```
[1] 5
```

De la même manière, assigner un objet à un autre ne crée pas de “lien” entre les deux. Cela copie juste la valeur de l’objet de droite dans celui de gauche :

```
x <- 1
y <- 3
x <- y
x
```

```
[1] 3
```

```
## Si on modifie y, cela ne modifie pas x
y <- 4
x
```

```
[1] 3
```

On le verra, les objets peuvent contenir tout un tas d’informations. Jusqu’ici on n’a stocké que des nombres, mais ils peuvent aussi contenir des chaînes de caractères (du texte), qu’on délimite avec des guillemets simples ou doubles (‘ ou ”) :

```
chien <- "Chihuahua"
chien
```

```
[1] "Chihuahua"
```

2.2.2 Vecteurs

Imaginons maintenant qu'on a demandé la taille en centimètres de 5 personnes et qu'on souhaite calculer leur taille moyenne. On pourrait créer autant d'objets que de tailles et faire l'opération mathématique qui va bien :

```
taille1 <- 156
taille2 <- 164
taille3 <- 197
taille4 <- 147
taille5 <- 173
(taille1 + taille2 + taille3 + taille4 + taille5) / 5
```

```
[1] 167.4
```

Cette manière de faire n'est évidemment pas pratique du tout. On va plutôt stocker l'ensemble de nos tailles dans un seul objet, de type *vecteur*, avec la syntaxe suivante :

```
tailles <- c(156, 164, 197, 147, 173)
```

Si on affiche le contenu de cet objet, on voit qu'il contient bien l'ensemble des tailles saisies :

```
tailles
```

```
[1] 156 164 197 147 173
```

Un *vecteur* dans R est un objet qui peut contenir plusieurs informations du même type, potentiellement en très grand nombre.

L'avantage d'un vecteur est que lorsqu'on lui applique une opération, celle-ci s'applique à toutes les valeurs qu'il contient. Ainsi, si on veut la taille en mètres plutôt qu'en centimètres, on peut faire :

```
tailles_m <- tailles / 100  
tailles_m
```

```
[1] 1.56 1.64 1.97 1.47 1.73
```

Cela fonctionne pour toutes les opérations de base :

```
tailles + 10
```

```
[1] 166 174 207 157 183
```

```
tailles^2
```

```
[1] 24336 26896 38809 21609 29929
```

Imaginons maintenant qu'on a aussi demandé aux cinq mêmes personnes leur poids en kilos. On peut alors créer un deuxième vecteur :

```
poids <- c(45, 59, 110, 44, 88)
```

On peut alors effectuer des calculs utilisant nos deux vecteurs `tailles` et `poids`. On peut par exemple calculer l'indice de masse corporelle (IMC) de chacun de nos enquêtés en divisant leur poids en kilo par leur taille en mètre au carré :

```
imc <- poids / (tailles / 100) ^ 2  
imc
```

```
[1] 18.49112 21.93635 28.34394 20.36189 29.40292
```

Un vecteur peut contenir des nombres, mais il peut aussi contenir du texte. Imaginons qu'on a demandé aux 5 mêmes personnes leur niveau de diplôme : on peut regrouper l'information dans un vecteur de *chaînes de caractères*. Une chaîne de caractère contient du texte libre, délimité par des guillemets simples ou doubles :

```
diplome <- c("CAP", "Bac", "Bac+2", "CAP", "Bac+3")
diplome
```

```
[1] "CAP"   "Bac"   "Bac+2" "CAP"   "Bac+3"
```

L'opérateur `:`, lui, permet de générer rapidement un vecteur comprenant tous les nombres entre deux valeurs, opération assez courante sous R :

```
x <- 1:10
x
```

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

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

```
diplome[2]
```

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

Cette opération, qui utilise l'opérateur `[]`, permet donc la sélection d'éléments d'un vecteur.

Dernière remarque, si on affiche dans la console un vecteur avec beaucoup d'éléments, ceux-ci seront répartis sur plusieurs lignes. Par exemple, si on a un vecteur de 50 nombres on peut obtenir quelque chose comme :

```
[1] 294 425 339 914 114 896 716 648 915 587 181 926 489
[14] 848 583 182 662 888 417 133 146 322 400 698 506 944
[27] 237 324 333 443 487 658 793 288 897 588 697 439 697
[40] 914 694 126 969 744 927 337 439 226 704 635
```

On remarque que R ajoute systématiquement un nombre entre crochets au début de chaque ligne : il s'agit en fait de la position du premier élément de la ligne dans le vecteur. Ainsi, le 848 de la deuxième ligne est le 14e élément du vecteur, le 914 de la dernière ligne est le 40e, etc.

Ceci explique le [1] qu'on obtient quand on affiche un simple nombre² :

```
[1] 4
```

2.3 Fonctions

2.3.1 Principe

Nous savons désormais effectuer des opérations arithmétiques de base sur des nombres et des vecteurs, et stocker des valeurs dans des objets pour pouvoir les réutiliser plus tard.

Pour aller plus loin, nous devons aborder les *fonctions* qui sont, avec les objets, un deuxième concept de base de R. On utilise des fonctions pour effectuer des calculs, obtenir des résultats et accomplir des actions.

Formellement, une fonction a un *nom*, elle prend en entrée entre parenthèses un ou plusieurs *arguments* (ou *paramètres*), et retourne un *résultat*.

Prenons tout de suite un exemple. Si on veut connaître le nombre d'éléments du vecteur **tailles** que nous avons construit précédemment, on peut utiliser la fonction **length**, de cette manière :

```
length(tailles)
```

```
[1] 5
```

Ici, **length** est le nom de la fonction, on l'appelle en lui passant un argument entre parenthèses (en l'occurrence notre vecteur **tailles**), et elle nous renvoie un résultat, à savoir le nombre d'éléments du vecteur passé en paramètre.

Autre exemple, les fonctions **min** et **max** retournent respectivement les valeurs minimales et maximales d'un vecteur de nombres :

```
min(tailles)
```

```
[1] 147
```

²Et permet de constater que pour R, un nombre est un vecteur à un seul élément.

```
max(tailles)
```

```
[1] 197
```

La fonction `mean` calcule et retourne la moyenne d'un vecteur de nombres :

```
mean(tailles)
```

```
[1] 167.4
```

La fonction `sum` retourne la somme de tous les éléments du vecteur :

```
sum(tailles)
```

```
[1] 837
```

Jusqu'à présent on n'a vu que des fonctions qui calculent et retournent un unique nombre. Mais une fonction peut renvoyer d'autres types de résultats. Par exemple, la fonction `range` (étendue) renvoie un vecteur de deux nombres, le minimum et le maximum :

```
range(tailles)
```

```
[1] 147 197
```

Ou encore, la fonction `unique`, qui supprime toutes les valeurs en double dans un vecteur, qu'il s'agisse de nombres ou de chaînes de caractères :

```
diplome <- c("CAP", "Bac", "Bac+2", "CAP", "Bac+3")
unique(diplome)
```

```
[1] "CAP"    "Bac"    "Bac+2"  "Bac+3"
```

2.3.2 Arguments

Une fonction peut prendre plusieurs arguments, dans ce cas on les indique toujours entre parenthèses, séparés par des virgules.

On a déjà rencontré un exemple de fonction acceptant plusieurs arguments : la fonction `c`, qui combine l'ensemble de ses arguments en un vecteur³ :

```
tailles <- c(156, 164, 197, 181, 173)
```

Ici, `c` est appelée en lui passant cinq arguments, les cinq tailles séparées par des virgules, et elle renvoie un vecteur numérique regroupant ces cinq valeurs.

Supposons maintenant que dans notre vecteur `tailles` nous avons une valeur manquante (une personne a refusé de répondre, ou notre mètre mesureur était en panne). On symbolise celle-ci dans R avec le code interne `NA` :

```
tailles <- c(156, 164, 197, NA, 173)
tailles
```

```
[1] 156 164 197 NA 173
```



`NA` est l'abréviation de *Not available*, non disponible. Cette valeur particulière peut être utilisée pour indiquer une valeur manquante, qu'il s'agisse d'un nombre, d'une chaîne de caractères, etc.

Si je calcule maintenant la taille moyenne à l'aide de la fonction `mean`, j'obtiens :

```
mean(tailles)
```

```
[1] NA
```

En effet, R considère par défaut qu'il ne peut pas calculer la moyenne si une des valeurs n'est pas disponible. Il considère alors que cette moyenne est elle-même "non disponible" et renvoie donc comme résultat `NA`.

On peut cependant indiquer à `mean` d'effectuer le calcul en ignorant les valeurs manquantes. Ceci se fait en ajoutant un argument supplémentaire, nommé `na.rm` (abréviation de *NA remove*, "enlever les NA"), et de lui attribuer la valeur `TRUE` (code interne de R signifiant *vrai*) :

³`c` est l'abréviation de *combine*, son nom est très court car on l'utilise très souvent

```
mean(tailles, na.rm = TRUE)
```

```
[1] 172.5
```

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

Si on ne dit rien à la fonction `mean`, cet argument a une valeur par défaut, en l'occurrence `FALSE` (faux), qui fait qu'il ne supprime pas les valeurs manquantes. Les deux commandes suivantes sont donc rigoureusement équivalentes :

```
mean(tailles)
```

```
[1] NA
```

```
mean(tailles, na.rm = FALSE)
```

```
[1] NA
```



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

2.3.3 Aide sur une fonction

Il est fréquent de ne pas savoir (ou d'avoir oublié) quels sont les arguments d'une fonction, ou comment ils se nomment. On peut à tout moment faire appel à l'aide intégrée à R en passant le nom de la fonction (entre guillemets) à la fonction `help` :

```
help("mean")
```

On peut aussi utiliser le raccourci `?mean`.

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

Dans RStudio, les pages d'aide en ligne s'ouvriront par défaut dans la zone en bas à droite, sous l'onglet *Help*. Un clic sur l'icône en forme de maison vous affichera la page d'accueil de l'aide.

2.4 Regrouper ses commandes dans des scripts

Jusqu'ici on a utilisé R de manière “interactive”, en saisissant des commandes directement dans la console. Ça n'est cependant pas la manière dont on va utiliser R au quotidien, pour une raison simple : lorsque R redémarre, tout ce qui a été effectué dans la console est perdu.

Plutôt que de saisir nos commandes dans la console, on va donc les regrouper dans des scripts (de simples fichiers texte), qui vont garder une trace de toutes les opérations effectuées, et ce sont ces scripts, sauvegardés régulièrement, qui seront le “coeur” de notre travail. C'est en rouvrant les scripts et en réexécutant les commandes qu'ils contiennent qu'on pourra “reproduire” les données, leur traitement, les analyses et leurs résultats.

Pour créer un script, il suffit de sélectionner le menu *File*, puis *New file* et *R script*. Une quatrième zone apparaît alors en haut à gauche de l'interface de RStudio. On peut enregistrer notre script à tout moment dans un fichier avec l'extension **.R**, en cliquant sur l'icône de disquette ou en choisissant *File* puis *Save*.

Un script est un fichier texte brut, qui s'édite de la manière habituelle. À la différence de la console, quand on appuie sur **Entrée**, cela n'exécute pas la commande en cours mais insère un saut de ligne (comme on pouvait s'y attendre).

Pour exécuter une commande saisie dans un script, il suffit de positionner le curseur sur la ligne de la commande en question, et de cliquer sur le bouton *Run* dans la barre d'outils juste au-dessus de la zone d'édition du script. On peut aussi utiliser le raccourci clavier **Ctrl + Entrée** (**Cmd + Entrée** sous Mac). On peut enfin sélectionner plusieurs lignes avec la souris ou le clavier et cliquer sur *Run* (ou utiliser le raccourci clavier), et l'ensemble des lignes est exécuté d'un coup.

Au final, un script pourra ressembler à quelque chose comme ça :

```
tailles <- c(156, 164, 197, 147, 173)
poids <- c(45, 59, 110, 44, 88)

mean(tailles)
mean(poids)
```

```
imc <- poids / (tailles / 100) ^ 2
min(imc)
max(imc)
```

2.4.1 Commentaires

Les commentaires sont un élément très important d'un script. Il s'agit de texte libre, ignoré par R, et qui permet de décrire les étapes du script, sa logique, les raisons pour lesquelles on a procédé de telle ou telle manière... Il est primordial de documenter ses scripts à l'aide de commentaires, car il est très facile de ne plus se retrouver dans un programme qu'on a produit soi-même, même après une courte interruption.

Pour ajouter un commentaire, il suffit de le faire précéder d'un ou plusieurs symboles `#`. En effet, dès que R rencontre ce caractère, il ignore tout ce qui se trouve derrière, jusqu'à la fin de la ligne.

On peut donc documenter le script précédent :

```
# Saisie des tailles et poids des enquêtés
tailles <- c(156, 164, 197, 147, 173)
poids <- c(45, 59, 110, 44, 88)

# Calcul des tailles et poids moyens
mean(tailles)
mean(poids)

# Calcul de l'IMC (poids en kilo divisé par les tailles en mètre au carré)
imc <- poids / (tailles / 100) ^ 2
# Valeurs extrêmes de l'IMC
min(imc)
max(imc)
```

2.5 Installer et charger des extensions (*packages*)

R étant un logiciel libre, il bénéficie d'un développement communautaire riche et dynamique. L'installation de base de R permet de faire énormément de choses, mais le langage dispose en plus d'un système d'extensions permettant d'ajouter facilement de nouvelles fonctionnalités. La plupart des extensions sont

développées et maintenues par la communauté des utilisateurs de R, et diffusées via un réseau de serveurs nommé CRAN (*Comprehensive R Archive Network*).

Pour installer une extension, si on dispose d'une connexion Internet, on peut utiliser le bouton *Install* de l'onglet *Packages* de RStudio.

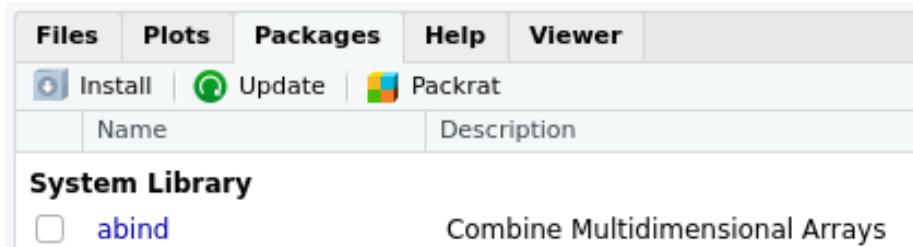


Figure 2.2: Installer une extension

Il suffit alors d'indiquer le nom de l'extension dans le champ *Package* et de cliquer sur *Install*.

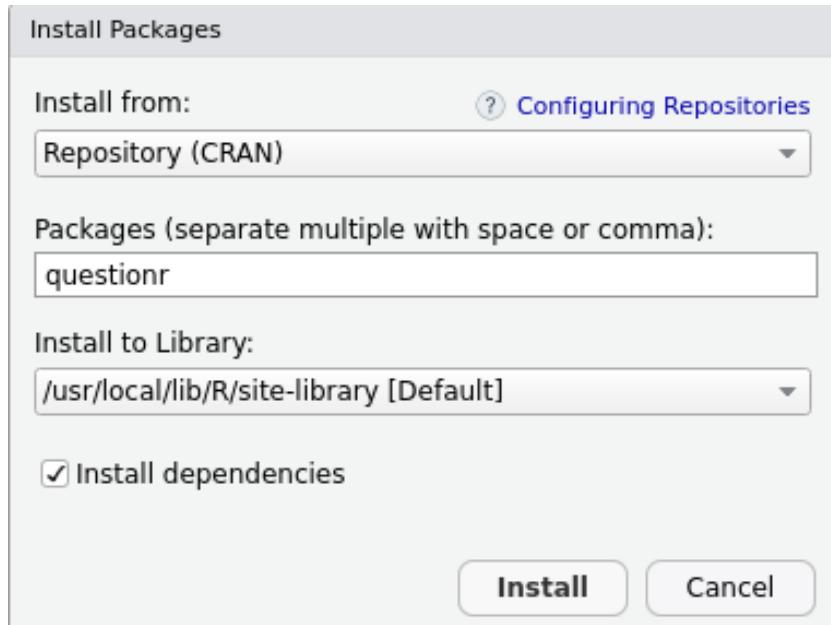


Figure 2.3: Installation d'une extension

On peut aussi installer des extensions en utilisant la fonction `install.packages()`

directement dans la console. Par exemple, pour installer le *package questionr* on peut exécuter la commande :

```
install.packages("questionr")
```

Installer une extension via l'une des deux méthodes précédentes va télécharger l'ensemble des fichiers nécessaires depuis l'une des machines du CRAN, puis installer tout ça sur le disque dur de votre ordinateur. Vous n'avez besoin de le faire qu'une fois, comme vous le faites pour installer un programme sur votre Mac ou PC.

Une fois l'extension installée, il faut la “charger” avant de pouvoir utiliser les fonctions qu'elle propose. Ceci se fait avec la fonction **library**. Par exemple, pour pouvoir utiliser les fonctions de **questionr**, vous devrez exécuter la commande suivante :

```
library(questionr)
```

Ainsi, bien souvent, on regroupe en début de script toute une série d'appels à **library** qui permettent de charger tous les packages utilisés dans le script. Quelque chose comme :

```
library(readxl)
library(ggplot2)
library(questionr)
```

Si vous essayez d'exécuter une fonction d'une extension et que vous obtenez le message d'erreur **impossible de trouver la fonction**, c'est certainement parce que vous n'avez pas exécuté la commande **library** correspondante.

2.6 Exercices

Exercice 1

Construire le vecteur **x** suivant :

```
[1] 120 134 256 12
```

Utiliser ce vecteur **x** pour générer les deux vecteurs suivants :

```
[1] 220 234 356 112
```

```
[1] 240 268 512 24
```

Exercice 2

On a demandé à 4 ménages le revenu des deux conjoints, et le nombre de personnes du ménage :

```
conjunto1 <- c(1200, 1180, 1750, 2100)
conjunto2 <- c(1450, 1870, 1690, 0)
nb_personnes <- c(4, 2, 3, 2)
```

Calculer le revenu total de chaque ménage, puis diviser par le nombre de personnes pour obtenir le revenu par personne de chaque ménage.

Exercice 3

Dans l'exercice précédent, calculer le revenu minimum et maximum parmi ceux du premier conjoint.

```
conjunto1 <- c(1200, 1180, 1750, 2100)
```

Recommencer avec les revenus suivants, parmi lesquels l'un des enquêtés n'a pas voulu répondre :

```
conjunto1 <- c(1200, 1180, 1750, NA)
```

Exercice 4

Les deux vecteurs suivants représentent les précipitations (en mm) et la température (en °C) moyennes sur la ville de Lyon, pour chaque mois de l'année, entre 1981 et 2010 :

```
temperature <- c(3.4, 4.8, 8.4, 11.4, 15.8, 19.4, 22.2, 21.6, 17.6, 13.4, 7.6, 4.4)
precipitations <- c(47.2, 44.1, 50.4, 74.9, 90.8, 75.6, 63.7, 62, 87.5, 98.6, 81.9, 55)
```

Calculer la température moyenne sur l'année.

Calculer la quantité totale de précipitations sur l'année.

À quoi correspond et comment peut-on interpréter le résultat de la fonction suivante ? Vous pouvez vous aider de la page d'aide de la fonction si nécessaire.

```
cumsum(precipitations)
```

```
[1] 47.2 91.3 141.7 216.6 307.4 383.0 446.7 508.7 596.2 694.8 776.7  
[12] 831.9
```

Même question pour :

```
diff(temperature)
```

```
[1] 1.4 3.6 3.0 4.4 3.6 2.8 -0.6 -4.0 -4.2 -5.8 -3.2
```

Exercice 5

On a relevé les notes en maths, anglais et sport d'une classe de 6 élèves et on a stocké ces données dans trois vecteurs :

```
maths <- c(12, 16, 8, 18, 6, 10)  
anglais <- c(14, 9, 13, 15, 17, 11)  
sport <- c(18, 11, 14, 10, 8, 12)
```

Calculer la moyenne des élèves de la classe en anglais.

Calculer la moyenne générale de chaque élève.

Essayez de comprendre le résultat des deux fonctions suivantes (vous pouvez vous aider de la page d'aide de ces fonctions) :

```
pmin(math, anglais, sport)
```

```
[1] 12 9 8 10 6 10
```

```
pmax(math, anglais, sport)
```

```
[1] 18 16 14 18 17 12
```


Chapitre 3

Premier travail avec des données

3.1 Jeu de données d'exemple

Dans cette partie nous allons (enfin) travailler sur des “vraies” données, et utiliser un jeu de données présent dans l’extension `questionr`. Nous devons donc avant toute chose installer cette extension.

Pour installer ce package, deux possibilités :

- Dans l’onglet *Packages* de la zone de l’écran en bas à droite, cliquez sur le bouton *Install*. Dans le dialogue qui s’ouvre, entrez “`questionr`” dans le champ *Packages* puis cliquez sur *Install*.
- Saisissez directement la commande suivante dans la console :
`install.packages("questionr")`

Dans les deux cas, tout un tas de messages devraient s’afficher dans la console. Attendez que l’invite de commandes > apparaisse à nouveau.

Pour plus d’informations sur les extensions et leur installation, voir la section [2.5](#).

Le jeu de données que nous allons utiliser est un extrait de l’enquête *Histoire de vie* réalisée par l’INSEE en 2003. Il contient 2000 individus et 20 variables. Pour une description plus complète et une liste des variables, voir la section [A.3.2.2](#).

Pour pouvoir utiliser ces données, il faut d’abord charger l’extension `questionr` (après l’avoir installée, bien entendu) :

```
library(questionr)
```

L'utilisation de `library` permet de rendre "disponibles", dans notre session R, les fonctions et jeux de données inclus dans l'extension.

Nous devons ensuite indiquer à R que nous souhaitons accéder au jeu de données à l'aide de la commande `data` :

```
data(hdv2003)
```

Cette commande ne renvoie aucun résultat particulier (sauf en cas d'erreur), mais vous devriez voir apparaître dans l'onglet *Environment* de RStudio un nouvel objet nommé `hdv2003` :



Figure 3.1: Onglet *Environment*

Cet objet est d'un type nouveau : il s'agit d'un tableau de données.

3.2 Tableau de données (*data frame*)

Un *data frame* (ou tableau de données, ou table) est un type d'objet R qui contient des données au format tabulaire, avec les observations en ligne et les variables en colonnes, comme dans une feuille de tableur de type LibreOffice ou Excel.

Si on se contente d'exécuter le nom de notre tableau de données :

```
hdv2003
```

R va, comme à son habitude, nous l'afficher dans la console, ce qui est tout sauf utile.

Une autre manière d'afficher le contenu du tableau est de cliquer sur l'icône en forme de tableau à droite du nom de l'objet dans l'onglet *Environment* :



Figure 3.2: View icon

Ou d'utiliser la fonction `View` :



Dans les deux cas votre tableau devrait s'afficher dans RStudio avec une interface de type tableauur :

					Filter						
	id	age	sex	nivetud	poids	occup	qualif	freres.s	mois.s	mois.y	mois.y.s
1	1	28	Femme	Enseignement supérieur y compris technique sup...	2634.3982	Exerce une profession	Employe				
2	2	23	Femme	NA	9738.3958	Etudiant, elevé	NA				
3	3	59	Homme	Dernière année d'études primaires	3994.1025	Exerce une profession	Technicien				
4	4	34	Homme	Enseignement supérieur y compris technique sup...	5731.6615	Exerce une profession	Technicien				
5	5	71	Femme	Dernière année d'études primaires	4329.0940	Retraite	Employe				
6	6	35	Femme	Enseignement technique ou professionnel court	8674.6994	Exerce une profession	Employe				
7	7	60	Femme	Dernière année d'études primaires	6165.8035	Au foyer	Ouvrier qualifié				
8	8	47	Homme	Enseignement technique ou professionnel court	12891.6408	Exerce une profession	Ouvrier qualifié				
9	9	20	Femme	NA	7808.8721	Etudiant, elevé	NA				
10	10	28	Homme	Enseignement technique ou professionnel long	2277.1605	Exerce une profession	Autre				
11	11	65	Femme	Enseignement supérieur y compris technique sup...	704.3227	Retraite	Employe				
12	12	47	Homme	2ème cycle	6697.8682	Exerce une profession	Ouvrier qualifié				
13	13	63	Femme	Dernière année d'études 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 arrête ses études, avant la dernière année d'ét...	11042.0774	Retraite	NA				
16	16	49	Femme	Enseignement technique ou professionnel court	9958.2287	Exerce une profession	Employe				
17	17	62	Homme	Enseignement supérieur y compris technique sup...	4836.1393	Retraite	Cadre				
18	18	20	Femme	NA	1551.4846	Etudiant, elevé	NA				

Figure 3.3: Interface “View”

Il est important de comprendre que l'objet `hdv2003` contient *l'intégralité* des données du tableau. On voit donc qu'un objet peut contenir des données de types très différents (simple nombre, texte, vecteur, tableau de données entier), et être potentiellement de très grande taille¹.

¹La seule limite pour la taille d'un objet étant la mémoire vive (RAM) de la machine sur laquelle tourne la session R.



Sous R, on peut importer ou créer autant de tableaux de données qu'on le souhaite, dans les limites des capacités de sa machine.

Un *data frame* peut être manipulé comme les autres objets vus précédemment. On peut par exemple faire :

```
d <- hdv2003
```

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

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

```
## Chargement des extensions nécessaires
library(questionr)

## Jeu de données hdv2003
data(hdv2003)
d <- hdv2003
```

3.2.1 Structure du tableau

Un tableau étant un objet comme un autre, on peut lui appliquer des fonctions. Par exemple, nrow et ncol retournent le nombre de lignes et de colonnes du tableau :

```
nrow(d)
```

```
[1] 2000
```

```
ncol(d)
```

```
[1] 20
```

La fonction `dim` renvoie ses dimensions, donc les deux nombres précédents :

```
dim(d)
```

```
[1] 2000    20
```

La fonction `names` retourne les noms des colonnes du tableau, c'est-à-dire la liste de nos *variables* :

```
names(d)
```

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

Enfin, la fonction `str` renvoie un descriptif plus détaillé de la structure du tableau. Elle liste les différentes variables, indique leur type ² et affiche les premières valeurs :

```
str(d)
```

```
'data.frame': 2000 obs. of 20 variables:
 $ id           : int 1 2 3 4 5 6 7 8 9 10 ...
 $ age          : int 28 23 59 34 71 35 60 47 20 28 ...
 $ sexe         : Factor w/ 2 levels "Homme","Femme": 2 2 1 1 2 2 2 1 2 1 ...
 $ nivetud      : Factor w/ 8 levels "N'a jamais fait d'etudes",...: 8 NA 3 8 3 6 3 6 NA 7 ...
 $ poids         : num 2634 9738 3994 5732 4329 ...
 $ occup         : Factor w/ 7 levels "Exerce une profession",...: 1 3 1 1 4 1 6 1 3 1 ...
 $ qualif        : Factor w/ 7 levels "Ouvrier specialise",...: 6 NA 3 3 6 6 2 2 NA 7 ...
 $ freres.soeurs: int 8 2 2 1 0 5 1 5 4 2 ...
 $ clso          : Factor w/ 3 levels "Oui","Non","Ne sait pas": 1 1 2 2 1 2 1 2 1 2 ...
```

²Les différents types de variables seront décrits plus en détail dans le chapitre 9 sur les recodages.

```
$ relig      : Factor w/ 6 levels "Pratiquant regulier",...: 4 4 4 3 1 4 3 4 3 2 ...
$ trav.imp   : Factor w/ 4 levels "Le plus important",...: 4 NA 2 3 NA 1 NA 4 NA 3 ...
$ trav.satisf: Factor w/ 3 levels "Satisfaction",...: 2 NA 3 1 NA 3 NA 2 NA 1 ...
$ hard.rock  : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 1 ...
$ lecture.bd: Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 1 ...
$ peche.chasse: Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 2 2 1 1 ...
$ cuisine    : Factor w/ 2 levels "Non","Oui": 2 1 1 2 1 1 2 2 1 1 ...
$ bricol     : Factor w/ 2 levels "Non","Oui": 1 1 1 2 1 1 1 2 1 1 ...
$ cinema     : Factor w/ 2 levels "Non","Oui": 1 2 1 2 1 2 1 1 2 2 ...
$ sport      : Factor w/ 2 levels "Non","Oui": 1 2 2 2 1 2 1 1 1 2 ...
$ heures.tv  : num  0 1 0 2 3 2 2.9 1 2 2 ...
```

Sous RStudio, on peut afficher à tout moment la structure d'un objet en cliquant sur l'icône de triangle sur fond bleu à gauche du nom de l'objet dans l'onglet *Environment* :

The screenshot shows the RStudio interface with the 'Environment' tab selected. Below the tabs, there are buttons for 'Import Dataset' and 'Global Environment'. Under the 'Data' section, the variable 'd' is selected (indicated by a blue circle). The structure of 'd' is displayed as follows:

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

Figure 3.4: Structure d'un objet

3.2.2 Accéder aux variables d'un tableau

Une opération très importante est l'accès aux variables du tableau (à ses colonnes) pour pouvoir les manipuler, effectuer des calculs, etc. On utilise pour cela l'opérateur \$, qui permet d'accéder aux colonnes du tableau. Ainsi, si l'on tape :

```
d$sexé
```

```
[1] Femme Femme Homme Homme Femme Femme Femme Homme Femme Homme Femme
[12] Homme Femme Femme Femme Femme Homme Femme Homme Femme Femme Homme
[23] Femme Femme Femme Homme Femme Homme Homme Homme Homme Homme Homme
[34] Homme Femme Femme Homme Femme Femme Homme Femme Femme Homme Femme Femme
[45] Femme Homme Femme Femme Femme Homme Femme Femme Homme Femme Homme
[56] Femme Femme Femme Homme Femme Femme Homme Homme Homme Femme Femme
[67] Homme Homme Femme Femme Homme Homme Femme Femme Femme Femme Homme
[78] Femme Femme Femme Femme Femme Homme Homme Femme Femme Homme Homme
[89] Homme Homme Homme Femme Homme Femme Femme Femme Homme Homme Femme
[100] Femme Femme Homme Femme Homme Femme Femme Femme Femme Femme Homme
[111] Homme Homme Homme Homme Femme Homme Femme Femme Homme Femme Homme
[122] Femme Femme Homme Femme Femme Homme Femme Femme Homme Femme Homme
[133] Femme Femme Femme Homme Homme Homme Homme Homme Homme Homme Homme
[144] Femme Homme Homme Homme Femme Femme Femme Homme Femme Femme Femme
[155] Femme Homme Femme Homme Homme Femme Homme Femme Femme Homme Femme
[166] Homme Homme Femme Femme Homme Femme Homme Femme Femme Femme Femme
[177] Homme Homme Homme Femme Homme Femme Femme Homme Femme Homme Femme Femme
[188] Femme Femme Femme Homme Homme Femme Homme Femme Homme Femme Femme
[199] Homme Femme
[ reached getOption("max.print") -- omitted 1800 entries ]
Levels: Homme Femme
```

R va nous afficher l'ensemble des valeurs de notre variable `sexé` dans la console, ce qui est à nouveau fort peu utile. Mais cela nous permet de constater que `d$sexé` est un vecteur de chaînes de caractères tels qu'on en a déjà rencontré précédemment.

La fonction `table$colonne` renvoie donc la colonne nommée `colonne` du tableau `table`, c'est-à-dire un vecteur, en général de nombres ou de chaînes de caractères.

Si on souhaite afficher seulement les premières ou dernières valeurs d'une variable, on peut utiliser les fonctions `head` et `tail` :

```
head(d$age)
```

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

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

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

Le deuxième argument numérique permet d'indiquer le nombre de valeurs à afficher.

3.2.3 Créer une nouvelle variable

On peut aussi utiliser l'opérateur `$` pour créer une nouvelle variable dans notre tableau : pour cela, il suffit de lui assigner une valeur.

Par exemple, la variable `heures.tv` contient le nombre d'heures passées quotidiennement devant la télé :

```
head(d$heures.tv, 10)
```

```
[1] 0.0 1.0 0.0 2.0 3.0 2.0 2.9 1.0 2.0 2.0
```

On peut vouloir créer une nouvelle variable dans notre tableau qui contienne la même durée mais en minutes. On va donc créer une nouvelle variables `minutes.tv` de la manière suivante :

```
d$minutes.tv <- d$heures.tv * 60
```

On peut alors constater, soit visuellement soit dans la console, qu'une nouvelle variable (une nouvelle colonne) a bien été ajoutée au tableau :

```
head(d$minutes.tv)
```

```
[1] 0 60 0 120 180 120
```

3.3 Analyse univariée

On a donc désormais accès à un tableau de données `d`, dont les lignes sont des observations (des individus enquêtés), et les colonnes des variables (des

caractéristiques de chacun de ces individus), et on sait accéder à ces variables grâce à l'opérateur \$.

Si on souhaite analyser ces variables, les méthodes et fonctions utilisées seront différentes selon qu'il s'agit d'une variable *quantitative* (variable numérique pouvant prendre un grand nombre de valeurs : l'âge, le revenu, un pourcentage...) ou d'une variable *qualitative* (variable pouvant prendre un nombre limité de valeurs appelées modalités : le sexe, la profession, le dernier diplôme obtenu, etc.).

3.3.1 Analyser une variable quantitative

Une variable quantitative est une variable de type numérique (un nombre) qui peut prendre un grand nombre de valeurs. On en a plusieurs dans notre jeu de données, notamment l'âge (variable `age`) ou le nombre d'heures passées devant la télé (`heures.tv`).

3.3.1.1 Indicateurs de centralité

Caractériser une variable quantitative, c'est essayer de décrire la manière dont ses valeurs se répartissent, ou se distribuent.

Pour cela on peut commencer par regarder les valeurs extrêmes, avec les fonctions `min`, `max` ou `range` :

```
min(d$age)
```

```
[1] 18
```

```
max(d$age)
```

```
[1] 97
```

```
range(d$age)
```

```
[1] 18 97
```

On peut aussi calculer des indicateurs de *centralité* : ceux-ci indiquent autour de quel nombre se répartissent les valeurs de la variable. Il y en a plusieurs, le plus connu étant la moyenne, qu'on peut calculer avec la fonction `mean` :

```
mean(d$age)
```

```
[1] 48.157
```

Il existe aussi la médiane, qui est la valeur qui sépare notre population en deux : on a la moitié de nos observations en-dessous, et la moitié au-dessus. Elle se calcule avec la fonction `median` :

```
median(d$age)
```

```
[1] 48
```

Une différence entre les deux indicateurs est que la médiane est beaucoup moins sensible aux valeurs “extrêmes” : on dit qu’elle est plus *robuste*. Ainsi, en 2013, le salaire net *moyen* des salariés à temps plein en France était de 2202 euros, tandis que le salaire net *médian* n’était que de 1772 euros. La différence étant due à des très hauts salaires qui “tirent” la moyenne vers le haut.

3.3.1.2 Indicateurs de dispersion

Les indicateurs de dispersion permettent de mesurer si les valeurs sont plutôt regroupées ou au contraire plutôt dispersées.

L’indicateur le plus simple est l’étendue de la distribution, qui décrit l’écart maximal observé entre les observations :

```
max(d$age) - min(d$age)
```

```
[1] 79
```

Les indicateurs de dispersion les plus utilisés sont la variance ou, de manière équivalente, l’écart-type (qui est égal à la racine carrée de la variance). On obtient la première avec la fonction `var`, et le second avec `sd` (abréviation de *standard deviation*) :

```
var(d$age)
```

```
[1] 287.0249
```

```
sd(d$age)
```

```
[1] 16.94181
```

Plus la variance ou l'écart-type sont élevés, plus les valeurs sont dispersées autour de la moyenne. À l'inverse, plus ils sont faibles et plus les valeurs sont regroupées.

Une autre manière de mesurer la dispersion est de calculer les quartiles :

- le premier quartile est la valeur pour laquelle on a 25% des observations en dessous et 75% au dessus
- le deuxième quartile est la valeur pour laquelle on a 50% des observations en dessous et 50% au dessus (c'est donc la médiane)
- le troisième quartile est la valeur pour laquelle on a 75% des observations en dessous et 25% au dessus

On peut les calculer avec la fonction `quantile` :

```
## Premier quartile  
quantile(d$age, prob = 0.25)
```

```
25%  
35
```

```
## Troisième quartile  
quantile(d$age, prob = 0.75)
```

```
75%  
60
```

`quantile` prend deux arguments principaux : le vecteur dont on veut calculer le quantile, et un argument `prob` qui indique quel quantile on souhaite obtenir.

`prob` prend une valeur entre 0 et 1 : 0.5 est la médiane, 0.25 le premier quartile, 0.1 le premier décile, etc.

Notons enfin que la fonction `summary` permet d'obtenir d'un coup plusieurs indicateurs classiques :

```
summary(d$age)
```

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

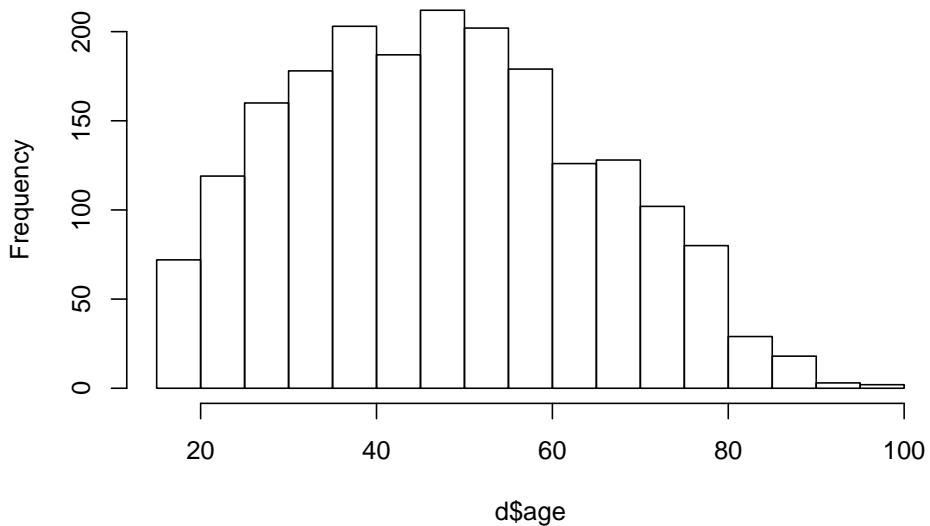
3.3.1.3 Représentation graphique

L'outil le plus utile pour étudier la distribution des valeurs d'une variable quantitative reste la représentation graphique.

La représentation la plus courante est sans doute l'histogramme. On peut l'obtenir avec la fonction `hist` :

```
hist(d$age)
```

Histogram of d\$age

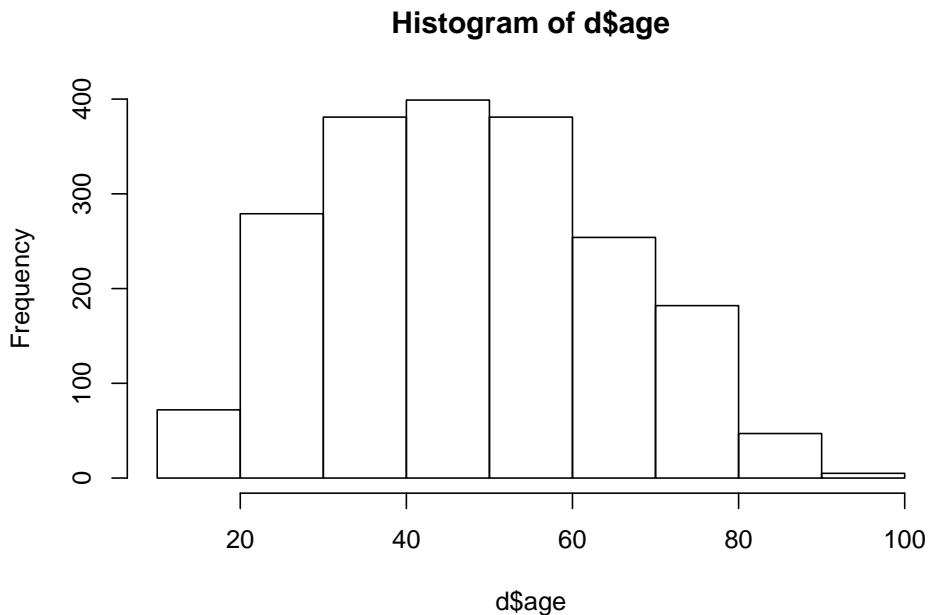


Cette fonction n'a pas pour effet direct d'effectuer un calcul ou de nous renvoyer un résultat : elle génère un graphique qui va s'afficher dans l'onglet *Plots* de

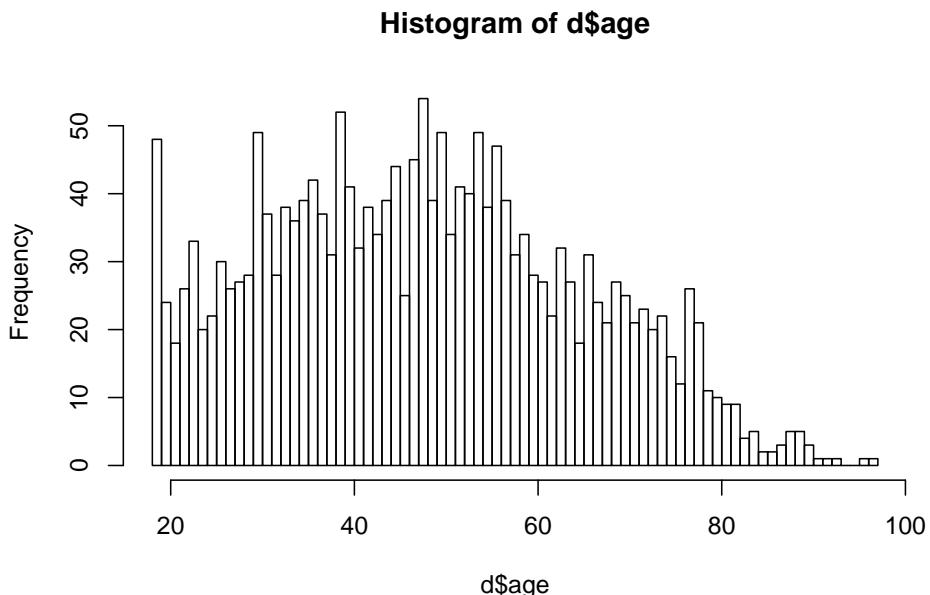
RStudio.

On peut personnaliser l'apparence de l'histogramme en ajoutant des arguments supplémentaires à la fonction `hist`. L'argument le plus important est `breaks`, qui permet d'indiquer le nombre de classes que l'on souhaite.

```
hist(d$age, breaks = 10)
```



```
hist(d$age, breaks = 70)
```

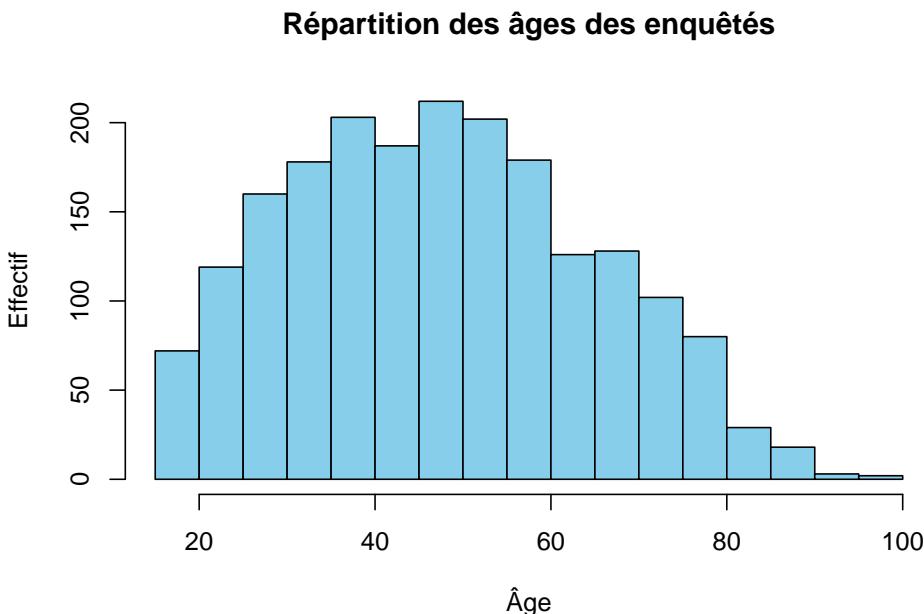


Le choix d'un “bon” nombre de classes pour un histogramme n'est pas un problème simple : si on a trop peu de classes, on risque d'effacer quasiment toutes les variations, et si on en a trop on risque d'avoir trop de détails et de masquer les grandes tendances.

Les arguments de `hist` permettent également de modifier la présentation du graphique. On peut ainsi changer la couleur des barres avec `col`³, le titre avec `main`, les étiquettes des axes avec `xlab` et `ylab`, etc. :

```
hist(d$age, col = "skyblue",
  main = "Répartition des âges des enquêtés",
  xlab = "Âge",
  ylab = "Effectif")
```

³Les différentes manières de spécifier des couleurs sont indiquées dans l'encadré de la section 8.7.3.



La fonction `hist` fait partie des fonctions graphiques de base de R. On verra plus en détail d'autres fonctions graphiques dans la partie 8 de ce document, consacrée à l'extension `ggplot2`, qui fait partie du *tidyverse* et qui permet la production et la personnalisation de graphiques complexes.

3.3.2 Analyser une variable qualitative

Une variable qualitative est une variable qui ne peut prendre qu'un nombre limité de valeurs, appelées modalités. Dans notre jeu de données on trouvera par exemple le sexe (`sex`), le niveau d'études (`niveau`), la catégorie socio-professionnelle (`qualif`)...

À noter qu'une variable qualitative peut tout-à-fait être numérique, et que certaines variables peuvent être traitées soit comme quantitatives, soit comme qualitatives : c'est le cas par exemple du nombre d'enfants ou du nombre de frères et soeurs.

3.3.2.1 Tri à plat

L'outil le plus utilisé pour représenter la répartition des valeurs d'une variable qualitative est le *tri à plat* : il s'agit simplement de compter, pour chacune des valeurs possibles de la variable (pour chacune des modalités), le nombre d'observations ayant cette valeur. Un tri à plat s'obtient sous R à l'aide de la fonction `table` :

```
table(d$sexe)
```

Homme	Femme
899	1101

Ce tableau nous indique donc que parmi nos enquêtés on trouve 899 hommes et 1101 femmes.

```
table(d$qualif)
```

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

Un tableau de ce type peut être affiché ou stocké dans un objet, et on peut à son tour lui appliquer des fonctions. Par exemple, la fonction `sort` permet de trier le tableau selon la valeur de l'effectif. On peut donc faire :

```
tab <- table(d$qualif)
sort(tab)
```

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



Attention, par défaut la fonction `table` n'affiche pas les valeurs manquantes (NA). Si on souhaite les inclure il faut utiliser l'argument `useNA = "always"`, soit : `table(d$qualif, useNA = "always")`.

À noter qu'on peut aussi appliquer `summary` à une variable qualitative. Le résultat est également le tri à plat de la variable, avec en plus le nombre de

valeurs manquantes éventuelles :

```
summary(d$qualif)
```

	Ouvrier specialise	Ouvrier qualifie	Technicien
	203	292	86
Profession intermediaire		Cadre	Employe
	160	260	594
Autre	58	NA's	
		347	

Par défaut ces tris à plat sont en effectifs et ne sont donc pas toujours très lisibles, notamment quand on a des effectifs importants. On leur rajoute donc en général la répartition en pourcentages. Pour cela, nous allons utiliser la fonction `freq` de l'extension `questionr`, qui devra donc avoir précédemment été chargée avec `library(questionr)` :

```
## À rajouter en haut de script et à exécuter
library(questionr)
```

On peut alors utiliser la fonction :

```
freq(d$qualif)
```

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

La colonne `n` représente les effectifs de chaque catégorie, la colonne `%` le pourcentage, et la colonne `val%` le pourcentage calculé sur les valeurs valides, donc en excluant les `NA`. Une ligne a également été rajoutée pour indiquer le nombre et la proportion de `NA`.

`freq` accepte un certain nombre d'arguments pour personnaliser son affichage. Par exemple :

- `valid` indique si on souhaite ou non afficher les pourcentages sur les valeurs valides
- `cum` indique si on souhaite ou non afficher les pourcentages cumulés
- `total` permet d'ajouter une ligne avec les effectifs totaux
- `sort` permet de trier le tableau par fréquence croissante (`sort="inc"`) ou décroissante (`sort="dec"`).

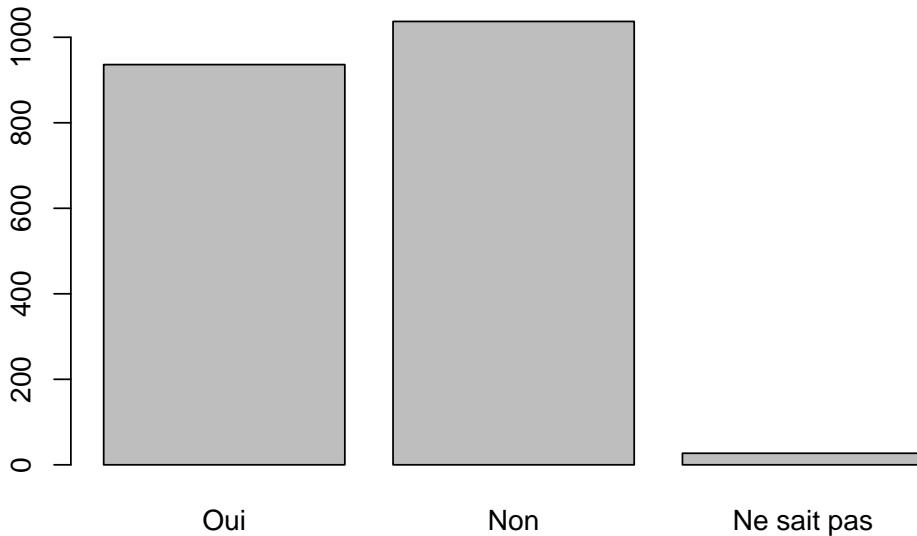
```
freq(d$qualif, valid= FALSE, total = TRUE, sort = "dec")
```

	n	%
Employe	594	29.7
Ouvrier qualifie	292	14.6
Cadre	260	13.0
Ouvrier specialise	203	10.2
Profession intermediaire	160	8.0
Technicien	86	4.3
Autre	58	2.9
NA	347	17.3
Total	2000	100.0

3.3.2.2 Représentations graphiques

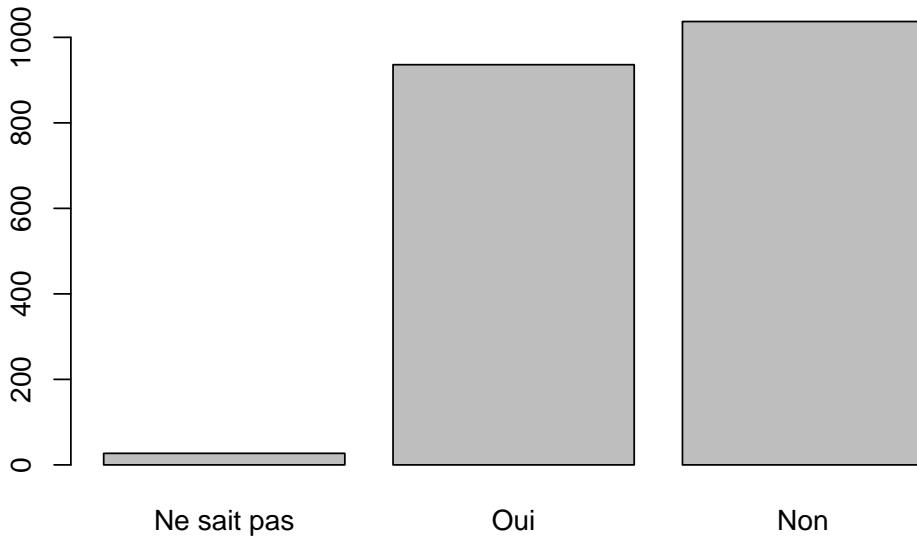
On peut représenter graphiquement le tri à plat d'une variable qualitative avec un diagramme en barres, obtenu avec la fonction `barplot`. Attention, contrairement à `hist` cette fonction ne s'applique pas directement à la variable mais au résultat du tri à plat de cette variable, calculé avec `table`. Il faut donc procéder en deux étapes :

```
tab <- table(d$cuso)
barplot(tab)
```



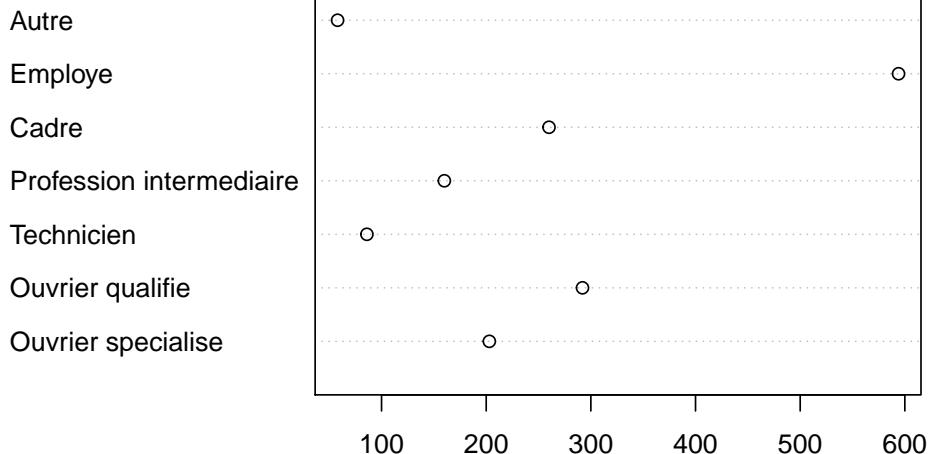
On peut aussi trier le tri à plat avec la fonction `sort` avant de le représenter graphiquement, ce qui peut faciliter la lecture du graphique :

```
barplot(sort(tab))
```



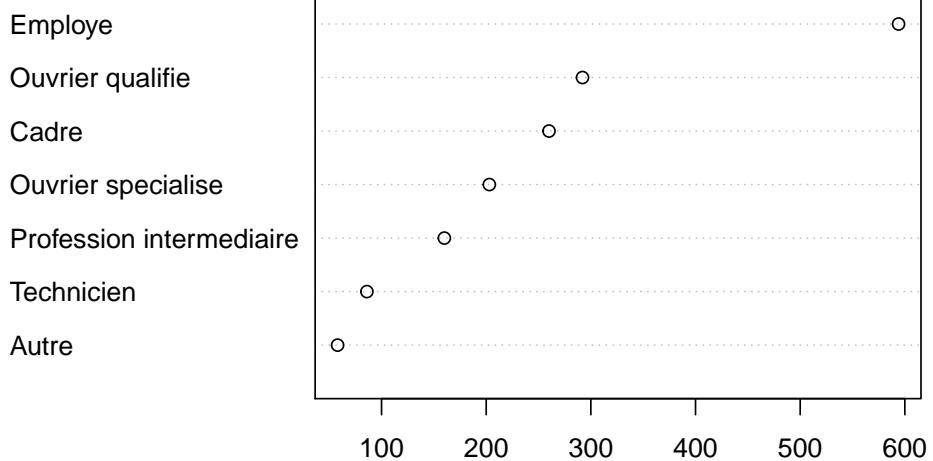
Une alternative au graphique en barres est le *diagramme de Cleveland*, qu'on peut obtenir avec la fonction `dotchart`. Celle-ci s'applique elle aussi au tri à plat de la variable calculé avec `table`.

```
dotchart(table(d$qualif))
```



Là aussi, pour améliorer la lisibilité du graphique il est préférable de trier le tri à plat de la variable avant de le représenter :

```
dotchart(sort(table(d$qualif)))
```



3.4 Exercices

Exercice 1

Créer un nouveau script qui effectue les actions suivantes :

- charger l'extension `questionr`
- charger le jeu de données nommé `hdv2003`
- copier le jeu de données dans un nouvel objet nommé `df`
- afficher les dimensions et la liste des variables de `df`

Exercice 2

On souhaite étudier la répartition du temps passé devant la télévision par les enquêtés (variable `heures.tv`). Pour cela, affichez les principaux indicateurs de cette variable : valeur minimale, maximale, moyenne, médiane et écart-type. Représentez ensuite sa distribution par un histogramme en 10 classes.

Exercice 3

On s'intéresse maintenant à l'importance accordée par les enquêtés à leur travail (variable `trav.imp`). Faites un tri à plat des effectifs des modalités de cette variable avec la commande `table`.

Faites un tri à plat affichant à la fois les effectifs et les pourcentages de chaque modalité. Y'a-t-il des valeurs manquantes ?

Représentez graphiquement les effectifs des modalités à l'aide d'un graphique en barres.

Utilisez l'argument `col` de la fonction `barplot` pour modifier la couleur du graphique en `tomato`.

Tapez `colors()` dans la console pour afficher l'ensemble des noms de couleurs disponibles dans R. Testez chaque couleur une à une pour trouver votre couleur préférée.

Chapitre 4

Analyse bivariée

Faire une analyse bivariée, c'est étudier la relation entre deux variables : sont-elles liées ? les valeurs de l'une influencent-elles les valeurs de l'autre ? ou sont-elles au contraire indépendantes ?

À noter qu'on va parler ici d'influence ou de lien, mais pas de relation de cause à effet : les outils présentés permettent de visualiser ou de déterminer une relation, mais des liens de causalité proprement dit sont plus difficiles à mettre en évidence. Il faut en effet vérifier que c'est bien telle variable qui influence telle autre et pas l'inverse, qu'il n'y a pas de “variable cachée”, etc.

Là encore, le type d'analyse ou de visualisation est déterminé par la nature qualitative ou quantitative des deux variables.

4.1 Croisement de deux variables qualitatives

4.1.1 Tableaux croisés

On va continuer à travailler avec le jeu de données tiré de l'enquête *Histoire de vie* inclus dans l'extension `questionr`. On commence donc par charger l'extension, le jeu de données, et à le renommer en un nom plus court pour gagner un peu de temps de saisie au clavier :

```
library(questionr)
data(hdv2003)
d <- hdv2003
```

Quand on veut croiser deux variables qualitatives, on fait un *tableau croisé*. Comme pour un tri à plat ceci s'obtient avec la fonction `table` de R, mais à

laquelle on passe cette fois deux variables en argument. Par exemple, si on veut croiser la catégorie socio-professionnelle et le sexe des enquêtés :

```
table(d$qualif, d$sex)
```

	Homme	Femme
Ouvrier specialise	96	107
Ouvrier qualifie	229	63
Technicien	66	20
Profession intermediaire	88	72
Cadre	145	115
Employe	96	498
Autre	21	37

Pour pouvoir interpréter ce tableau on doit passer du tableau en effectifs au tableau en pourcentages ligne ou colonne. Pour cela, on peut utiliser les fonctions `lprop` et `cprop` de l'extension `questionr`, qu'on applique au tableau croisé précédent.

Pour calculer les pourcentages ligne :

```
tab <- table(d$qualif, d$sex)
lprop(tab)
```

	Homme	Femme	Total
Ouvrier specialise	47.3	52.7	100.0
Ouvrier qualifie	78.4	21.6	100.0
Technicien	76.7	23.3	100.0
Profession intermediaire	55.0	45.0	100.0
Cadre	55.8	44.2	100.0
Employe	16.2	83.8	100.0
Autre	36.2	63.8	100.0
Ensemble	44.8	55.2	100.0

Et pour les pourcentages colonne :

```
cprop(tab)
```

	Homme	Femme	Ensemble
Ouvrier specialise	13.0	11.7	12.3
Ouvrier qualifie	30.9	6.9	17.7
Technicien	8.9	2.2	5.2
Profession intermediaire	11.9	7.9	9.7
Cadre	19.6	12.6	15.7
Employe	13.0	54.6	35.9
Autre	2.8	4.1	3.5
Total	100.0	100.0	100.0



Pour savoir si on doit faire des pourcentages ligne ou colonne, on pourra se référer à l'article suivant :

<http://alain-leger.lescigales.org/textes/lignecolonne.pdf>

En résumé, quand on fait un tableau croisé, celui-ci est parfaitement symétrique : on peut inverser les lignes et les colonnes, ça ne change pas son interprétation. Par contre, on a toujours en tête un “sens” de lecture dans le sens où on considère que l’une des variables *dépend* de l’autre. Par exemple, si on croise sexe et type de profession, on dira que le type de profession dépend du sexe, et non l’inverse : le type de profession est alors la variable *dépendante* (à expliquer), et le sexe la variable *indépendante* (explicative).

Pour faciliter la lecture d’un tableau croisé, il est recommandé de **faire les pourcentages sur la variable indépendante**. Dans notre exemple, la variable indépendante est le sexe, elle est en colonne, on calcule donc les pourcentages colonnes qui permettent de comparer directement, pour chaque sexe, la répartition des catégories socio-professionnelles.

4.1.2 Test du χ^2

Comme on travaille sur un échantillon et pas sur une population entière, on peut compléter ce tableau croisé par un test d’indépendance du χ^2 . Celui-ci permet de rejeter l’hypothèse d’indépendance des lignes et des colonnes du tableau, c’est à dire de rejeter l’hypothèse que les écarts à l’indépendance observés seraient uniquement dus au biais d’échantillonnage (au fait qu’on n’a pas interrogé toute notre population).

Pour effectuer un test de ce type, on applique la fonction `chisq.test` au tableau croisé calculé précédemment :

```
chisq.test(tab)
```

```
Pearson's Chi-squared test

data: tab
X-squared = 387.56, df = 6, p-value < 2.2e-16
```

Le résultat nous indique trois valeurs :

- **X-squared**, la valeur de la statistique du χ^2 pour notre tableau, c'est-à-dire une "distance" entre notre tableau observé et celui attendu si les deux variables étaient indépendantes.
- **df**, le nombre de degrés de libertés du test.
- **p-value**, le fameux p , qui indique la probabilité d'obtenir une valeur de la statistique du χ^2 au moins aussi extrême sous l'hypothèse d'indépendance.

Ici, le p est extrêmement petit (la notation $< 2.2e-16$ indique qu'il est plus petit que la plus petite valeur proche de zéro calculable par R), donc certainement en-dessous du seuil de décision choisi préalablement au test (souvent 5%, soit 0.05). On peut donc rejeter l'hypothèse d'indépendance des lignes et des colonnes du tableau.

En complément du test du χ^2 , on peut aussi regarder les *résidus* de ce test pour affiner la lecture du tableau. Ceux-ci s'obtiennent avec la fonction `chisq.residuals` de `questionr` :

```
chisq.residuals(tab)
```

	Homme	Femme
Ouvrier specialise	0.52	-0.47
Ouvrier qualifie	8.57	-7.73
Technicien	4.42	-3.98
Profession intermediaire	1.92	-1.73
Cadre	2.64	-2.38
Employe	-10.43	9.41
Autre	-0.98	0.88

L'interprétation des résidus est la suivante :

- si la valeur du résidu pour une case est inférieure à -2, alors il y a une sous-représentation de cette case dans le tableau : les effectifs sont significativement plus faibles que ceux attendus sous l'hypothèse d'indépendance

- à l'inverse, si le résidu est supérieur à 2, il y a sur-représentation de cette case
- si le résidu est compris entre -2 et 2, il n'y a pas d'écart à l'indépendance significatif

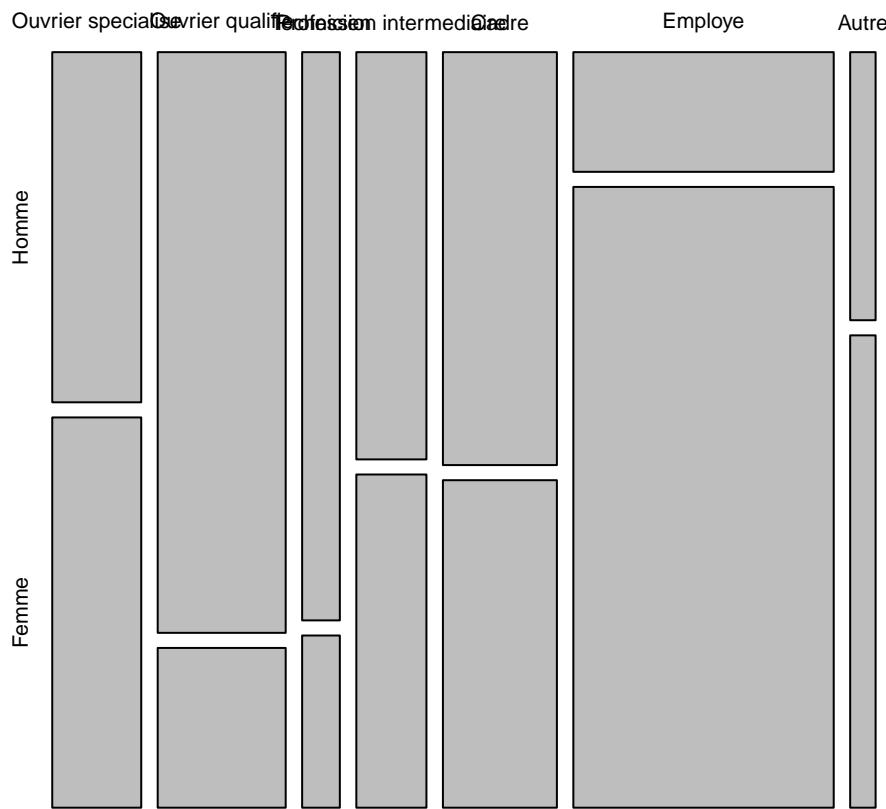
Les résidus peuvent être une aide utile à l'interprétation, notamment pour des tableaux de grande dimension.

4.1.3 Représentation graphique

Il est possible de faire une représentation graphique d'un tableau croisé, par exemple avec la fonction `mosaicplot` :

```
mosaicplot(tab)
```

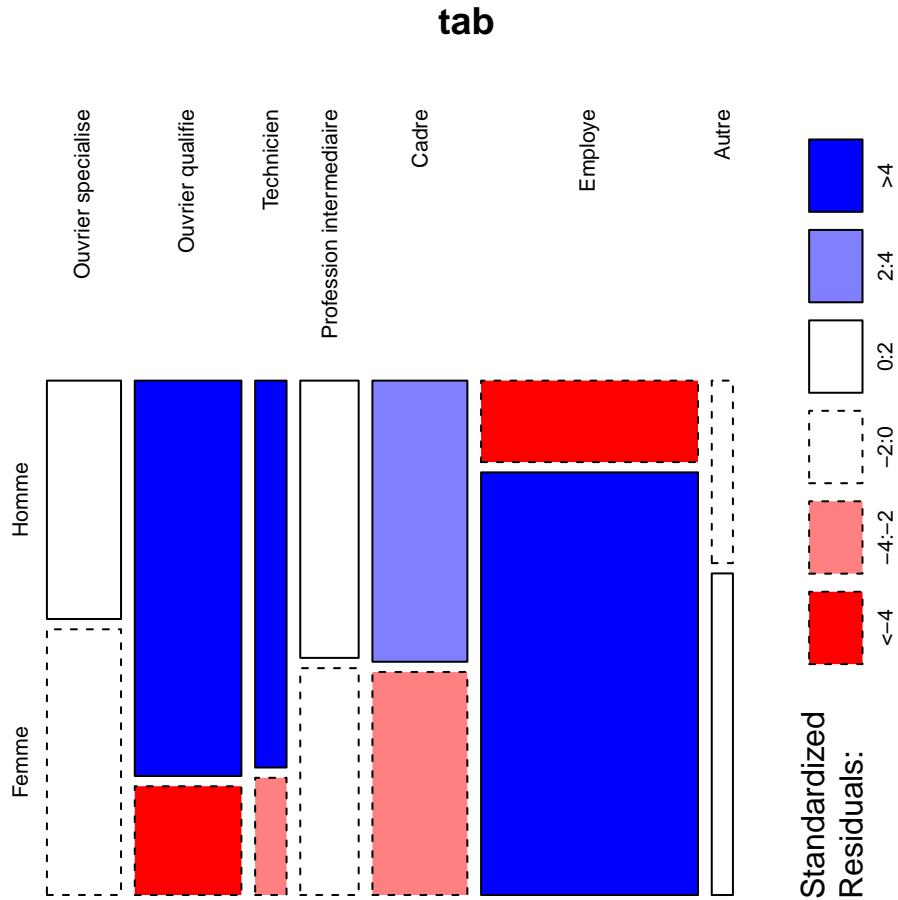
tab



On peut améliorer ce graphique en colorant les cases selon les résidus du test du

² (argument `shade = TRUE`) et en orientant verticalement les labels de colonnes (argument `las = 3`) :

```
mosaicplot(tab, las = 3, shade = TRUE)
```



Chaque rectangle de ce graphique représente une case de tableau. Sa largeur correspond au pourcentage des modalités en colonnes (il y'a beaucoup d'employés et d'ouvriers et très peu d'“autres”). Sa hauteur correspond aux pourcentages colonnes : la proportion d'hommes chez les cadres est plus élevée que chez les employés. Enfin, la couleur de la case correspond au résidu du test du ² correspondant : les cases en rouge sont sous-représentées, les cases en bleu sur-représentées, et les cases blanches sont proches des effectifs attendus sous l'hypothèse d'indépendance.

4.2 Croisement d'une variable quantitative et d'une variable qualitative

4.2.1 Représentation graphique

Croiser une variable quantitative et une variable qualitative, c'est essayer de voir si les valeurs de la variable quantitative se répartissent différemment selon la catégorie d'appartenance de la variable qualitative.

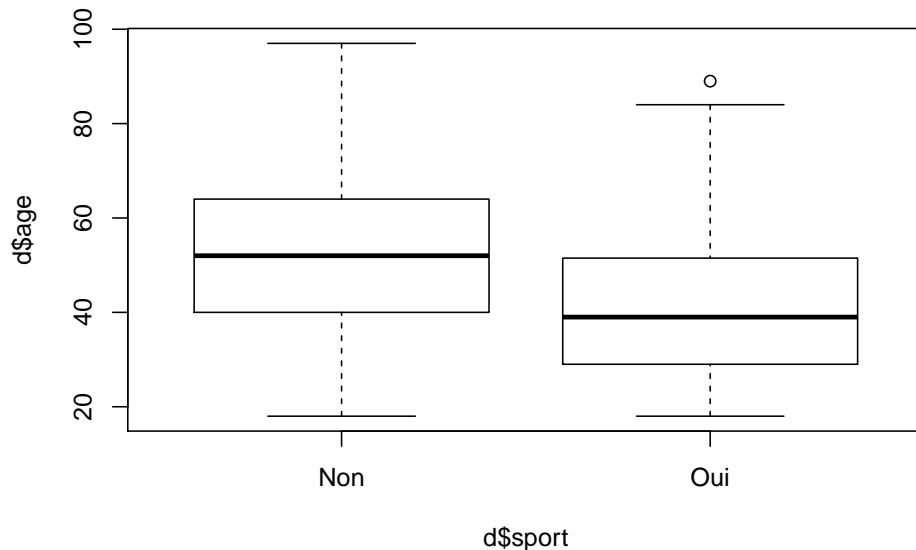
Pour cela, l'idéal est de commencer par une représentation graphique de type "boîte à moustache" à l'aide de la fonction `boxplot`. Par exemple, si on veut visualiser la répartition des âges selon la pratique ou non d'un sport, on va utiliser la syntaxe suivante :

```
boxplot(d$age ~ d$sport)
```



Cette syntaxe de `boxplot` utilise une nouvelle notation de type "formule". Celle-ci est utilisée notamment pour la spécification des modèles de régression. Ici le `~` peut se lire comme "en fonction de" : on veut représenter le boxplot de l'âge en fonction du sport.

Ce qui va nous donner le résultat suivant :





L'interprétation d'un boxplot est la suivante : Les bords inférieurs et supérieurs du carré central représentent le premier et le troisième quartile de la variable représentée sur l'axe vertical. On a donc 50% de nos observations dans cet intervalle. Le trait horizontal dans le carré représente la médiane. Enfin, des "moustaches" s'étendent de chaque côté du carré, jusqu'aux valeurs minimales et maximales, avec une exception : si des valeurs sont éloignées du carré de plus de 1,5 fois l'écart interquartile (la hauteur du carré), alors on les représente sous forme de points (symbolisant des valeurs considérées comme "extrêmes").

Dans le graphique ci-dessus, on voit que ceux qui ont pratiqué un sport au cours des douze derniers mois ont l'air d'être sensiblement plus jeunes que les autres.

4.2.2 Calculs d'indicateurs

On peut aussi vouloir comparer certains indicateurs (moyenne, médiane) d'une variable quantitative selon les modalités d'une variable qualitative. Si on reprend l'exemple précédent, on peut calculer la moyenne d'âge pour ceux qui pratiquent un sport et pour ceux qui n'en pratiquent pas.

Une première méthode pour cela est d'extraire de notre population autant de sous-populations qu'il y a de modalités dans la variable qualitative. On peut le faire notamment avec la fonction `filter` du package `dplyr`¹.

On commence par charger `dplyr` (en l'ayant préalablement installé) :

```
library(dplyr)
```

Puis on applique `filter` pour créer deux sous-populations, stockées dans deux nouveaux tableaux de données :

```
d_sport <- filter(d, sport == "Oui")
d_nonsport <- filter(d, sport == "Non")
```

On peut ensuite utiliser ces deux nouveaux tableaux de données comme on en a l'habitude, et calculer les deux moyennes d'âge :

```
mean(d_sport$age)
```

¹Le package en question est présenté en détail dans la partie 10.

```
[1] 40.92531
```

```
mean(d_nonsport$age)
```

```
[1] 52.25137
```

Une autre possibilité est d'utiliser la fonction `tapply`, qui prend en paramètre une variable quantitative, une variable qualitative et une fonction, puis applique automatiquement la fonction aux valeurs de la variables quantitative pour chaque niveau de la variable qualitative :

```
tapply(d$age, d$sport, mean)
```

	Non	Oui
52.25137	40.92531	

On verra dans la partie 10 d'autres méthodes basées sur `dplyr` pour effectuer ce genre d'opérations.

4.2.3 Tests statistiques

Un des tests les plus connus est le test du *t* de Student, qui permet de tester si les moyennes de deux sous-populations peuvent être considérées comme différentes (compte tenu des fluctuations aléatoires provenant du biais d'échantillonnage).

Un test *t* s'effectue à l'aide de la fonction `t.test`. Ainsi, on peut tester l'hypothèse d'égalité des âges moyens selon la pratique ou non d'un sport avec la commande suivante :

```
t.test(d$age ~ d$sport)
```

```
Welch Two Sample t-test

data: d$age by d$sport
t = 15.503, df = 1600.4, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
```

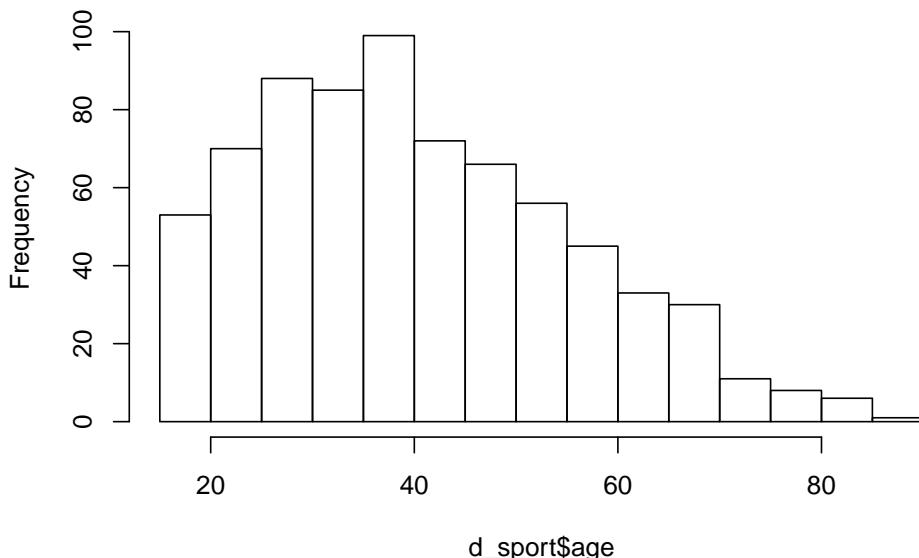
```
95 percent confidence interval:
 9.893117 12.759002
sample estimates:
mean in group Non mean in group Oui
      52.25137        40.92531
```

Le résultat du test est significatif, avec un p extrêmement petit, et on peut rejeter l'hypothèse nulle d'égalité des moyennes des deux groupes. Le test nous donne même un intervalle de confiance à 95% pour la valeur de la différence entre les deux moyennes.

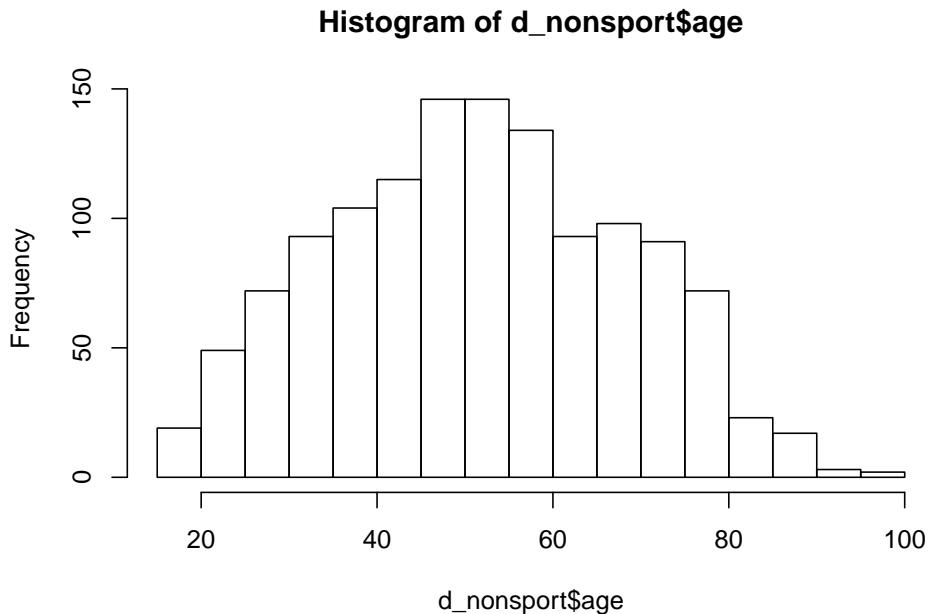
Nous sommes cependant allés un peu vite, et avons négligé le fait que le test t s'applique normalement à des distributions normales. On peut se faire un premier aperçu visuel de cette normalité en traçant les histogrammes des deux répartitions :

```
hist(d_sport$age)
```

Histogram of d_sport\$age



```
hist(d_nonsport$age)
```



Si l'âge dans le groupe des non sportifs se rapproche d'une distribution normale, celui des sportifs en semble assez éloigné, notamment du fait de la limite d'âge à 18 ans imposée par construction de l'enquête.

On peut tester cette normalité à l'aide du test de Shapiro-Wilk et de la fonction `shapiro.test` :

```
shapiro.test(d_sport$age)
```

```
Shapiro-Wilk normality test

data: d_sport$age
W = 0.96203, p-value = 9.734e-13
```

```
shapiro.test(d_nonsport$age)
```

```
Shapiro-Wilk normality test

data: d_nonsport$age
W = 0.98844, p-value = 1.654e-08
```

Le test est significatif dans les deux cas et rejette l'hypothèse d'une normalité des deux distributions.

Dans ce cas on peut faire appel à un test non-paramétrique, qui ne fait donc pas d'hypothèses sur les lois de distribution des variables testées, en l'occurrence le test des rangs de Wilcoxon, à l'aide de la fonction `wilcox.test` :

```
wilcox.test(d$age ~ d$sport)
```

```
Wilcoxon rank sum test with continuity correction
```

```
data: d$age by d$sport
W = 640577, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0
```

La valeur p étant à nouveau extrêmement petite, on peut rejeter l'hypothèse d'indépendance et considérer que les distributions des âges dans les deux sous-populations sont différentes.

4.3 Croisement de deux variables quantitatives

Le jeu de données `hdv2003` comportant assez peu de variables quantitatives, on va s'intéresser maintenant à un autre jeu de données comportant des informations du recensement de la population de 2012. On le charge avec :

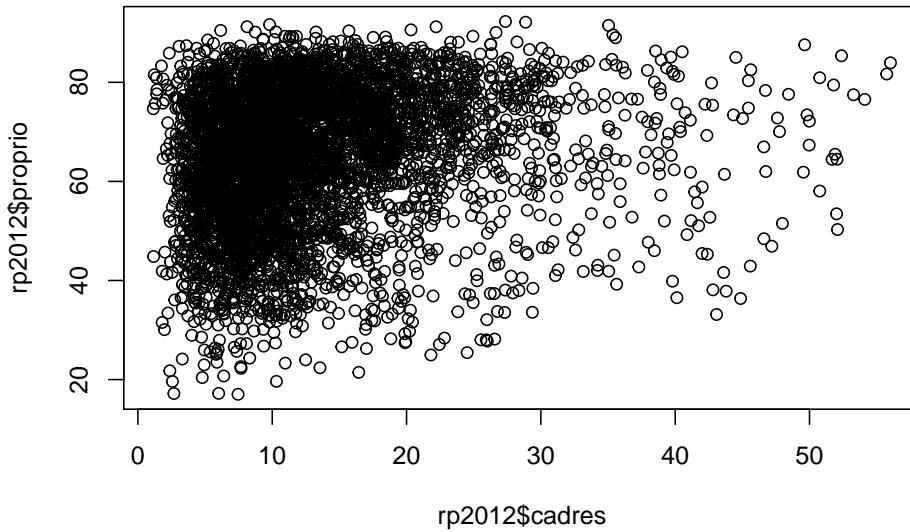
```
data(rp2012)
```

Un nouveau tableau de données `rp2012` devrait apparaître dans votre environnement. Celui-ci comprend les 5170 communes de France métropolitaine de plus de 2000 habitants, et une soixantaine de variables telles que le département, la population, le taux de chômage, etc. Pour une description plus complète et une liste des variables, voir section A.3.2.3.

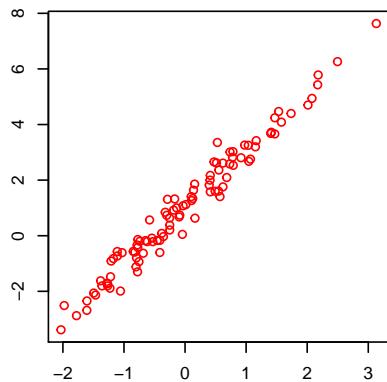
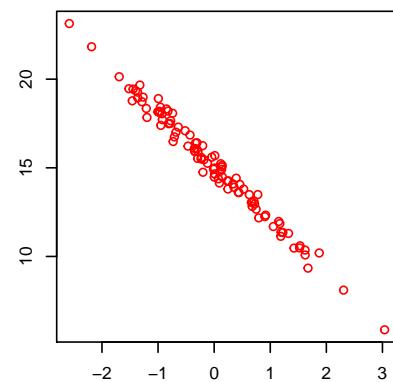
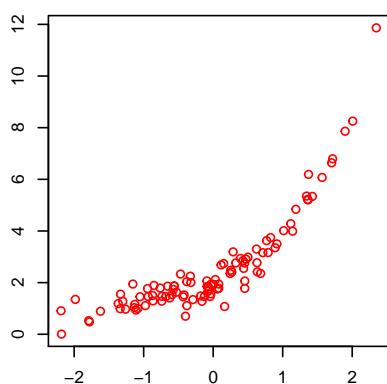
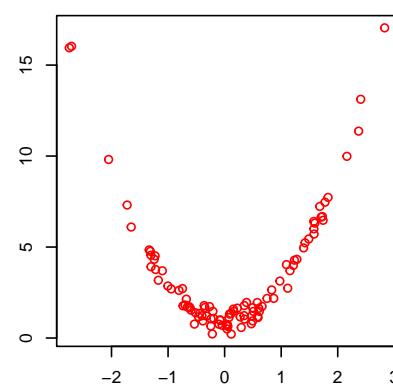
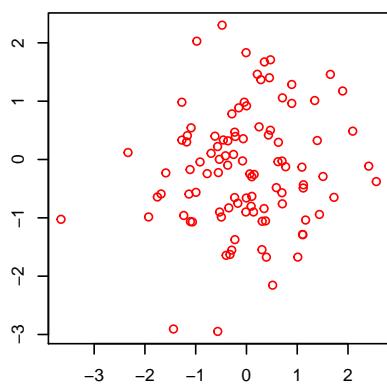
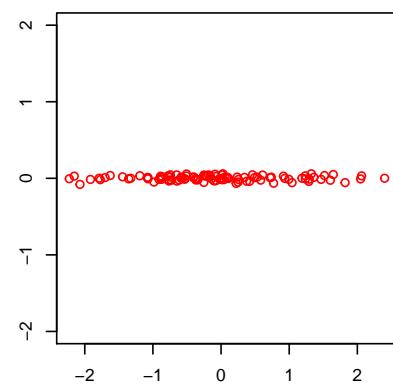
4.3.1 Représentation graphique

Quand on croise deux variables quantitatives, l'idéal est de faire une représentation graphique sous forme de nuage de points à l'aide de la fonction `plot`. On va représenter le croisement entre le pourcentage de cadres et le pourcentage de propriétaires dans la commune :

```
plot(rp2012$cadres, rp2012$proprio)
```

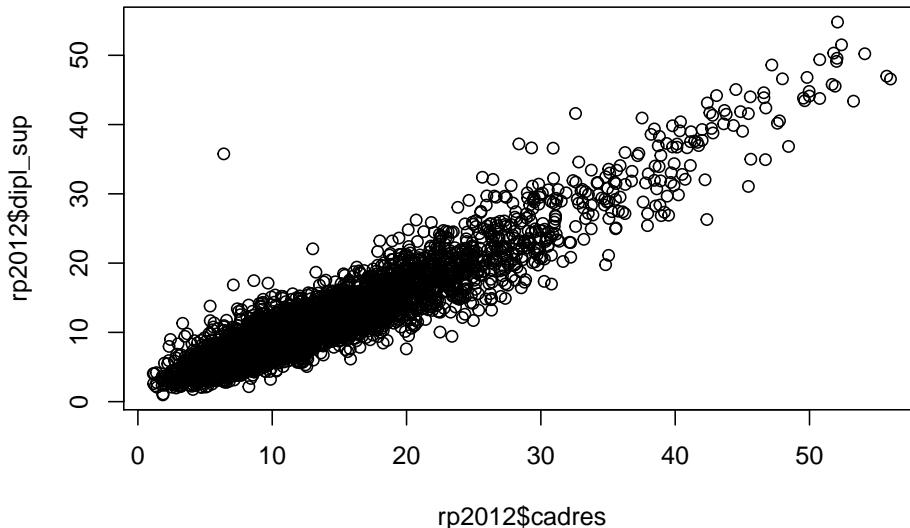


Une représentation graphique est l'idéal pour visualiser l'existence d'un lien entre les deux variables. Voici quelques exemples d'interprétation :

Dépendance linéaire positive**Dépendance linéaire négative****Dépendance non-linéaire monotone****Dépendance non-linéaire non monotone****Indépendance****Indépendance**

Dans ce premier graphique généré sur nos données, il semble difficile de mettre en évidence une relation de dépendance. Si par contre on croise le pourcentage de cadres et celui de diplômés du supérieur, on obtient une belle relation de dépendance linéaire.

```
plot(rp2012$cadres, rp2012$dipl_sup)
```



4.3.2 Calcul d'indicateurs

En plus d'une représentation graphique, on peut calculer certains indicateurs permettant de mesurer le degré d'association de deux variables quantitatives.

4.3.2.1 Corrélation linéaire (Pearson)

La corrélation est une mesure du lien d'association *linéaire* entre deux variables quantitatives. Sa valeur varie entre -1 et 1. Si la corrélation vaut -1, il s'agit d'une association linéaire négative parfaite. Si elle vaut 1, il s'agit d'une association linéaire positive parfaite. Si elle vaut 0, il n'y a aucune association linéaire entre les variables.

On la calcule dans R à l'aide de la fonction `cor`.

Ainsi la corrélation entre le pourcentage de cadres et celui de diplômés du supérieur vaut :

```
cor(rp2012$cadres, rp2012$dipl_sup)
```

```
[1] 0.9371629
```

Ce qui est extrêmement fort. Il y a donc un lien linéaire et positif entre les deux variables (quand la valeur de l'une augmente, la valeur de l'autre augmente également).

À l'inverse, la corrélation entre le pourcentage de cadres et le pourcentage de propriétaires vaut :

```
cor(rp2012$cadres, rp2012$proprio)
```

```
[1] 0.1622786
```

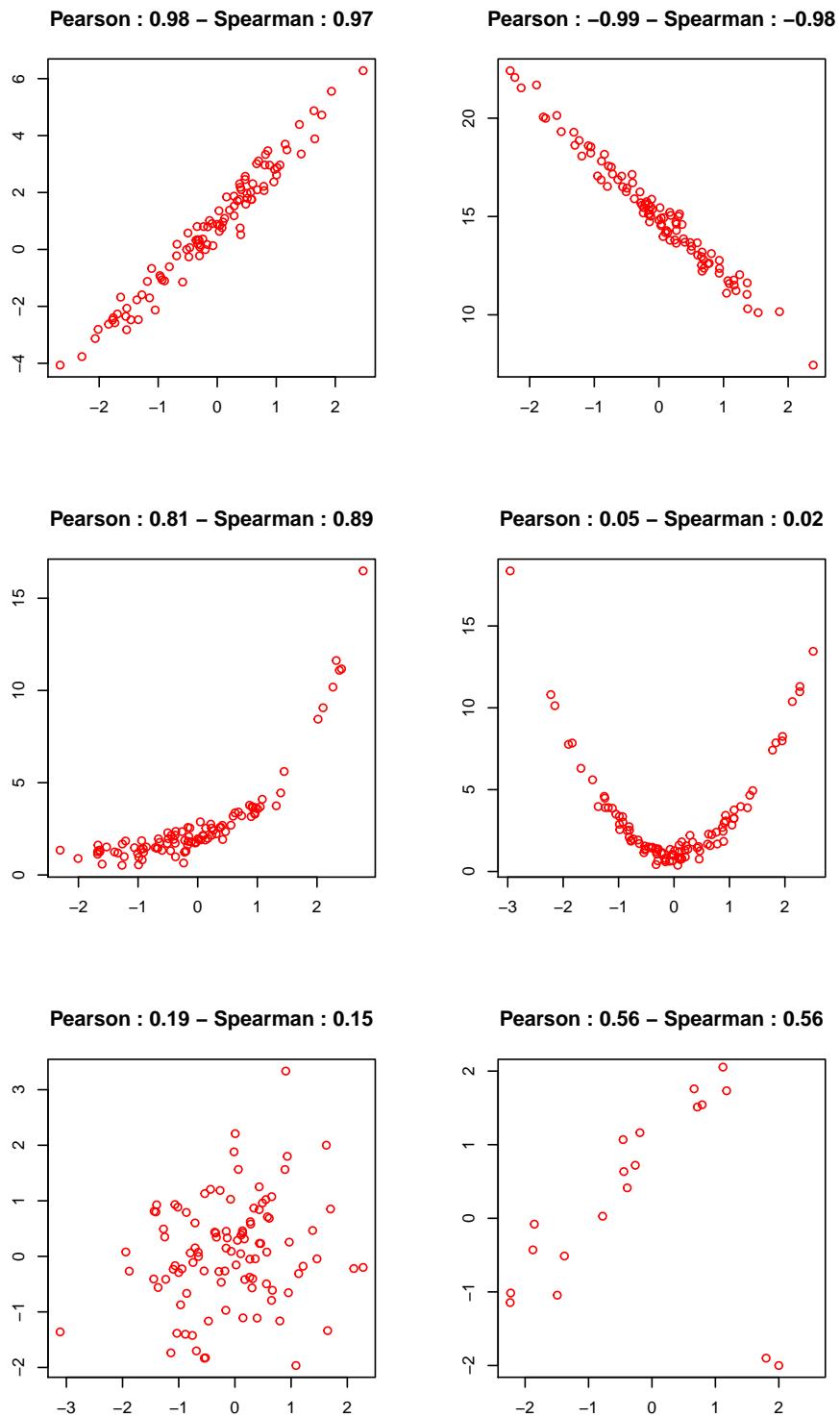
Ce qui indique, pour nos données, une absence de liaison linéaire entre les deux variables.

4.3.2.2 Corrélation des rangs (Spearman)

Le coefficient de corrélation de Pearson ci-dessus fait une hypothèse forte sur les données : elles doivent être liées par une association linéaire. Quand ça n'est pas le cas mais qu'on est en présence d'une association monotone, on peut utiliser un autre coefficient, le coefficient de corrélation des rangs de Spearman.

Plutôt que de se baser sur les valeurs des variables, cette corrélation va se baser sur leurs rangs, c'est-à-dire sur leur position parmi les différentes valeurs prises par les variables.

Ainsi, si la valeur la plus basse de la première variable est associée à la valeur la plus basse de la deuxième, et ainsi de suite jusqu'à la valeur la plus haute, on obtiendra une corrélation de 1. Si la valeur la plus forte de la première variable est associée à la valeur la plus faible de la seconde, et ainsi de suite, et que la valeur la plus faible de la première est associée à la plus forte de la deuxième, on obtiendra une corrélation de -1. Si les rangs sont "mélangés", sans rapports entre eux, on obtiendra une corrélation autour de 0.



La corrélation des rangs a aussi pour avantage d'être moins sensibles aux valeurs extrêmes ou aux points isolés. On dit qu'elle est plus "robuste".

Pour calculer une corrélation de Spearman, on utilise la fonction `cor` mais avec l'argument `method = "spearman"` :

```
cor(rp2012$cadres, rp2012$dipl_sup, method = "spearman")
```

```
[1] 0.9036273
```

4.3.3 Régression linéaire

Quand on est en présence d'une association linéaire entre deux variables, on peut vouloir faire la régression linéaire d'une des variables sur l'autres.

Une régression linéaire simple se fait à l'aide de la fonction `lm` :

```
lm(rp2012$cadres ~ rp2012$dipl_sup)
```

```
Call:
lm(formula = rp2012$cadres ~ rp2012$dipl_sup)
```

```
Coefficients:
```

(Intercept)	rp2012\$dipl_sup
0.9217	1.0816



On retrouve avec `lm` la syntaxe "formule" déjà rencontrée avec `boxplot`. Elle permet ici de spécifier des modèles de régression : la variable dépendante se place à gauche du `~`, et la variable indépendante à droite. Si on souhaite faire une régression multiple avec plusieurs variables indépendantes, on aura une formule du type `dep ~ indep1 + indep2`. Il est également possible de spécifier des termes plus complexes, des interactions, etc.

`lm` nous renvoie par défaut les coefficients de la droite de régression :

- l'ordonnée à l'origine (`Intercept`) vaut 0.92
- le coefficient associé à `dip1_sup` vaut 1.08

Pour des résultats plus détaillés, on peut stocker le résultat de la régression dans un objet et utiliser la fonction `summary` :

```
reg <- lm(rp2012$cadres ~ rp2012$dipl_sup)
summary(reg)
```

```
Call:
lm(formula = rp2012$cadres ~ rp2012$dipl_sup)

Residuals:
    Min      1Q  Median      3Q     Max 
-33.218 -1.606 -0.172  1.491 13.001 

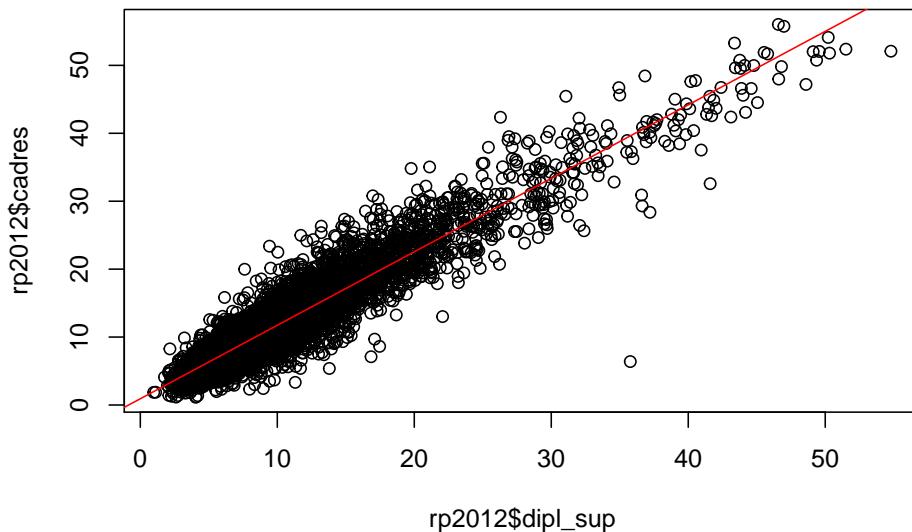
Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.921661  0.071814 12.83   <2e-16 ***
rp2012$dipl_sup 1.081636  0.005601 193.10  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.701 on 5168 degrees of freedom
Multiple R-squared:  0.8783,    Adjusted R-squared:  0.8783 
F-statistic: 3.729e+04 on 1 and 5168 DF,  p-value: < 2.2e-16
```

Ces résultats montrent notamment que les coefficients sont significativement différents de 0. La part de cadres augmente donc bien avec celle de diplômés du supérieur.

On peut enfin représenter la droite de régression sur notre nuage de points à l'aide de la fonction `abline` :

```
plot(rp2012$dipl_sup, rp2012$cadres)
abline(reg, col="red")
```



4.4 Exercices

Exercice 1

Dans le jeu de données `hdv2003`, faire le tableau croisé entre la catégorie socio-professionnelle (variable `qualif`) et le fait de croire ou non en l'existence des classes sociales (variable `c1so`). Identifier la variable indépendante et la variable dépendante, et calculer les pourcentages ligne ou colonne. Interpréter le résultat.

Faire un test du χ^2 . Peut-on rejeter l'hypothèse d'indépendance ?

Représenter ce tableau croisé sous la forme d'un `mosaicplot` en colorant les cases selon les résidus du test du χ^2 .

Exercice 2

Toujours sur le jeu de données `hdv2003`, faire le boxplot qui croise le nombre d'heures passées devant la télévision (variable `heures.tv`) avec le statut d'occupation (variable `occup`).

Calculer la durée moyenne devant la télévision en fonction du statut d'occupation à l'aide de `tapply`.

Exercice 3

Sur le jeu de données `rp2012`, représenter le nuage de points croisant le pourcentage de personnes sans diplôme (variable `dipl_aucun`) et le pourcentage de propriétaires (variable `proprio`).

Calculer le coefficient de corrélation linéaire correspondant.

Chapitre 5

Organiser ses scripts

On l'a vu, le script est l'élément central de toute analyse dans R. C'est lui qui contient l'ensemble des opérations constitutives d'une analyse, dans leur ordre d'exécution : chargement des données, recodages, manipulations, analyses, exports de résultats, etc.

Une conséquence est qu'un script peut rapidement devenir très long, et on peut finir par s'y perdre. Il est donc nécessaire d'organiser son travail pour pouvoir se retrouver facilement parmi les différentes étapes d'un projet d'analyse.

5.1 Les projets dans RStudio

La notion de projet est une fonctionnalité très pratique de RStudio, qui permet d'organiser son travail et de faciliter l'accès à l'ensemble des fichiers constitutifs d'une analyse (données, scripts, documentation, etc.).

En pratique, un projet est un dossier que vous avez créé où bon vous semble sur votre disque dur, et dans lequel vous regrouperez tous les fichiers en question. Utiliser des projets procure plusieurs avantages :

- RStudio lance automatiquement R dans le dossier du projet et facilite ainsi grandement l'accès aux fichiers de données à importer (plus besoin de taper le chemin d'accès complet). De même, si vous déplacez votre dossier sur votre disque, le projet continuera à fonctionner.
- L'onglet *Files* de la zone en bas à droite de l'interface de RStudio vous permet de naviguer facilement dans les fichiers de votre projet.
- Vous pouvez très facilement passer d'un projet à l'autre si vous travaillez sur plusieurs jeux de données en parallèle.

Pour créer un projet, il faut aller dans le menu *File* puis sélectionner *New project*.

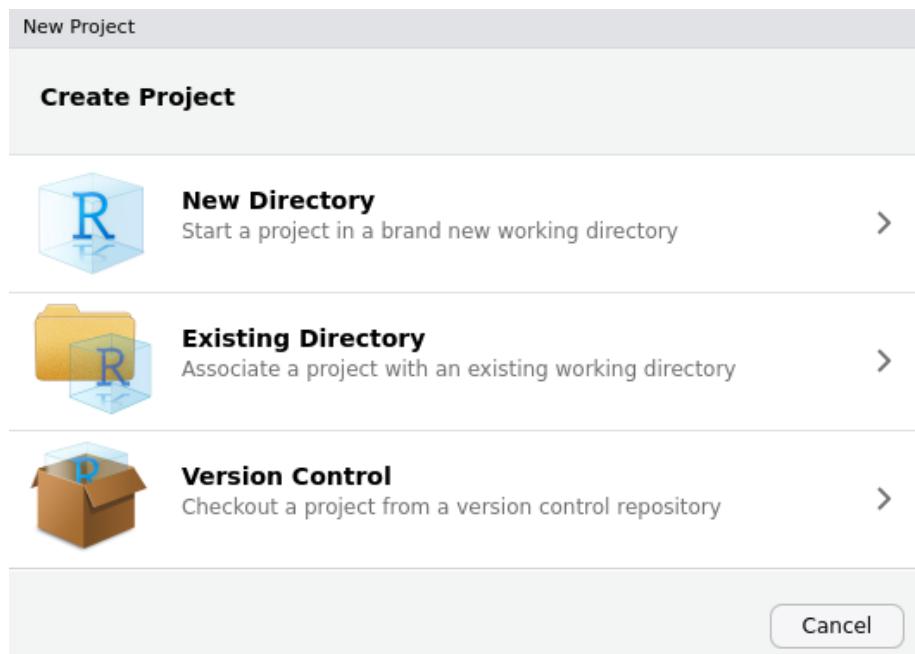


Figure 5.1: Création d'un nouveau projet

Selon que le dossier du projet existe déjà ou pas, on choisira *Existing directory* ou *New directory*. L'étape d'après consiste à créer ou sélectionner le dossier, puis on n'a plus qu'à cliquer sur *Create project*.

À la création du projet, et chaque fois que vous l'ouvrirez, une nouvelle session R est lancée dans la fenêtre *Console* avec le dossier du projet comme répertoire de travail, et l'onglet *Files* affiche les fichiers contenus dans ce dossier.

Une fois le projet créé, son nom est affiché dans un petit menu déroulant en haut à droite de l'interface de RStudio (menu qui permet de passer facilement d'un projet à un autre).

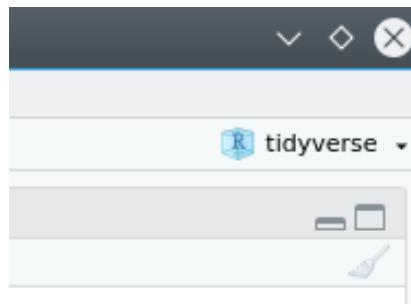


Figure 5.2: Menu projets



Si vous ne retrouvez pas le nom du projet dans ce menu, vous pouvez l'ouvrir en sélectionnant *File* puis *Open Project...* et en allant sélectionner le fichier `.Rproj` qui se trouve dans le dossier du projet à ouvrir.

5.2 Créer des sections dans un script

Lorsqu'un script est long, RStudio permet de créer des "sections" facilitant la navigation.

Pour créer une section, il suffit de faire suivre une ligne de commentaires par plusieurs tirets -, comme ceci :

```
## Titre de la section -----
```

Le nombre de tirets n'a pas d'importance, il doit juste y'en avoir plus de quatre. RStudio affiche alors dans la marge de gauche du script un petit triangle noir qui permet de replier ou déplier le contenu de la section :

```
5
6 ▾ ## Chargement des données -----
7
8 data(hdv2003)
9
10 ▾ ## Analyse -----
11
12 hist(hdv2003$age)
```

Figure 5.3: Section de script dépliée

```

5
6 #> ## Chargement des données
10 ## Analyse -----
11
12 hist(hdv2003$age)

```

Figure 5.4: Section de script repliée

De plus, en cliquant sur l’icône *Show document outline* (la plus à droite de la barre d’outils de la fenêtre du script), ou en utilisant le raccourci clavier **Ctrl+Maj+0**, RStudio affiche une “table des matières” automatiquement mise à jour qui liste les sections existantes et permet de naviguer facilement dans le script :



Figure 5.5: Liste dynamique des sections

5.3 Répartir son travail entre plusieurs scripts

Si le script devient très long, les sections peuvent ne plus être suffisantes. De plus, il est souvent intéressant d’isoler certaines parties d’un script, par exemple pour pouvoir les mutualiser. On peut alors répartir les étapes d’une analyse entre plusieurs scripts.

Un exemple courant concerne les recodages et la manipulation des données. Il est fréquent, au cours d’une analyse, de calculer de nouvelles variables, recoder des variables qualitatives existantes, etc. Il peut alors être intéressant de regrouper tous ces recodages dans un script à part (nommé, par exemple, `recodages.R`). Ce fichier contient alors l’ensemble des recodages “validés”, ceux qu’on a testé et qu’on sait vouloir conserver.

Pour exécuter ces recodages, on peut évidemment ouvrir le script `recodages.R` dans RStudio et lancer l’ensemble du code qu’il contient. Mais une méthode

plus pratique est d'utiliser la fonction `source` : celle-ci prend en paramètre un nom de fichier .R, et quand on l'exécute elle va exécuter l'ensemble du code contenu dans ce fichier.

Ainsi, un début de script `analyse.R` pourra ressembler à ceci :

```
# Analyse des données Histoire de vie 2003

# Chargement des extensions et des données -----
library(questionr)

data(hdv2003)
source("recodages.R")

# Analyse de l'âge -----
hist(hdv2003$age)

(...)
```

L'avantage principal est qu'on peut à tout moment revenir à nos données d'origine et aux recodages “validés” simplement en exécutant les deux lignes :

```
data(hdv2003)
source("recodages.R")
```

L'autre avantage est qu'on peut répartir nos analyses entre différents scripts, et conserver ces deux lignes en haut de chaque script, ce qui permet de “mutualiser” les recodages validés. On pourrait ainsi créer un deuxième script `analyse_qualif.R` qui pourrait ressembler à ceci :

```
# Analyse des données Histoire de vie 2003 - Qualifications

# Chargement des extensions et des données -----
library(questionr)

data(hdv2003)
source("recodages.R")

# Analyse des qualifications -----
```

```
freq(hdv2003$qualif)
(...)
```

On peut évidemment répartir les recodages entre plusieurs fichiers et faire appel à autant de **source** que l'on souhaite.



Cette organisation recalcule l'ensemble des recodages à chaque début de script. C'est intéressant et pratique pour des données de taille raisonnable, mais pour des fichiers plus volumineux les calculs peuvent être trop longs. Dans ce cas il est préférable de créer des scripts dédiés qui chargent les données source, effectuent calculs et recodages, et enregistrent les données résultantes dans un fichier de données (voir le chapitre sur l'import/export de données). Et c'est ce fichier résultat qui sera chargé par les scripts d'analyse.

Enfin, pour des projets un peu complexes, on pourra se référer à l'extension [ProjectTemplate](#), qui propose une architecture de dossiers spécifique et des outils associés.

5.4 Désactiver la sauvegarde de l'espace de travail

Vous avez sans doute remarqué qu'au moment de quitter RStudio, une boîte de dialogue semblable à celle-ci s'affiche parfois :

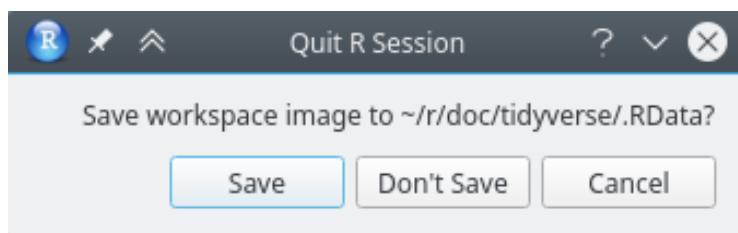


Figure 5.6: Dialogue d'enregistrement de l'espace de travail

Et il est bien difficile de comprendre de quoi cela parle.

Il s'agit en fait d'une fonctionnalité de R lui-même qui propose d'enregistrer notre espace de travail (*workspace*), c'est-à-dire l'ensemble des objets qui existent actuellement dans notre environnement, dans un fichier nommé **.RData**. La prochaine fois que R est lancé dans le même dossier (par exemple à la réouverture

du projet), s'il trouve un fichier `.RData` il va le lire automatiquement et restaurer l'ensemble des objets dans l'état où ils étaient.

Ceci peut sembler pratique, mais c'est en fait une mauvaise idée, pour deux raisons :

- on peut se retrouver avec des objets dont on ne sait plus d'où ils viennent et comment ils ont été calculés
- cette manière de faire casse la logique principale de R, qui est que c'est le script qui est central, et que c'est lui qui retrace toutes les étapes de notre analyse et permet de les reproduire

Il est donc *fortement recommandé*, juste après l'installation de RStudio, de désactiver cette fonctionnalité. Pour cela, aller dans le menu *Tools*, puis *Global Options*, et s'assurer que :

- la case *Restore .RData into workspace at startup* est *décochée*
- le champ *Save workspace to .RData on exit* vaut *Never*

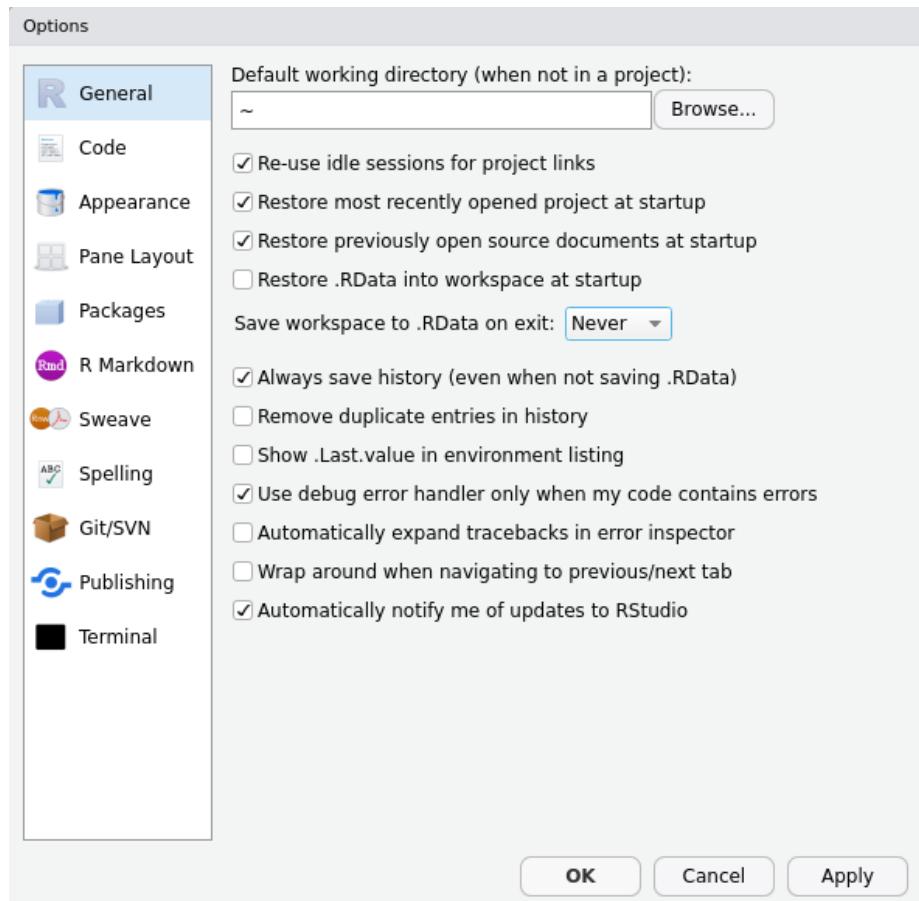


Figure 5.7: Options d'enregistrement de l'espace de travail

Partie II

Introduction au tidyverse

Chapitre 6

Le tidyverse

6.1 Extensions

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 d’extensions conçues pour travailler ensemble et basées sur une philosophie commune.

Elles abordent un très grand nombre d’opérations courantes dans R (la liste n’est pas exhaustive) :

- visualisation
- manipulation des tableaux de données
- import/export de données
- manipulation de variables
- extraction de données du Web
- programmation

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](#).

6.2 Installation

tidyverse est également le nom d’une extension qu’on peut installer de manière classique, soit via le bouton *Install* de l’onglet *Packages* de RStudio, soit en utilisant la commande :

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

Cette commande va en fait installer plusieurs extensions qui constituent le “coeur” du *tidyverse*, à savoir :

- **ggplot2** (visualisation)
- **dplyr** (manipulation des données)
- **tidyr** (remise en forme des données)
- **purrr** (programmation)
- **readr** (importation de données)
- **tibble** (tableaux de données)
- **forcats** (variables qualitatives)
- **stringr** (chaînes de caractères)



Figure 6.1: Packages de l’extension tidyverse

De la même manière, charger l’extension avec :

```
library(tidyverse)
```

Chargera l’ensemble des extensions précédentes.

Il existe d’autres extensions qui font partie du *tidyverse* mais qui doivent être chargées explicitement, comme par exemple **readxl** (pour l’importation de données depuis des fichiers Excel).

La liste complète des extensions se trouve sur [le site officiel du tidyverse](#).

Ce document est basé sur les versions d’extension suivantes :

```
ggplot2 3.1.1      purrr   0.3.2
tibble   2.1.1      dplyr    0.8.1
tidyrm   0.8.3      stringr 1.4.0
readr    1.3.1      forcats 0.4.0
```

6.3 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

On verra plus précisément dans la section [12](#) comment définir et rendre des données *tidy* avec l'extension `tidyrm`.

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

6.4 tibbles

Une autre particularité du *tidyverse* est que ces extensions travaillent avec des tableaux de données au format *tibble*, qui est une évolution plus moderne du classique *data frame* du 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 objet de classe `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...) ¹
- s'affichent plus intelligemment que les *data frames* : seules les premières lignes sont affichées, ainsi que quelques informations supplémentaires utiles (dimensions, types des colonnes...)

¹Quand on veut utiliser des noms de ce type, on doit les entourer avec des *backticks* (`)

- ne font pas de *partial matching* sur les noms de colonnes ²
- affichent un avertissement si on essaie d'accéder à une colonne qui n'existe pas

Pour autant, les tibbles restent compatibles avec les *data frames*. On peut ainsi facilement convertir un *data frame* en tibble avec `as_tibble` :

```
as_tibble(mtcars)
```

```
# A tibble: 32 x 11
  mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <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
# ... with 22 more rows
```

Si le *data frame* d'origine a des *rownames*, on peut d'abord les convertir en colonnes avec `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
  <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Mazda ~  21     6   160   110  3.9   2.62  16.5    0     1     4
2 Mazda ~  21     6   160   110  3.9   2.88  17.0    0     1     4
3 Datsun~  22.8   4   108   93   3.85  2.32  18.6    1     1     4
4 Hornet~  21.4   6   258   110  3.08  3.22  19.4    1     0     3
5 Hornet~  18.7   8   360   175  3.15  3.44  17.0    0     0     3
6 Valiant  18.1   6   225   105  2.76  3.46  20.2    1     0     3
```

²Dans R de base, si une table `d` contient une colonne `qualif`, `d$qual` retournera cette colonne.

```

7 Duster~ 14.3     8 360     245 3.21 3.57 15.8     0     0     3
8 Merc 2~ 24.4     4 147.    62 3.69 3.19 20       1     0     4
9 Merc 2~ 22.8     4 141.    95 3.92 3.15 22.9     1     0     4
10 Merc 2~ 19.2    6 168.   123 3.92 3.44 18.3     1     0     4
# ... with 22 more rows, and 1 more variable: carb <dbl>

```

À l'inverse, on peut à tout moment convertir un tibble en *data frame* avec `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
	[reached 'max' / getOption("max.print") -- omitted 16 rows]											

Là encore, on peut convertir la colonne `rowname` en “vrais” *rownames* avec `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
[ reached 'max' / getOption("max.print") -- omitted 14 rows ]
```



Les deux fonctions `column_to_rrownames` et `rrownames_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.

Chapitre 7

Importer et exporter des données

R n'est pas prévu pour la saisie de données, mais il bénéficie de nombreuses fonctions et packages permettant l'import de données depuis un grand nombre de formats. Seuls les plus courants seront abordés ici.

Il est très vivement conseillé de travailler avec les projets de RStudio pour faciliter l'accès aux fichiers et pouvoir regrouper l'ensemble des éléments d'une analyse dans un dossier (voir partie 5.1).



Les projets permettent notamment de ne pas avoir à spécifier un chemin complet vers un fichier (sous Windows, quelque chose du genre C:\\\\Users\\\\toto\\\\Documents\\\\quanti\\\\projet\\\\data\\\\donnees.xls) mais un chemin relatif au dossier du projet (juste donnees.xls si le fichier se trouve à la racine du projet, data/donnees.xls s'il se trouve dans un sous-dossier data, etc.)

7.1 Import de fichiers textes

L'extension `readr`, qui fait partie du *tidyverse*, permet l'importation de fichiers texte, notamment au format CSV (*Comma separated values*), format standard pour l'échange de données tabulaires entre logiciels.

Cette extension fait partie du “coeur” du *tidyverse*, elle est donc automatiquement chargée avec :

```
library(tidyverse)
```

Si votre fichier CSV suit un format CSV standard (c'est le cas s'il a été exporté depuis LibreOffice par exemple), avec des champs séparés par des virgules, vous pouvez utiliser la fonction `read_csv` en lui passant en argument le nom du fichier :

```
d <- read_csv("fichier.csv")
```

Si votre fichier vient d'Excel, avec des valeurs séparées par des points virgule, utilisez la fonction `read_csv2` :

```
d <- read_csv2("fichier.csv")
```

Dans la même famille de fonction, `read_tsv` permet d'importer des fichiers dont les valeurs sont séparées par des tabulations, et `read_delim` des fichiers délimités par un séparateur indiqué en argument.

Chaque fonction dispose de plusieurs arguments, parmi lesquels :

- `col_names` indique si la première ligne contient le nom des colonnes (TRUE par défaut)
- `col_types` permet de spécifier manuellement le type des colonnes si `readr` ne les identifie pas correctement
- `na` est un vecteur de chaînes de caractères indiquant les valeurs devant être considérées comme manquantes. Ce vecteur vaut `c("", "NA")` par défaut

Il peut arriver, notamment sous Windows, que l'encodage des caractères accentués ne soit pas correct au moment de l'importation. On peut alors spécifier manuellement l'encodage du fichier importé à l'aide de l'option `locale`. Par exemple, si l'on est sous Mac ou Linux et que le fichier a été créé sous Windows, il est possible qu'il soit encodé au format iso-8859-1. On peut alors l'importer avec :

```
d <- read_csv("fichier.csv", locale = locale(encoding = "ISO-8859-1"))
```

À l'inverse, si vous importez un fichier sous Windows et que les accents ne sont pas affichés correctement, il est sans doute encodé en UTF-8 :

```
d <- read_csv("fichier.csv", locale = locale(encoding = "UTF-8"))
```

Pour plus d'informations sur ces fonctions, voir [le site de l'extension readr](#).



À noter que si vous souhaitez importer des fichiers textes très volumineux le plus rapidement possible, la fonction `fread` de l'extension `data.table` est plus rapide que `read_csv`.

7.1.1 Interface interactive d'import de fichiers

RStudio propose une interface permettant d'importer un fichier de données de manière interactive. Pour y accéder, dans l'onglet *Environment*, cliquez sur le bouton *Import Dataset* :

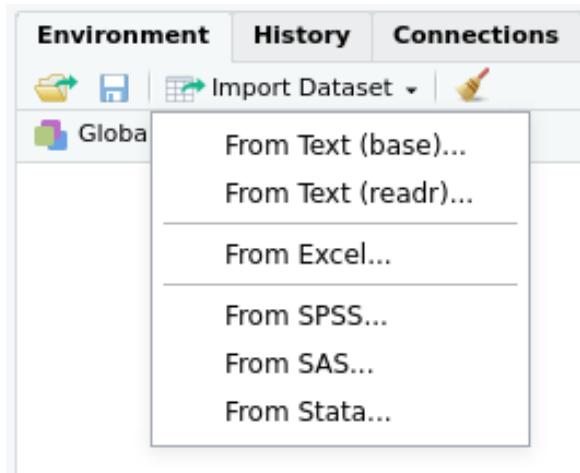


Figure 7.1: Menu *Import Dataset*

Sélectionnez *From Text (readr)...*. Une nouvelle fenêtre s'affiche :

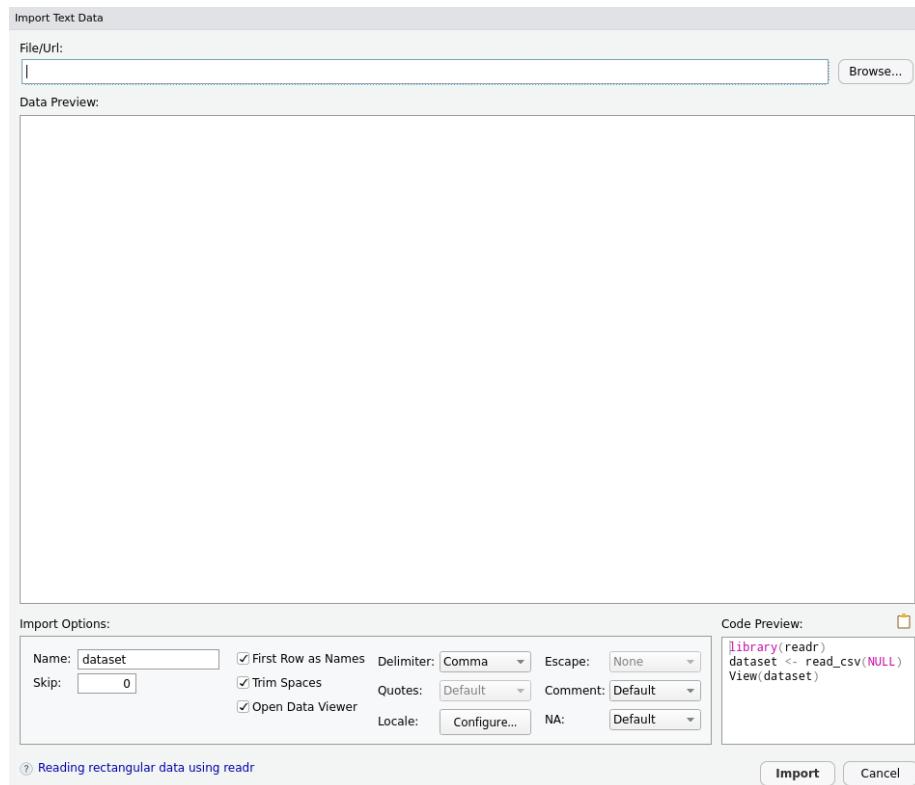


Figure 7.2: Dialogue d'importation

Il vous suffit d'indiquer le fichier à importer dans le champ *File/URL* tout en haut (vous pouvez même indiquer un lien vers un fichier distant via HTTP). Un aperçu s'ouvre dans la partie *Data Preview* et vous permet de vérifier si l'import est correct :

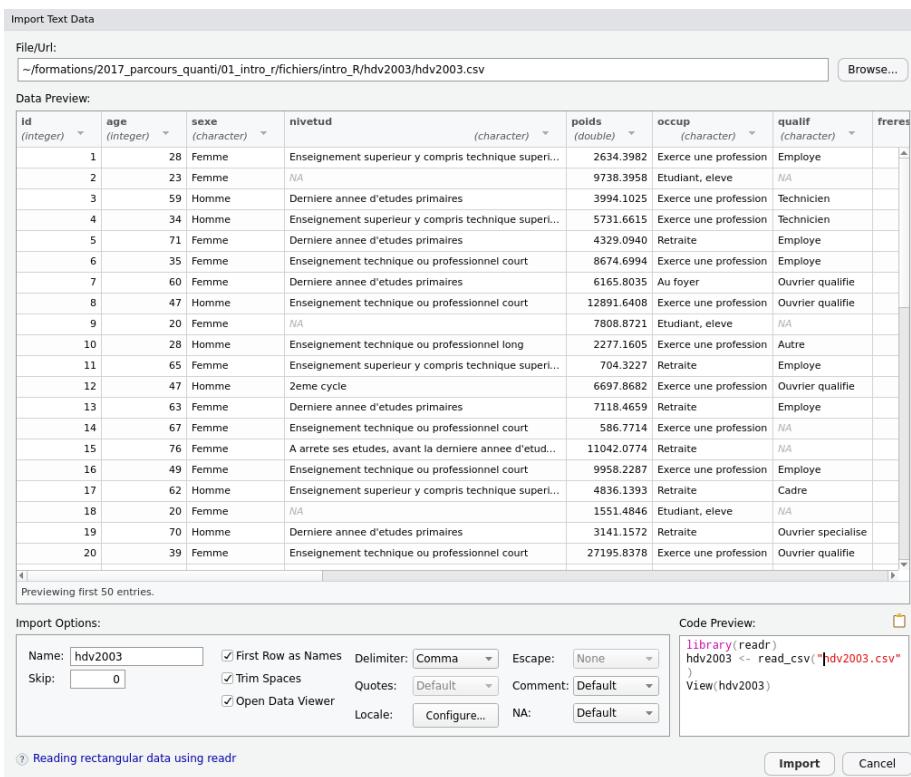


Figure 7.3: Exemple de dialogue d'importation

Vous pouvez modifier les options d'importation, changer le type des colonnes, etc. et l'aperçu se met à jour. De même, le code correspondant à l'importation du fichier avec les options sélectionnées est affiché dans la partie *Code Preview*.



Important : une fois que l'import semble correct, ne cliquez pas sur le bouton *Import*. À la place, sélectionnez le code généré et copiez-le (ou cliquez sur l'icône en forme de presse papier) et choisissez *Cancel*. Ensuite, collez le code dans votre script et exécutez-le (vous pouvez supprimer la ligne commençant par *View*).

Cette manière de faire permet “d’automatiser” l’importation des données, puisqu’à la prochaine ouverture du script vous aurez juste à exécuter le code en question, sans repasser par l’interface d’import.

7.2 Import depuis un fichier Excel

L’extension `readxl`, qui fait également partie du *tidyverse*, permet d’importer des données directement depuis un fichier au format `xls`ou `xlsx`.

Elle ne fait pas partie du “coeur” du *tidyverse*, il faut donc la charger explicitement avec :

```
library(readxl)
```

On peut alors utiliser la fonction `read_excel` en lui spécifiant le nom du fichier :

```
d <- read_excel("fichier.xls")
```

Il est possible de spécifier la feuille et la plage de cellules que l’on souhaite importer avec les arguments `sheet` et `range` :

```
d <- read_excel("fichier.xls", sheet = "Feuille2", range = "C1:F124")
```

Comme pour l’import de fichiers texte, une interface interactive d’import de fichiers Excel est disponible dans RStudio dans l’onglet *Environment*. Pour y accéder, cliquez sur *Import Dataset* puis *From Excel....*

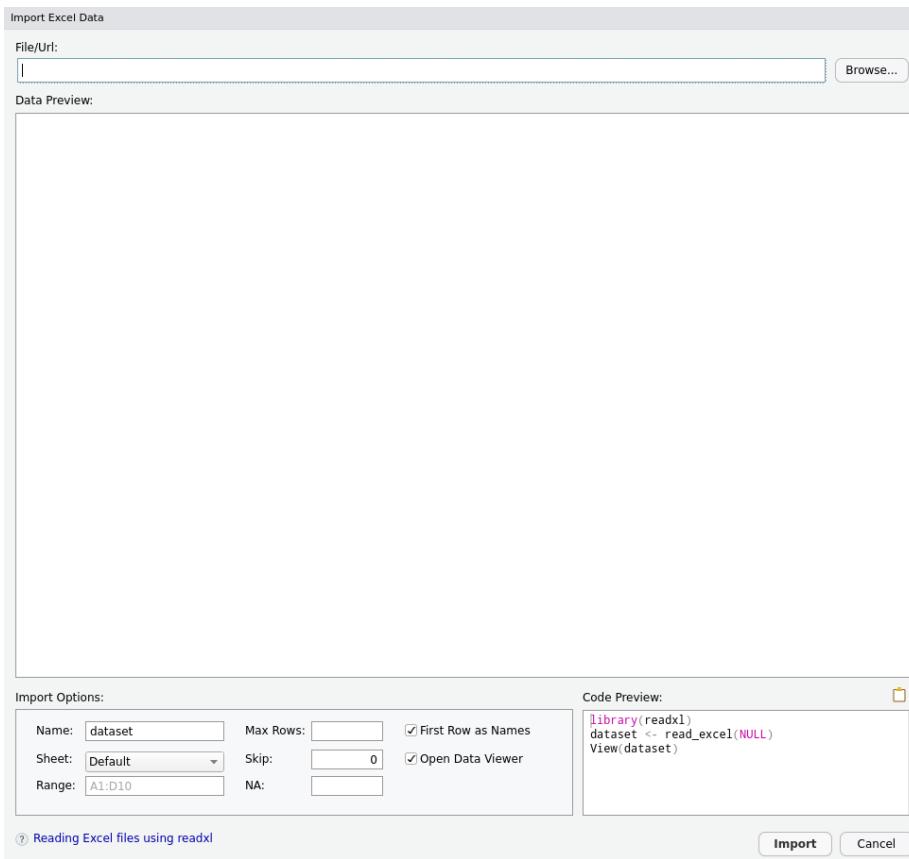


Figure 7.4: Dialogue d'importation d'un fichier Excel

Spécifiez le chemin ou l'URL du fichier dans le premier champ, vérifiez l'import dans la partie *Data Preview*, modifiez si besoin les options d'importation, copiez le code d'importation généré dans la partie *Code Preview* et collez le dans votre script.

Pour plus d'informations, voir [le site de l'extension `readxl`](#).

7.3 Import de fichiers SAS, SPSS et Stata

L'import de fichiers de données au format SAS, SPSS ou Stata se fait via les fonctions de l'extension `haven`.

Celle-ci fait partie du *tidyverse*, mais doit être chargée explicitement avec :

```
library(haven)
```

- Pour les fichiers provenant de SAS, vous pouvez utiliser les fonctions `read_sas` ou `read_xpt`
- Pour les fichiers provenant de SPSS, vous pouvez utiliser `read_sav` ou `read_por`
- Pour les fichiers provenant de Stata, utilisez `read_dta`

Chaque fonction dispose de plusieurs options. Le plus simple est d'utiliser, là aussi l'interface interactive d'importation de données de RStudio : dans l'onglet *Environment*, sélectionnez *Import Dataset* puis *From SPSS*, *From SAS* ou *From Stata*. Indiquez le chemin ou l'url du fichier, réglez les options d'importation, puis copiez le code d'importation généré et collez le dans votre script.

Pour plus d'informations, voir [le site de l'extension haven](#)

7.4 Import de fichiers dBBase

Le format dBBase est encore utilisé, notamment par l'INSEE, pour la diffusion de données volumineuses.

Les fichiers au format `dbf` peuvent être importées à l'aide de la fonction `read.dbf` de l'extension `foreign`¹ :

```
library(foreign)
d <- read.dbf("fichier.dbf")
```

La fonction `read.dbf` n'admet qu'un seul argument, `as.is`. Si `as.is = FALSE` (valeur par défaut), les chaînes de caractères sont automatiquement converties en `factor` à l'importation. Si `as.is = TRUE`, elles sont conservées telles quelles.

7.5 Connexion à des bases de données

7.5.1 Interfaçage via l'extension DBI

R est capable de s'interfacer avec différents systèmes de bases de données relationnelles, dont SQLite, MS SQL Server, PostgreSQL, MariaDB, etc.

¹`foreign` est une extension installée de base avec R, vous n'avez pas besoin de l'installer, il vous suffit de la charger avec `library`

Pour illustrer rapidement l'utilisation de bases de données, on va créer une base SQLite d'exemple à l'aide du code R suivant, qui copie la table du jeu de données `mtcars` dans une base de données `bdd.sqlite` :

```
library(DBI)
library(RSQLite)
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "bdd.sqlite")
data(mtcars)
mtcars$name <- rownames(mtcars)
dbWriteTable(con, "mtcars", mtcars)
dbDisconnect(con)
```

Si on souhaite se connecter à cette base de données par la suite, on peut utiliser l'extension `DBI`, qui propose une interface générique entre R et différents systèmes de bases de données. On doit aussi avoir installé et chargé l'extension spécifique à notre base, ici `RSQLite`. On commence par ouvrir une connexion à l'aide de la fonction `dbConnect` de `DBI` :

```
library(DBI)
library(RSQLite)
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "bdd.sqlite")
```

La connexion est stockée dans un objet `con`, qu'on va utiliser à chaque fois qu'on voudra interroger la base.

On peut vérifier la liste des tables présentes et les champs de ces tables avec `dbListTables` et `dbListFields` :

```
dbListTables(con)
```

```
[1] "mtcars"
```

```
dbListFields(con, "mtcars")
```

```
[1] "mpg"   "cyl"   "disp"  "hp"    "drat"  "wt"    "qsec"  "vs"    "am"    "gear"
[11] "carb"  "name"
```

On peut également lire le contenu d'une table dans un objet de notre environnement avec `dbReadTable` :

```
cars <- dbReadTable(con, "mtcars")
```

On peut également envoyer une requête SQL directement à la base et récupérer le résultat :

```
dbGetQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	name
1	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	Datsun 710
2	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2	Merc 240D
3	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2	Merc 230
4	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1	Fiat 128
5	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2	Honda Civic
6	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1	Toyota Corolla
7	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1	Toyota Corona
8	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1	Fiat X1-9
9	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2	Porsche 914-2
10	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2	Lotus Europa
11	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2	Volvo 142E

Enfin, quand on a terminé, on peut se déconnecter à l'aide de `dbDisconnect` :

```
dbDisconnect(con)
```

Ceci n'est évidemment qu'un tout petit aperçu des fonctionnalités de DBI.

7.5.2 Utilisation de `dplyr` et `dbplyr`

L'extension `dplyr` est dédiée à la manipulation de données, elle est présentée au chapitre 10. En installant l'extension complémentaire `dbplyr`, on peut utiliser `dplyr` directement sur une connection à une base de données générée par DBI :

```
library(DBI)
library(RSQLite)
library(dplyr)
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "bdd.sqlite")
```

La fonction `tbl` notamment permet de créer un nouvel objet qui représente une table de la base de données :

```
cars_tbl <- tbl(con, "mtcars")
```



Ici l'objet `cars_tbl` n'est *pas* un tableau de données, c'est juste un objet permettant d'interroger la table de notre base de données.

On peut utiliser cet objet avec les verbes de `dplyr` :

```
cars_tbl %>%
  filter(cyl == 4) %>%
  select(name, mpg, cyl)
```

```
# Source:  lazy query [?? x 3]
# Database: sqlite 3.22.0 [/home/julien/r/doc/tidyverse/bdd.sqlite]
  name      mpg cyl
  <chr>    <dbl> <dbl>
1 Datsun 710 22.8   4
2 Merc 240D  24.4   4
3 Merc 230   22.8   4
4 Fiat 128   32.4   4
5 Honda Civic 30.4   4
6 Toyota Corolla 33.9   4
7 Toyota Corona 21.5   4
8 Fiat X1-9   27.3   4
9 Porsche 914-2 26.0   4
10 Lotus Europa 30.4   4
# ... with more rows
```

`dbplyr` s'occupe, de manière transparente, de transformer les instructions `dplyr` en requête SQL, d'interroger la base de données et de renvoyer le résultat. De plus, tout est fait pour qu'un minimum d'opérations sur la base, parfois coûteuses en temps de calcul, ne soient effectuées.



Il est possible de modifier des objets de type `tbl`, par exemple avec `mutate` :

```
cars_tbl <- cars_tbl %>% mutate(type = "voiture")
```

Dans ce cas la nouvelle colonne `type` est bien créée et on peut y accéder par la suite. Mais **cette création se fait dans une table temporaire** : elle n'existe que le temps de la connexion à la base de données. À la prochaine connexion, cette nouvelle colonne n'apparaîtra pas dans la table.

Bien souvent on utilisera une base de données quand les données sont trop volumineuses pour être gérées par un ordinateur de bureau. Mais si les données ne sont pas trop importantes, il sera toujours plus rapide de récupérer l'intégralité de la table dans notre session R pour pouvoir la manipuler comme les tableaux de données habituels. Ceci se fait grâce à la fonction `collect` de `dplyr` :

```
cars <- cars_tbl %>% collect
```

Ici, `cars` est bien un tableau de données classique, copie de la table de la base au moment du `collect`.

Et dans tous les cas, on n'oubliera pas de se déconnecter avec :

```
dbDisconnect(con)
```

7.5.3 Ressources

Pour plus d'informations, voir la [documentation très complète](#) (en anglais) proposée par RStudio.

Par ailleurs, depuis la version 1.1, RStudio facilite la connexion à certaines bases de données grâce à l'onglet *Connections*. Pour plus d'informations on pourra se référer à l'article (en anglais) [Using RStudio Connections](#).

7.6 Export de données

7.6.1 Export de tableaux de données

On peut avoir besoin d'exporter un tableau de données dans R vers un fichier dans différents formats. La plupart des fonctions d'import disposent d'un équivalent permettant l'export de données. On citera notamment :

- `write_csv`, `write_csv2`, `read_tsv` permettent d'enregistrer un *data frame* ou un tibble dans un fichier au format texte délimité
- `write_sas` permet d'exporter au format SAS
- `write_sav` permet d'exporter au format SPSS
- `write_dta` permet d'exporter au format Stata

Il n'existe par contre pas de fonctions permettant d'enregistrer directement au format `xls` ou `xlsx`. On peut dans ce cas passer par un fichier CSV.

Ces fonctions sont utiles si on souhaite diffuser des données à quelqu'un d'autre, ou entre deux logiciels.

Si vous travaillez sur des données de grandes dimensions, les formats texte peuvent être lents à exporter et importer. Dans ce cas, l'extension **feather** peut être utile : elle permet d'enregistrer un *data frame* au format feather, qui n'est pas le plus compact mais qui est extrêmement rapide à lire et écrire².

Les fonctions `read_feather` et `write_feather` permettent d'importer et exporter des tableaux de données dans ce format.

7.6.2 Sauvegarder des objets

Une autre manière de sauvegarder des données est de les enregistrer au format **RData**. Ce format propre à R est compact, rapide, et permet d'enregistrer plusieurs objets R, quel que soit leur type, dans un même fichier.

Pour enregistrer des objets, il suffit d'utiliser la fonction `save` et de lui fournir la liste des objets à sauvegarder et le nom du fichier :

```
save(d, rp2012, tab, file = "fichier.RData")
```

Pour charger des objets préalablement enregistrés, utiliser `load` :

```
load("fichier.RData")
```

Les objets `d`, `rp2012` et `tab` devraient alors apparaître dans votre environnement.



Attention, quand on utilise `load`, les objets chargés sont importés directement dans l'environnement en cours avec leur nom d'origine. Si d'autres objets du même nom existaient déjà, ils sont écrasés sans avertissement.

²**feather** est un format compatible avec Python, R et Julia. Pour plus d'informations voir <https://github.com/wesm/feather>

Chapitre 8

Visualiser avec `ggplot2`

`ggplot2` est une extension du *tidyverse* qui permet de générer des graphiques avec une syntaxe cohérente et puissante. Elle nécessite l'apprentissage d'un "mini-langage" supplémentaire, mais permet la construction de graphiques complexes de manière efficace.

Une des particularités de `ggplot2` est qu'elle part du principe que les données relatives à un graphique sont stockées dans un tableau de données (*data frame*, *tibble* ou autre).

8.1 Préparation

`ggplot2` fait partie du cœur du *tidyverse*, elle est donc chargée automatiquement avec :

```
library(tidyverse)
```

On peut également la charger explicitement avec :

```
library(ggplot2)
```

Dans ce qui suit on utilisera le jeu de données issu du recensement de la population de 2012 inclus dans l'extension `questionr` (résultats partiels concernant les communes de plus de 2000 habitants de France métropolitaine). On charge ces données et on en extrait les données de 5 départements (l'utilisation de la fonction `filter` sera expliquée dans la section 10.2.2 de la partie sur `dplyr` :

```
library(questionr)
data(rp2012)

rp <- filter(rp2012, departement %in% c("Oise", "Rhône", "Hauts-de-Seine", "Lozère", "P...")
```

8.2 Initialisation

Un graphique *ggplot2* s'initialise à l'aide de la fonction `ggplot()`. Les données représentées graphiquement sont toujours issues d'un tableau de données (*data frame* ou *tibble*), qu'on passe en argument `data` à la fonction :

```
ggplot(data = rp)
## Ou, équivalent
ggplot(rp)
```

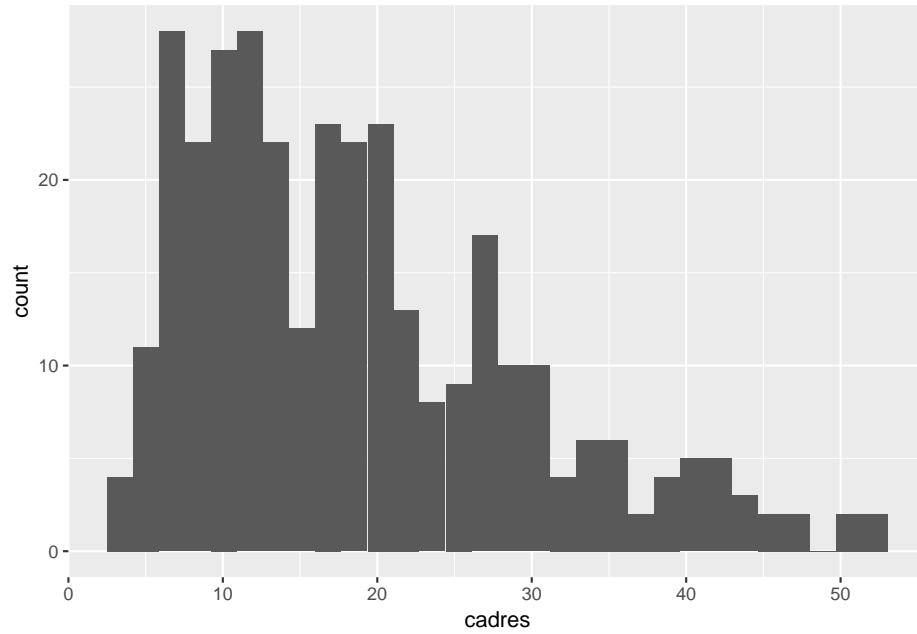
On a défini la source de données, il faut maintenant ajouter des éléments de représentation graphique. Ces éléments sont appelés des `geom`, et on les ajoute à l'objet graphique de base avec l'opérateur `+`.

Un des `geom` les plus simples est `geom_histogram`. On peut l'ajouter de la manière suivante :

```
ggplot(rp) + geom_histogram()
```

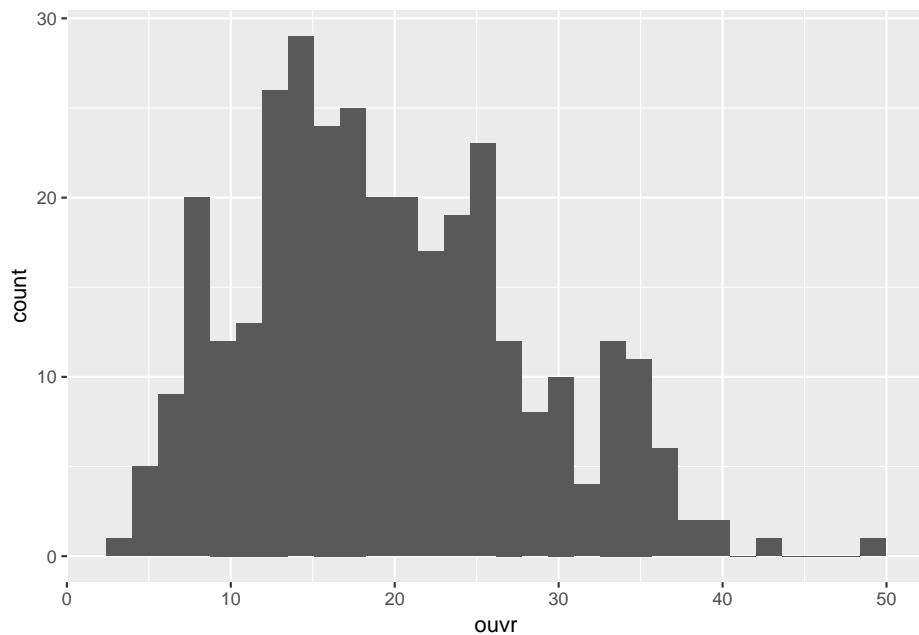
Reste à indiquer quelle donnée nous voulons représenter sous forme d'histogramme. Cela se fait à l'aide d'arguments passés via la fonction `aes()`. Ici nous avons un paramètre à renseigner, `x`, qui indique la variable à représenter sur l'axe des `x` (l'axe horizontal). Ainsi, si on souhaite représenter la distribution des communes du jeu de données selon le pourcentage de cadres dans leur population active (variable `cadres`), on pourra faire :

```
ggplot(rp) + geom_histogram(aes(x = cadres))
```



Si on veut représenter une autre variable, il suffit de changer la valeur de x :

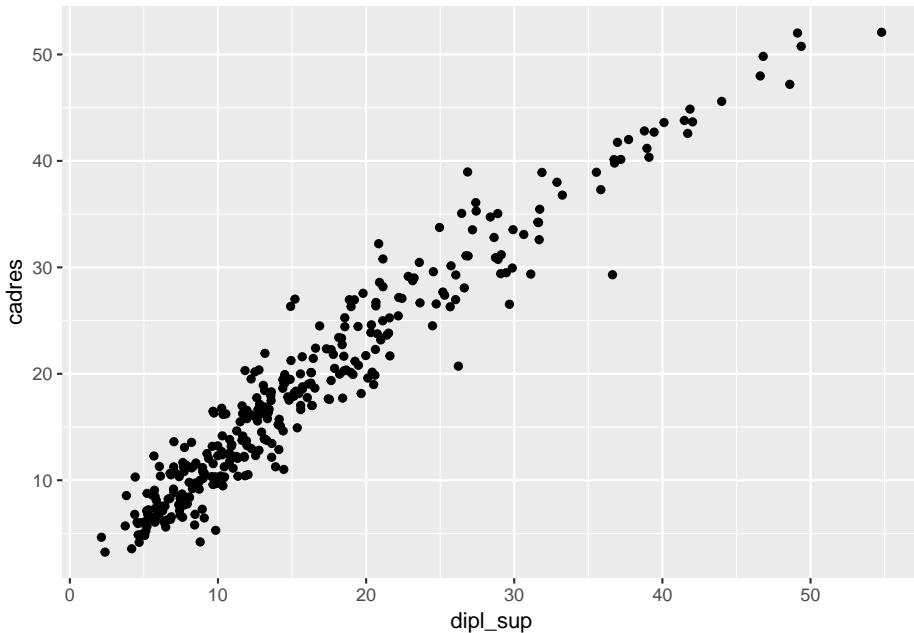
```
ggplot(rp) + geom_histogram(aes(x = ouvr))
```



Quand on spécifie une variable, inutile d'indiquer le nom du tableau de données sous la forme `rp$ouvr`, car `ggplot2` recherche automatiquement la variable dans le tableau de données indiqué avec le paramètre `data`. On peut donc se contenter de `ouvr`.

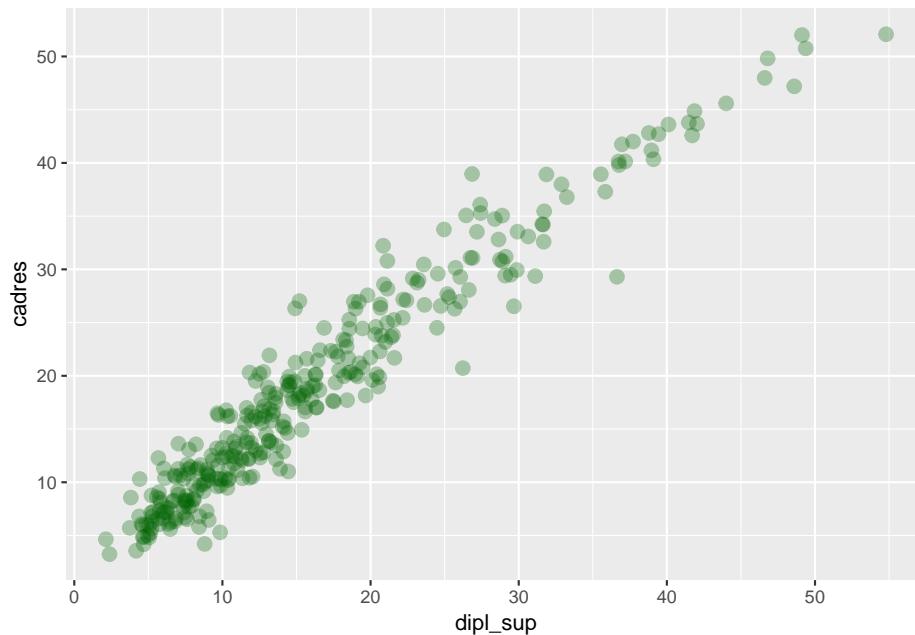
Certains `geom` prennent plusieurs paramètres. Ainsi, si on veut représenter un nuage de points, on peut le faire en ajoutant un `geom_point`. On doit alors indiquer à la fois la position en `x` (la variable sur l'axe horizontal) et en `y` (la variable sur l'axe vertical) de ces points, il faut donc passer ces deux arguments à `aes()` :

```
ggplot(rp) + geom_point(aes(x = dipl_sup, y = cadres))
```



On peut modifier certains attributs graphiques d'un `geom` en lui passant des arguments supplémentaires. Par exemple, pour un nuage de points, on peut modifier la couleur des points avec l'argument `color`, leur taille avec l'argument `size`, et leur transparence avec l'argument `alpha` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres),
             color = "darkgreen", size = 3, alpha = 0.3)
```



On notera que dans ce cas les arguments sont dans la fonction `geom` mais à l'extérieur du `aes()`. Plus d'explications sur ce point dans quelques instants.

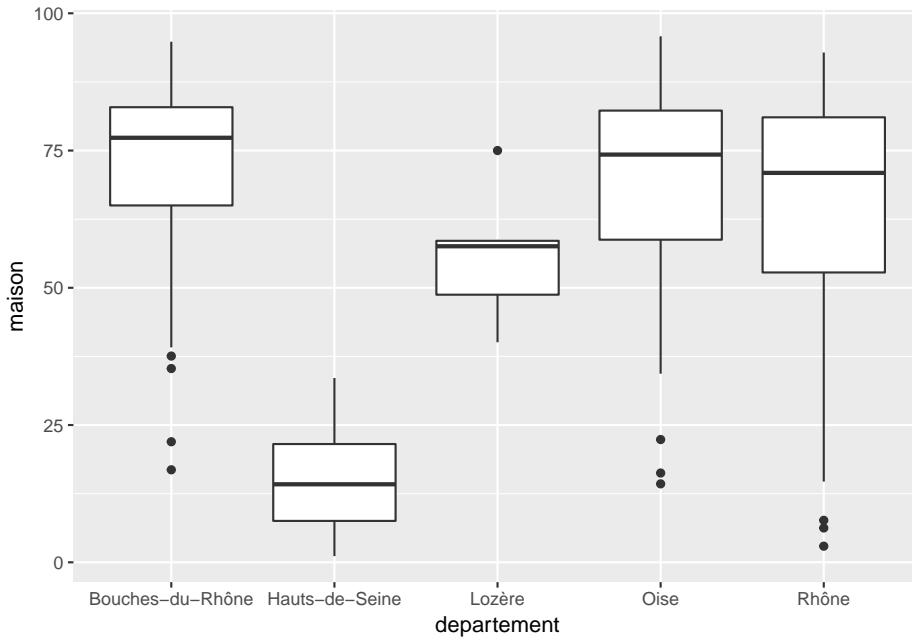
8.3 Exemples de `geom`

Il existe un grand nombre de `geom`, décrits en détail dans la [documentation officielle](#). Outre les `geom_histogram` et `geom_point` que l'on vient de voir, on pourra noter les `geom` suivants.

8.3.1 `geom_boxplot`

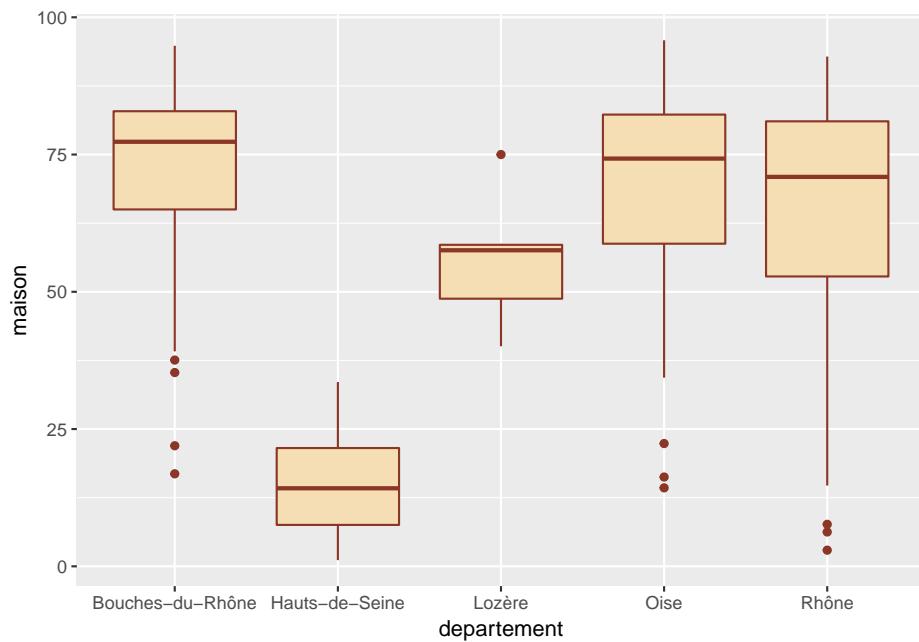
`geom_boxplot` permet de représenter des boîtes à moustaches. On lui passe en `y` la variable numérique dont on veut étudier la répartition, et en `x` la variable qualitative contenant les classes qu'on souhaite comparer. Ainsi, si on veut comparer la répartition du pourcentage de maisons en fonction du département de la commune, on pourra faire :

```
ggplot(rp) + geom_boxplot(aes(x = departement, y = maison))
```



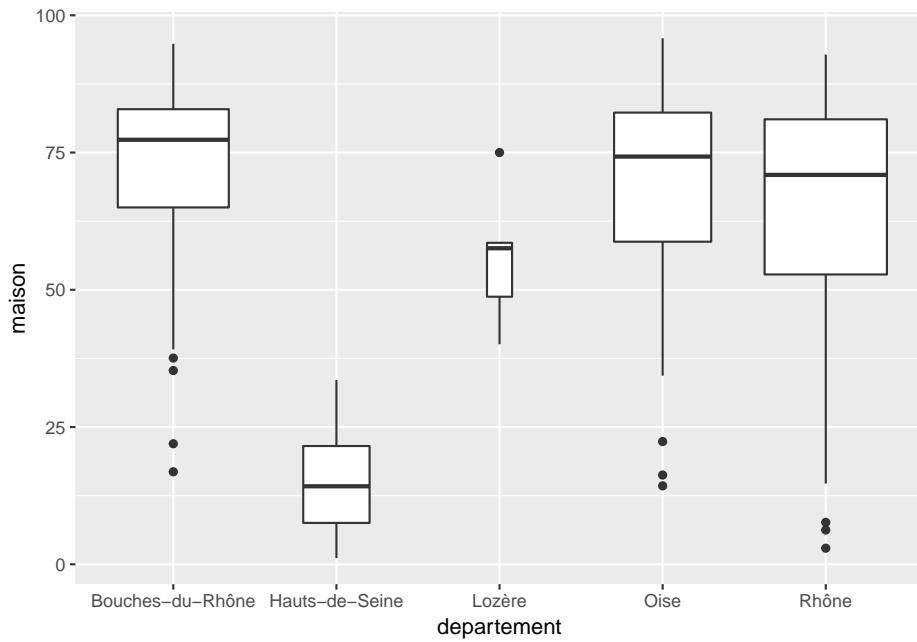
On peut personnaliser la présentation avec différents arguments supplémentaires comme `fill` ou `color` :

```
ggplot(rp) +
  geom_boxplot(aes(x = departement, y = maison), fill = "wheat", color = "tomato4")
```



Un autre argument utile, `varwidth`, permet de faire varier la largeur des boîtes en fonction des effectifs de la classe (donc, ici, en fonction du nombre de communes de chaque département) :

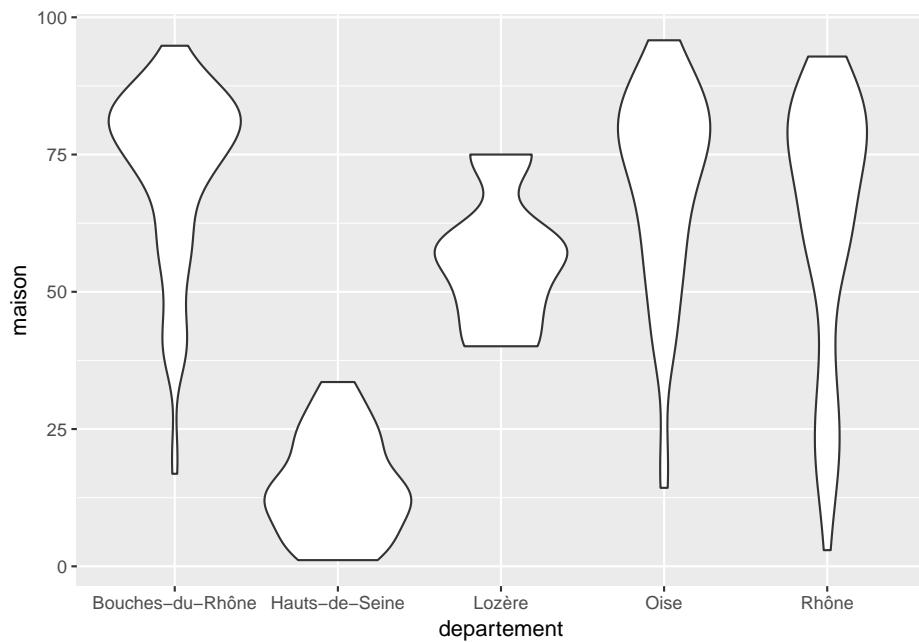
```
ggplot(rp) +
  geom_boxplot(aes(x = département, y = maison), varwidth = TRUE)
```



8.3.2 geom_violin

`geom_violin` est très semblable à `geom_boxplot`, mais utilise des graphes en violon à la place des boîtes à moustache.

```
ggplot(rp) + geom_violin(aes(x = departement, y = maison))
```



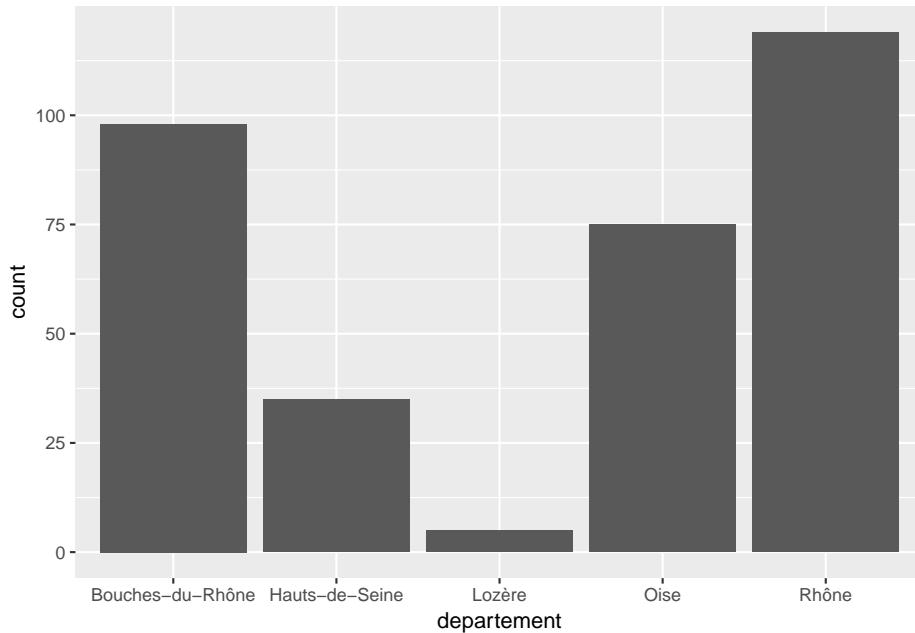
Les graphes en violon peuvent donner une lecture plus fine des différences de distribution selon les classes.

8.3.3 `geom_bar`

`geom_bar` permet de produire un graphique en bâtons (*barplot*). On lui passe en `x` la variable qualitative dont on souhaite représenter l'effectif de chaque modalité.

Par exemple, si on veut afficher le nombre de communes de notre jeu de données pour chaque département :

```
ggplot(rp) + geom_bar(aes(x = département))
```



Un cas assez fréquent mais un peu plus complexe survient quand on a déjà calculé le tri à plat de la variable à représenter. Dans ce cas on souhaite que `geom_bar` représente les effectifs sans les calculer : cela se fait en indiquant un mappage `y` pour la variable contenant les effectifs précalculés, et en ajoutant l'argument `stat = "identity"`.

Par exemple, si on a les données sous cette forme :

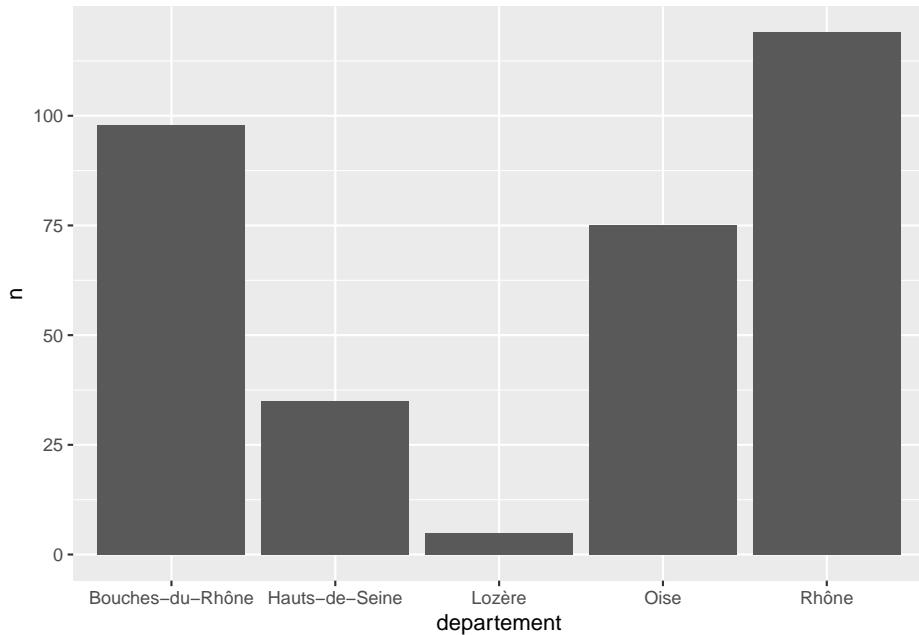
```

departement      n
1 Bouches-du-Rhône 98
2   Hauts-de-Seine 35
3       Lozère     5
4           Oise    75
5       Rhône    119

```

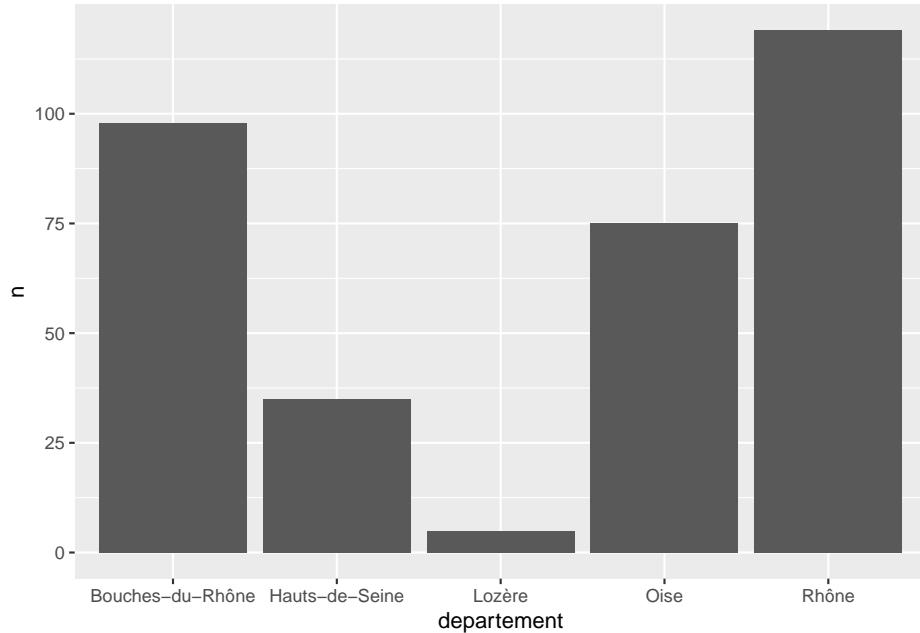
On peut obtenir le graphique souhaité ainsi :

```
ggplot(df) + geom_bar(aes(x = departement, y = n), stat = "identity")
```



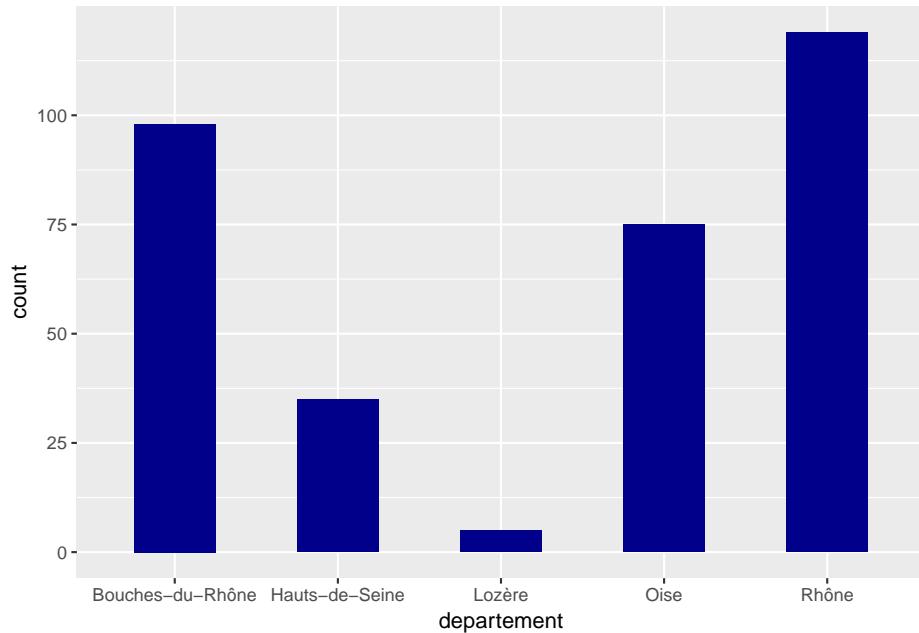
À noter qu'on peut aussi utiliser `geom_col` qui est un raccourci pour appliquer un `geom_bar` avec `stat = "identity"`. La commande précédente est donc équivalente à :

```
ggplot(df) + geom_col(aes(x = departement, y = n))
```



Là aussi, on peut modifier l'apparence du graphique avec des arguments supplémentaires comme `fill` ou `width` :

```
ggplot(rp) + geom_bar(aes(x = departement), fill = "darkblue", width = .5)
```

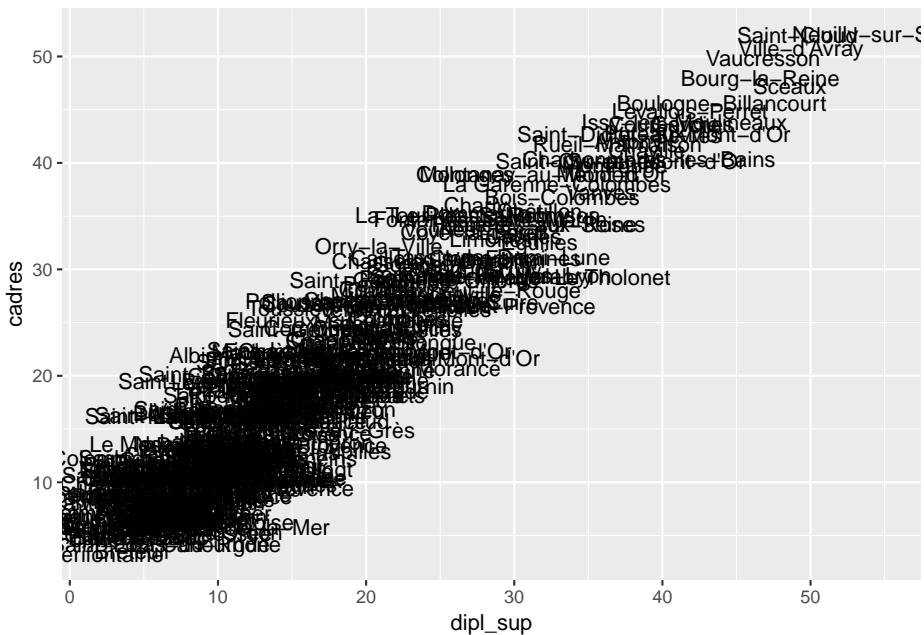


8.3.4 geom_text

`geom_text` permet d'afficher des étiquettes de texte. On doit lui passer `x` et `y` pour la position des étiquettes, et `label` pour leur texte.

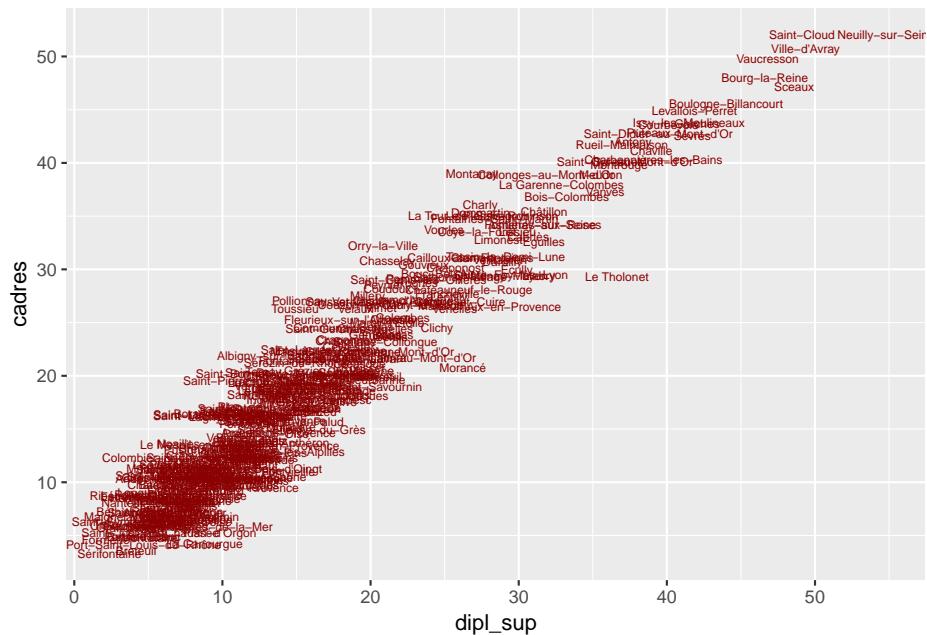
Par exemple, si on souhaite représenter le nuage croisant la part des diplômés du supérieur et la part de cadres, mais en affichant le nom de la commune (variable `commune`) plutôt qu'un simple point, on peut faire :

```
ggplot(rp) + geom_text(aes(x = dipl_sup, y = cadres, label = commune))
```



On peut personnaliser l'apparence et la position du texte avec des arguments comme `size`, `color`, etc.

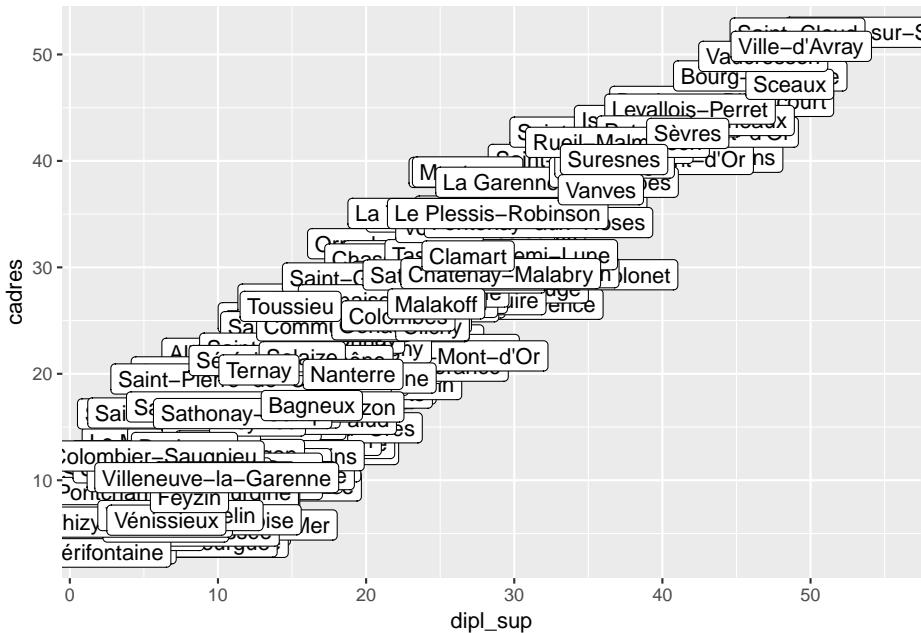
```
ggplot(rp) +  
  geom_text(aes(x = dipl_sup, y = cadres, label = commune),  
            color = "darkred", size = 2)
```



8.3.5 geom_label

`geom_label` est identique à `geom_text`, mais avec une présentation un peu différente.

```
ggplot(rp) + geom_label(aes(x = dipl_sup, y = cadres, label = commune))
```

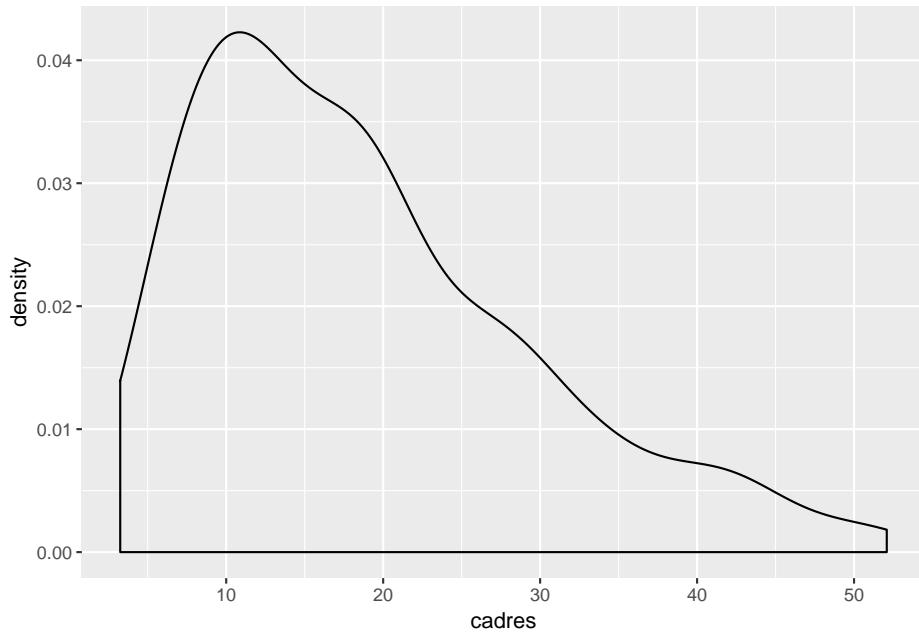


8.3.6 geom_density

`geom_density` permet d'afficher l'estimation de densité d'une variable numérique. Son usage est similaire à celui de `geom_histogram`.

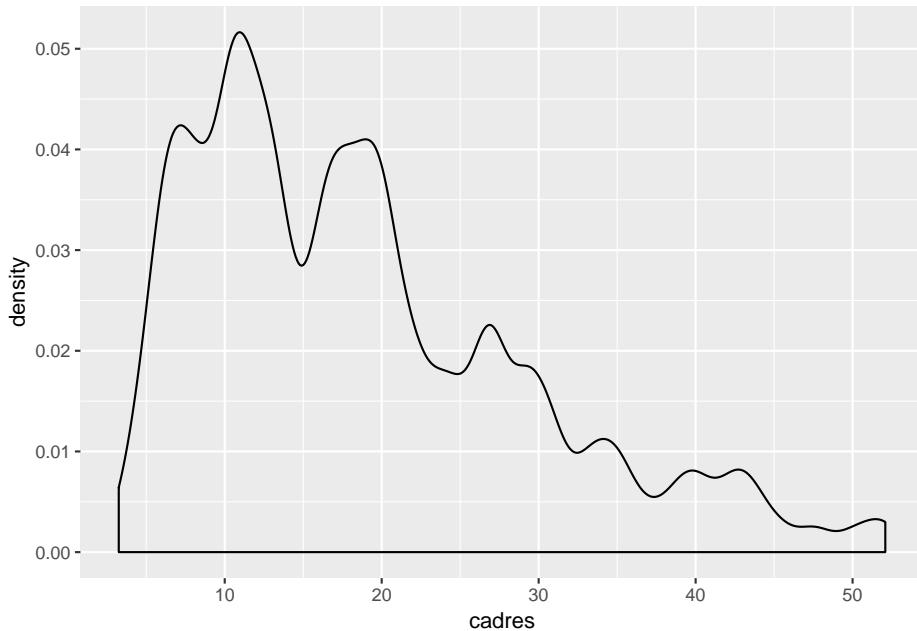
Ainsi, si on veut afficher la densité de la répartition de la part des cadres dans les communes de notre jeu de données :

```
ggplot(rp) + geom_density(aes(x = cadres))
```



On peut utiliser différents arguments pour ajuster le calcul de l'estimation de densité, parmi lesquels `kernel` et `bw` (voir la page d'aide de la fonction `density` pour plus de détails). `bw` (abréviation de *bandwidth*, bande passante) permet de régler la “finesse” de l'estimation de densité, un peu comme le choix du nombre de classes dans un histogramme :

```
ggplot(rp) + geom_density(aes(x = cadres), bw = 1)
```



8.3.7 geom_line

`geom_line` trace des lignes connectant les différentes observations entre elles. Il est notamment utilisé pour la représentation de séries temporelles. On passe à `geom_line` deux paramètres : `x` et `y`. Les observations sont alors connectées selon l'ordre des valeurs passées en `x`.

Comme il n'y a pas de données adaptées pour ce type de représentation dans notre jeu de données d'exemple, on va utiliser ici le jeu de données `economics` inclus dans `ggplot2` et représenter l'évolution du taux de chômage aux États-Unis (variable `unemploy`) dans le temps (variable `date`) :

```
data("economics")
economics
```

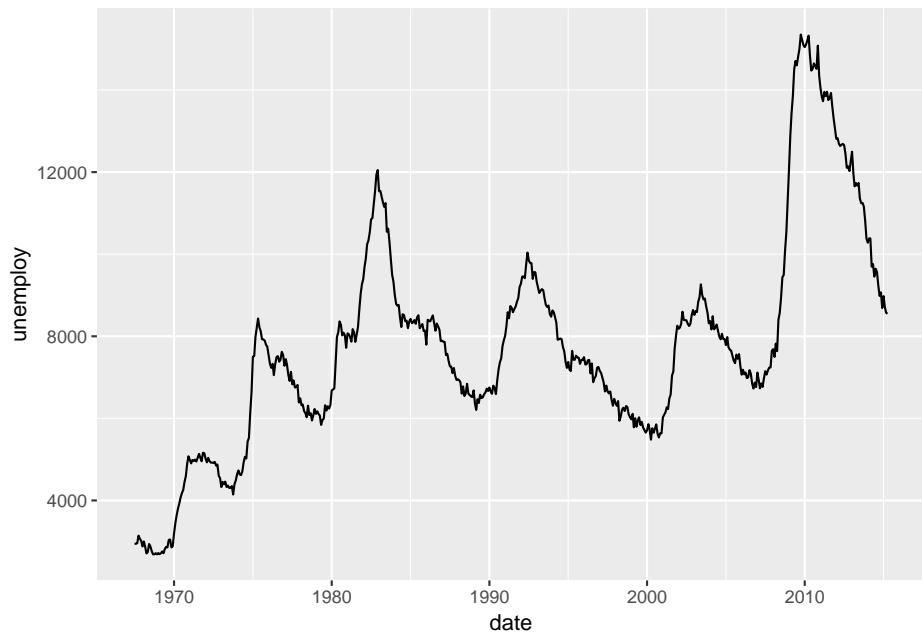
	date	pce	pop	psavert	uempmed	unemploy
	<date>	<dbl>	<int>	<dbl>	<dbl>	<int>
1	1967-07-01	507.	198712	12.5	4.5	2944
2	1967-08-01	510.	198911	12.5	4.7	2945
3	1967-09-01	516.	199113	11.7	4.6	2958
4	1967-10-01	513.	199311	12.5	4.9	3143

```

5 1967-11-01 518. 199498    12.5    4.7    3066
6 1967-12-01 526. 199657    12.1    4.8    3018
7 1968-01-01 532. 199808    11.7    5.1    2878
8 1968-02-01 534. 199920    12.2    4.5    3001
9 1968-03-01 545. 200056    11.6    4.1    2877
10 1968-04-01 545. 200208   12.2    4.6    2709
# ... with 564 more rows

```

```
ggplot(economics) + geom_line(aes(x = date, y = unemploy))
```



8.4 Mappages

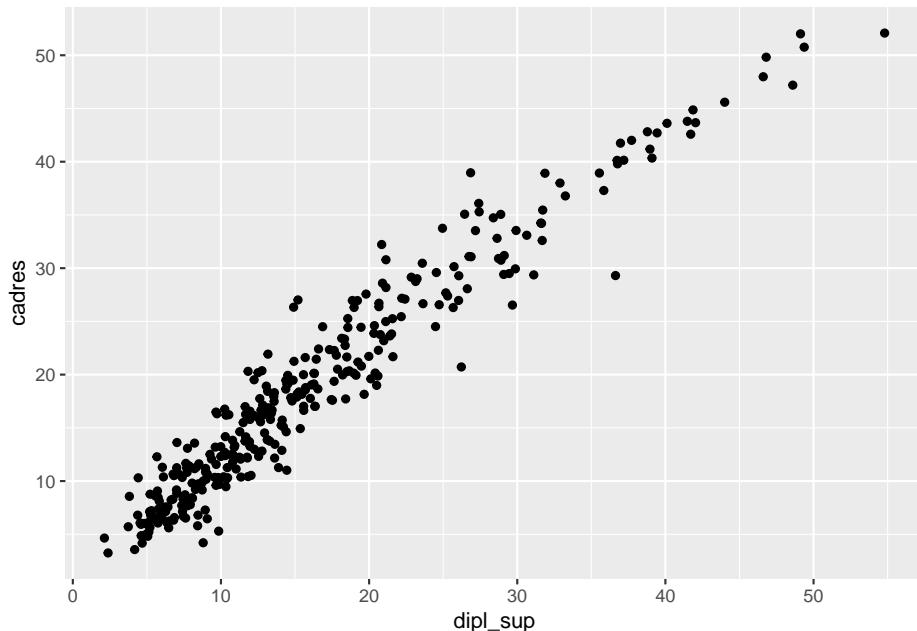
Un *mappage*, dans *ggplot2*, est une mise en relation entre **un attribut graphique** du *geom* (position, couleur, taille...) et **une variable** du tableau de données.

Ces mappages sont passés aux différents *geom* via la fonction *aes()* (abréviation d'*aesthetic*).

8.4.1 Exemples de mappages

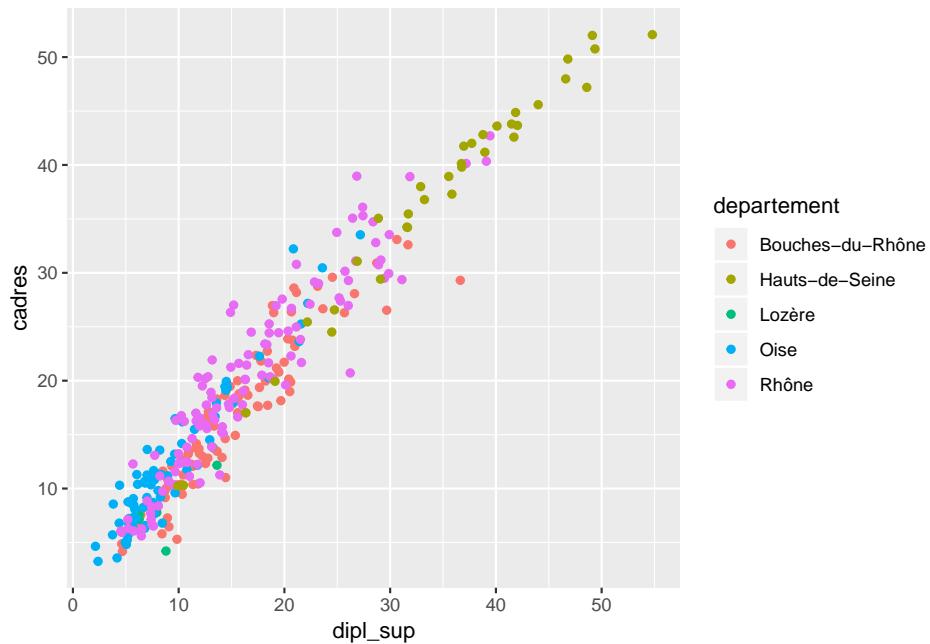
On a déjà vu les mappages `x` et `y` pour un nuage de points. Ceux-ci signifient que la position d'un point donné horizontalement (`x`) et verticalement (`y`) dépend de la valeur des variables passées comme arguments `x` et `y` dans `aes()` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres))
```



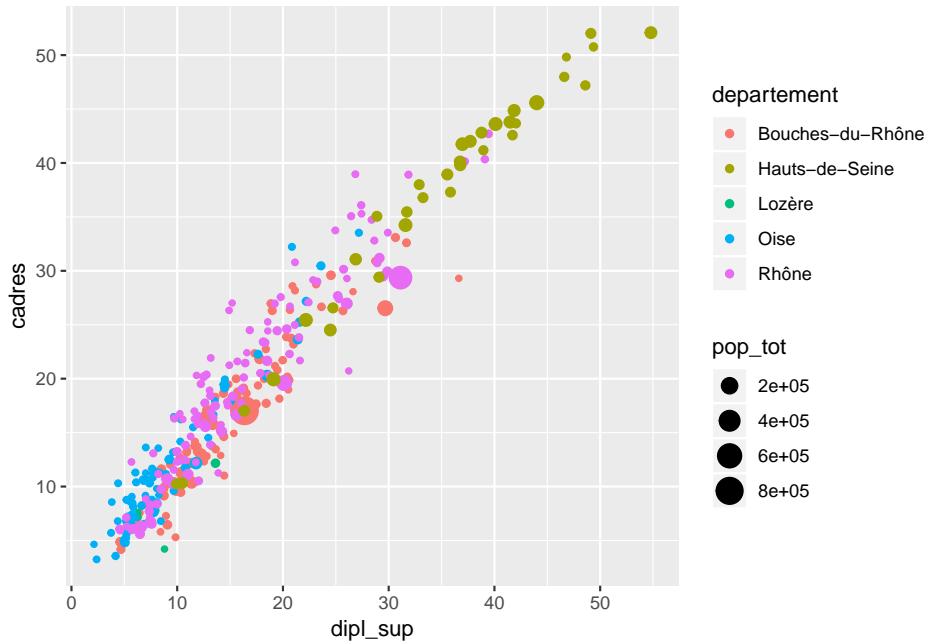
Mais on peut ajouter d'autres mappages. Par exemple, `color` permet de faire varier la couleur des points automatiquement en fonction des valeurs d'une troisième variable. Ainsi, on peut vouloir colorer les points selon le département de la commune correspondante :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = departement))
```



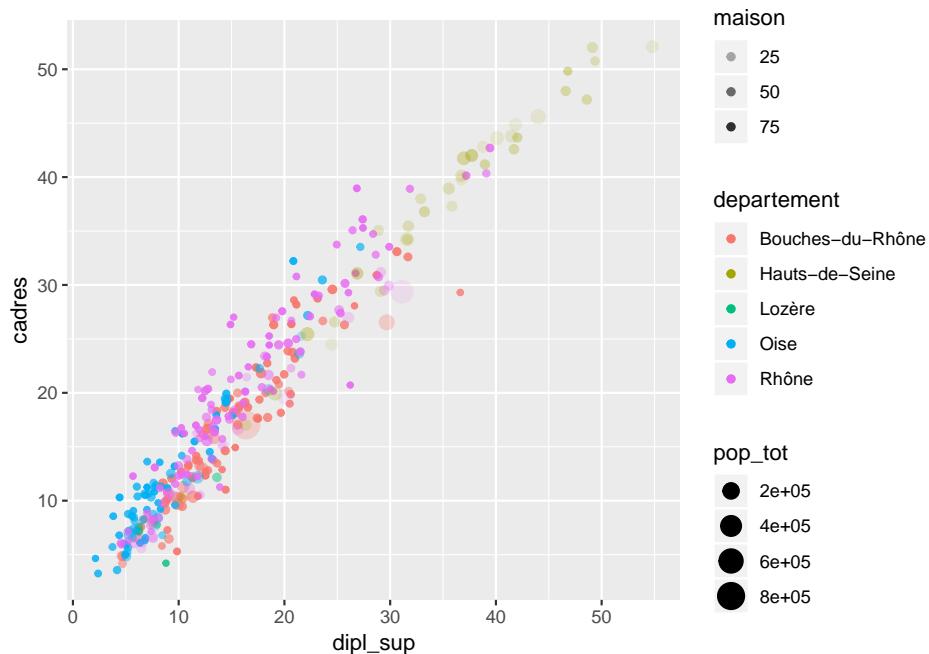
On peut aussi faire varier la taille des points avec `size`. Ici, la taille dépend de la population totale de la commune :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres,  
                 color = departement, size = pop_tot))
```



On peut même associer la transparence des points à une variable avec alpha :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres,
                 color = departement, size = pop_tot, alpha = maison))
```

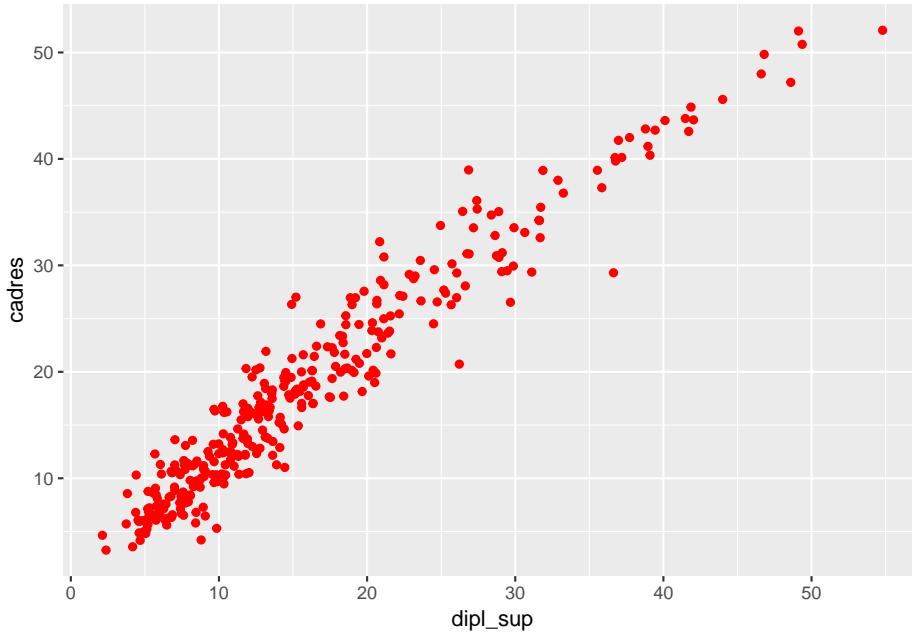


Chaque geom possède sa propre liste de mappages.

8.4.2 `aes()` or not `aes()` ?

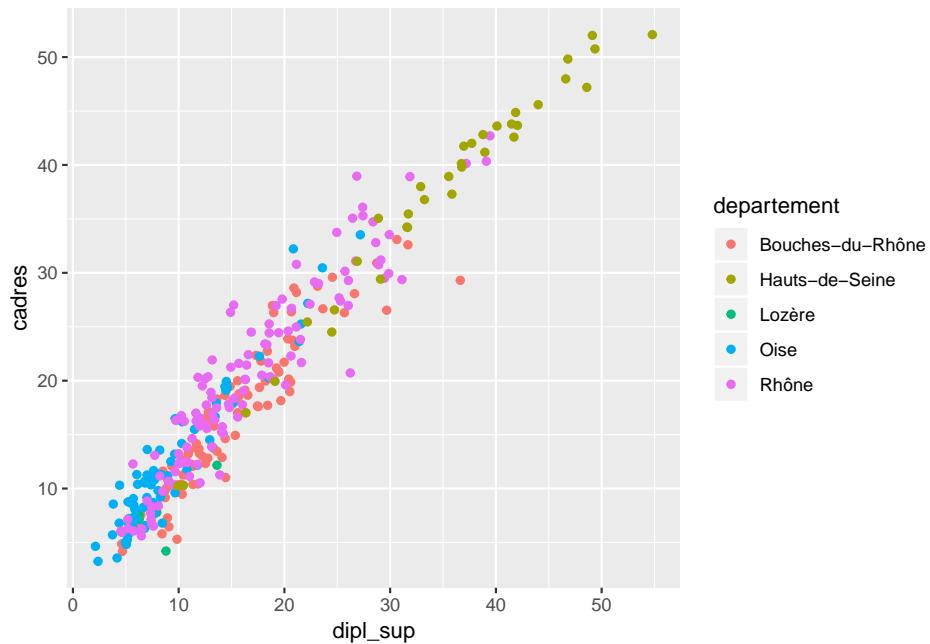
Comme on l'a déjà vu, parfois on souhaite changer un attribut sans le relier à une variable. Par exemple, on veut représenter tous les points en rouge. Dans ce cas on utilise toujours l'attribut `color`, mais comme il ne s'agit pas d'un mappage, on le définit **à l'extérieur** de la fonction `aes()` :

```
ggplot(rp) + geom_point(aes(x = dipl_sup, y = cadres), color = "red")
```



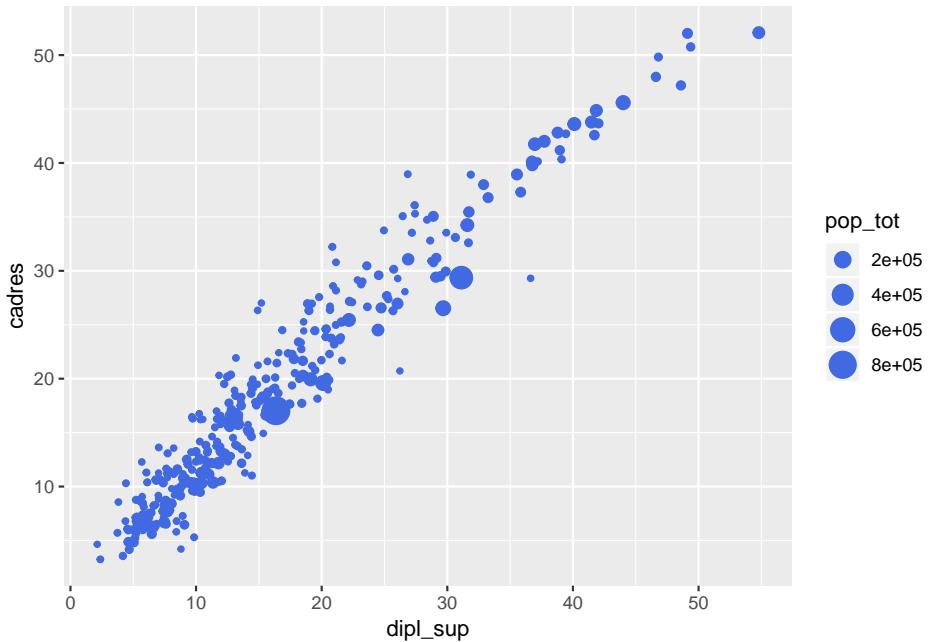
Par contre, si on veut faire varier la couleur en fonction des valeurs prises par une variable, on réalise un mappage, et on doit donc placer l'attribut `color` à l'intérieur de `aes()`.

```
ggplot(rp) + geom_point(aes(x = dipl_sup, y = cadres, color = departement))
```



On peut évidemment mélanger attributs liés à une variable (mappage, donc dans `aes()`) et attributs constants (donc à l'extérieur). Dans l'exemple suivant, la taille varie en fonction de la variable `pop_tot`, mais la couleur est constante pour tous les points :

```
ggplot(rp) + geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot), color = "royalbi
```



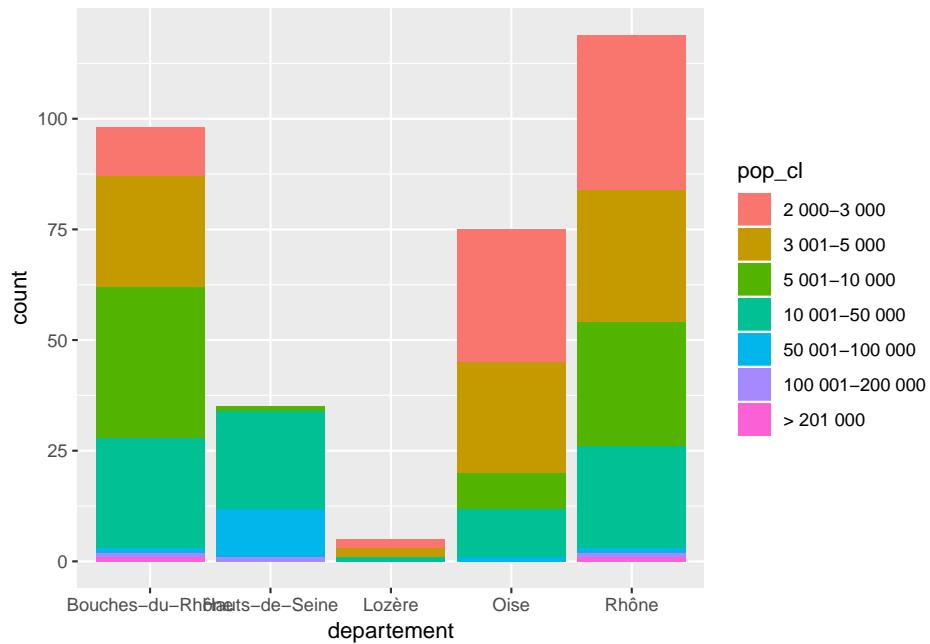
La règle est donc simple mais très importante :

Si on établit un lien entre les valeurs d'une variable et un attribut graphique, on définit un mappage, et on le déclare dans `aes()`. Sinon, on modifie l'attribut de la même manière pour tous les points, et on le définit en-dehors de la fonction `aes()`.

8.4.3 geom_bar et position

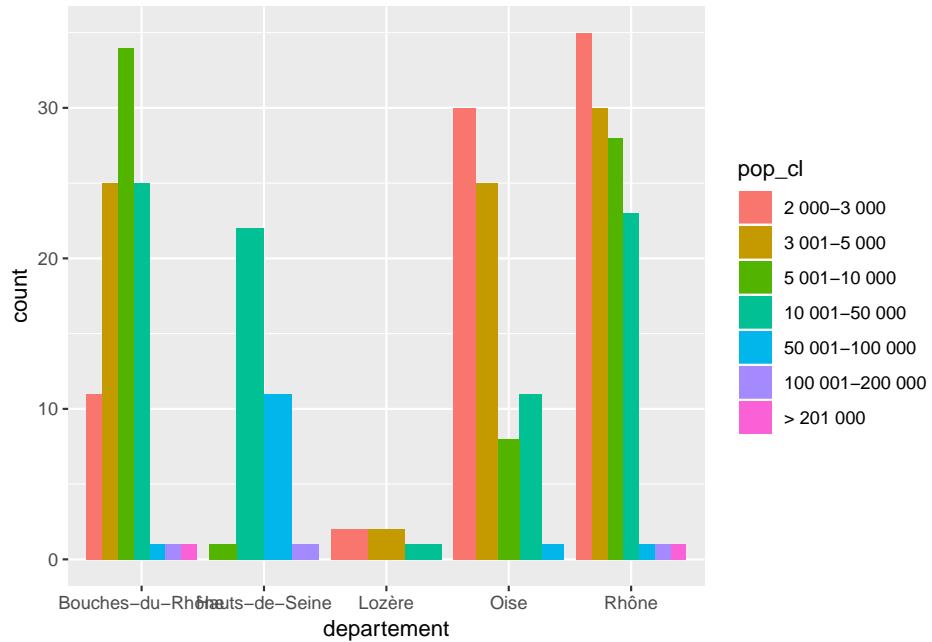
Un des mappages possibles de `geom_bar` est l'attribut `fill`, qui permet de tracer des barres de couleur différentes selon les modalités d'une deuxième variable :

```
ggplot(rp) + geom_bar(aes(x = departement, fill = pop_c1))
```



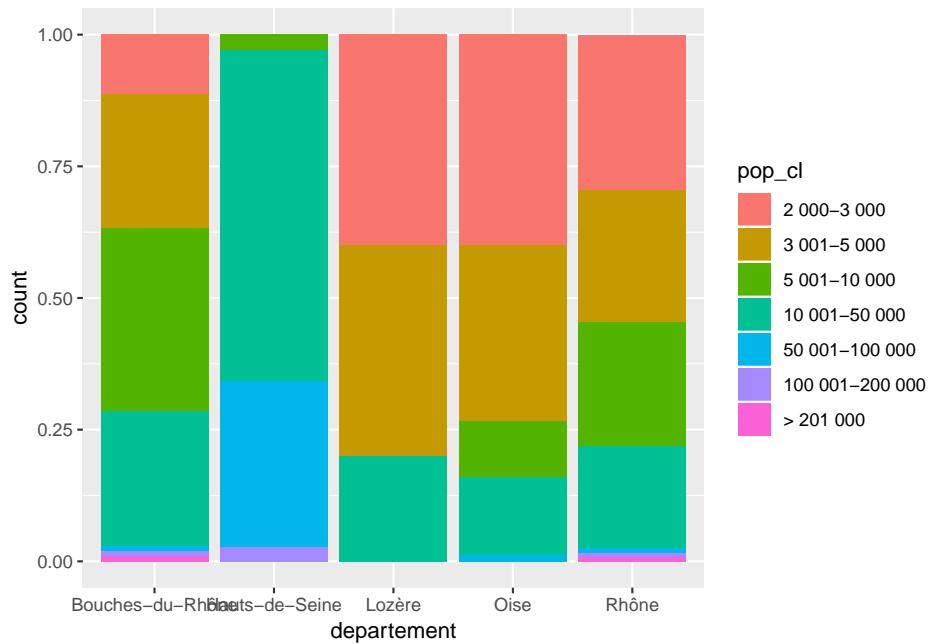
L’attribut `position` de `geom_bar` permet d’indiquer comment les différentes barres doivent être positionnées. Par défaut on a `position = "stack"` et elles sont donc “empilées”. Mais on peut préciser `position = "dodge"` pour les mettre côté à côté :

```
ggplot(rp) + geom_bar(aes(x = departement, fill = pop_cl), position = "dodge")
```



Ou encore `position = "fill"` pour représenter non plus des effectifs, mais des proportions :

```
ggplot(rp) + geom_bar(aes(x = departement, fill = pop_cl), position = "fill")
```

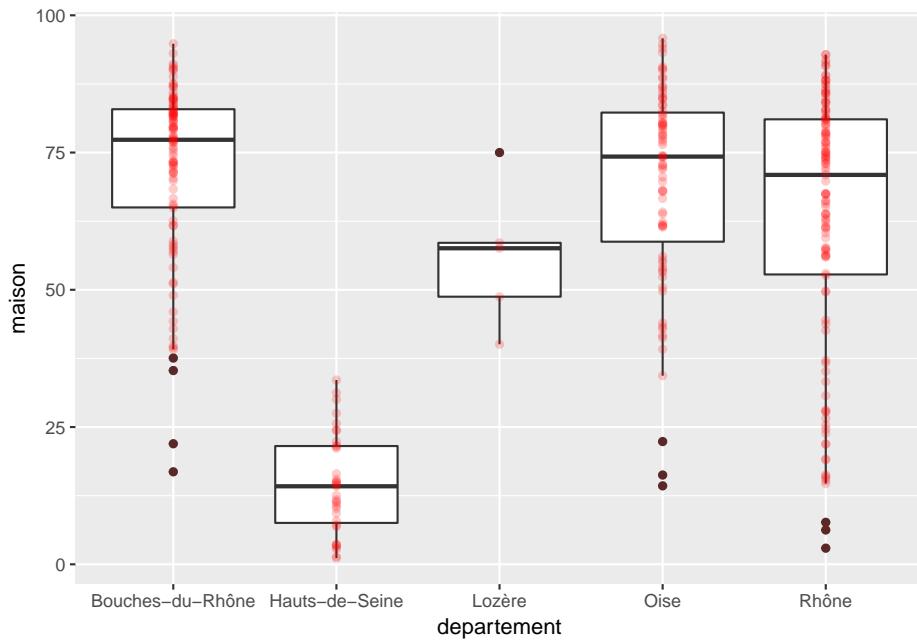


8.5 Représentation de plusieurs geom

On peut représenter plusieurs `geom` simultanément sur un même graphique, il suffit de les ajouter à tour de rôle avec l'opérateur `+`.

Par exemple, on peut superposer la position des points au-dessus d'un boxplot. On va pour cela ajouter un `geom_point` après avoir ajouté notre `geom_boxplot` :

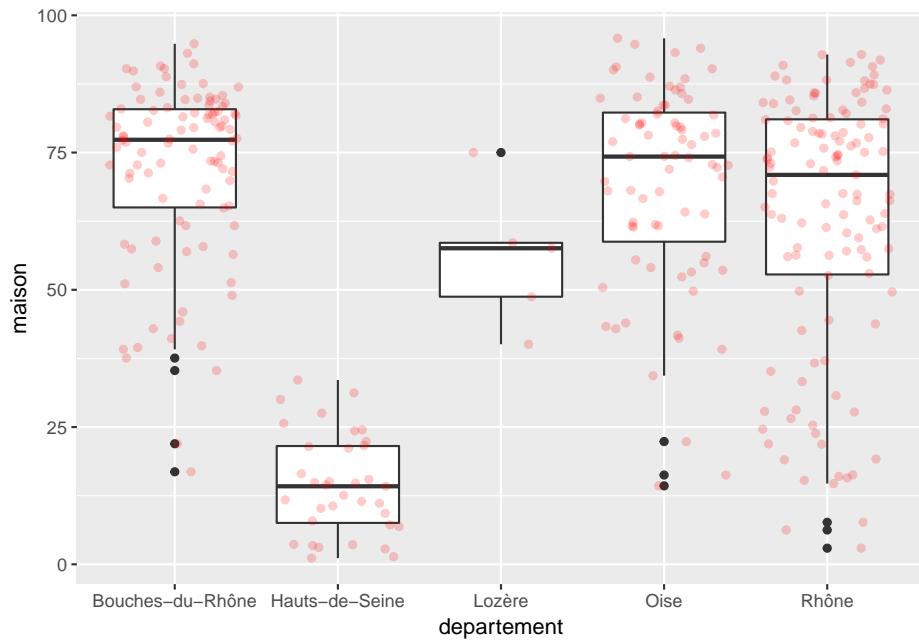
```
ggplot(rp) +
  geom_boxplot(aes(x = departement, y = maison)) +
  geom_point(aes(x = departement, y = maison), col = "red", alpha = 0.2)
```



Quand une commande `ggplot2` devient longue, il peut être plus lisible de la répartir sur plusieurs lignes. Dans ce cas, il faut penser à placer l'opérateur `+` en fin de ligne, afin que R comprenne que la commande n'est pas complète et qu'il prenne en compte la suite.

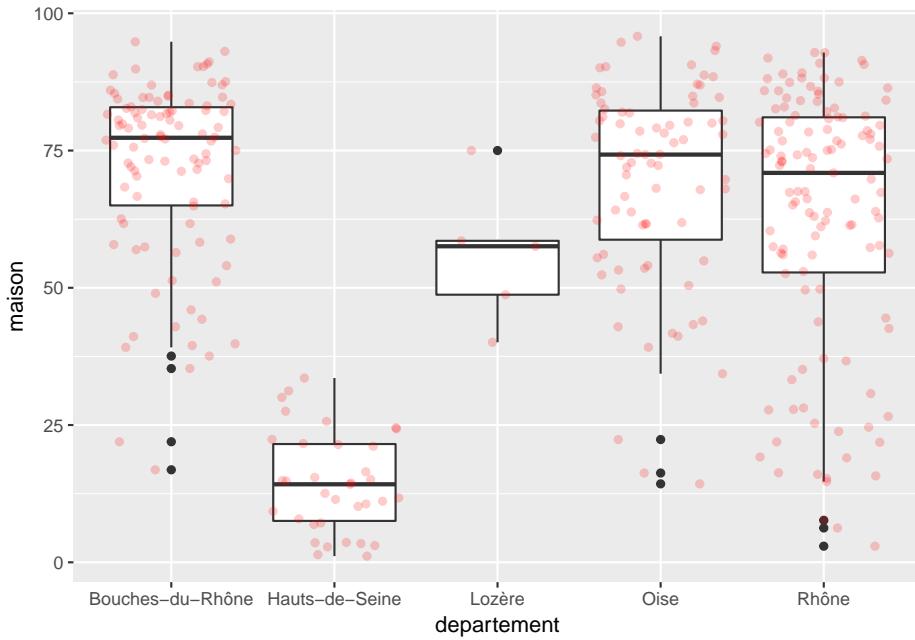
Pour un résultat un peu plus lisible, on peut remplacer `geom_point` par `geom_jitter`, qui disperse les points horizontalement et facilite leur visualisation :

```
ggplot(rp) +
  geom_boxplot(aes(x = departement, y = maison)) +
  geom_jitter(aes(x = departement, y = maison), col = "red", alpha = 0.2)
```



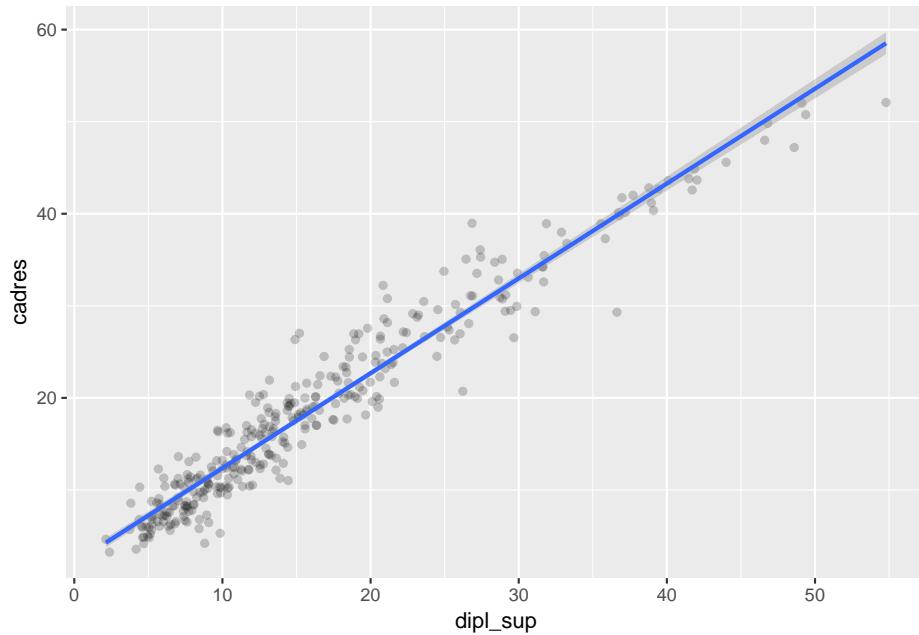
Pour simplifier un peu le code, plutôt que de déclarer les mappages dans chaque `geom`, on peut les déclarer dans l'appel à `ggplot()`. Ils seront automatiquement “hérités” par les `geom` ajoutés (sauf s'ils redéfinissent les mêmes mappages) :

```
ggplot(rp, aes(x = departement, y = maison)) +
  geom_boxplot() +
  geom_jitter(color = "red", alpha = 0.2)
```



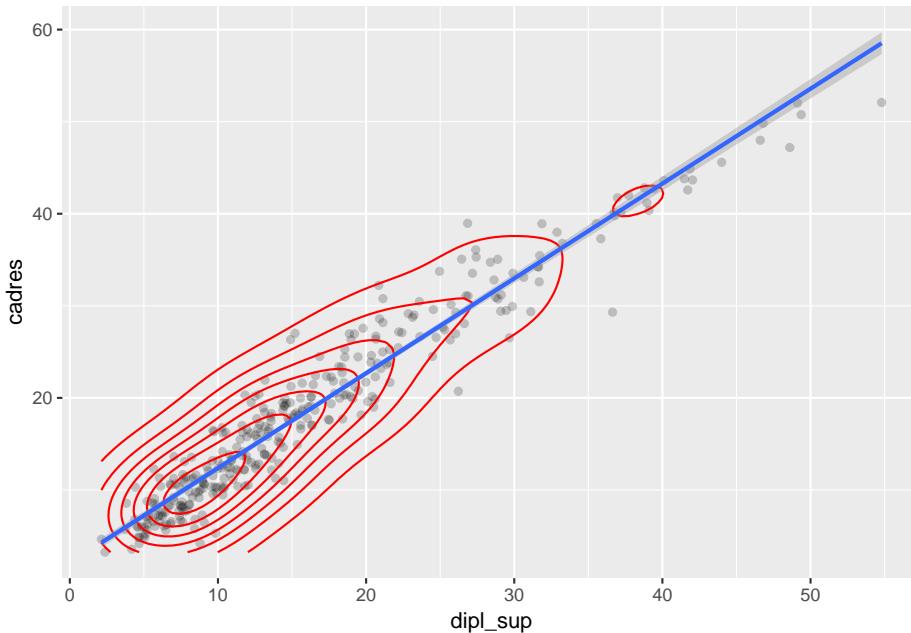
Autre exemple, on peut vouloir ajouter à un nuage de points une ligne de régression linéaire à l'aide de `geom_smooth` :

```
ggplot(rp, aes(x = dipl_sup, y = adres)) +  
  geom_point(alpha = 0.2) +  
  geom_smooth(method = "lm")
```



Et on peut même superposer une troisième visualisation de la répartition des points dans l'espace avec `geom_density2d` :

```
ggplot(rp, aes(x = dipl_sup, y = cadres)) +  
  geom_point(alpha = 0.2) +  
  geom_density2d(color = "red") +  
  geom_smooth(method = "lm")
```



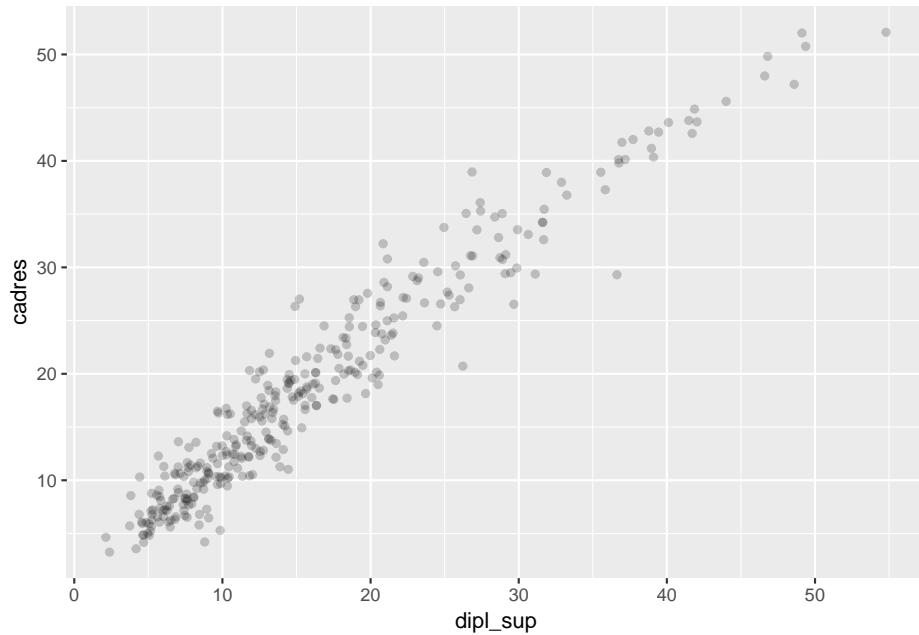
8.5.1 Plusieurs sources de données

On peut aussi associer à différents *geom* des sources de données différentes. Supposons qu'on souhaite afficher sur un nuage de points les noms des communes de plus de 50000 habitants. On peut commencer par créer un tableau de données avec seulement ces communes à l'aide de la fonction *filter* :

```
com50 <- filter(rp, pop_tot >= 50000)
```

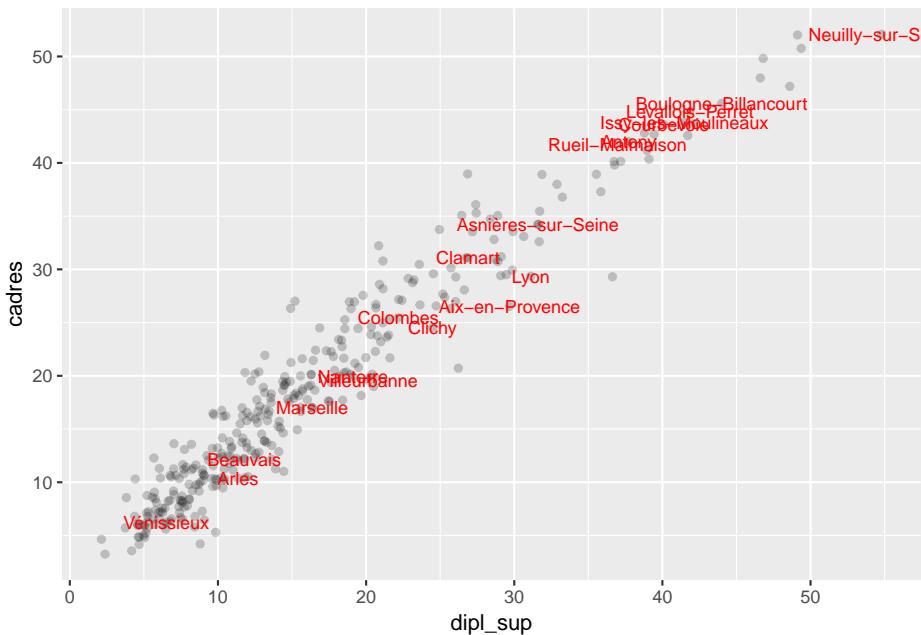
On fait ensuite le nuage de points comme précédemment :

```
ggplot(data = rp, aes(x = dipl_sup, y = cadres)) +  
  geom_point(alpha = 0.2)
```



Pour superposer les noms de communes de plus de 50 000 habitants, on peut ajouter un `geom_text`, mais en spécifiant que les données proviennent du nouveau tableau `com50` et non de notre tableau initial `rp`. On le fait en passant un argument `data` spécifique à `geom_text` :

```
ggplot(data = rp, aes(x = dipl_sup, y = cadres)) +
  geom_point(alpha = 0.2) +
  geom_text(data = com50, aes(label = commune), color = "red", size = 3)
```



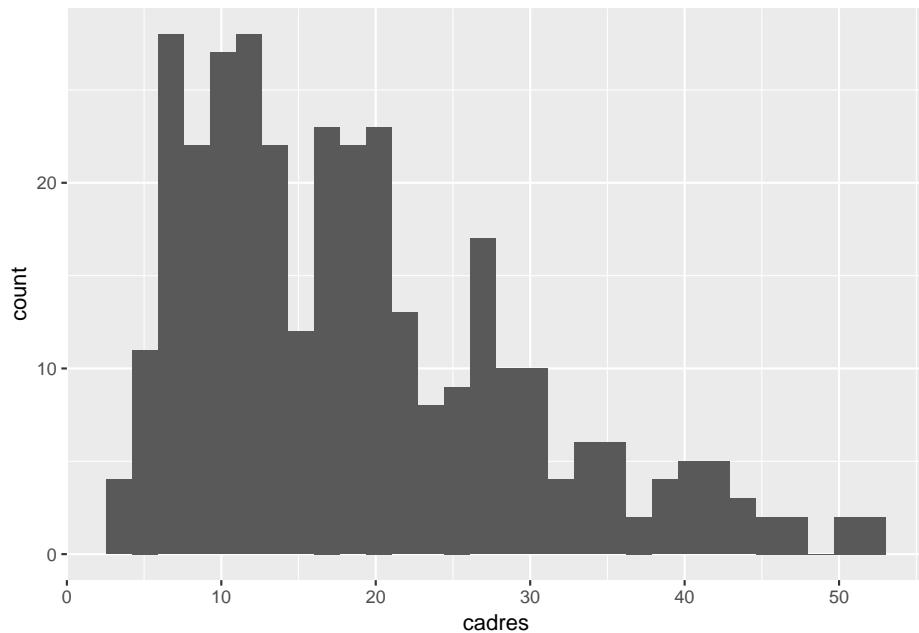
Ainsi, on obtient un graphique avec deux `geom` superposés, mais dont les données proviennent de deux tableaux différents.

8.6 Faceting

Le *faceting* permet d'effectuer plusieurs fois le même graphique selon les valeurs d'une ou plusieurs variables qualitatives.

Par exemple, on a vu qu'on peut représenter l'histogramme du pourcentage de cadres dans nos communes avec le code suivant :

```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres))
```

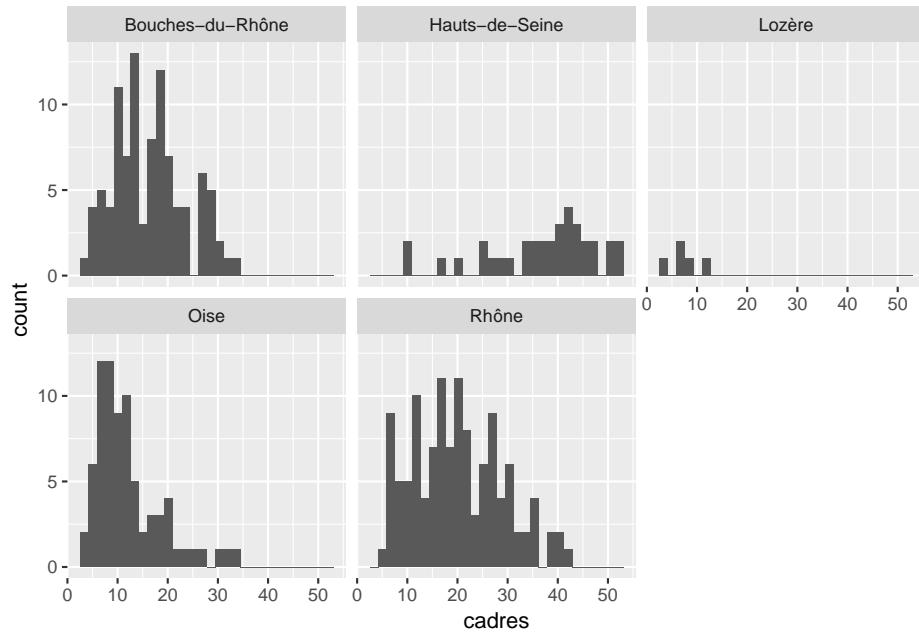


On peut vouloir comparer cette répartition de la part des cadres selon le département, et donc faire un histogramme pour chacun de ces départements. C'est ce que permettent les fonctions `facet_wrap` et `facet_grid`.

Les deux fonctions prennent en paramètre une formule de la forme `~variable`, où `variable` est le nom de la variable en fonction de laquelle on souhaite faire les différents graphiques.

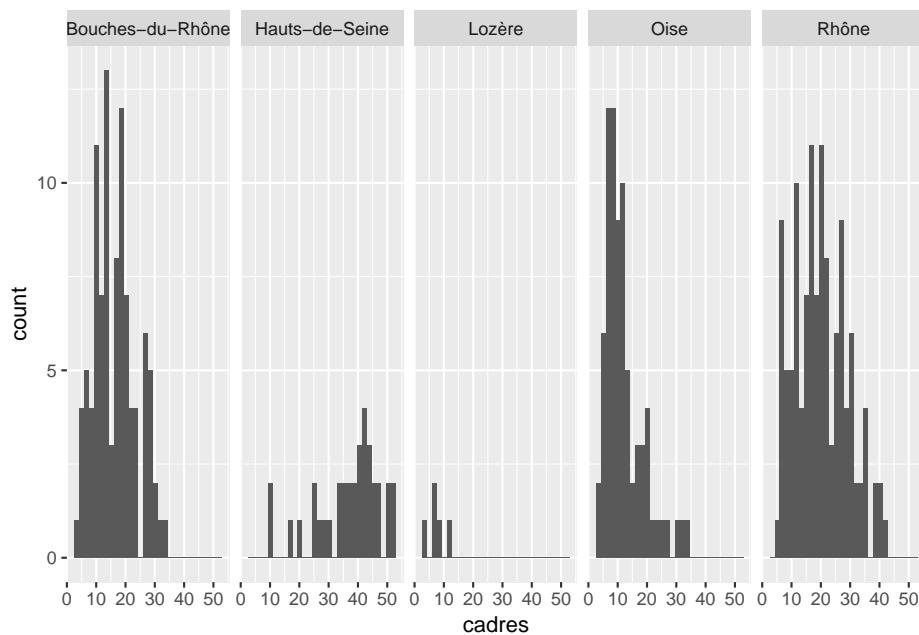
Avec `facet_wrap`, les différents graphiques sont affichés les uns à côté des autres et répartis automatiquement dans la page :

```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  facet_wrap(~departement)
```



Pour `facet_grid`, les graphiques sont disposés selon une grille. La formule est alors de la forme `variable en ligne ~ variable en colonne`. Si on n'a pas de variable dans l'une des deux dimensions, on met un point (.) :

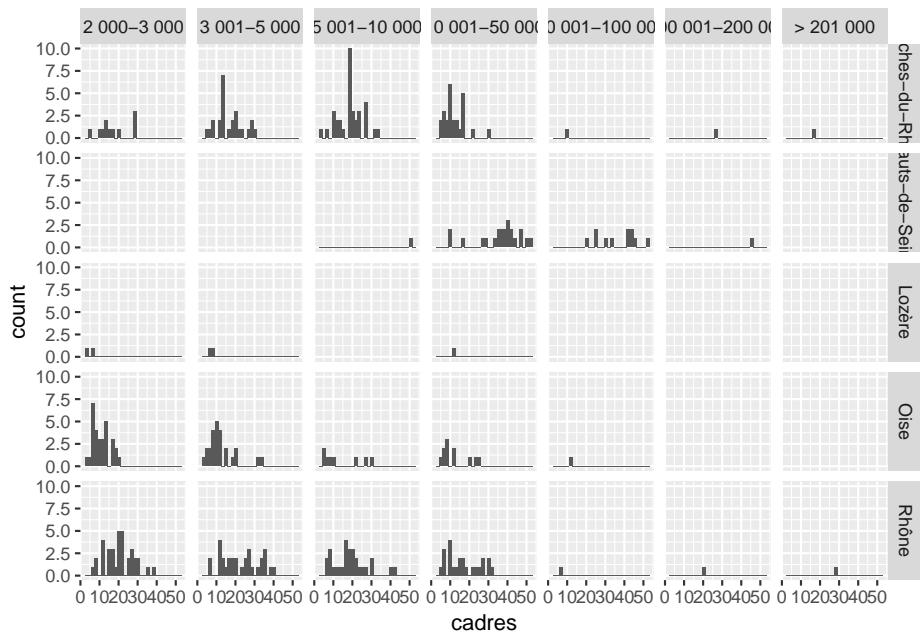
```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  facet_grid(.~departement)
```



Un des intérêts du faceting dans *ggplot2* est que tous les graphiques générés ont les mêmes échelles, ce qui permet une comparaison directe.

Enfin, notons qu'on peut même faire du faceting sur plusieurs variables à la fois. On peut par exemple faire des histogrammes de la répartition de la part des cadres pour chaque croisement des variables `departement` et `pop_cl` :

```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  facet_grid(departement ~ pop_cl)
```



L’histogramme en haut à gauche représente la répartition du pourcentage de cadres parmi les communes de 2000 à 3000 habitants dans les Bouches-du-Rhône, etc.

8.7 Scales

On a vu qu’avec `ggplot2` on définit des mappages entre des attributs graphiques (position, taille, couleur, etc.) et des variables d’un tableau de données. Ces mappages sont définis, pour chaque `geom`, via la fonction `aes()`.

Les *scales* dans `ggplot2` permettent de modifier la manière dont un attribut graphique va être relié aux valeurs d’une variable, et dont la légende correspondante va être affichée. Par exemple, pour l’attribut `color`, on pourra définir la palette de couleur utilisée. Pour `size`, les tailles minimales et maximales, etc.

Pour modifier une *scale* existante, on ajoute un nouvel élément à notre objet `ggplot2` avec l’opérateur `+`. Cet élément prend la forme `scale_<attribut>_<type>`.

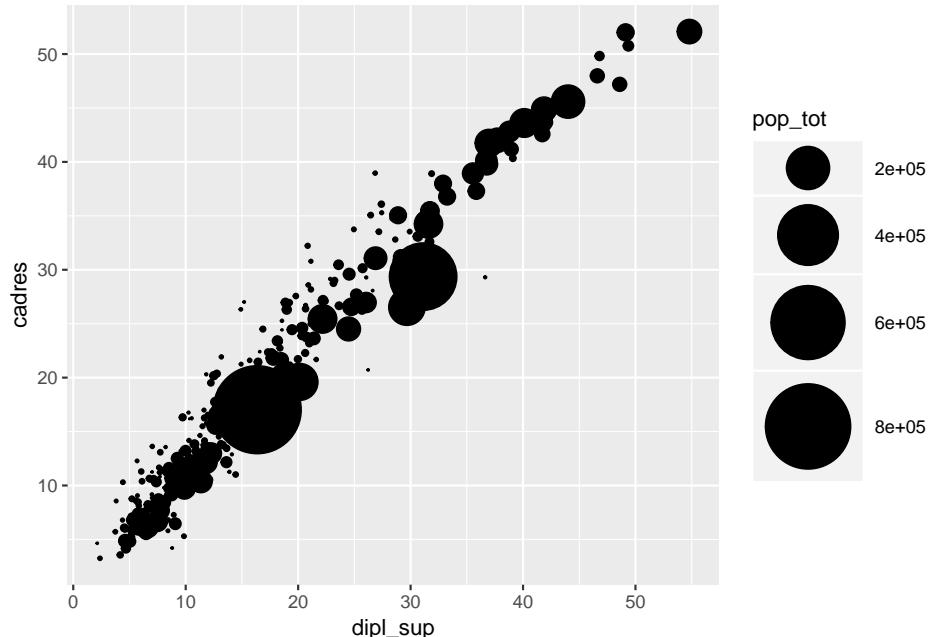
Voyons tout de suite quelques exemples.

8.7.1 scale_size

Si on souhaite modifier les tailles minimales et maximales des objets quand on a effectué un mappage de type `size`, on peut utiliser la fonction `scale_size` et

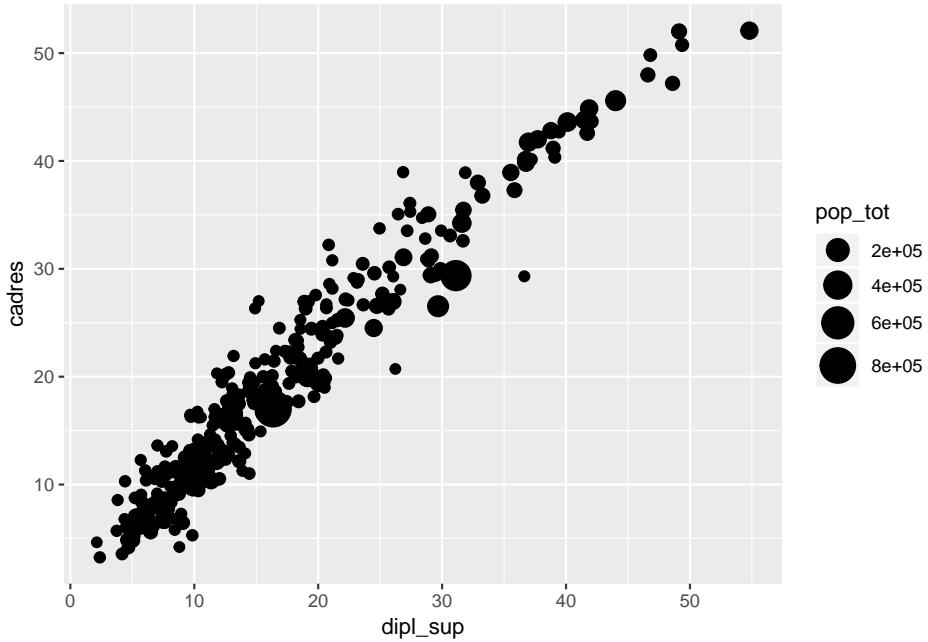
son argument `range` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +
  scale_size(range = c(0,20))
```



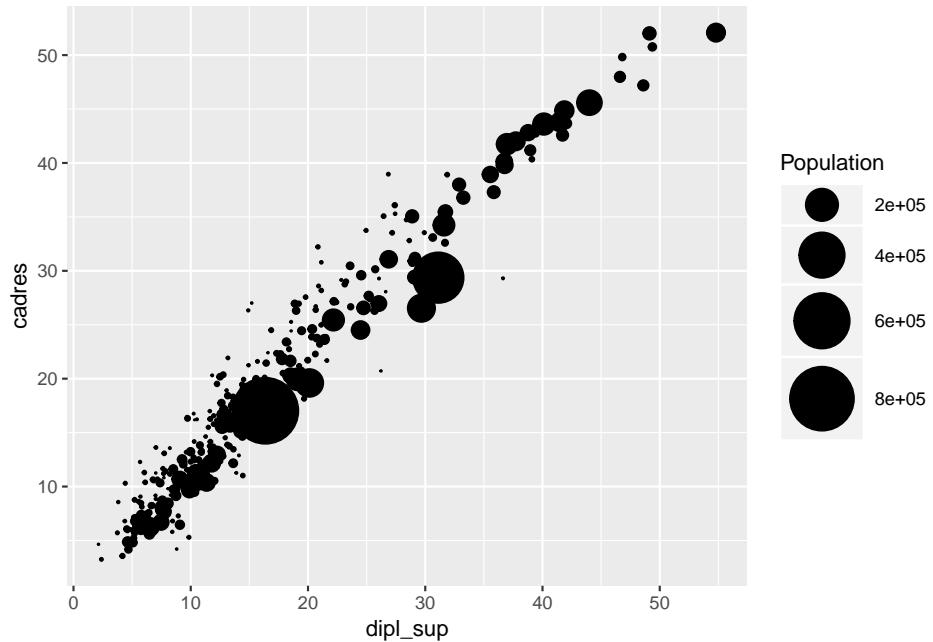
À comparer par exemple à :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +
  scale_size(range = c(2,8))
```



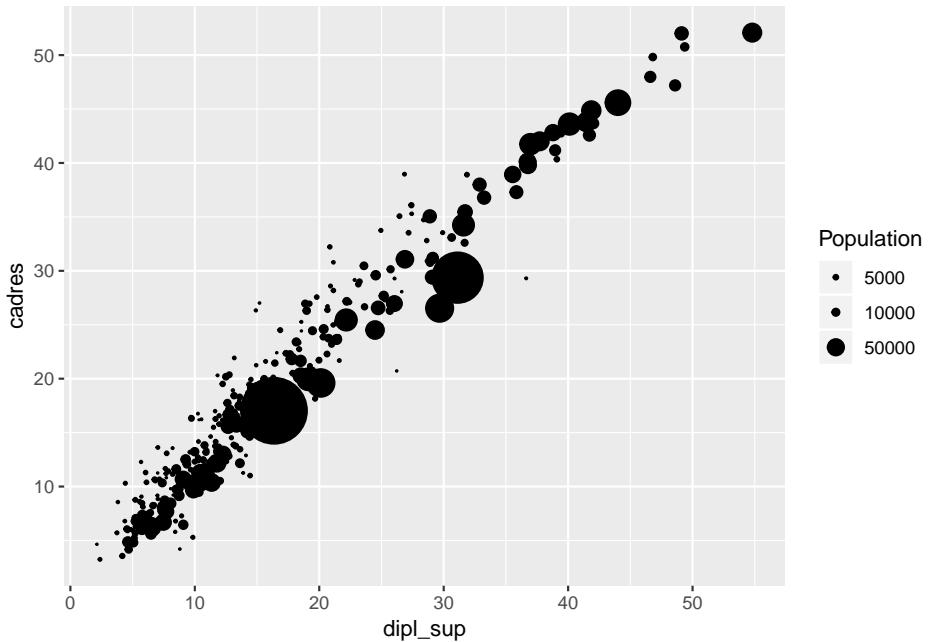
On peut ajouter d'autres paramètres à `scale_size`. Le premier argument est toujours le titre donné à la légende :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +
  scale_size("Population", range = c(0,15))
```



On peut aussi définir manuellement les éléments de légende représentés :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +  
  scale_size("Population", range = c(0,15), breaks = c(1000,5000,10000,50000))
```



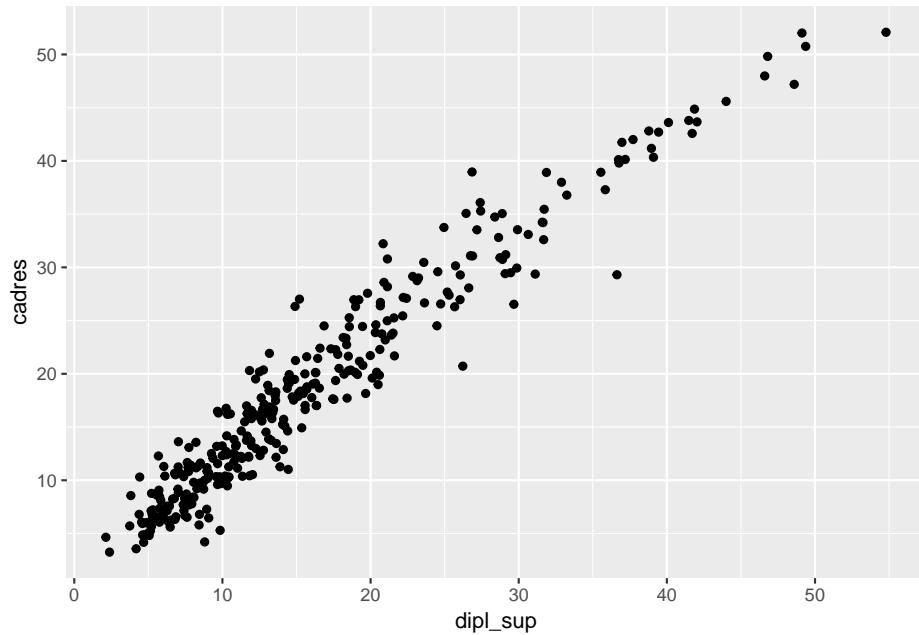
8.7.2 scale_x, scale_y

Les *scales* `scale_x` et `scale_y` modifient les axes x et y du graphique.

`scale_x_continuous` et `scale_y_continuous` s'appliquent lorsque la variable x ou y est numérique (quantitative).

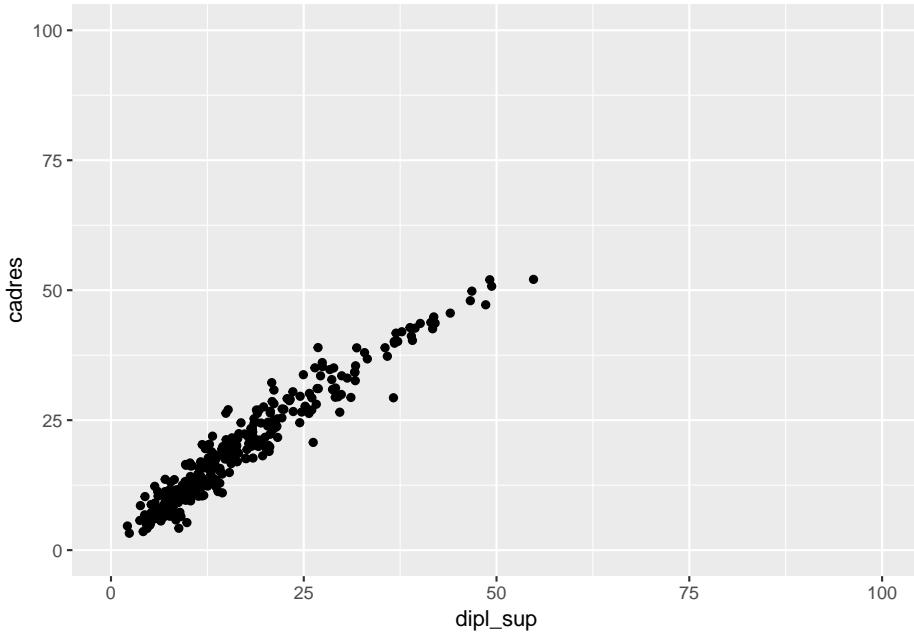
C'est le cas de notre nuage de points croisant part de cadres et part de diplômés du supérieur :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres))
```



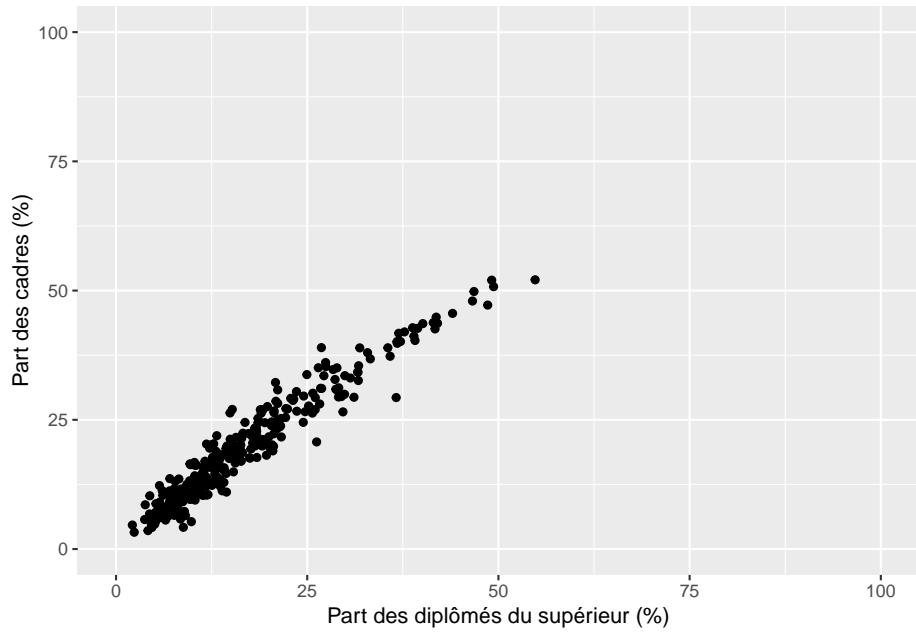
Comme on représente des pourcentages, on peut vouloir forcer les axes x et y à s'étendre des valeurs 0 à 100. On peut le faire en ajoutant un élément `scale_x_continuous` et un élément `scale_y_continuous`, et en utilisant leur argument `limits` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres)) +
  scale_x_continuous(limits = c(0,100)) +
  scale_y_continuous(limits = c(0,100))
```



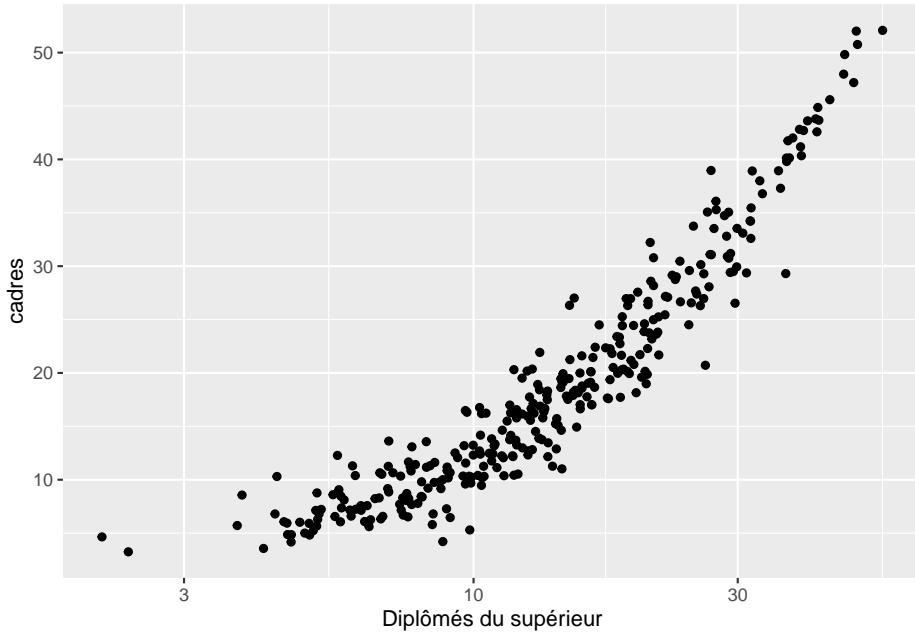
Là aussi, on peut modifier les étiquettes des axes en indiquant une chaîne de caractères en premier argument :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres)) +
  scale_x_continuous("Part des diplômés du supérieur (%)", limits = c(0,100)) +
  scale_y_continuous("Part des cadres (%)", limits = c(0,100))
```



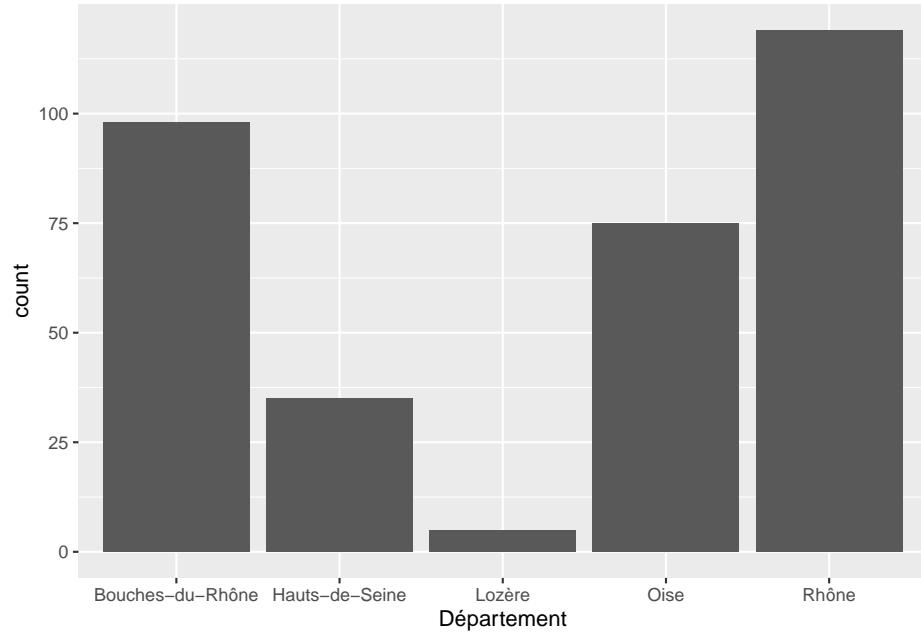
On peut utiliser `scale_x_log10` et `scale_y_log10` pour passer un axe à une échelle logarithmique :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres)) +  
  scale_x_log10("Diplômés du supérieur")
```



`scale_x_discrete` et `scale_y_discrete` s'appliquent quant à elles lorsque l'axe correspond à une variable discrète (qualitative). C'est le cas de l'axe des x dans un diagramme en bâtons :

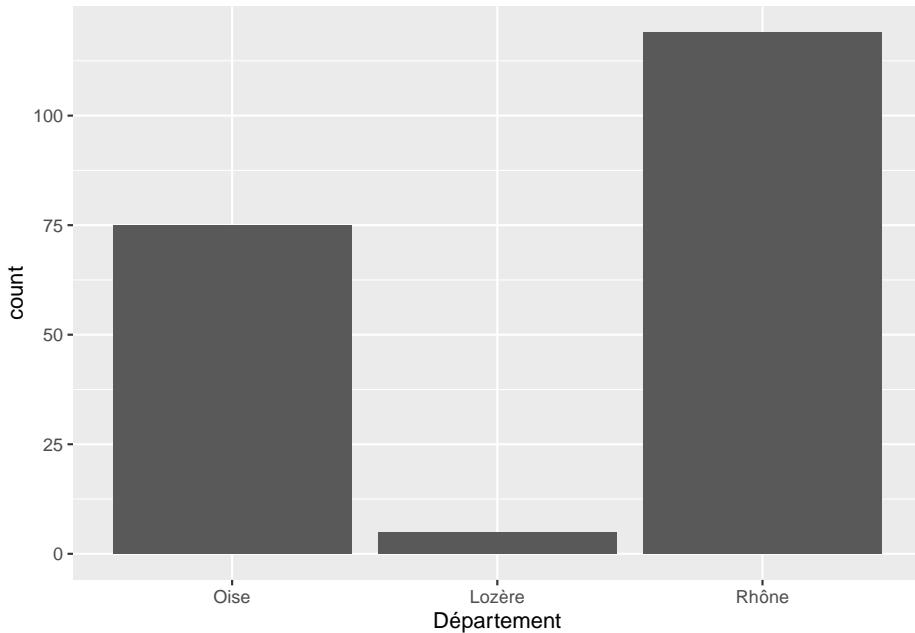
```
ggplot(rp) +
  geom_bar(aes(x = departement)) +
  scale_x_discrete("Département")
```



L’argument `limits` de `scale_x_discrete` permet d’indiquer quelles valeurs sont affichées et dans quel ordre.

```
ggplot(rp) +  
  geom_bar(aes(x = departement)) +  
  scale_x_discrete("Département", limits = c("Oise", "Lozère", "Rhône"))
```

```
Warning: Removed 133 rows containing non-finite values (stat_count).
```



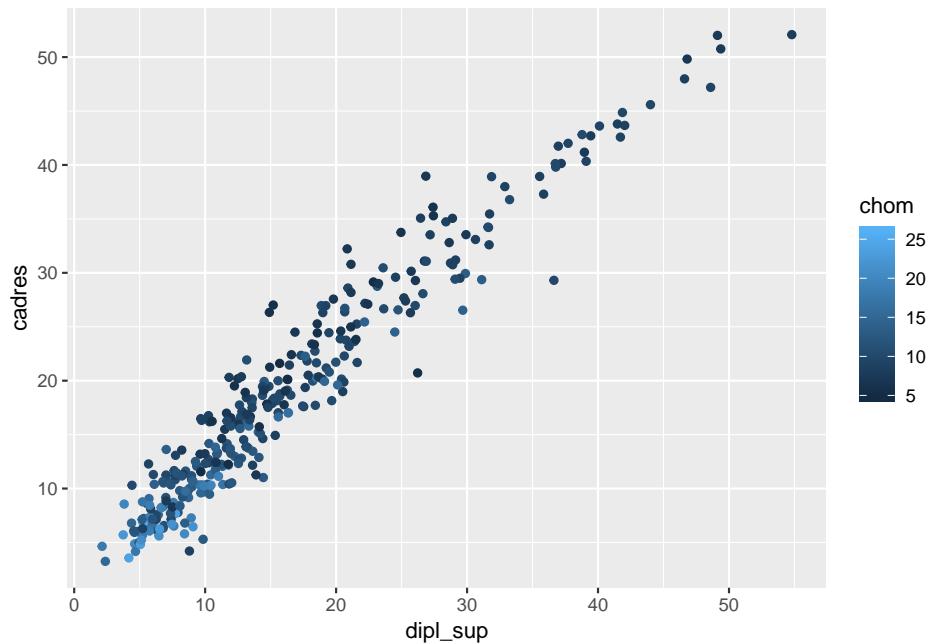
8.7.3 `scale_color`, `scale_fill`

Ces *scales* permettent, entre autre, de modifier les palettes de couleur utilisées pour le dessin (`color`) ou le remplissage (`fill`) des éléments graphiques. Dans ce qui suit, pour chaque fonction `scale_color` présentée il existe une fonction `scale_fill` équivalente et avec en général les mêmes arguments.

8.7.3.1 Variables quantitatives

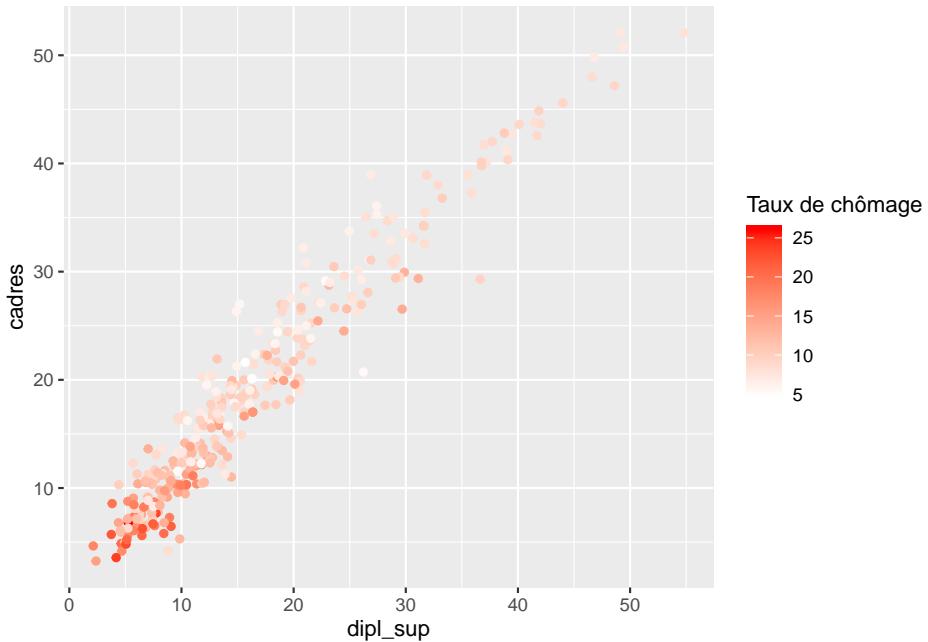
Le graphique suivant colore les points selon la valeur d'une variable numérique quantitative (ici la part de chômeurs) :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = chom))
```



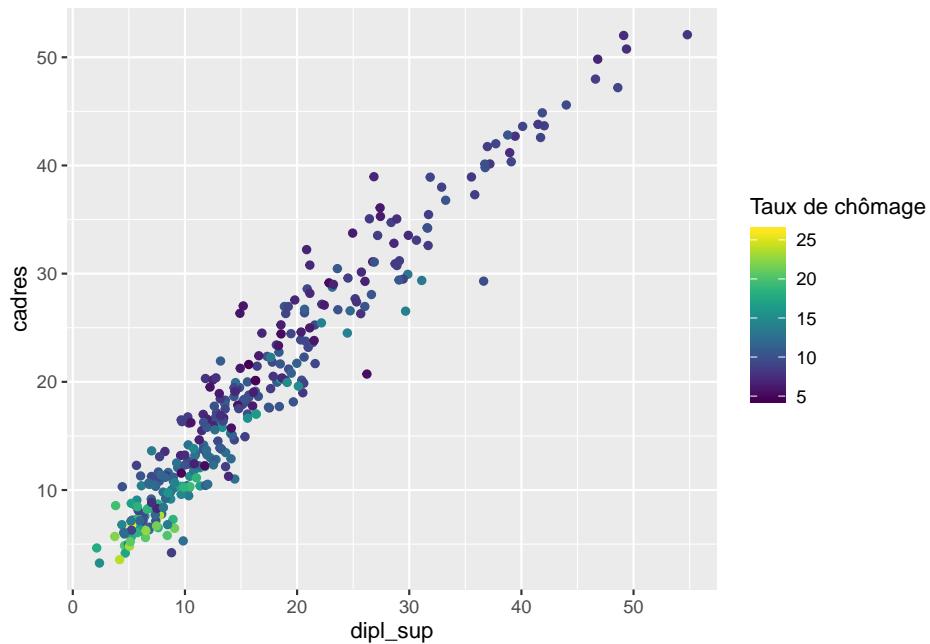
On peut modifier les couleurs utilisées avec les arguments `low` et `high` de la fonction `scale_color_gradient`. Ici on souhaite que la valeur la plus faible soit blanche, et la plus élevée rouge :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +  
  scale_color_gradient("Taux de chômage", low = "white", high = "red")
```



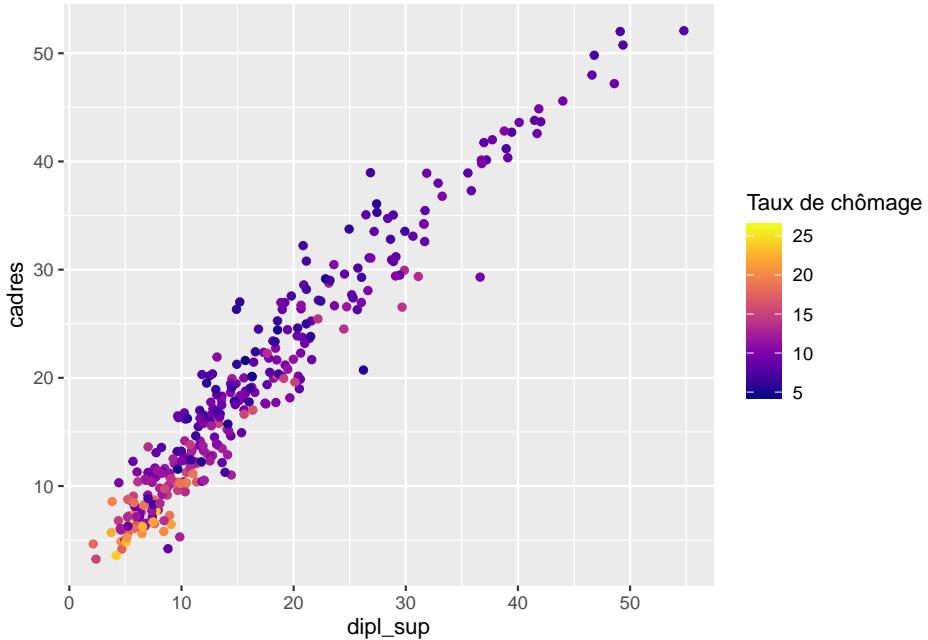
On peut aussi utiliser des palettes prédéfinies. L'une des plus populaires est la palette *viridis*, accessible depuis l'extension du même nom. On l'ajoute en utilisant `scale_color_viridis` :

```
library(viridis)
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +
  scale_color_viridis("Taux de chômage")
```



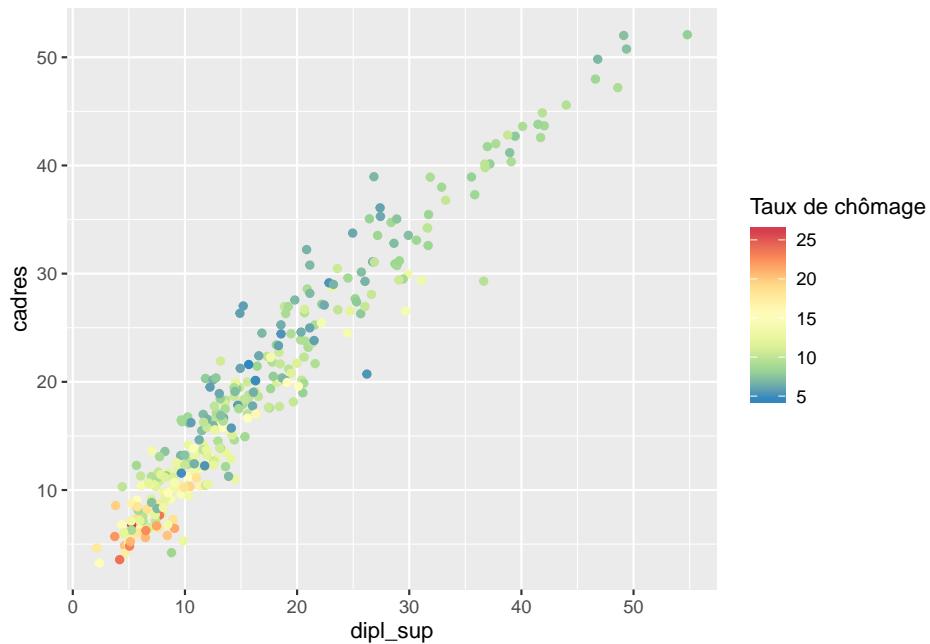
L’extension *viridis* propose également trois autres palettes, *magma*, *inferno* et *plasma*, accessibles via l’argument *option* :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +
  scale_color_viridis("Taux de chômage", option = "plasma")
```



On peut aussi utiliser `scale_color_distiller`, qui transforme une des palettes pour variable qualitative de `scale_color_brewer` en palette continue pour variable numérique :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +
  scale_color_distiller("Taux de chômage", palette = "Spectral")
```

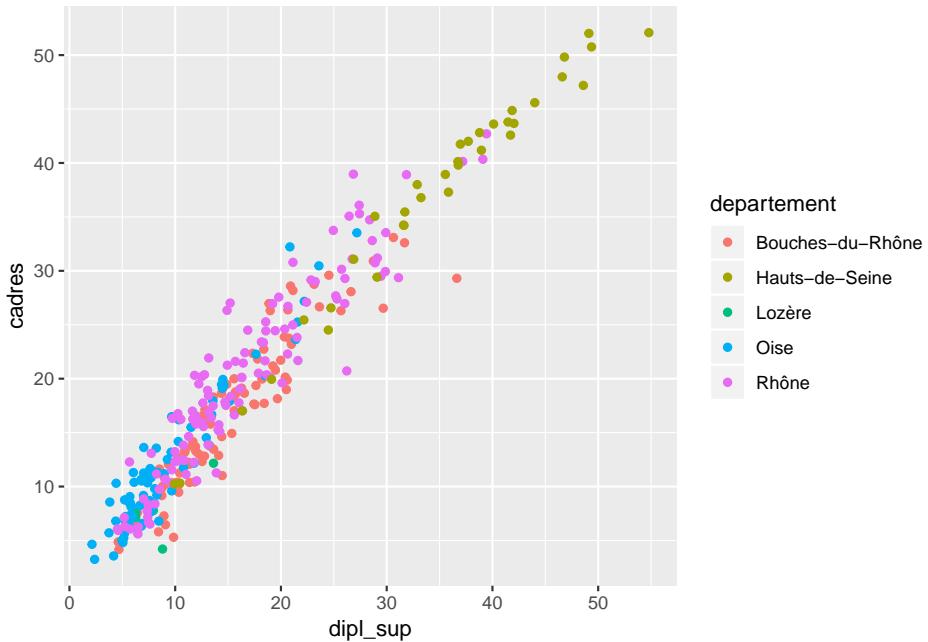


La liste des palettes de `scale_color_brewer` est indiquée en fin de section suivante.

8.7.3.2 Variables qualitatives

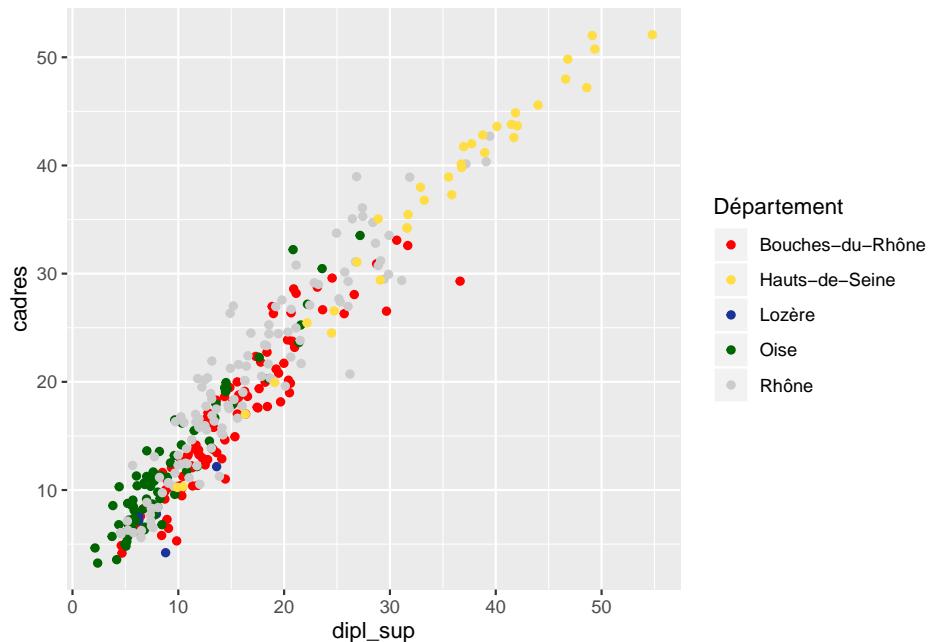
Si on a fait un mappage avec une variable discrète (qualitative), comme ici avec le département :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, color = departement))
```



Une première possibilité est de modifier la palette manuellement avec `scale_color_manual` et son argument `values` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = departement)) +
  scale_color_manual("Département",
                     values = c("red", "#FFDD45", rgb(0.1,0.2,0.6), "darkgreen", "grey80"))
```



L'exemple précédent montre plusieurs manières de définir manuellement des couleurs dans R :

- Par code hexadécimal : "#FFDD45"
- En utilisant la fonction `rgb` et en spécifiant les composantes rouge, vert, bleu par des nombres entre 0 et 1 (et optionnellement une quatrième composante d'opacité, toujours entre 0 et 1) : `rgb(0.1,0.2,0.6)`
- En donnant un nom de couleur : "red", "darkgreen"

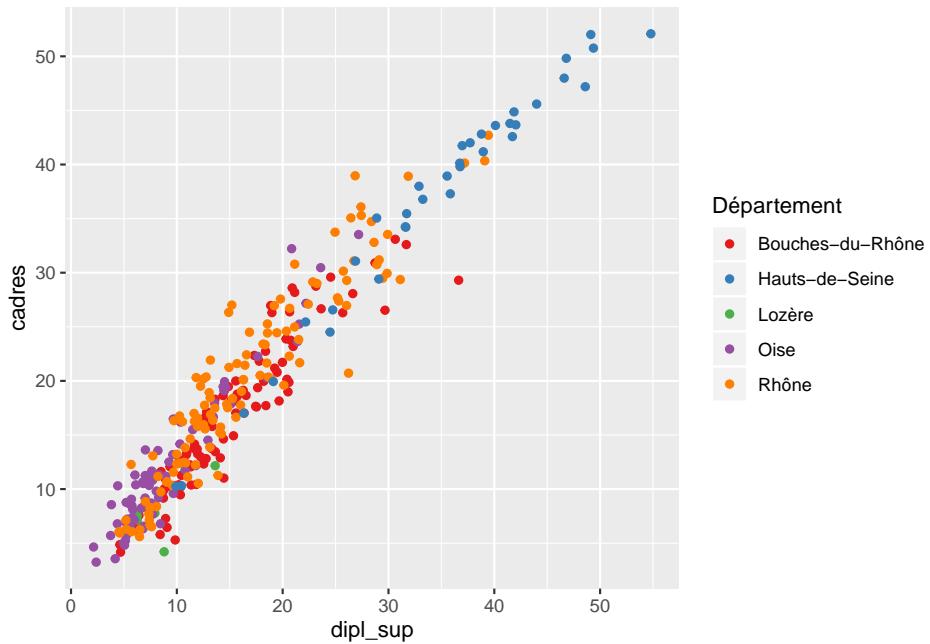
La liste complète des noms de couleurs connus par R peut être obtenu avec la fonction `colors()`. Vous pouvez aussi retrouver en ligne [la liste des couleurs et leur nom \(PDF\)](#).

Il est cependant souvent plus pertinent d'utiliser des palettes prédéfinies. Celles du site [Colorbrewer](#), initialement prévues pour la cartographie, permettent une bonne lisibilité, et peuvent être adaptées pour certains types de daltonisme.

Ces palettes s'utilisent via la fonction `scale_color_brewer`, en passant le nom de la palette via l'argument `palette`. Par exemple, si on veut utiliser la palette `Set1` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = departement)) +
```

```
scale_color_brewer("Département", palette = "Set1")
```



Le graphique suivant, accessible via la fonction `display.brewer.all()`, montre la liste de toutes les palettes disponibles via `scale_color_brewer`. Elles sont réparties en trois familles : les palettes séquentielles (pour une variable quantitative), les palettes qualitatives, et les palettes divergentes (typiquement pour une variable quantitative avec une valeur de référence, souvent 0, et deux palettes continues distinctes pour les valeurs inférieures et pour les valeurs supérieures).

```
RColorBrewer::display.brewer.all()
```



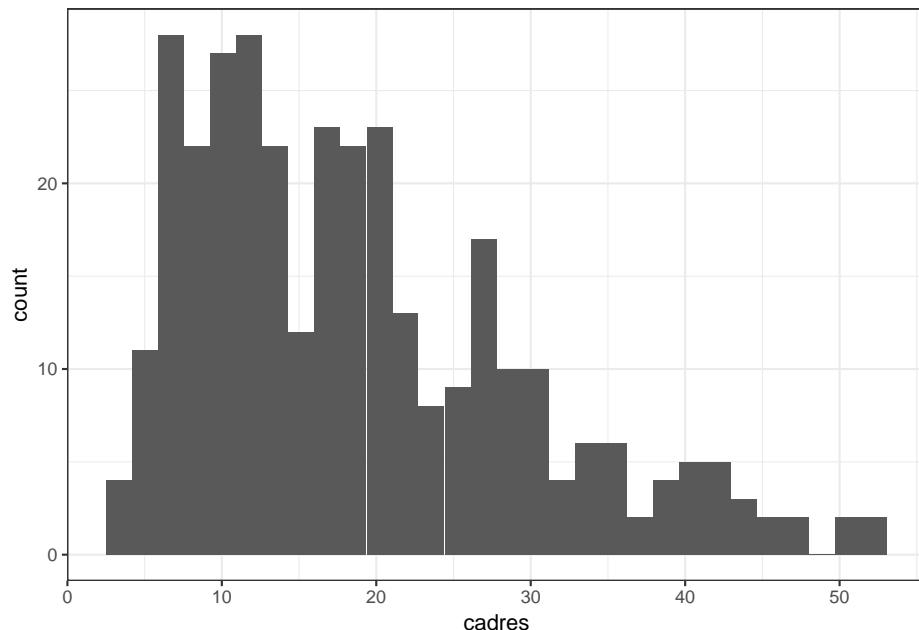
Il existe d'autres méthodes pour définir les couleurs : pour plus d'informations on pourra se reporter à [l'article de la documentation officielle sur ce sujet](#).

8.8 Thèmes

Les thèmes permettent de contrôler l'affichage de tous les éléments du graphique qui ne sont pas reliés aux données : titres, grilles, fonds, etc.

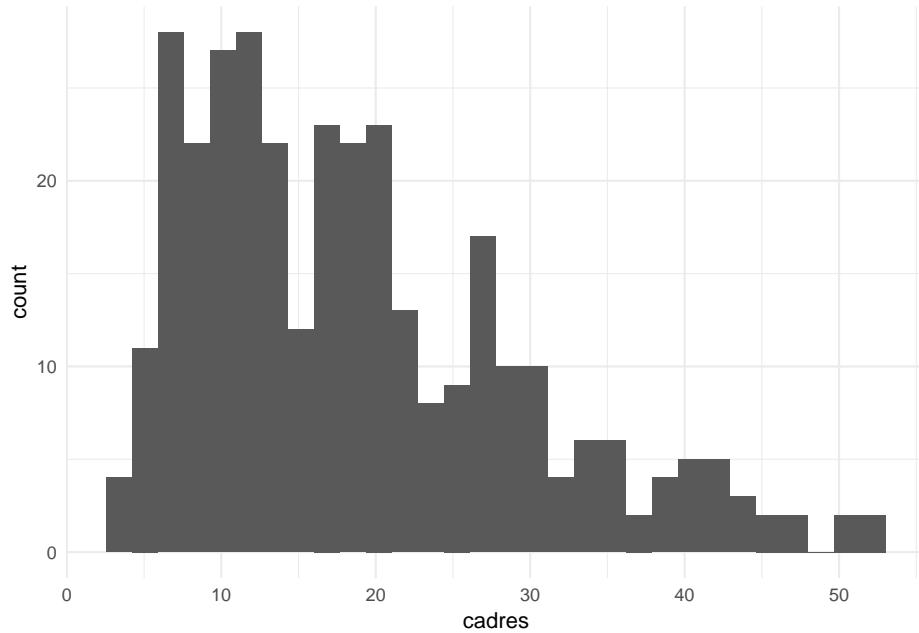
Il existe un certain nombre de thèmes préexistants, par exemple le thème `theme_bw` :

```
ggplot(data = rp) +
  geom_histogram(aes(x = adres)) +
  theme_bw()
```



Ou le thème `theme_minimal` :

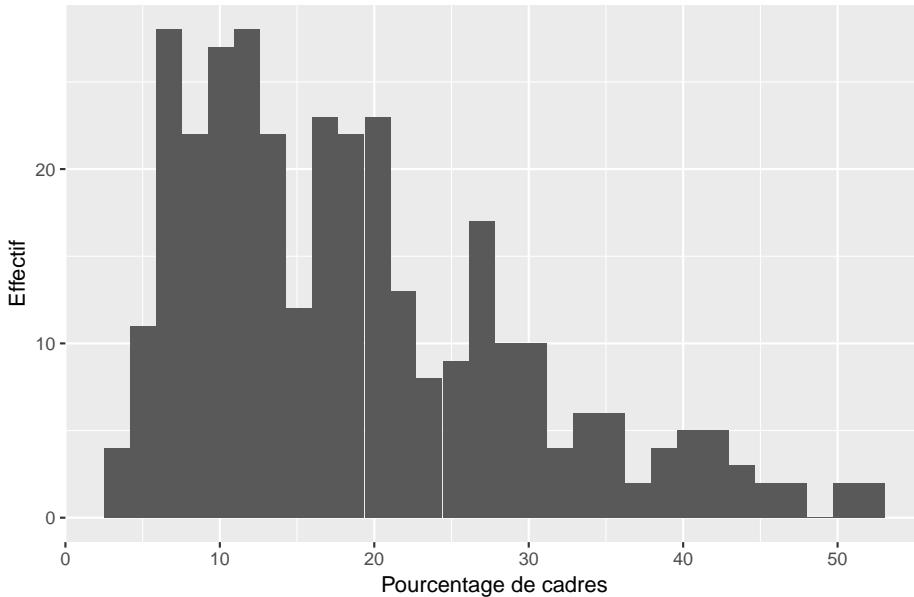
```
ggplot(data = rp) +
  geom_histogram(aes(x = adres)) +
  theme_minimal()
```



On peut cependant modifier manuellement les différents éléments. Par exemple, les fonctions `ggtitle`, `xlab` et `ylab` permettent d'ajouter ou de modifier le titre du graphique, ainsi que les étiquettes des axes x et y :

```
ggplot(data = rp) +  
  geom_histogram(aes(x = cadres)) +  
  ggtitle("Un bien bel histogramme") +  
  xlab("Pourcentage de cadres") +  
  ylab("Effectif")
```

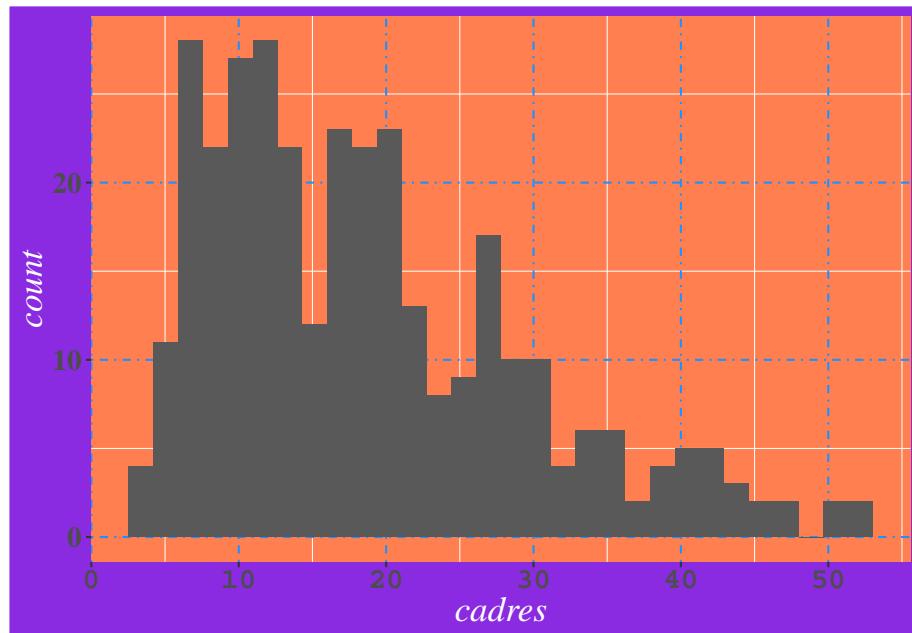
Un bien bel histogramme



Les éléments personnalisables étant nombreux, un bon moyen de se familiariser avec tous les arguments est sans doute l'addin RStudio `ggThemeAssist`. Pour l'utiliser il suffit d'installer le package du même nom, de sélectionner dans son script RStudio le code correspondant à un graphique `ggplot2`, puis d'aller dans le menu *Addins* et choisir *ggplot Theme Assistant*. Une interface graphique s'affiche alors permettant de modifier les différents éléments. Si on clique sur *Done*, le code sélectionné dans le script est alors automatiquement mis à jour pour correspondre aux modifications effectuées.

Ce qui permet d'obtenir très facilement des résultats extrêmement moches :

```
ggplot(data = rp) + geom_histogram(aes(x = cadres)) +
  theme(panel.grid.major = element_line(colour = "dodgerblue",
                                         size = 0.5, linetype = "dotdash"),
        axis.title = element_text(family = "serif",
                                   size = 18, face = "italic", colour = "white"),
        axis.text = element_text(family = "serif",
                                   size = 15, face = "bold"), axis.text.x = element_text(family = "mono"),
        plot.title = element_text(family = "serif"),
        legend.text = element_text(family = "serif"),
        legend.title = element_text(family = "serif"),
        panel.background = element_rect(fill = "coral"),
        plot.background = element_rect(fill = "blueviolet"))
```



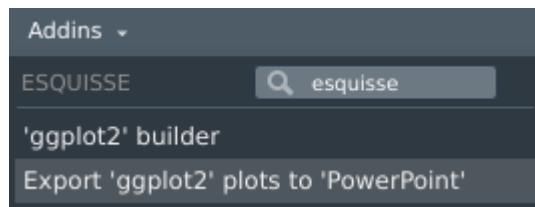
8.9 L'add-in esquisse

`esquisse` est un package développé notamment par [Victor Perrier](#) de dreamRs et qui fournit une interface graphique pour la construction de graphiques avec `ggplot2`.

Pour l'utiliser, il faut évidemment préalablement installer l'extension :

```
install.packages("esquisse")
```

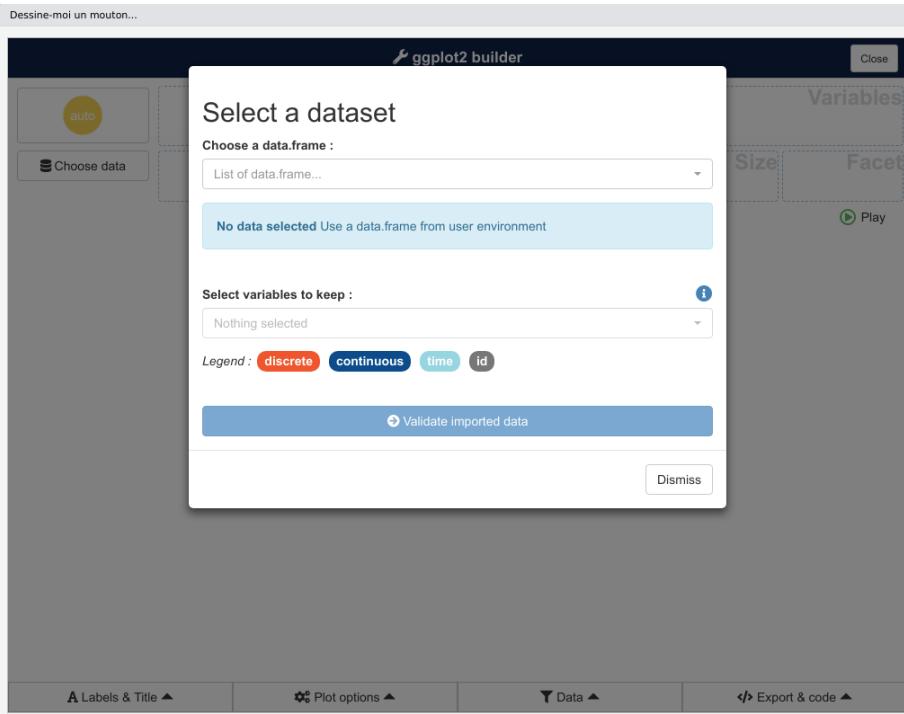
Pour lancer l'interface, ouvrez le menu *Addins* dans la barre d'outils de RStudio, et cliquez sur '`ggplot2`' *builder*¹.



Une fenêtre s'ouvre : la première étape consiste à choisir un *data frame* de

¹Vous pouvez aussi lancer la commande `esquisser::esquisse()` dans la Console.

votre environnement, et éventuellement à ne sélectionner que certaines de ses variables.



Une fois le choix effectué, cliquez sur *Validate imported data*.

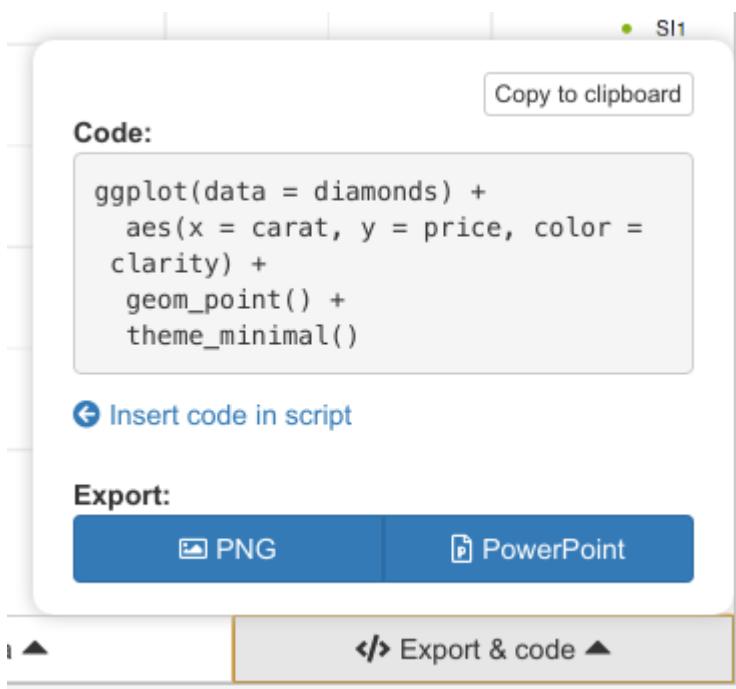
L'interface principale s'affiche alors. La liste des variables du *data frame* apparaît en haut, et vous pouvez les faire glisser dans les zones *X*, *Y*, *Fill*, *Color*, *Size* et *Facet* pour créer des mappages. Le graphique se met automatiquement à jour.



Par défaut, `esquisse` sélectionne le type de graphique le plus approprié selon la nature de vos variables. Mais vous pouvez choisir un autre type de graphique à l'aide de l'icône en haut à gauche.

Enfin, une série de menus en bas de l'interface vous permettent de personnaliser les titres, les labels, la présentation ou de filtrer des valeurs de vos variables.

Quand vous avez généré un graphique que vous souhaitez conserver, ouvrez le menu `Export & code` :



Vous y trouverez le code R correspondant au graphique actuellement affiché. Vous pouvez dès lors le copier pour le coller dans votre script, ou cliquer sur *Insert code in script* pour l'insérer directement dans votre script à l'endroit où se trouve votre curseur.

esquisse ne propose pas (encore) tous les `geom` ou toutes les possibilités de `ggplot2`, mais ça peut être un outil très utile et pratique pour une exploration rapide de données ou lorsqu'on est un peu perdu dans la syntaxe et les fonctions de l'extension.

Pour plus d'informations, vous pouvez vous référer à la [page du projet sur GitHub](#) (en anglais).

8.10 Ressources

[La documentation officielle](#) (en anglais) de `ggplot2` est très complète et accessible en ligne.

Une “antisèche” (en anglais) résumant en deux pages l’ensemble des fonctions et arguments et disponible soit directement depuis RStudio (menu *Help > Cheat-sheets > Data visualization with ggplot2*) ou [en ligne](#).

Les parties [Data visualisation](#) et [Graphics for communication](#) de l’ouvrage en ligne *R for data science*, de Hadley Wickham, sont une très bonne introduction

à `ggplot2`.

Plusieurs ouvrages, toujours en anglais, abordent en détail l'utilisation de `ggplot2`, en particulier [ggplot2: Elegant Graphics for Data Analysis](#), toujours de Hadley Wickham, et le [R Graphics Cookbook](#) de Winston Chang.

Le [site associé](#) à ce dernier ouvrage comporte aussi pas mal d'exemples et d'informations intéressantes.

Enfin, si `ggplot2` présente déjà un très grand nombre de fonctionnalités, il existe aussi un système d'extensions permettant d'ajouter des `geom`, des thèmes, etc. Le site [ggplot2 extensions](#) est une très bonne ressource pour les parcourir et les découvrir, notamment grâce à sa [galerie](#).

8.11 Exercices

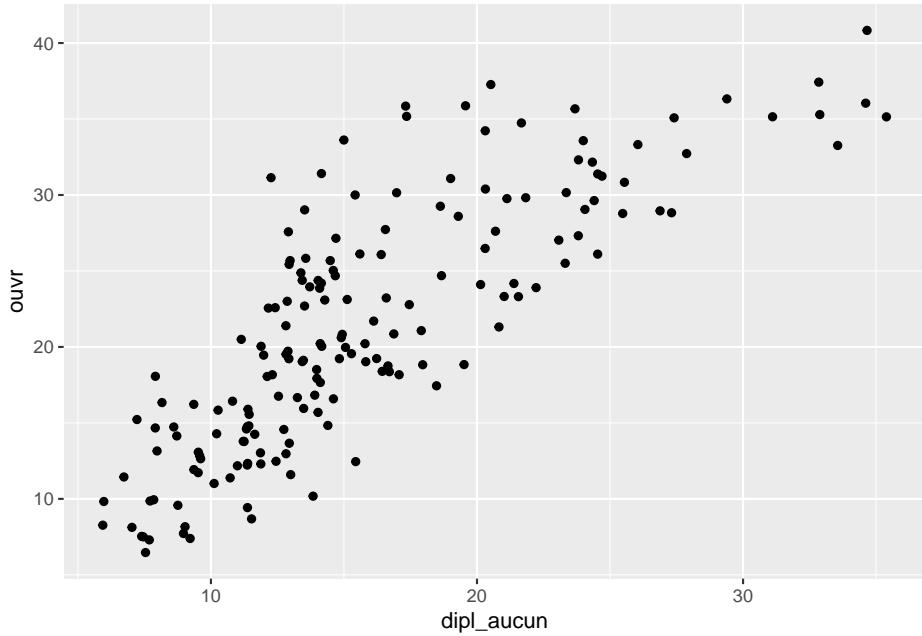
Pour les exercices qui suivent, on commence par charger les extensions nécessaires et les données du jeu de données `rp2012`. On crée alors un objet `rp69` comprenant uniquement les communes du Rhône et de la Loire.

```
library(tidyverse)
library(questionr)
data(rp2012)

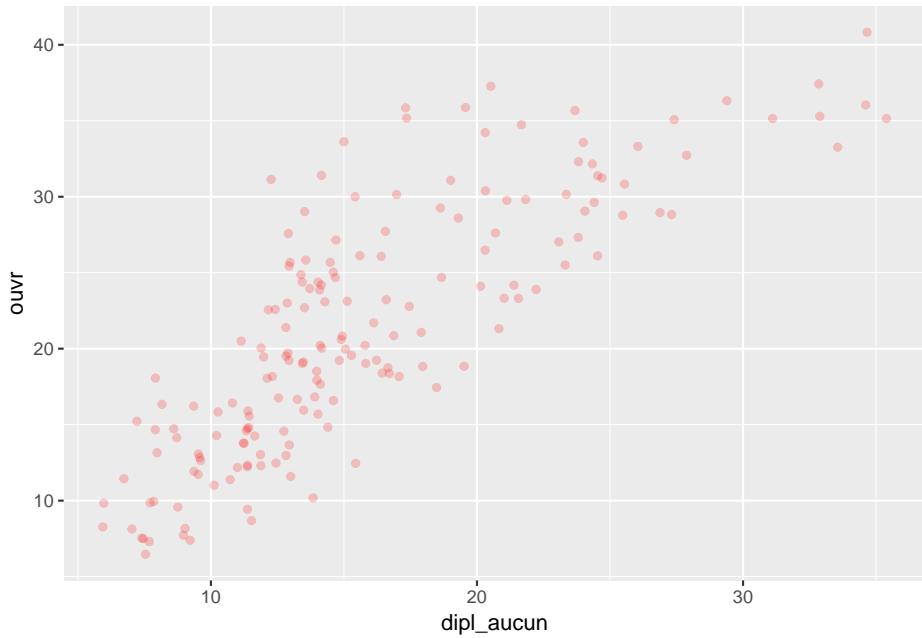
rp69 <- filter(rp2012, département %in% c("Rhône", "Loire"))
```

Exercice 1

Faire un nuage de points croisant le pourcentage de sans diplôme (`dipl_aucun`) et le pourcentage d'ouvriers (`ouvr`).

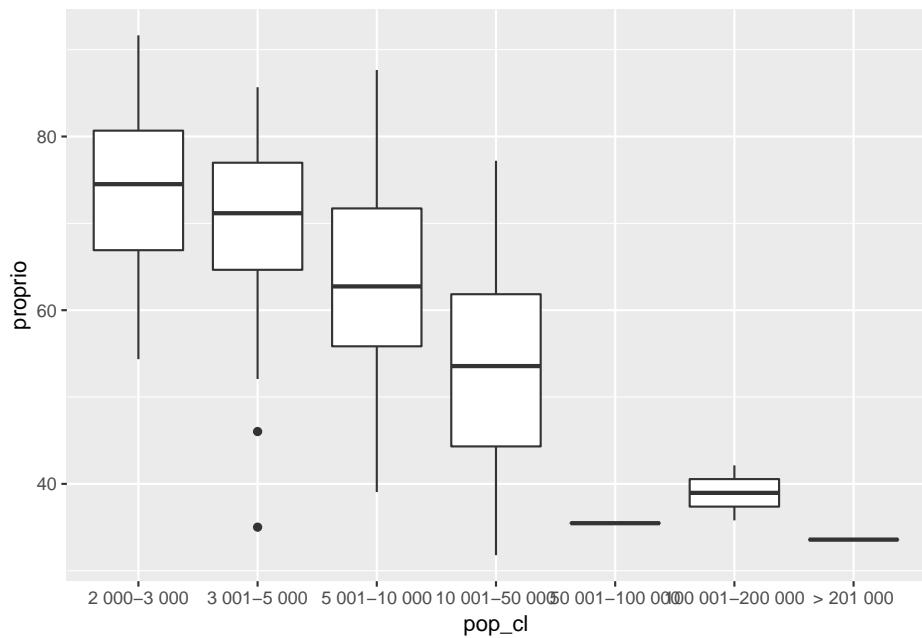
**Exercice 2**

Faire un nuage de points croisant le pourcentage de sans diplôme et le pourcentage d'ouvriers, avec les points en rouge et de transparence 0.2.

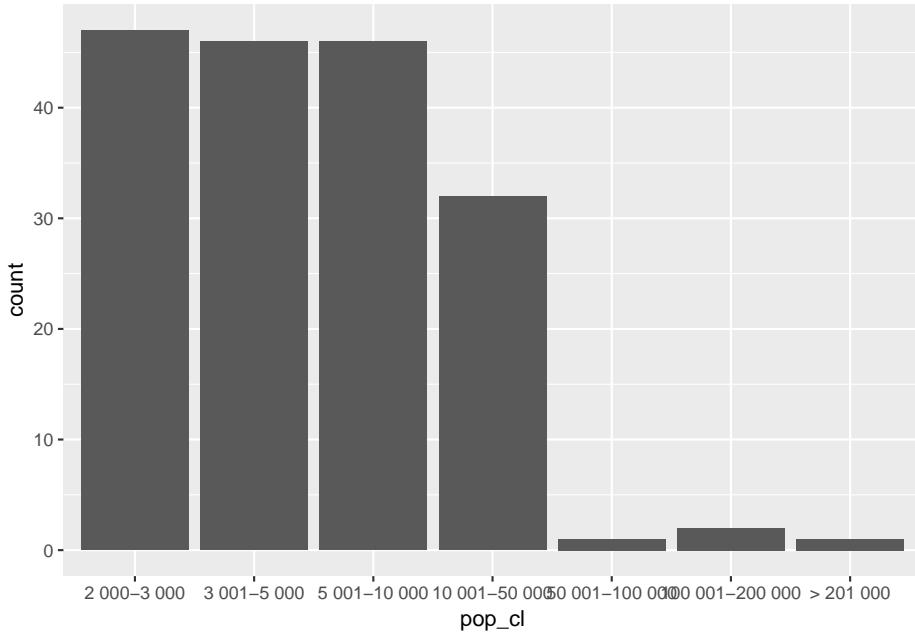


Exercice 3

Représenter la répartition du pourcentage de propriétaires (variable `proprio`) selon la taille de la commune en classes (variable `pop_cl`) sous forme de boîtes à moustache.

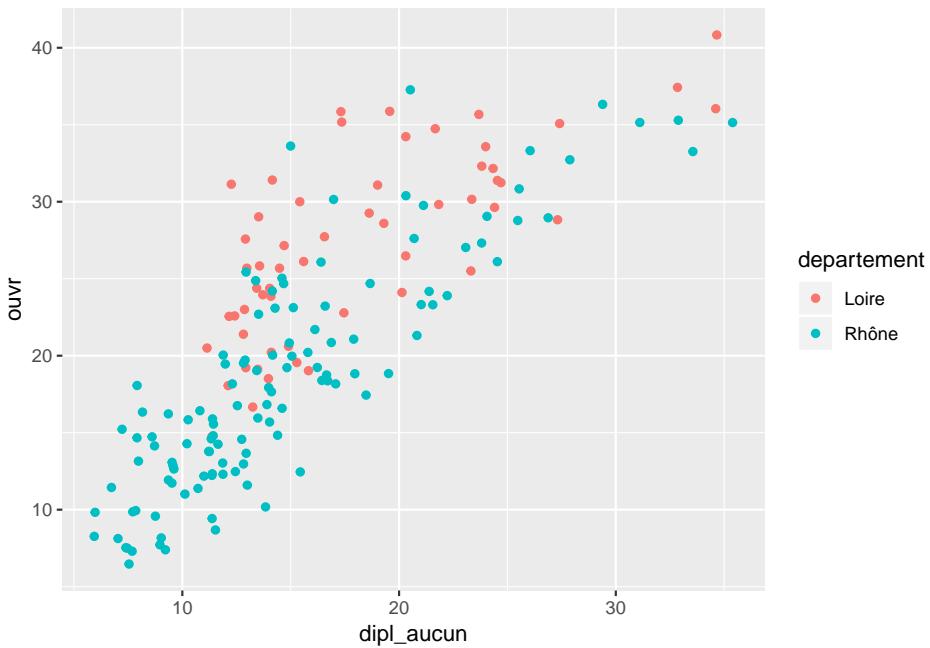
**Exercice 4**

Représenter la répartition du nombre de communes selon la taille de la commune en classes sous la forme d'un diagramme en bâtons.

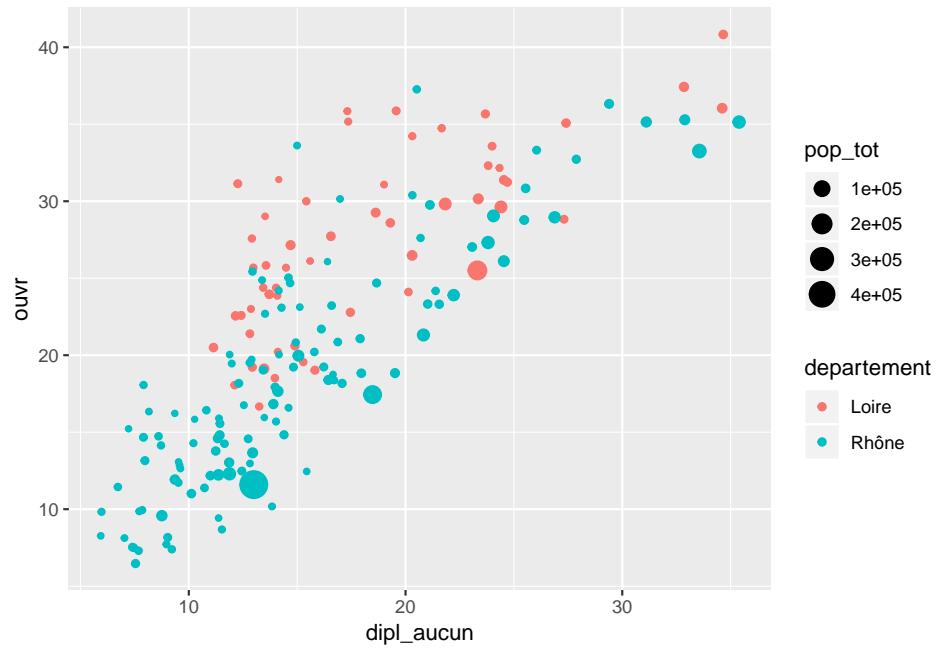


Exercice 5

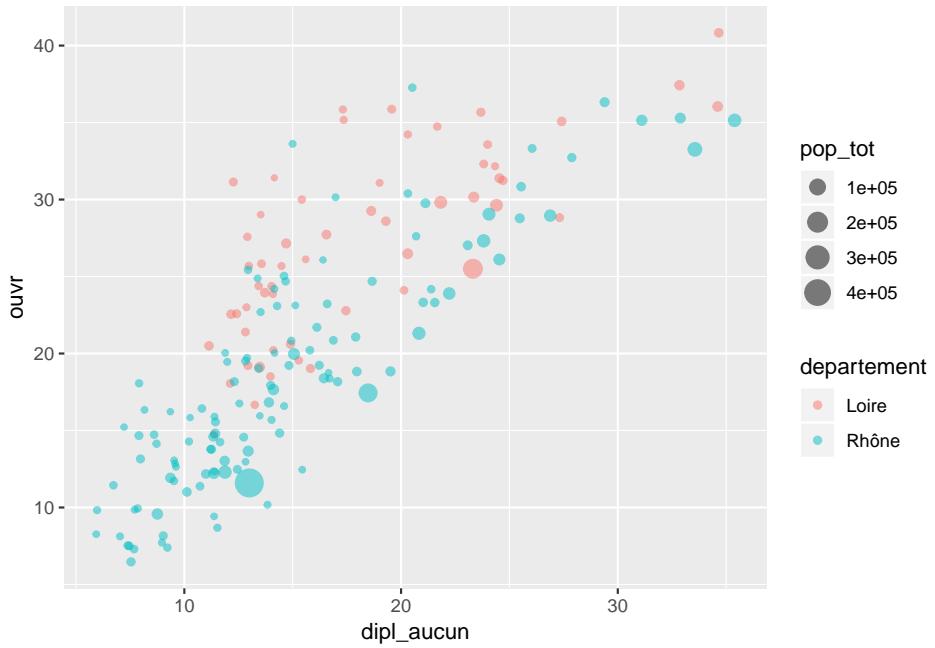
Faire un nuage de points croisant le pourcentage de sans diplôme et le pourcentage d'ouvriers. Faire varier la couleur selon le département (`departement`).



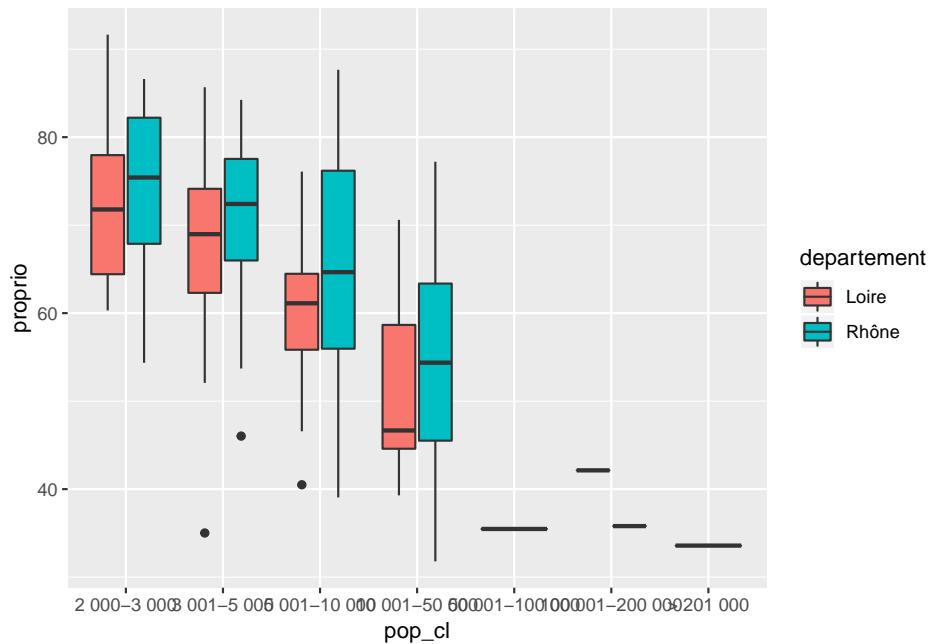
Sur le même graphique, faire varier la taille des points selon la population totale de la commune (`pop_tot`).



Enfin, toujours sur le même graphique, rendre les points transparents en plaçant leur opacité à 0.5.

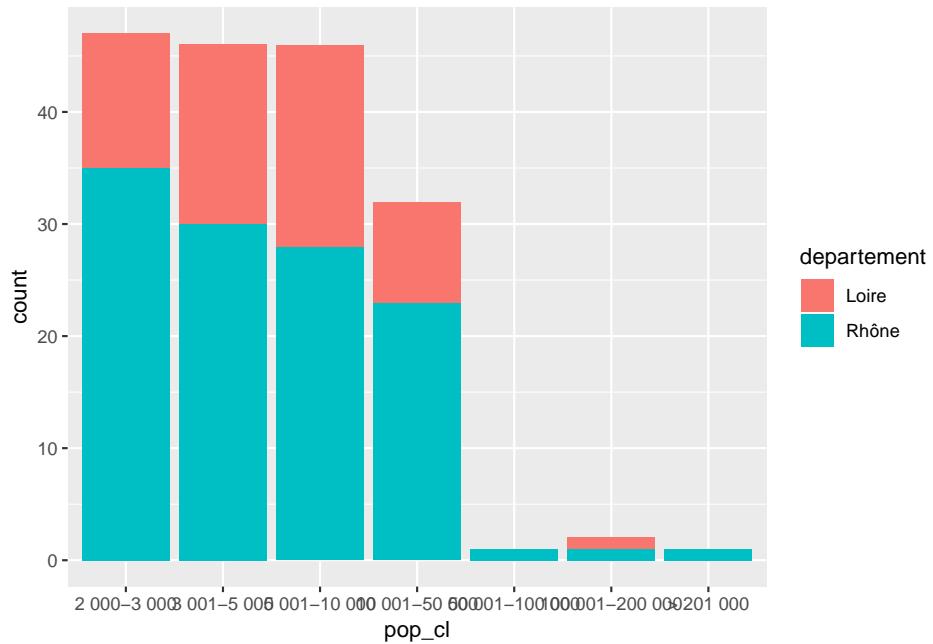
**Exercice 6**

Représenter la répartition du pourcentage de propriétaires (variable `proprio`) selon la taille de la commune en classes (variable `pop_c1`) sous forme de boîtes à moustache. Faire varier la couleur de remplissage (attribut `fill`) selon le département.

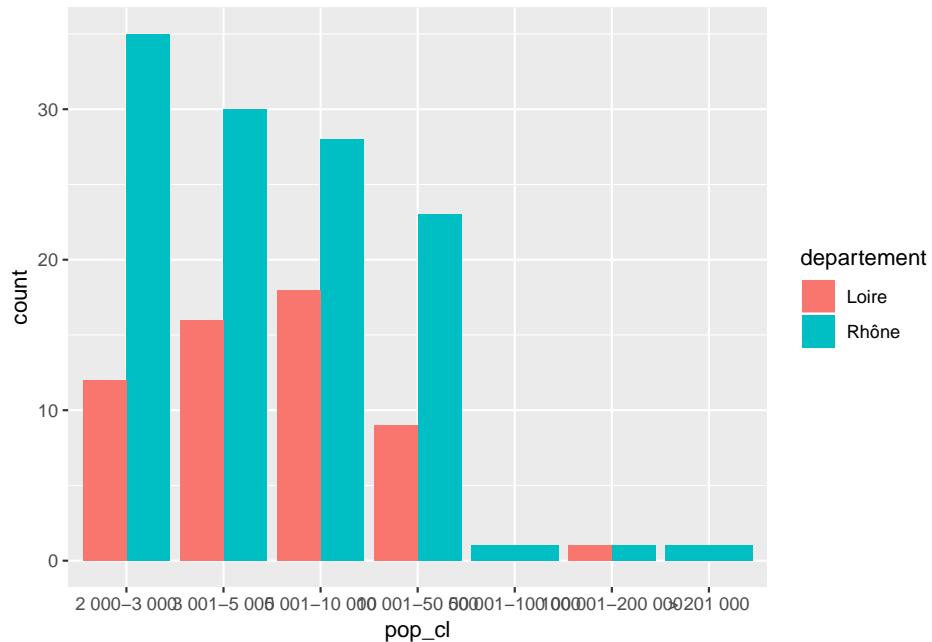


Exercice 7

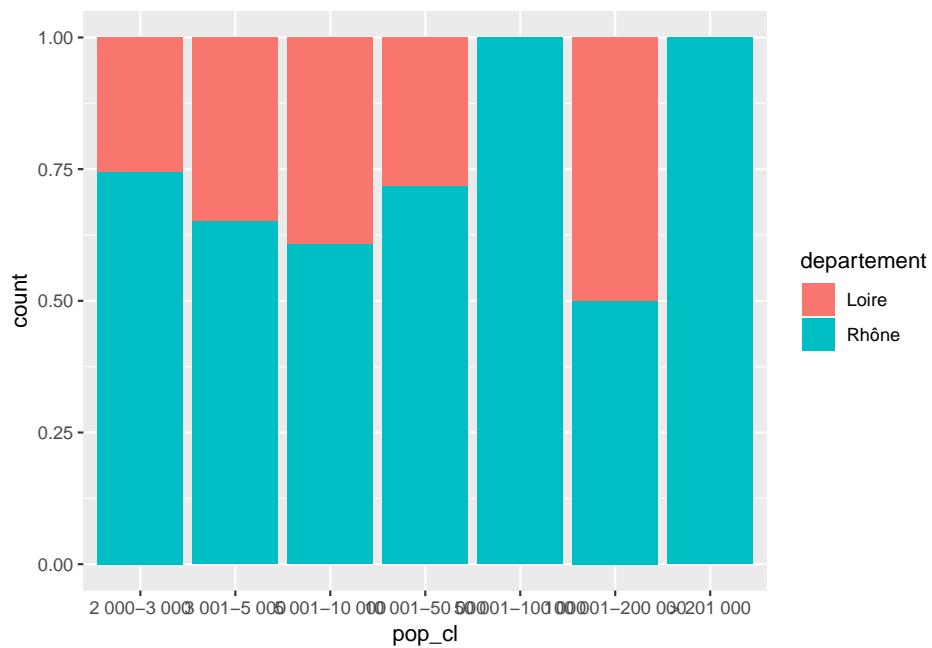
Représenter la répartition du nombre de communes selon la taille de la commune en classes (variable *pop_cl*) sous forme de diagramme en bâtons, avec une couleur différente selon le département.



Faire varier la valeur du paramètre `position` pour afficher les barres les unes à côté des autres.

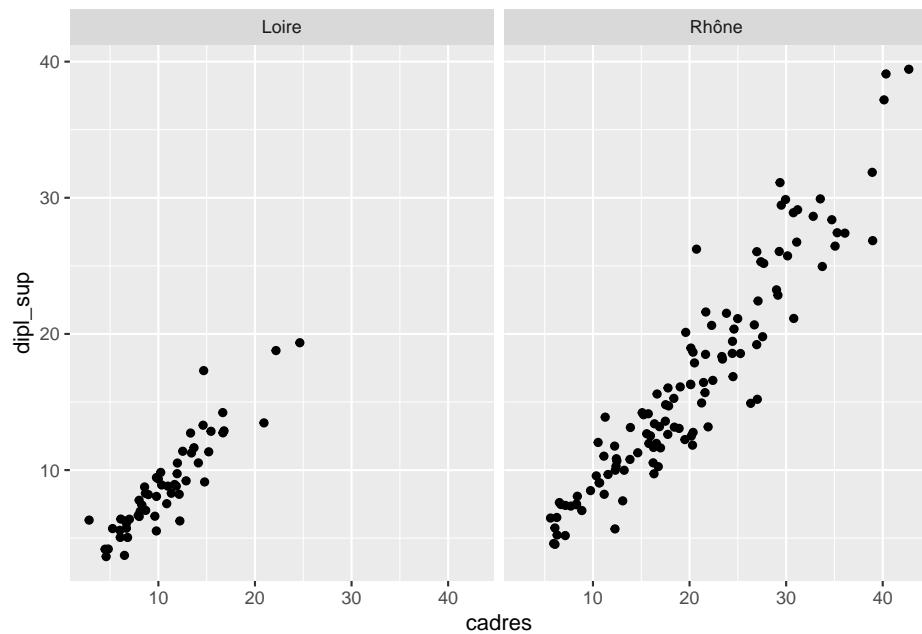


Changer à nouveau la valeur du paramètre `position` pour représenter les proportions de communes de chaque département pour chaque catégorie de taille.

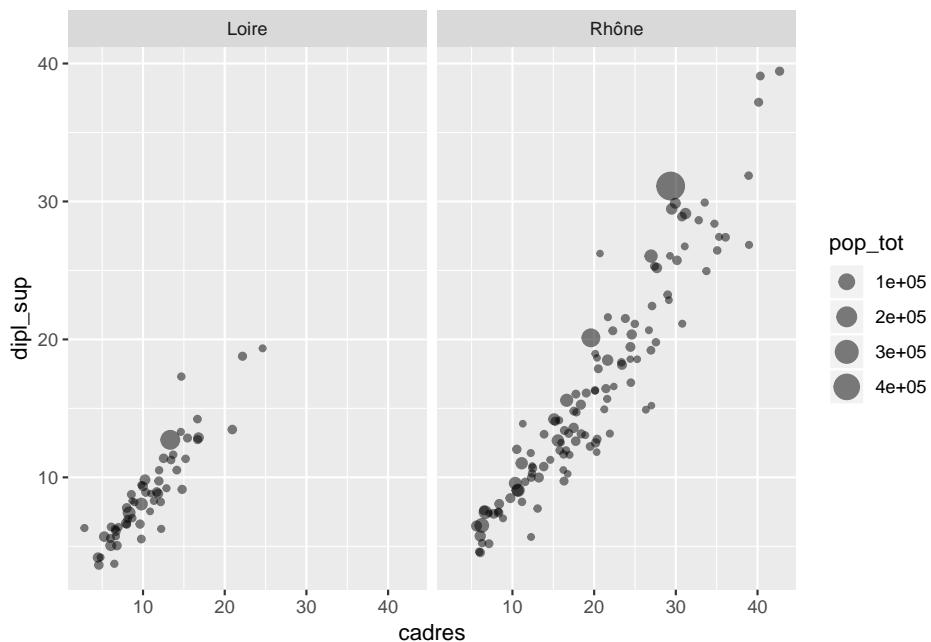


Exercice 8

Faire le nuage de points du pourcentage de cadres (`cadres`) par le pourcentage de diplômés du supérieur (`dipl_sup`). Représenter ce nuage par deux graphiques différents selon le département en utilisant `facet_grid`.

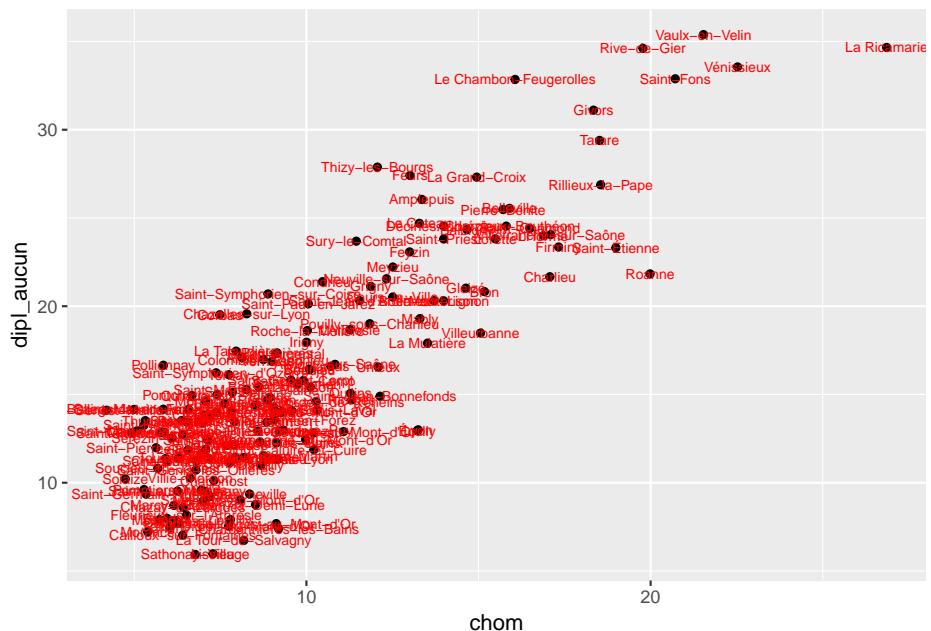


Sur le même graphique, faire varier la taille des points selon la population totale de la communes (variable `pop_tot`) et rendre les points transparents.



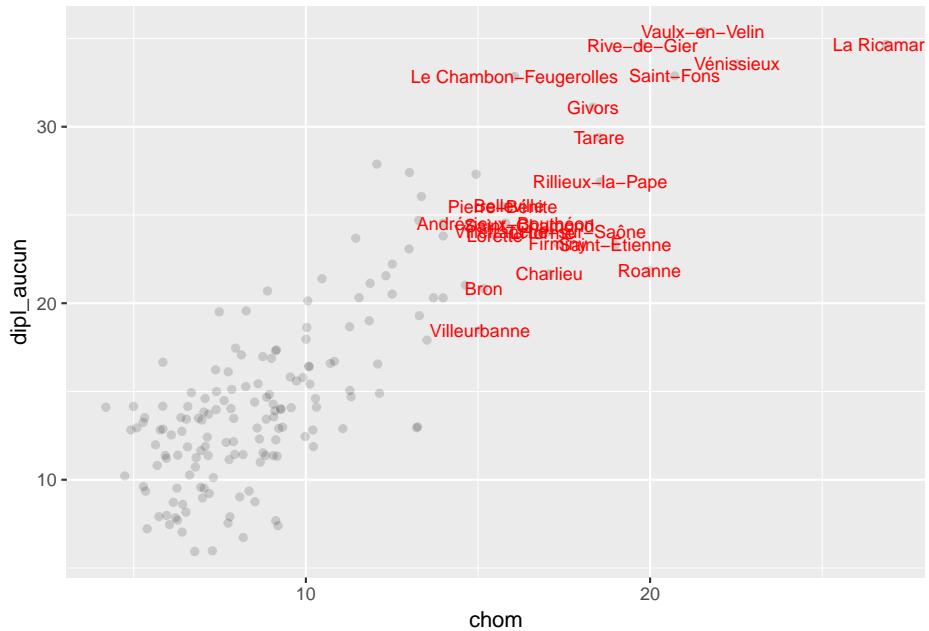
Exercice 9

Faire le nuage de points croisant pourcentage de chômeurs (**chom**) et pourcentage de sans diplôme. Y ajouter les noms des communes correspondant (variable **commune**), en rouge et en taille 2.5 :



Exercice 10

Dans le graphique précédent, n'afficher que le nom des communes ayant plus de 15% de chômage.



Chapitre 9

Recoder des variables

9.1 Rappel sur les variables et les vecteurs

Dans R, une variable, en général une colonne d'un tableau de données, est un objet de type *vecteur*. Un vecteur est un ensemble d'éléments, tous du même type.

On a vu qu'on peut construire un vecteur manuellement de différentes manières :

```
couleur <- c("Jaune", "Jaune", "Rouge", "Vert")
nombres <- 1:10
```

Mais le plus souvent on manipule des vecteurs faisant partie d'une table importée dans R. Dans ce qui suit on va utiliser le jeu de données d'exemple `hdv2003` de l'extension `questionr`.

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

Quand on veut accéder à un vecteur d'un tableau de données, on peut utiliser l'opérateur `$` :

```
hdv2003$qualif
```

On peut facilement créer de nouvelles variables (ou colonnes) dans un tableau de données en utilisant le `$` dans une assignation :

```
hdv2003$minutes.tv <- hdv2003$heures.tv * 60
```

Les vecteurs peuvent être de classes différentes, selon le type de données qu'ils contiennent.

On a ainsi des vecteurs de type `integer` ou `double`, qui contiennent respectivement des nombres entiers ou décimaux :

```
typeof(hdv2003$age)
```

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

```
typeof(hdv2003$heures.tv)
```

```
[1] "double"
```

Des vecteurs de type `character`, qui contiennent des chaînes de caractères :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
typeof(vec)
```

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

Et des vecteurs de type `logical`, qui ne peuvent contenir que les valeurs vraie (`TRUE`) ou fausse (`FALSE`).

```
vec <- c(TRUE, FALSE, FALSE, TRUE)
typeof(vec)
```

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

On peut convertir un vecteur d'un type en un autre en utilisant les fonctions `as.numeric`, `as.character` ou `as.logical`. Les valeurs qui n'ont pas pu être converties sont automatiquement transformées en `NA`.

```
x <- c("1", "2.35", "8.2e+03", "foo")
as.numeric(x)
```

```
Warning: NAs introduits lors de la conversion automatique
```

```
[1] 1.00 2.35 8200.00 NA
```

```
y <- 2:6
as.character(y)
```

```
[1] "2" "3" "4" "5" "6"
```

On peut sélectionner certains éléments d'un vecteur à l'aide de l'opérateur `[]`. La manière la plus simple est d'indiquer la position des éléments qu'on veut sélectionner :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
vec[1,3]
```

```
[1] "Jaune" "Rouge"
```

La sélection peut aussi être utilisée pour modifier certains éléments d'un vecteur, par exemple :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
vec[2] <- "Violet"
vec
```

```
[1] "Jaune" "Violet" "Rouge" "Vert"
```

9.2 Tests et comparaison

Un test est une opération logique de comparaison qui renvoie vrai (`TRUE`) ou faux (`FALSE`) pour chacun des éléments d'un vecteur.

Parmi les opérateurs de comparaison disponibles, on trouve notamment :

- `==` qui teste l'égalité
- `!=` qui teste la différence
- `>`, `<`, `<=`, `>=` qui testent la supériorité ou l'infériorité
- `%in%` qui teste l'appartenance à un ensemble de valeurs

Exemple le plus simple :

```
2 == 3
```

```
[1] FALSE
```

```
2 != 3
```

```
[1] TRUE
```

Exemple appliqué à un vecteur :

```
x <- 1:10
x < 5
```

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

On peut combiner plusieurs tests avec les opérateurs logiques *et* (`&`) et *ou* (`|`). Ainsi, si on veut tester qu'une valeur est comprise entre 3 et 6 inclus, on peut faire :

```
x >= 3 & x <= 6
```

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

Si on veut tester qu'une valeur est égale à "Bleu" ou à "Vert", on peut faire :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
vec == "Jaune" | vec == "Vert"
```

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

À noter que dans ce cas, on peut utiliser l'opérateur `%in%`, qui teste si une valeur fait partie des éléments d'un vecteur :

```
vec %in% c("Jaune", "Vert")
```

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



Attention, si on souhaite tester si une valeur est égale à `NA`, faire `x == NA` ne fonctionnera pas. En effet, fidèle à sa réputation de rigueur informatienne, pour R `NA == NA` ne vaut pas `TRUE` mais... `NA`.

Pour tester l'égalité avec `NA`, il faut utiliser la fonction dédiée `is.na` et faire `is.na(x)`.

Enfin, on peut inverser un test avec l'opérateur *non* (!) :

```
!(vec %in% c("Jaune", "Vert"))
```

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

Les tests sont notamment utilisés par le verbe `filter` de `dplyr` (voir section 10.2.2) qui permet de sélectionner certaines lignes d'un tableau de données. On peut ainsi sélectionner les individus ayant entre 20 et 40 ans en filtrant sur la variable `age` :

```
filter(hdv2003, age >= 20 & age <= 40)
```

Ou sélectionner les personnes ayant comme catégorie socio-professionnelle `Ouvrier specialise` ou `Ouvrier qualifie` en filtrant sur la variable `qualif` :

```
filter(hdv2003, qualif %in% c("Ouvrier specialise", "Ouvrier qualifie"))
```

On peut utiliser les tests pour sélectionner certains éléments d'un vecteur. Si on passe un test à l'opérateur de sélection [], seuls les éléments pour lesquels ce test est vrai seront conservés :

```
x <- c(12, 8, 14, 7, 6, 18)
x[x > 10]
```

```
[1] 12 14 18
```

Enfin, on peut aussi utiliser les tests et la sélection pour modifier les valeurs d'un vecteur. Ainsi, si on assigne une valeur à une sélection, les éléments pour lesquels le test est vrai sont remplacés par cette valeur :

```
x <- c(12, 8, 14, 7, 6, 18)
x[x > 10] <- 100
x
```

```
[1] 100    8 100    7    6 100
```

En utilisant cette assignation via un test, on peut effectuer des recodages de variables. Soit le vecteur suivant :

```
vec <- c("Femme", "Homme", "Femme", "Garçon")
```

Si on souhaite recoder la modalité “Garçon” en “Homme”, on peut utiliser la syntaxe suivante :

```
vec[vec == "Garçon"] <- "Homme"
vec
```

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

Cette syntaxe est tout à fait valable et couramment utilisée, cependant dans la section suivante on va voir différentes fonctions qui facilitent ces opérations de recodage.

9.3 Recoder une variable qualitative

Pour rappel, on appelle variable qualitative une variable pouvant prendre un nombre limité de modalités (de valeurs possibles).

9.3.1 Facteurs et `forcats`

Dans R, les variables qualitatives peuvent être de deux types : ou bien des vecteurs de type `character` (des chaînes de caractères), ou bien des `factor` (facteurs). Si vous utilisez les fonctions des extensions du *tidyverse* comme `readr`, `readxl` ou `haven` pour importer vos données, vos variables qualitatives seront importées sous forme de `character`. Mais dans les autres cas elles se retrouveront souvent sous forme de `factor`. C'est le cas dans notre jeu de données d'exemple :

```
class(hdv2003$qualif)
```

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

Les facteurs sont un type de variable ne pouvant prendre qu'un nombre défini de modalités nommés *levels* :

```
levels(hdv2003$qualif)
```

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

Ceci complique les opérations de recodage car du coup l'opération suivante, qui tente de modifier une modalité de la variable, aboutit à un avertissement, et l'opération n'est pas effectuée :

```
hdv2003$qualif[hdv2003$qualif == "Ouvrier specialise"] <- "Ouvrier"
```

```
Warning in `<- .factor`(`*tmp*`, hdv2003$qualif == "Ouvrier specialise", :  
invalid factor level, NA generated
```

`forcats` est une extension facilitant la manipulation des variables qualitatives, qu'elles soient sous forme de vecteurs `character` ou de facteurs. Elle fait partie du *tidyverse*, et est donc automatiquement chargée par :

```
library(tidyverse)
```

9.3.2 Modifier les modalités d'une variable qualitative

Une opération courante consiste à modifier les valeurs d'une variable qualitative, que ce soit pour avoir des intitulés plus courts ou plus clairs ou pour regrouper des modalités entre elles.

Il existe plusieurs possibilités pour effectuer ce type de recodage, mais ici on va utiliser la fonction `fct_recode` de l'extension `forcats`. Celle-ci prend en argument une liste de recodages sous la forme "`Nouvelle valeur`" = "`Ancienne valeur`".

Un exemple :

```
f <- c("Pomme", "Poire", "Pomme", "Cerise")
f <- fct_recode(f,
                 "Fraise" = "Pomme",
                 "Ananas" = "Poire")
f
```

```
[1] Fraise Ananas Fraise Cerise
Levels: Cerise Ananas Fraise
```

Autre exemple sur une “vraie” variable :

```
freq(hdv2003$qualif)
```

	n	%	val%
Ouvrier specialise	0	0.0	0.0
Ouvrier qualifie	292	14.6	20.1
Technicien	86	4.3	5.9
Profession intermediaire	160	8.0	11.0
Cadre	260	13.0	17.9
Employe	594	29.7	41.0
Autre	58	2.9	4.0
NA	550	27.5	NA

```
hdv2003$qualif5 <- fct_recode(hdv2003$qualif,
                                "Ouvrier" = "Ouvrier specialise",
                                "Ouvrier" = "Ouvrier qualifie",
                                "Interm" = "Technicien",
                                "Interm" = "Profession intermediaire")
```

```
freq(hdv2003$qualif5)
```

	n	% val%
Ouvrier	292	14.6 20.1
Interm	246	12.3 17.0
Cadre	260	13.0 17.9
Employe	594	29.7 41.0
Autre	58	2.9 4.0
NA	550	27.5 NA

Attention, les anciennes valeurs saisies doivent être exactement égales aux valeurs des modalités de la variable recodée : toute différence d'accent ou d'espace fera que ce recodage ne sera pas pris en compte. Dans ce cas, `forcats` affiche un avertissement nous indiquant qu'une valeur saisie n'a pas été trouvée dans les modalités de la variable :

```
hdv2003$qualif_test <- fct_recode(hdv2003$qualif,
                                    "Ouvrier" = "Ouvrier spécialisé",
                                    "Ouvrier" = "Ouvrier qualifié")
```

```
Warning: Unknown levels in `f`: Ouvrier spécialisé, Ouvrier qualifié
```

Si on souhaite recoder une modalité de la variable en `NA`, il faut (contre intuitivement) lui assigner la valeur `NULL` :

```
hdv2003$qualif_rec <- fct_recode(hdv2003$qualif, NULL = "Autre")
```

```
freq(hdv2003$qualif_rec)
```

	n	% val%
--	---	--------

Ouvrier specialise	0	0.0	0.0
Ouvrier qualifie	292	14.6	21.0
Technicien	86	4.3	6.2
Profession intermediaire	160	8.0	11.5
Cadre	260	13.0	18.7
Employe	594	29.7	42.7
NA	608	30.4	NA

À l'inverse, si on souhaite recoder les NA d'une variable, on utilisera la fonction `fct_explicit_na`, qui convertit toutes les valeurs manquantes (NA) d'un facteur en une modalité spécifique :

```
hdv2003$qualif_rec <- fct_explicit_na(hdv2003$qualif, na_level = "(Manquant)")
```

```
freq(hdv2003$qualif_rec)
```

	n	%	val%
Ouvrier specialise	0	0.0	0.0
Ouvrier qualifie	292	14.6	14.6
Technicien	86	4.3	4.3
Profession intermediaire	160	8.0	8.0
Cadre	260	13.0	13.0
Employe	594	29.7	29.7
Autre	58	2.9	2.9
(Manquant)	550	27.5	27.5

D'autres fonctions sont proposées par `forcats` pour faciliter certains recodage, comme `fct_collapse`, qui propose une autre syntaxe pratique quand on doit regrouper ensemble des modalités :

```
hdv2003$qualif_rec <- fct_collapse(hdv2003$qualif,
                                      "Ouvrier" = c("Ouvrier specialise", "Ouvrier qualifie",
                                      "Interm" = c("Technicien", "Profession intermediaire"))
```

```
freq(hdv2003$qualif_rec)
```

n	%	val%
---	---	------

```
Ouvrier 292 14.6 20.1
Interm 246 12.3 17.0
Cadre 260 13.0 17.9
Employe 594 29.7 41.0
Autre 58 2.9 4.0
NA 550 27.5 NA
```

`fct_other`, qui regroupe une liste de modalités en une seule modalité “Other” :

```
hdv2003$qualif_rec <- fct_other(hdv2003$qualif,
                                     drop = c("Ouvrier specialise", "Ouvrier qualifie",
                                             "Cadre", "Autre"))
```

```
freq(hdv2003$qualif_rec)
```

	n	%	val%
Technicien	86	4.3	5.9
Profession intermediaire	160	8.0	11.0
Employe	594	29.7	41.0
Other	610	30.5	42.1
NA	550	27.5	NA

`fct_lump`, qui regroupe automatiquement les modalités les moins fréquentes en une seule modalité “Other” (avec possibilité d’indiquer des seuils de regroupement) :

```
hdv2003$qualif_rec <- fct_lump(hdv2003$qualif)
```

```
freq(hdv2003$qualif_rec)
```

	n	%	val%
Ouvrier qualifie	292	14.6	20.1
Profession intermediaire	160	8.0	11.0
Cadre	260	13.0	17.9
Employe	594	29.7	41.0
Other	144	7.2	9.9
NA	550	27.5	NA

9.3.2.1 Interface graphique de recodage

L'extension `questionr` propose une interface graphique facilitant le recodage des modalités d'une variable qualitative. L'objectif est de permettre à l'utilisateur 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 `irec()` en lui passant comme paramètre la variable à recoder.

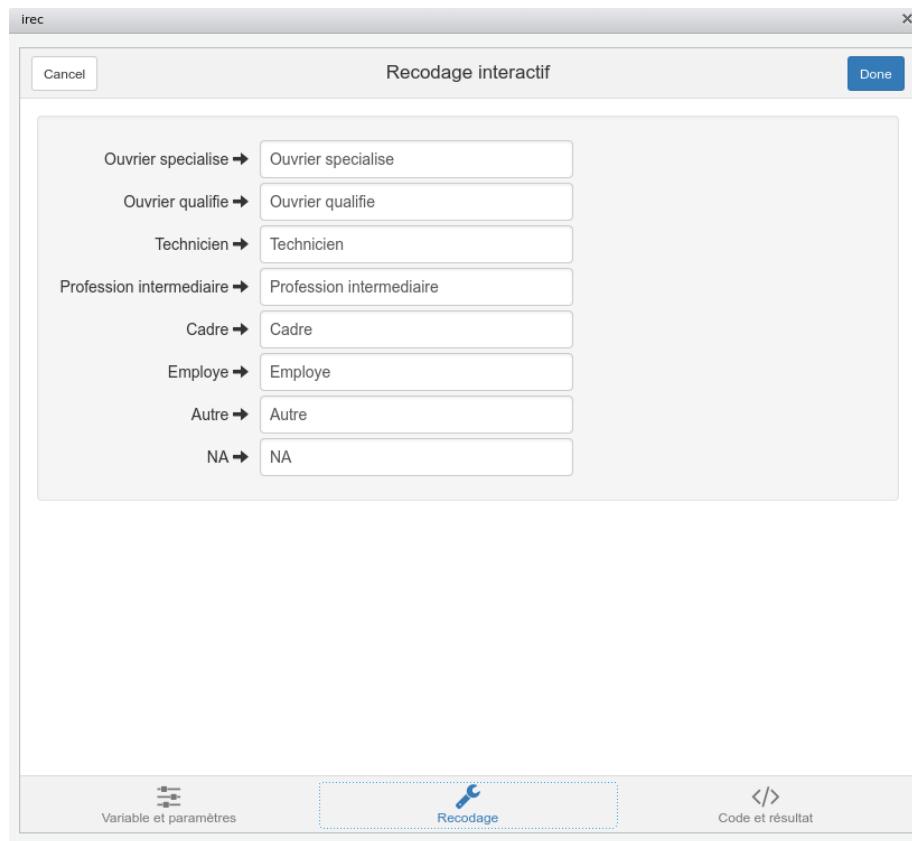


Figure 9.1: Interface graphique de `irec`

L'interface se compose de trois onglets : l'onglet *Variable et paramètres* vous permet de sélectionner la variable à recoder, le nom de la nouvelle variable et d'autres paramètres, l'onglet *Recodages* vous permet de saisir les nouvelles

valeurs des modalités, et l'onglet *Code et résultat* affiche le code R correspondant ainsi qu'un tableau permettant de vérifier les résultats.

Une fois votre recodage terminé, cliquez sur le bouton *Done* et le code R sera inséré dans votre script R ou affiché dans la console.



Attention, cette interface est prévue pour ne pas modifier vos données. C'est donc à vous d'exécuter le code généré pour que le recodage soit réellement effectif.

9.3.3 Ordonner les modalités d'une variable qualitative

L'avantage des facteurs (par rapport aux vecteurs de type `character`) est que leurs modalités peuvent être ordonnées, ce qui peut faciliter la lecture de tableaux ou graphiques.

On peut ordonner les modalités d'un facteur manuellement, par exemple avec la fonction `fct_relevel()` de l'extension `forcats` :

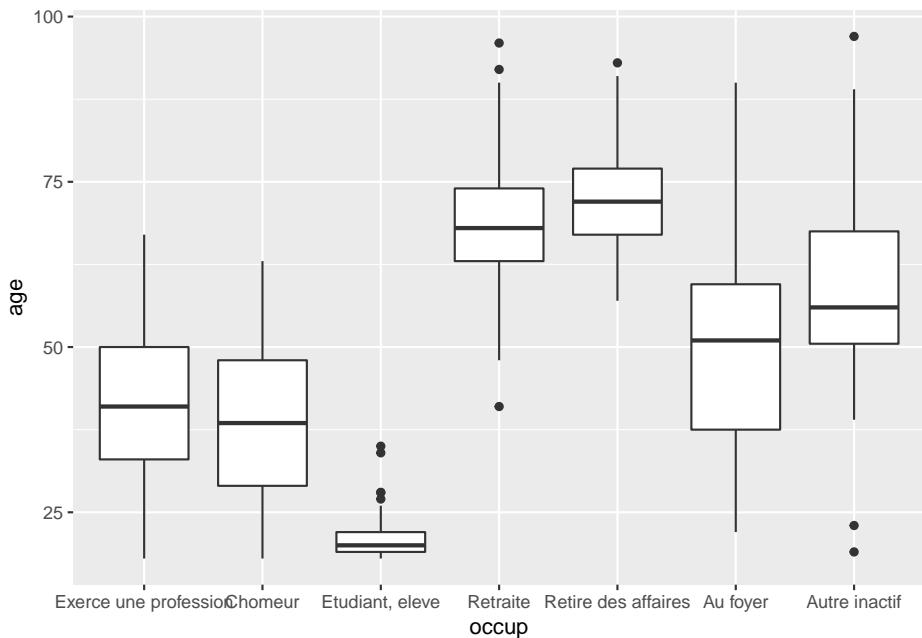
```
hdv2003$qualif_rec <- fct_relevel(hdv2003$qualif,
                                      "Cadre", "Profession intermediaire", "Technicien",
                                      "Employe", "Ouvrier qualifie", "Ouvrier specialise",
                                      "Autre")
```

```
freq(hdv2003$qualif_rec)
```

	n	%	val%
Cadre	260	13.0	17.9
Profession intermediaire	160	8.0	11.0
Technicien	86	4.3	5.9
Employe	594	29.7	41.0
Ouvrier qualifie	292	14.6	20.1
Ouvrier specialise	0	0.0	0.0
Autre	58	2.9	4.0
NA	550	27.5	NA

Une autre possibilité est d'ordonner les modalités d'un facteur selon les valeurs d'une autre variable. Par exemple, si on représente le boxplot de la répartition de l'âge selon le statut d'occupation :

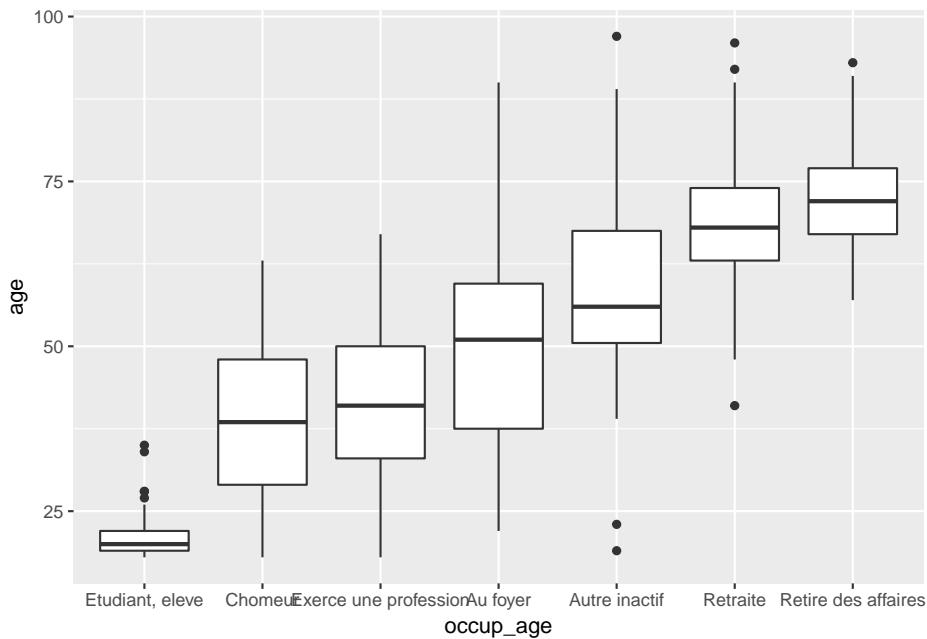
```
library(ggplot2)
ggplot(hdv2003) +
  geom_boxplot(aes(x=occup, y=age))
```



Le graphique pourrait être plus lisible si les modalités étaient triées par âge median croissant. Ceci est possible en utilisant `fct_reorder`. Celle-ci prend 3 arguments : le facteur à réordonner, la variable dont les valeurs doivent être utilisées pour ce réordonnement, et enfin une fonction à appliquer à cette deuxième variable.

```
hdv2003$occup_age <- fct_reorder(hdv2003$occup, hdv2003$age, median)

ggplot(hdv2003) +
  geom_boxplot(aes(x = occup_age, y = age))
```

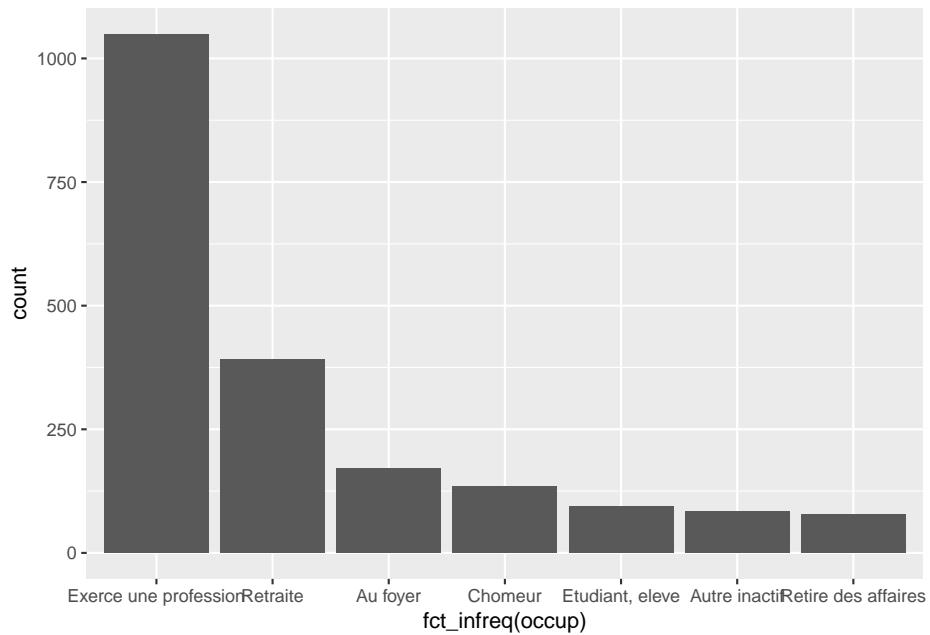


On peut aussi effectuer le réordonnement directement dans l'appel à `ggplot2`, sans créer de nouvelle variable :

```
ggplot(hdv2003) +
  geom_boxplot(aes(x = fct_reorder(occup, age, median),
                  y = age))
```

Lorsqu'on effectue un diagramme en barres avec `geom_bar`, on peut aussi réordonner les modalités selon leurs effectifs à l'aide de `fct_infreq` :

```
ggplot(hdv2003) +
  geom_bar(aes(x = fct_infreq(occup)))
```



9.3.3.1 Interface graphique

`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 `iorder()` en lui passant comme paramètre le facteur à réordonner.

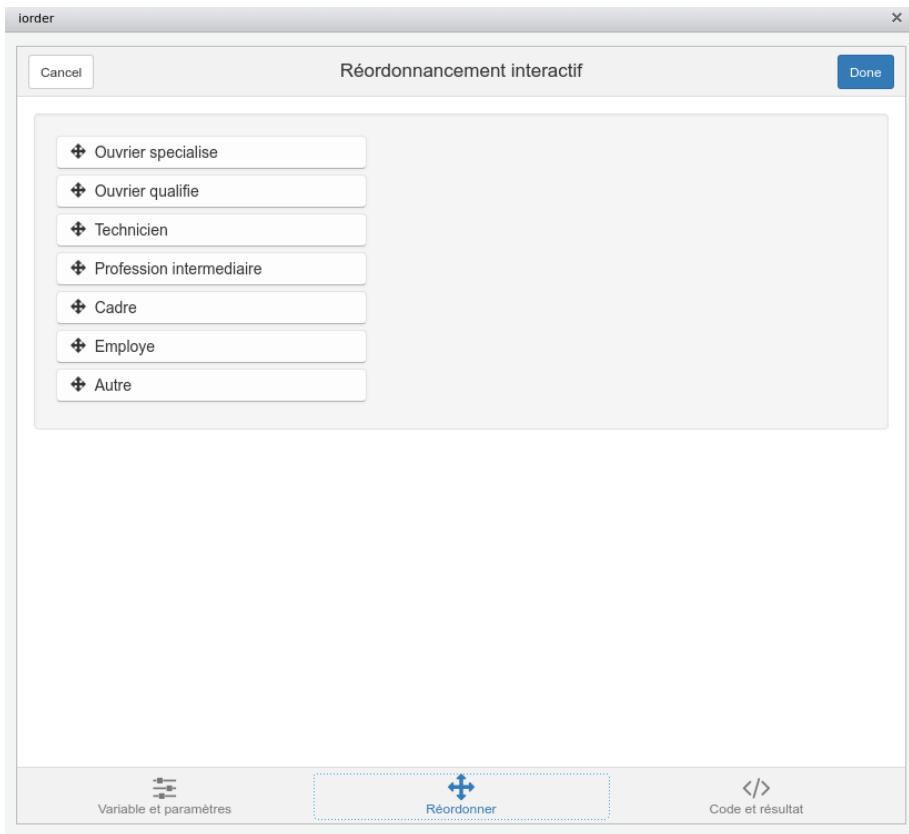


Figure 9.2: Interface graphique de `iorder`

Le fonctionnement de l'interface est similaire à celui de [l'interface de recodage](#). Vous pouvez réordonner les modalités en les faisant glisser avec la souris, puis récupérer et exécuter le code R généré.

9.4 Combiner plusieurs variables

Parfois, on veut créer une nouvelle variable en partant des valeurs d'une ou plusieurs autres variables. Dans ce cas on peut utiliser les fonctions `if_else` pour les cas les plus simples, ou `case_when` pour les cas plus complexes. Ces deux fonctions sont incluses dans l'extension `dplyr`, qu'il faut donc avoir chargé précédemment.

9.4.1 if_else

`if_else` prend trois arguments : un test, une valeur à renvoyer si le test est vrai, et une valeur 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$statut <- if_else(hdv2003$sex == "Homme" & hdv2003$age > 60,
                            "Homme de plus de 60 ans",
                            "Autre")
```

```
freq(hdv2003$statut)
```

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

9.4.2 case_when

`case_when` est une génération du `if_else` qui permet d'indiquer plusieurs tests et leurs valeurs associées.

Imaginons qu'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$statut <- case_when(
  hdv2003$age > 60 & hdv2003$sex == "Homme" ~ "Homme de plus de 60 ans",
```

```
hdv2003$age > 60 & hdv2003$sex == "Femme" ~ "Femme de plus de 60 ans",
TRUE ~ "Autre")
```

```
freq(hdv2003$statut)
```

	n	%	val%
Autre	1512	75.6	75.6
Femme de plus de 60 ans	266	13.3	13.3
Homme de plus de 60 ans	222	11.1	11.1

`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 renvoie 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.



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$statut <- case_when(
  hdv2003$sex == "Homme" ~ "Homme",
  hdv2003$sex == "Homme" & hdv2003$age > 60 ~ "Homme de plus de 60 ans",
  TRUE ~ "Autre")
```

```
freq(hdv2003$statut)
```

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

Comme la condition `sex == "Homme"` est plus générale que `sex == "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$statut <- case_when(
  hdv2003$sexe == "Homme" & hdv2003$age > 60 ~ "Homme de plus de 60 ans",
  hdv2003$sexe == "Homme" ~ "Homme",
  TRUE ~ "Autre")
```

```
freq(hdv2003$statut)
```

	n	%	val%
Autre	1101	55.0	55.0
Homme	677	33.9	33.9
Homme de plus de 60 ans	222	11.1	11.1

9.5 Découper une variable numérique en classes

Une autre opération relativement courante consiste à découper une variable numérique en classes. Par exemple, on voudra transformer une variable *revenu* contenant le revenu mensuel en une variable qualitative avec des catégories *Moins de 500 euros*, *500-1000 euros*, etc.

On utilise pour cela la fonction `cut()` :

```
hdv2003$agecl <- cut(hdv2003$age, breaks = 5)
```

```
freq(hdv2003$agecl)
```

	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

Si on donne un nombre entier à l'argument `breaks`, un nombre correspondant de classes d'amplitudes égales sont automatiquement calculées. Il est souvent

préférable cependant d'avoir des limites "rondes", on peut alors spécifier ces dernières manuellement en passant un vecteur à **breaks** :

```
hdv2003$agecl <- cut(hdv2003$age,  
                      breaks = c(18, 25, 35, 45, 55, 65, 97),  
                      include.lowest = TRUE)
```

```
freq(hdv2003$agecl)
```

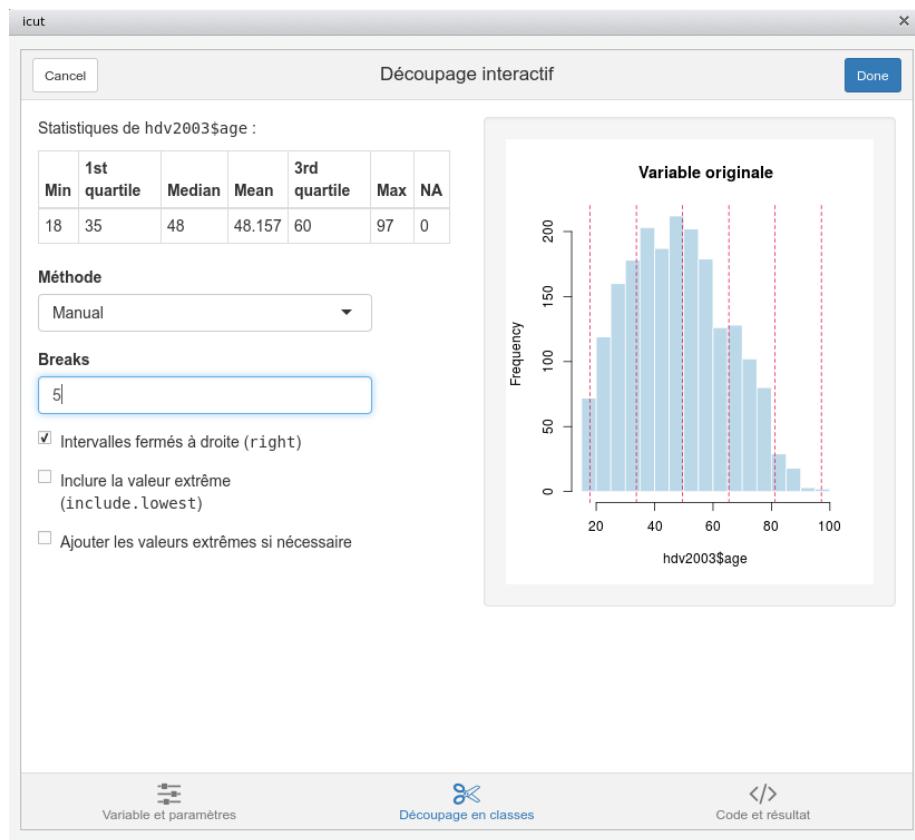
	n	%	val%
[18,25]	191	9.6	9.6
(25,35]	338	16.9	16.9
(35,45]	390	19.5	19.5
(45,55]	414	20.7	20.7
(55,65]	305	15.2	15.2
(65,97]	362	18.1	18.1

Ici on a été obligé d'ajouter l'argument `include.lowest = TRUE` car sinon la valeur 18 n'aurait pas été incluse, et on aurait eu des valeurs manquantes.

9.5.1 Interface graphique

Comme l'utilisation des arguments de `cut` n'est pas toujours très intuitive, l'extension `questionr` propose une interface graphique facilitant cette opération de découpage en classes d'une variable numérique.

Pour lancer cette interface, sous RStudio ouvrir le menu *Addins* et sélectionner *Numeric range dividing*, ou exécuter la fonction `icut()` dans la console en lui passant comme argument la variable quantitative à découper.

Figure 9.3: Interface graphique de `icut`

Vous pouvez alors choisir la variable à découper dans l'onglet *Variable et paramètres*, indiquer les limites de vos classes ainsi que quelques options complémentaires dans l'onglet *Découpage en classes*, et vérifier le résultat dans l'onglet *Code et résultat*. Une fois le résultat satisfaisant, cliquez sur *Done* : si vous êtes sous RStudio le code généré sera directement inséré dans votre script actuel à l'emplacement du curseur. Sinon, ce code sera affiché dans la console et vous pourrez le copier/coller pour l'inclure dans votre script.



Attention, cette interface est prévue pour ne pas modifier vos données. C'est donc à vous d'exécuter le code généré pour que le découpage soit réellement effectif.

9.6 Exercices

9.6.1 Préparation

Pour la plupart de ces exercices, on a besoin des extensions `forcats` et `questionr`, et du jeu de données d'exemple `hdv2003`.

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

9.6.2 Vecteurs et tests

Exercice 1.1

Construire le vecteur suivant :

```
x <- c("12", "3.5", "421", "2,4")
```

Et le convertir en vecteur numérique. Que remarquez-vous ?

Exercice 1.2

Construire le vecteur suivant :

```
x <- c(1, 20, 21, 15.5, 14, 12, 8)
```

- Écrire le test qui indique si les éléments du vecteur sont strictement supérieurs à 15.
- Utiliser ce test pour extraire du vecteur les éléments correspondants.

Exercice 1.3

Le code suivant génère un vecteur de 1000 nombres aléatoires compris entre 0 et 10 :

```
x <- runif(1000, 0, 10)
```

Combien d'éléments de ce vecteur sont compris entre 2 et 4 ?

9.6.3 Recodages de variable qualitative

Exercice 2.1

Construire un vecteur `f` à l'aide du code suivant :

```
f <- c("Jan", "Jan", "Fev", "Juil")
```

Recoder le vecteur à l'aide de la fonction `fct_recode` pour obtenir le résultat suivant :

```
[1] Janvier Janvier Février Juillet
Levels: Février Janvier Juillet
```

Exercice 2.2

À l'aide de l'interface graphique de `questionr`, recoder la variable `relig` du jeu de données `hdv2003` pour obtenir le tri à plat suivant (il se peut que l'ordre des modalités dans le tri à plat soit différent) :

	n	%	val%
Pratiquant	708	35.4	35.4
Appartenance	760	38.0	38.0
Ni croyance ni appartenance	399	20.0	20.0
Rejet	93	4.7	4.7
NSP	40	2.0	2.0

Exercice 2.3

À l'aide de l'interface graphique de `questionr`, recoder la variable `nivetud` pour obtenir le tri à plat suivant (il se peut que l'ordre des modalités dans le tri à plat soit différent) :

	n	%	val%
N'a jamais fait d'etudes	39	2.0	2.1
Études primaires	427	21.3	22.6
1er cycle	204	10.2	10.8
2eme cycle	183	9.2	9.7
Enseignement technique ou professionnel	594	29.7	31.5
Enseignement supérieur	441	22.1	23.4
NA	112	5.6	NA

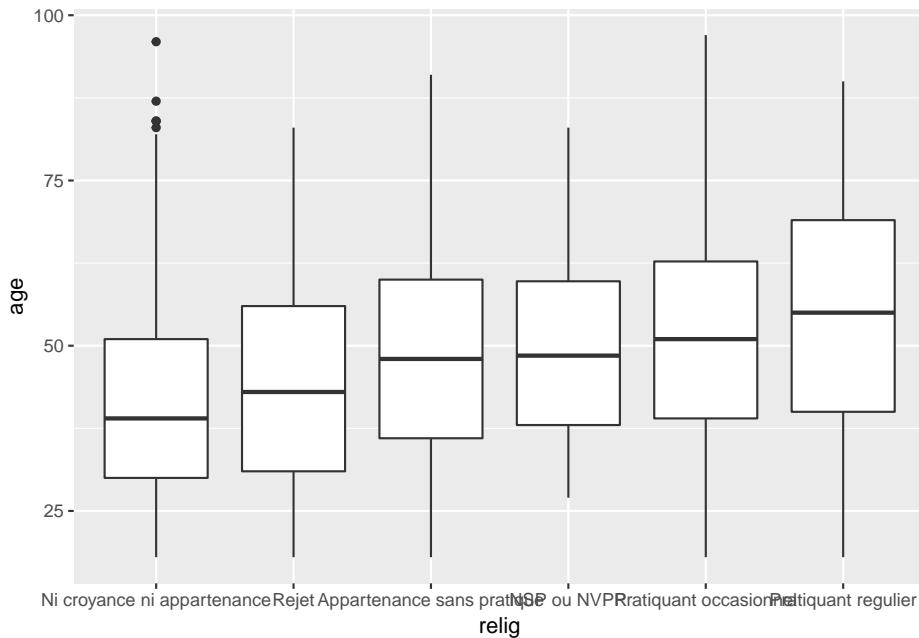
Toujours à l'aide de l'interface graphique, réordonner les modalités de cette variable recodée pour obtenir le tri à plat suivant :

	n	%	val%
Enseignement supérieur	441	22.1	23.4
Enseignement technique ou professionnel	594	29.7	31.5
2eme cycle	183	9.2	9.7
1er cycle	204	10.2	10.8
Études primaires	427	21.3	22.6
N'a jamais fait d'etudes	39	2.0	2.1
NA	112	5.6	NA

Exercice 2.4

À l'aide de la fonction `fct_reorder`, trier les modalités de la variable `relig` du jeu de données `hdv2003` selon leur âge médian.

Vérifier en générant le boxplot suivant :

**9.6.4 Combiner plusieurs variables****Exercice 3.1**

À l'aide de la fonction `if_else`, créer une nouvelle variable `cinema_bd` permettant d'identifier les personnes qui vont au cinéma et déclarent lire des bandes dessinées.

Vous devriez obtenir le tri à plat suivant pour cette nouvelle variable :

	n	%	val%
Autre	1971	98.6	98.6
Cinéma et BD	29	1.5	1.5

Exercice 3.2

À l'aide de la fonction `case_when`, créer une nouvelle variable ayant les modalités suivantes :

- Homme ayant plus de 2 frères et soeurs
- Femme ayant plus de 2 frères et soeurs
- Autre

Vous devriez obtenir le tri à plat suivant :

	n	%	val%
Autre	1001	50.0	50.0
Femme ayant plus de 2 frères et soeurs	546	27.3	27.3
Homme ayant plus de 2 frères et soeurs	453	22.7	22.7

Exercice 3.3

À l'aide de la fonction `case_when`, créer une nouvelle variable ayant les modalités suivantes :

- Homme de plus de 30 ans
- Homme de plus de 40 ans satisfait par son travail
- Femme pratiquant le sport ou le bricolage
- Autre

Vous devriez obtenir le tri à plat suivant :

	n	%	val%
Autre	714	35.7	35.7
Femme pratiquant le sport ou le bricolage	549	27.5	27.5
Homme de plus de 30 ans	610	30.5	30.5
Homme de plus de 40 ans satisfait par son travail	127	6.3	6.3

9.6.5 Découper une variable numérique

Exercice 4.1

Dans le jeu de données `hdv2003`, découper la variable `heures.tv` en classes de manière à obtenir au final le tri à plat suivant :

	n	%	val%
[0,1]	684	34.2	34.3
(1,2]	535	26.8	26.8
(2,4]	594	29.7	29.8
(4,6]	138	6.9	6.9
(6,12]	44	2.2	2.2
NA	5	0.2	NA

Chapitre 10

Manipuler les données avec `dplyr`

`dplyr` est une extension facilitant le traitement et la manipulation de données contenues dans une ou plusieurs tables. Elle propose une syntaxe claire et cohérente, sous formes de verbes, pour la plupart des opérations de ce type. Ses fonctions sont en général plus rapides que leur équivalent sous R de base, elles permettent donc de traiter efficacement des données de grande dimension.

`dplyr` part du principe que les données sont organisées selon le modèle des *tidy data* (voir la section 6.3). Les fonctions de l'extension peuvent s'appliquer à des tableaux de type `data.frame` ou `tibble`, et elles retournent systématiquement un `tibble` (voir la section 6.4).

10.1 Préparation

`dplyr` fait partie du cœur du *tidyverse*, elle est donc chargée automatiquement avec :

```
library(tidyverse)
```

On peut également la charger individuellement avec :

```
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 installé). 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 :

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

On va charger les trois tables du jeu de données :

```
library(nycflights13)
## Chargement des trois tables
data(flights)
data(airports)
data(airlines)
```

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

10.2 Les verbes de dplyr

La manipulation de données avec **dplyr** se fait en utilisant un nombre réduit de verbes, qui correspondent chacun à une action différente appliquée à un tableau de données.

10.2.1 slice

Le verbe **slice** sélectionne des lignes du tableau selon leur position. On lui passe un chiffre ou un vecteur de chiffres.

Si on souhaite sélectionner la 345e ligne du tableau **airports** :

```
slice(airports, 345)

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

Si on veut sélectionner les 5 premières lignes :

```
slice(airports, 1:5)
```

```
# A tibble: 5 x 8
  faa      name          lat    lon    alt    tz dst   tzone
  <chr> <chr>        <dbl> <dbl> <int> <dbl> <chr> <chr>
1 04G  Lansdowne Airport     41.1 -80.6  1044    -5 A America/New_~
2 06A  Moton Field Municipal ~ 32.5 -85.7   264    -6 A America/Chic~
3 06C  Schaumburg Regional    42.0 -88.1   801    -6 A America/Chic~
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_~
```

10.2.2 filter

`filter` sélectionne des lignes d'une table selon une condition. On lui passe en paramètre un test, et seules les lignes pour lesquelles ce test renvoie TRUE (vrai) sont conservées. Pour plus d'informations sur les tests et leur syntaxe, voir la section 9.2.

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

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

```
# A tibble: 27,004 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>           <int>     <dbl>    <int>
1 2013     1     1      517            515        2     830
2 2013     1     1      533            529        4     850
3 2013     1     1      542            540        2     923
4 2013     1     1      544            545       -1    1004
5 2013     1     1      554            600       -6     812
6 2013     1     1      554            558       -4     740
7 2013     1     1      555            600       -5     913
8 2013     1     1      557            600       -3     709
9 2013     1     1      557            600       -3     838
10 2013    1     1      558            600       -2     753
# ... with 26,994 more rows, and 12 more variables: sched_arr_time <int>,
#   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 on veut uniquement les vols avec un retard au départ (variable `dep_delay`) compris entre 10 et 15 minutes :

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

```
# A tibble: 14,919 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>           <int>     <dbl>     <int>
1 2013     1     1      611            600      11       945
2 2013     1     1      623            610      13       920
3 2013     1     1      743            730      13      1107
4 2013     1     1      743            730      13      1059
5 2013     1     1      851            840      11      1215
6 2013     1     1      912            900      12      1241
7 2013     1     1      914            900      14      1058
8 2013     1     1      920            905      15      1039
9 2013     1     1     1011            1001     10      1133
10 2013    1     1     1112            1100     12      1440
# ... with 14,909 more rows, and 12 more variables: sched_arr_time <int>,
#   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 on passe plusieurs arguments à `filter`, celui-ci rajoute automatiquement une condition *et* entre les conditions. La commande précédente peut donc être écrite de la manière suivante, avec le même résultat :

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

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

```
filter(flights, distance == max(distance))
```

```
# A tibble: 342 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>           <int>     <dbl>     <int>
1 2013     1     1      857            900      -3      1516
2 2013     1     2      909            900       9      1525
3 2013     1     3      914            900      14      1504
```

```

4 2013    1    4    900      900      0    1516
5 2013    1    5    858      900     -2    1519
6 2013    1    6   1019      900      79    1558
7 2013    1    7   1042      900     102    1620
8 2013    1    8    901      900      1    1504
9 2013    1    9    641      900    1301    1242
10 2013   10    859      900     -1    1449
# ... with 332 more rows, and 12 more variables: sched_arr_time <int>,
#   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>

```

10.2.3 `select` et `rename`

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

```
select(airports, 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.
# ... with 1,448 more rows

```

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

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

```
# A tibble: 1,458 x 6
  faa      name          alt   tz dst tzone
  <chr>    <chr>        <int> <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 Airpo~ 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_Ange~-
```

... with 1,448 more rows

`select` comprend toute une série de fonctions facilitant la sélection de colonnes multiples. Par exemple, `starts_with`, `ends_width`, `contains` ou `matches` permettent d'exprimer des conditions sur les noms de variables :

```
select(flights, 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
# ... with 336,766 more rows
```

La syntaxe `colonne1:colonne2` permet de sélectionner toutes les colonnes situées entre `colonne1` et `colonne2` incluses¹ :

¹À 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.

```
select(flights, 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
# ... with 336,766 more rows
```

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

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

```
# A tibble: 1,458 x 8
  name          faa      lat      lon      alt      tz dst tzone
  <chr>        <chr>  <dbl>  <dbl>  <int>  <dbl> <chr> <chr>
1 Lansdowne Airport 04G    41.1  -80.6  1044    -5 A  America/New_~
2 Moton Field Municipa~ 06A    32.5  -85.7   264    -6 A  America/Chic~
3 Schaumburg Regional  06C    42.0  -88.1   801    -6 A  America/Chic~
4 Randall Airport       06N    41.4  -74.4   523    -5 A  America/New_~
5 Jekyll Island Airport 09J    31.1  -81.4    11    -5 A  America/New_~
6 Elizabethhton Municip~ 0A9    36.4  -82.2  1593    -5 A  America/New_~
7 Williams County Airp~ 0G6    41.5  -84.5   730    -5 A  America/New_~
8 Finger Lakes Regiona~ 0G7    42.9  -76.8   492    -5 A  America/New_~
9 Shoestring Aviation ~ 0P2    39.8  -76.6  1000    -5 U  America/New_~
10 Jefferson County Intl OS9   48.1  -123.    108    -8 A  America/Los_~
# ... with 1,448 more rows
```

Une variante de `select` est `rename2`, qui permet de renommer des colonnes. On

²Il est également possible de renommer des colonnes directement avec `select`, avec la même

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` :

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

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

... with 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 (`) :

```
tmp <- rename(flights,
              "retard départ" = dep_delay,
              "retard arrivée" = arr_delay)
select(tmp, `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
```

syntaxe que pour `rename`.

```

8          -3         -14
9          -3          -8
10         -2           8
# ... with 336,766 more rows

```

10.2.4 `arrange`

`arrange` réordonne les lignes d'un tableau selon une ou plusieurs colonnes.

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

```
arrange(flights, dep_delay)
```

```

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013     12      7     2040        2123      -43       40
2 2013      2      3     2022        2055      -33      2240
3 2013     11     10     1408        1440      -32      1549
4 2013      1     11     1900        1930      -30      2233
5 2013      1     29     1703        1730      -27      1947
6 2013      8      9      729        755      -26      1002
7 2013     10     23     1907        1932      -25      2143
8 2013      3     30     2030        2055      -25      2213
9 2013      3      2     1431        1455      -24      1601
10 2013     5      5     934        958      -24      1225
# ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   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 :

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

```

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>

```

```

1 2013     1    11    1900      1930    -30    2233
2 2013     1    29    1703      1730    -27    1947
3 2013     1    12    1354      1416    -22    1606
4 2013     1    21    2137      2159    -22    2232
5 2013     1    20     704       725    -21    1025
6 2013     1    12    2050      2110    -20    2310
7 2013     1    12    2134      2154    -20      4
8 2013     1    14    2050      2110    -20    2329
9 2013     1     4    2140      2159    -19    2241
10 2013    1    11    1947      2005   -18    2209
# ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   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 on veut trier selon une colonne par ordre décroissant, on lui applique la fonction `desc()` :

```
arrange(flights, desc(dep_delay))
```

```

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>        <int>     <dbl>    <int>
1 2013     1     9      641        900     1301    1242
2 2013     6    15     1432      1935     1137    1607
3 2013     1    10     1121      1635     1126    1239
4 2013     9    20     1139      1845     1014    1457
5 2013     7    22      845      1600     1005    1044
6 2013     4    10     1100      1900      960    1342
7 2013     3    17     2321      810      911     135
8 2013     6    27      959      1900      899    1236
9 2013     7    22     2257      759      898     121
10 2013    12     5      756     1700      896    1058
# ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   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 `slice`, `arrange` permet par exemple de sélectionner les trois vols ayant eu le plus de retard :

```
tmp <- arrange(flights, desc(dep_delay))
slice(tmp, 1:3)

# A tibble: 3 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>     <int>        <int>    <dbl>    <int>
#> 1  2013      1     9       641          900     1301    1242
#> 2  2013      6    15      1432         1935     1137    1607
#> 3  2013      1    10      1121         1635     1126    1239
# ... with 12 more variables: sched_arr_time <int>, 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>
```

10.2.5 `mutate`

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

Par exemple, la table `flights` contient la durée du vol en minutes.. Si on veut créer une nouvelle variable `duree_h` avec cette durée en heures, on peut faire :

```
flights <- mutate(flights, duree_h = air_time / 60)
select(flights, air_time, duree_h)
```

```
# A tibble: 336,776 x 2
#>   air_time duree_h
#>   <dbl>    <dbl>
#> 1     227    3.78
#> 2     227    3.78
#> 3     160    2.67
#> 4     183    3.05
#> 5     116    1.93
#> 6     150    2.5
#> 7     158    2.63
#> 8      53    0.883
#> 9     140    2.33
#> 10    138    2.3
# ... with 336,766 more rows
```

On peut créer plusieurs nouvelles colonnes en une seule commande, et les expressions successives peuvent prendre en compte les résultats des calculs précédents. L'exemple suivant convertit d'abord la durée en heures dans une variable `duree_h` et la distance en kilomètres dans une variable `distance_km`, puis utilise ces nouvelles colonnes pour calculer la vitesse en km/h.

```
flights <- mutate(flights,
                    duree_h = air_time / 60,
                    distance_km = distance / 0.62137,
                    vitesse = distance_km / duree_h)
select(flights, air_time, duree_h, distance, distance_km, vitesse)
```

```
# A tibble: 336,776 x 5
  air_time duree_h distance distance_km vitesse
    <dbl>    <dbl>    <dbl>      <dbl>    <dbl>
1     227     3.78     1400     2253.     596.
2     227     3.78     1416     2279.     602.
3     160     2.67     1089     1753.     657.
4     183     3.05     1576     2536.     832.
5     116     1.93      762     1226.     634.
6     150     2.5       719     1157.     463.
7     158     2.63     1065     1714.     651.
8      53     0.883     229      369.     417.
9     140     2.33     944     1519.     651.
10    138     2.3      733     1180.     513.
# ... with 336,766 more rows
```

À noter que `mutate` est évidemment parfaitement compatible avec les fonctions vues dans le chapitre 9 sur les recodages : `fct_recode`, `if_else`, `case_when`...

L'avantage d'utiliser `mutate` est double. D'abord il permet d'éviter d'avoir à saisir le nom du tableau de données dans les conditions d'un `if_else` ou d'un `case_when` :

```
flights <- mutate(flights,
                  type_retard = case_when(
                    dep_delay > 0 & arr_delay > 0 ~ "Retard départ et arrivée",
                    dep_delay > 0 & arr_delay <= 0 ~ "Retard départ",
                    dep_delay <= 0 & arr_delay > 0 ~ "Retard arrivée",
                    TRUE ~ "Aucun retard"))
```

Ensuite, il permet aussi d'intégrer ces recodages dans un *pipeline* de traitement de données, concept présenté dans la section suivante.

10.3 Enchaîner les opérations avec le *pipe*

Quand on manipule un tableau de données, il est très fréquent d'enchaîner plusieurs opérations. On va par exemple extraire une sous-population avec `filter`, sélectionner des colonnes avec `select` puis trier selon une variable avec `arrange`, etc.

Quand on veut enchaîner des opérations, on peut le faire de différentes manières. La première est d'effectuer toutes les opérations en une fois en les “emboîtant” :

```
arrange(select(filter(flights, dest == "LAX"), dep_delay, arr_delay), dep_delay)
```

Cette notation a plusieurs inconvénients :

- elle est peu lisible
- les opérations apparaissent dans l'ordre inverse de leur réalisation. Ici on effectue d'abord le `filter`, puis le `select`, puis le `arrange`, alors qu'à la lecture du code c'est le `arrange` qui apparaît en premier.
- Il est difficile de voir quel paramètre se rapporte à quelle fonction

Une autre manière de faire est d'effectuer les opérations les unes après les autres, en stockant les résultats intermédiaires dans un objet temporaire :

```
tmp <- filter(flights, dest == "LAX")
tmp <- select(tmp, dep_delay, arr_delay)
arrange(tmp, dep_delay)
```

C'est nettement plus lisible, l'ordre des opérations est le bon, et les paramètres sont bien rattachés à leur fonction. Par contre, ça reste un peu “verbeux”, et on crée un objet temporaire `tmp` dont on n'a pas réellement besoin.

Pour simplifier et améliorer encore la lisibilité du code, on va utiliser un nouvel opérateur, baptisé *pipe*³. Le *pipe* se note `%>%`, et son fonctionnement est le suivant : si j'exécute `expr %>% f`, alors le résultat de l'expression `expr`, à gauche du *pipe*, sera passé comme premier argument à la fonction `f`, à droite du *pipe*, ce qui revient à exécuter `f(expr)`.

Ainsi les deux expressions suivantes sont rigoureusement équivalentes :

```
filter(flights, dest == "LAX")
```

³Le *pipe* a été introduit à l'origine par l'extension `magrittr`, et repris par `dplyr`

```
flights %>% filter(dest == "LAX")
```

Ce qui est intéressant dans cette histoire, c'est qu'on va pouvoir enchaîner les *pipes*. Plutôt que d'écrire :

```
select(filter(flights, dest == "LAX"), dep_delay, arr_delay)
```

On va pouvoir faire :

```
flights %>% filter(dest == "LAX") %>% select(dep_delay, arr_delay)
```

À chaque fois, le résultat de ce qui se trouve à gauche du *pipe* est passé comme premier argument à ce qui se trouve à droite : on part de l'objet `flights`, qu'on passe comme premier argument à la fonction `filter`, puis on passe le résultat de ce `filter` comme premier argument du `select`.

Le résultat final est le même avec les deux syntaxes, mais avec le *pipe* l'ordre des opérations correspond à l'ordre naturel de leur exécution, et on n'a pas eu besoin de créer d'objet intermédiaire.

Si la liste des fonctions enchaînées est longue, on peut les répartir sur plusieurs lignes à condition que l'opérateur `%>%` soit en fin de ligne :

```
flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay) %>%
  arrange(dep_delay)
```

 On appelle une suite d'instructions de ce type un *pipeline*.

Évidemment, il est naturel de vouloir récupérer le résultat final d'un *pipeline* pour le stocker dans un objet. Par exemple, on peut stocker le résultat du *pipeline* ci-dessus dans un nouveau tableau `delay_la` de la manière suivante :

```
delay_la <- flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay) %>%
  arrange(dep_delay)
```

Dans ce cas, `delay_la` contiendra le tableau final, obtenu après application des trois instructions `filter`, `select` et `arrange`.

Cette notation n'est pas forcément très intuitive au départ. Il faut bien comprendre que c'est le résultat final, une fois application de toutes les opérations du *pipeline*, qui est renvoyé et stocké dans l'objet en début de ligne.

Une manière de le comprendre peut être de voir que la notation suivante :

```
delay_la <- flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay)
```

est équivalente à :

```
delay_la <- (flights %>% filter(dest == "LAX") %>% select(dep_delay, arr_delay))
```



L'utilisation du *pipe* n'est pas obligatoire, mais elle rend les scripts plus lisibles et plus rapides à saisir. On l'utilisera donc dans ce qui suit.

10.4 Opérations groupées

10.4.1 group_by

Un élément très important de `dplyr` est la fonction `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 22
# Groups:   month [12]
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>          <int>    <dbl>     <int>
1 2013     1     1      517          515       2     830
2 2013     1     1      533          529       4     850
3 2013     1     1      542          540       2     923
4 2013     1     1      544          545      -1    1004
5 2013     1     1      554          600      -6     812
6 2013     1     1      554          558      -4     740
7 2013     1     1      555          600      -5     913
```

```

8 2013     1     1      557          600       -3     709
9 2013     1     1      557          600       -3     838
10 2013    1     1      558          600       -2     753
# ... with 336,766 more rows, and 15 more variables: sched_arr_time <int>,
#   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>, duree_h <dbl>, 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 `slice`, `mutate` ou `summarise` vont en tenir compte lors de leurs opérations.

Par exemple, si on applique `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 22
# Groups:   month [12]
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>           <int>     <dbl>     <int>
1 2013     1     1      517          515       2     830
2 2013     2     1      456          500      -4     652
3 2013     3     1       4          2159      125     318
4 2013     4     1      454          500      -6     636
5 2013     5     1       9          1655      434     308
6 2013     6     1       2          2359       3     341
7 2013     7     1       1          2029      212     236
8 2013     8     1      12          2130      162     257
9 2013     9     1       9          2359      10     343
10 2013    10     1      447          500     -13     614
11 2013    11     1       5          2359       6     352
12 2013    12     1      13          2359      14     446
# ... with 15 more variables: sched_arr_time <int>, 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>, duree_h <dbl>, distance_km <dbl>, vitesse <dbl>

```

Idem pour `mutate` : les opérations appliquées lors du calcul des valeurs des

nouvelles colonnes sont appliquées groupe de lignes par groupe de lignes. Dans l'exemple suivant, on ajoute une nouvelle colonne qui contient le retard moyen pour chaque compagnie aérienne. Cette valeur est donc différente d'une compagnie à une autre, mais identique pour tous les vols d'une même compagnie :

```
flights %>%
  group_by(carrier) %>%
  mutate(mean_delay_carrier = mean(dep_delay, na.rm = TRUE)) %>%
  select(dep_delay, month, mean_delay_carrier)
```

Adding missing grouping variables: `carrier`

```
# A tibble: 336,776 x 4
# Groups:   carrier [16]
  carrier dep_delay month mean_delay_carrier
  <chr>     <dbl> <int>          <dbl>
1 UA         2     1            12.1
2 UA         4     1            12.1
3 AA         2     1            8.59
4 B6        -1     1            13.0
5 DL         -6     1            9.26
6 UA         -4     1            12.1
7 B6        -5     1            13.0
8 EV         -3     1            20.0
9 B6        -3     1            13.0
10 AA        -2    1            8.59
# ... with 336,766 more rows
```

Ceci peut permettre, par exemple, de déterminer si un retard donné est supérieur ou inférieur au retard médian de la compagnie :

```
flights %>%
  group_by(carrier) %>%
  mutate(median_delay = median(dep_delay, na.rm = TRUE),
         delay_carrier = if_else(dep_delay > median_delay,
                                  "Supérieur", "Inférieur ou égal")) %>%
  select(dep_delay, month, median_delay, delay_carrier)
```

Adding missing grouping variables: `carrier`

```
# A tibble: 336,776 x 5
# Groups:   carrier [16]
  carrier dep_delay month median_delay delay_carrier
  <chr>      <dbl> <int>        <dbl> <chr>
1 UA          2     1           0 Supérieur
2 UA          4     1           0 Supérieur
3 AA          2     1          -3 Supérieur
4 B6         -1     1          -1 Inférieur ou égal
5 DL          -6     1          -2 Inférieur ou égal
6 UA          -4     1           0 Inférieur ou égal
7 B6         -5     1          -1 Inférieur ou égal
8 EV          -3     1          -1 Inférieur ou égal
9 B6         -3     1          -1 Inférieur ou égal
10 AA         -2    1          -3 Supérieur
# ... with 336,766 more rows
```

`group_by` peut aussi être utile avec `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 22
# Groups:   month [12]
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>            <int>    <dbl>    <int>
1 2013     1     9      641            900    1301    1242
2 2013    10    14     2042            900     702    2255
3 2013    11     3      603            1645     798     829
4 2013    12     5      756            1700     896    1058
5 2013     2    10     2243            830     853     100
6 2013     3    17     2321            810     911     135
7 2013     4    10     1100            1900     960    1342
8 2013     5     3     1133            2055     878    1250
9 2013     6    15     1432            1935    1137    1607
10 2013    7    22      845            1600    1005    1044
11 2013    8     8     2334            1454     520     120
12 2013    9    20     1139            1845    1014    1457
# ... with 15 more variables: sched_arr_time <int>, 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>, duree_h <dbl>, distance_km <dbl>, vitesse <dbl>
```



Attention : la clause `group_by` marche pour les verbes déjà vus précédemment, *sauf* pour `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 22
# Groups:   month [12]
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>        <int>    <dbl>    <int>
1 2013     1     9      641          900    1301    1242
2 2013     6    15     1432         1935    1137    1607
3 2013     1    10     1121         1635    1126    1239
4 2013     9    20     1139         1845    1014    1457
5 2013     7    22      845         1600    1005    1044
6 2013     4    10     1100         1900     960    1342
7 2013     3    17     2321         810     911     135
8 2013     6    27      959         1900     899    1236
9 2013     7    22     2257         759     898     121
10 2013    12     5      756        1700     896    1058
# ... with 336,766 more rows, and 15 more variables: sched_arr_time <int>,
# 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>, duree_h <dbl>, distance_km <dbl>,
# vitesse <dbl>
```

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

```
# A tibble: 336,776 x 22
# Groups: month [12]
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>    <dbl>    <int>
1 2013     1     9      641           900    1301    1242
2 2013     1    10     1121          1635    1126    1239
3 2013     1     1      848          1835     853    1001
4 2013     1    13     1809          810     599    2054
5 2013     1    16     1622          800     502    1911
6 2013     1    23     1551          753     478    1812
7 2013     1    10     1525          900     385    1713
8 2013     1     1     2343         1724     379     314
9 2013     1     2     2131         1512     379    2340
10 2013    1     7     2021         1415     366    2332
# ... with 336,766 more rows, and 15 more variables: sched_arr_time <int>,
#   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>, duree_h <dbl>, distance_km <dbl>,
#   vitesse <dbl>
```

10.4.2 `summarise` et `count`

`summarise` permet d'agréger les lignes du tableau en effectuant une opération “résumée” sur une ou plusieurs colonnes. Par exemple, si on souhaite connaître les retards moyens au départ et à l'arrivée pour l'ensemble des vols du tableau `flights` :

```
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 `group_by`, puisqu'elle permet du coup d'agréger et résumer les lignes du tableau groupe par groupe. Si on souhaite calculer le délai maximum, le délai minimum et le délai moyen au départ pour chaque mois, on pourra faire :

```
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
```

`summarise` dispose d'un opérateur spécial, `n()`, qui retourne le nombre de lignes du groupe. Ainsi si on veut le nombre de vols par destination, on peut utiliser :

```
flights %>%
  group_by(dest) %>%
  summarise(nb = n())
```

```
# A tibble: 105 x 2
  dest      nb
  <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
# ... with 95 more rows
```

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

À noter que quand on veut compter le nombre de lignes par groupe, on peut utiliser directement la fonction `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
# ... with 95 more rows
```

Depuis la version 0.8 de `dplyr`, lorsque la variable de groupage est un facteur et que certaines valeurs du facteur ne sont pas présentes dans le tableau, l'argument `.drop = FALSE` de `group_by` permet de conserver ces niveaux dans le résultat d'une opération groupée.

Par exemple, si on transforme la variable `origin` en facteur pour conserver la liste de ses modalités, et on ne garde que les vols à destination de San Francisco (code `SFO`) :

```
ff <- flights %>%
  mutate(origin = factor(origin)) %>%
  filter(dest == "SFO")
```

Par défaut, si on compte le nombre de vols selon l'aéroport de départ, La Guardia n'apparaît pas car il ne compte aucun vol :

```
ff %>%
  group_by(origin) %>%
  summarise(n = n())
```

```
# A tibble: 2 x 2
  origin     n
  <fct>   <int>
1 EWR      5127
2 JFK      8204
```

Si on souhaite faire apparaître cette information dans la sortie du `summarise`, on peut ajouter l'argument `.drop = FALSE` au `group_by` :

```
ff %>%
  group_by(origin, .drop = FALSE) %>%
  summarise(n = n())
```

```
# A tibble: 3 x 2
  origin     n
  <fct>   <int>
1 EWR      5127
2 JFK      8204
3 LGA       0
```

Cet argument fonctionne aussi avec `count` :

```
ff %>%
  count(origin, .drop = FALSE)
```

```
# A tibble: 3 x 2
  origin     n
  <fct>   <int>
1 EWR      5127
2 JFK      8204
3 LGA       0
```

10.4.3 Grouper selon plusieurs variables

On peut grouper selon plusieurs variables à la fois, il suffit de les indiquer dans la clause du `group_by`. Le *pipeline* suivant le nombre de vols pour chaque mois et pour chaque destination, et trie le résultat par nombre de vols décroissant :

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

```
# 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
# ... with 1,103 more rows
```

On peut également utiliser `count` sur plusieurs variables. Les commandes suivantes comptent le nombre de vols pour chaque couple aéroport de départ / aéroport d'arrivée, et trie le résultat par nombre de vols décroissant :

```
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
# ... with 214 more rows

```

On peut utiliser plusieurs opérations de groupage dans le même *pipeline*. Ainsi, si on souhaite déterminer le triplet compagnie aérienne / aéroport de départ / aéroport d'arrivée ayant le plus grand nombre de vols selon le mois de l'année, on devra procéder en deux étapes :

- d'abord grouper selon mois, compagnie, aéroports d'origine et d'arrivée 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, carrier, origin, dest) %>%
  summarise(nb = n()) %>%
  group_by(month) %>%
  filter(nb == max(nb))

```

```

# A tibble: 13 x 5
# Groups:   month [12]
  month carrier origin dest     nb
  <int> <chr>   <chr> <chr> <int>
1     1 AA       LGA    DFW    437
2     1 DL       LGA    ATL    437
3     2 DL       LGA    ATL    402
4     3 DL       LGA    ATL    461
5     4 AA       LGA    ORD    501
6     5 AA       LGA    ORD    518
7     6 AA       LGA    ORD    520
8     7 AA       LGA    ORD    551
9     8 AA       LGA    ORD    545
10    9 AA       LGA    ORD    492
11   10 AA      LGA    ORD    516
12   11 DL      LGA    ATL    471

```

```
13     12 DL      LGA      ATL      489
```

Lorsqu'on effectue un `group_by` suivi d'un `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` :

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

```
# 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
# ... with 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)
```

```
# 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
# ... with 1,103 more rows

```

On peut à tout moment “dégroupier” un tableau à l'aide de `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)

```

```

# 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
# ... with 1,103 more rows

```

À noter que `count`, par contre, renvoie 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
# ... with 1,103 more rows
```

10.5 Autres fonctions utiles

`dplyr` contient beaucoup d'autres fonctions utiles pour la manipulation de données.

10.5.1 `sample_n`, `sample_frac`

Ces verbes permettent de sélectionner un nombre de lignes ou une fraction des lignes d'un tableau aléatoirement. Ainsi si on veut choisir 5 lignes au hasard dans le tableau `airports` :

```
airports %>% sample_n(5)
```

```
# A tibble: 5 x 8
  faa    name                  lat    lon    alt    tz dst  tzone
  <chr> <chr>              <dbl>  <dbl> <int> <dbl> <chr> <chr>
1 AUG    Augusta State        44.3  -69.8   352   -5 A   America/New-
2 PBX    Pike County Airport - ~ 37.6  -82.6  1473   -5 A   America/New-
3 70J    Cairo-Grady County Air~ 30.9  -84.2   265   -5 A   America/New-
4 PUW    Pullman-Moscow Rgnl   46.7  -117.  2556   -8 A   America/Los-
5 3W2    Put-in-Bay Airport   41.4  -82.5   595   -5 A   America/New-
```

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

```
flights %>% sample_frac(0.1)
```

```
# A tibble: 33,678 x 22
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>      <int>        <dbl>    <int>
1 2013     8    24      501        500      1       636
2 2013     2     4     2240       2250     -10      2343
3 2013     7    14     1909       1859      10      2142
4 2013     6    26     1452       1450      2       1741
5 2013    10    29     1303       1310     -7       1717
6 2013     2    19     1225       1216      9       1559
7 2013    12     2     733        740     -7       851
8 2013    11     8     1525       1529     -4       1811
9 2013     6    29     2146       2149     -3       2232
10 2013    9    16     1451       1455     -4       1602
# ... with 33,668 more rows, and 15 more variables: sched_arr_time <int>,
#   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>, duree_h <dbl>, distance_km <dbl>,
#   vitesse <dbl>
```

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

10.5.2 lead et lag

`lead` et `lag` permettent de décaler les observations d'une variable d'un cran vers l'arrière (pour `lead`) ou vers l'avant (pour `lag`).

```
lead(1:5)
```

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

```
lag(1:5)
```

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

Ceci peut être utile pour des données de type “séries temporelles”. Par exemple,

on peut facilement calculer l'écart entre le retard au départ de chaque vol et celui du vol précédent :

```
flights %>%
  mutate(dep_delay_prev = lag(dep_delay),
        dep_delay_diff = dep_delay - dep_delay_prev) %>%
  select(dep_delay_prev, dep_delay, dep_delay_diff)

# # A tibble: 336,776 x 3
#   dep_delay_prev dep_delay dep_delay_diff
#       <dbl>      <dbl>          <dbl>
# 1           NA        2            NA
# 2            2        4             2
# 3            4        2            -2
# 4            2       -1            -3
# 5           -1       -6            -5
# 6           -6       -4             2
# 7           -4       -5            -1
# 8           -5       -3             2
# 9           -3       -3             0
# 10          -3       -2             1
# ... with 336,766 more rows
```

10.5.3 tally

tally est une fonction qui permet de compter le nombre d'observations d'un groupe :

```
flights %>%
  group_by(month, origin, dest) %>%
  tally

# # A tibble: 2,313 x 4
# Groups:   month, origin [36]
#   month origin dest     n
#       <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
# ... with 2,303 more rows

```

Lors de son premier appel, elle sera équivalente à un `summarise(n = n())` ou à un `count()`. Là où la fonction est intelligente, c'est que si on l'appelle plusieurs fois successivement, elle prendra en compte l'existence d'un `n` déjà calculé et effectuera automatiquement un `summarise(n = sum(n))` :

```

flights %>%
  group_by(month, origin, dest) %>%
  tally %>%
  tally

```

Using `n` as weighting variable

```

# A tibble: 36 x 3
# Groups:   month [12]
  month origin     n
  <int> <chr>   <int>
1     1 EWR     9893
2     1 JFK     9161
3     1 LGA     7950
4     2 EWR     9107
5     2 JFK     8421
6     2 LGA     7423
7     3 EWR    10420
8     3 JFK     9697
9     3 LGA     8717
10    4 EWR    10531
# ... with 26 more rows

```

Et ainsi de suite :

```

flights %>%
  group_by(month, origin, dest) %>%

```

```
tally %>%
tally %>%
tally
```

```
Using `n` as weighting variable
Using `n` as weighting variable
```

```
# A tibble: 12 x 2
  month     n
  <int> <int>
1     1 27004
2     2 24951
3     3 28834
4     4 28330
5     5 28796
6     6 28243
7     7 29425
8     8 29327
9     9 27574
10    10 28889
11    11 27268
12    12 28135
```

10.5.4 distinct

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

```
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
# ... with 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, `distinct` ne conservera que la première d'entre elles.

```

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
# ... with 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 22
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>    <dbl>    <int>
1  2013     1     1      517              515        2     830

```

```

2 2013    1    2     42      2359    43    518
3 2013    1    3     32      2359    33    504
4 2013    1    4     25      2359    26    505
5 2013    1    5     14      2359    15    503
6 2013    1    6     16      2359    17    451
7 2013    1    7     49      2359    50    531
8 2013    1    8    454      500    -6    625
9 2013    1    9      2      2359     3    432
10 2013   1   10      3      2359     4    426
# ... with 355 more rows, and 15 more variables: sched_arr_time <int>,
#   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>, duree_h <dbl>, distance_km <dbl>,
#   vitesse <dbl>

```

10.6 Tables multiples

Le jeu de données `nycflights13` est un exemple de données réparties en plusieurs tables. Ici on en a trois : les informations sur les vols dans `flights`, celles sur les aéroports dans `airports` et celles sur les compagnies aériennes dans `airlines`.

`dplyr` propose différentes fonctions permettant de travailler avec des données structurées de cette manière.

10.6.1 Concaténation : `bind_rows` et `bind_cols`

Les fonctions `bind_rows` et `bind_cols` permettent d'ajouter des lignes (respectivement des colonnes) à une table à partir d'une ou plusieurs autres tables.

L'exemple suivant (certes très artificiel) montre l'utilisation de `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 EDDA  EDDA Airport  40.7  -74.0
2 EDDW  EDDW Airport  40.7  -74.0

```

```

<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(faa, name, lat, lon) %>%
  slice(5:6)

t2

```

```

# A tibble: 2 x 4
  faa   name           lat   lon
  <chr> <chr>         <dbl> <dbl>
1 09J  Jekyll Island Airport 31.1 -81.4
2 0A9  Elizabethton Municipal Airport 36.4 -82.2

```

```

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 concaténe ensuite les trois tables avec `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 OA9    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 `NA` sont automatiquement insérées.

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 `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     OA9    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 `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 OA9    Elizabethton Municipal Airport 36.4 -82.2
5 table3 ADW    Andrews Afb      NA   NA
6 table3 AET    Allakaket Airport  NA   NA

```

`bind_cols` permet de concaténer des colonnes et fonctionne de manière similaire :

```
t1 <- flights %>% slice(1:5) %>% select(dep_delay, dep_time)
t2 <- flights %>% slice(1:5) %>% select(origin, dest)
t3 <- flights %>% slice(1:5) %>% select(arr_delay, arr_time)
bind_cols(t1, t2, t3)
```

	dep_delay	dep_time	origin	dest	arr_delay	arr_time
	<dbl>	<int>	<chr>	<chr>	<dbl>	<int>
1	2	517	EWR	IAH	11	830
2	4	533	LGA	IAH	20	850
3	2	542	JFK	MIA	33	923
4	-1	544	JFK	BQN	-18	1004
5	-6	554	LGA	ATL	-25	812

À noter que `bind_cols` associe les lignes uniquement *par position*. Les lignes des différents tableaux associés doivent donc correspondre (et leur nombre doit être identique). Pour associer des tables *par valeur*, on doit utiliser des jointures.

10.6.2 Jointures

10.6.2.1 Clés implicites

Très souvent, les données relatives à une analyse sont réparties dans plusieurs tables différentes. Dans notre exemple, on peut voir que la table `flights` contient seulement le code de la compagnie aérienne du vol dans la variable `carrier` :

```
flights %>% select(carrier)
```

	carrier
	<chr>
1	UA
2	UA
3	AA
4	B6
5	DL
6	UA

```

7 B6
8 EV
9 B6
10 AA
# ... with 336,766 more rows

```

Et que par ailleurs la table `airlines` contient une information supplémentaire relative à ces compagnies, à savoir le nom complet.

```
airlines
```

```

# A tibble: 16 x 2
  carrier name
  <chr>   <chr>
1 9E      Endeavor Air Inc.
2 AA      American Airlines Inc.
3 AS      Alaska Airlines Inc.
4 B6      JetBlue Airways
5 DL      Delta Air Lines Inc.
6 EV      ExpressJet Airlines Inc.
7 F9      Frontier Airlines Inc.
8 FL      AirTran Airways Corporation
9 HA      Hawaiian Airlines Inc.
10 MQ     Envoy Air
11 OO     SkyWest Airlines Inc.
12 UA     United Air Lines Inc.
13 US     US Airways Inc.
14 VX     Virgin America
15 WN     Southwest Airlines Co.
16 YV     Mesa Airlines Inc.

```

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 `left_join` :

```
left_join(flights, airlines)
```

Pour faciliter la lecture, on va afficher seulement certaines colonnes du résultat :

```
left_join(flights, airlines) %>%
  select(month, day, carrier, name)
```

Joining, by = "carrier"

```
# A tibble: 336,776 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.
# ... with 336,766 more rows
```

On voit que la table résultat 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.



À noter qu'on peut tout à fait utiliser le *pipe* avec les fonctions de jointure :
`flights %>% left_join(airlines)`.

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`.

10.6.2.2 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 jointure 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 `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 :

```
left_join(flights_ex, airports_ex)
```

```
`by` required, because the data sources have no common variables
```

On doit donc spécifier explicitement les clés avec l'argument `by` de `left_join`. Ici la clé est nommée `origin` dans la première table, et `faa` dans la seconde. La syntaxe est donc la suivante :

```
left_join(flights_ex, airports_ex, by = c("origin" = "faa"))
```

```
# A tibble: 336,776 x 6
  month   day origin dest    alt name
  <int> <int> <chr>  <chr> <int> <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
```

```
# ... with 336,766 more rows
```

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 la table `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 :

```
left_join(flights_ex, airports_ex, by=c("dest" = "faa"))
```

```
# A tibble: 336,776 x 8
  month   day origin dest alt.x name.x      alt.y name.y
  <int> <int> <chr>  <chr> <int> <chr>      <int> <chr>
1     1     1 EWR    IAH      18 Newark Libert~    97 George Bush Interco~
2     1     1 LGA    IAH      22 La Guardia     97 George Bush Interco~
3     1     1 JFK    MIA      13 John F Kenned~     8 Miami Intl
4     1     1 JFK    BQN      13 John F Kenned~    NA <NA>
5     1     1 LGA    ATL      22 La Guardia    1026 Hartsfield Jackson ~
6     1     1 EWR    ORD      18 Newark Libert~    668 Chicago Ohare Intl
7     1     1 EWR    FLL      18 Newark Libert~     9 Fort Lauderdale Hol~
8     1     1 LGA    IAD      22 La Guardia    313 Washington Dulles I~
9     1     1 JFK    MCO      13 John F Kenned~    96 Orlando Intl
10    1     1 LGA    ORD      22 La Guardia    668 Chicago Ohare Intl
# ... with 336,766 more rows
```

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 avec `rename` avant de faire la jointure pour avoir des intitulés plus explicites, mais on peut aussi utiliser l'argument `suffix` de

`left_join`, qui permet d'indiquer les suffixes à ajouter aux colonnes. Ainsi, on peut faire :

```
left_join(flights_ex, airports_ex,
          by = c("dest" = "faa"),
          suffix = c("_depart", "_arrivee"))

# # A tibble: 336,776 x 8
#   month day origin dest alt_depart name_depart alt_arrivee name_arrivee
#   <int> <int> <chr>  <chr>    <int> <chr>      <int> <chr>
# 1     1     1 EWR    IAH        18 Newark Lib~      97 George Bush-
# 2     1     1 LGA    IAH        22 La Guardia     97 George Bush-
# 3     1     1 JFK    MIA        13 John F Ken~      8 Miami Intl
# 4     1     1 JFK    BQN        13 John F Ken~      NA <NA>
# 5     1     1 LGA    ATL        22 La Guardia     1026 Hartsfield ~
# 6     1     1 EWR    ORD        18 Newark Lib~      668 Chicago Oha-
# 7     1     1 EWR    FLL        18 Newark Lib~      9 Fort Lauder-
# 8     1     1 LGA    IAD        22 La Guardia     313 Washington ~
# 9     1     1 JFK    MCO        13 John F Ken~      96 Orlando Intl
# 10    1     1 LGA    ORD        22 La Guardia     668 Chicago Oha-
# ... with 336,766 more rows
```

On obtient ainsi directement des noms de colonnes nettement plus clairs.

10.6.3 Types de jointures

Jusqu'à présent nous avons utilisé la fonction `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"))
```

nom	voiture
Sylvie	Twingo
Sylvie	Ferrari
Monique	Scenic
Gunter	Lada
Rayan	Twingo
Rayan	Clio

```
voitures <- tibble(voiture = c("Twingo", "Ferrari", "Clio", "Lada", "208"),
                     vitesse = c("140", "280", "160", "85", "160"))
```

voiture	vitesse
Twingo	140
Ferrari	280
Clio	160
Lada	85
208	160

10.6.3.1 left_join

Si on fait un `left_join` de `voitures` sur `personnes` :

```
left_join(personnes, voitures)
```

```
Joining, by = "voiture"
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Monique	Scenic	NA
Gunter	Lada	85
Rayan	Twingo	140
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 pas de ligne dans `voitures`, donc `vitesse` a été mise à `NA`. Dans le cas de `208`, présente dans `voitures` mais pas dans `personnes`, la ligne n'apparaît pas.

Si on fait un `left_join` cette fois de `personnes` sur `voitures`, c'est l'inverse :

```
left_join(voitures, personnes)
```

```
Joining, by = "voiture"
```

voiture	vitesse	nom
Twingo	140	Sylvie
Twingo	140	Rayan
Ferrari	280	Sylvie
Clio	160	Rayan
Lada	85	Gunter
208	160	NA

La ligne 208 est là, mais nom est à NA. Par contre Monique est absente. Et 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.

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 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, `left_join` sera le type de jointures le plus fréquemment utilisé.

10.6.3.2 right_join

La jointure `right_join` est l’exacte symétrique de `left_join`, c’est-à dire que `right_join(x, y)` est équivalent à `left_join(y, x)` :

```
right_join(personnes, voitures)
```

```
Joining, by = "voiture"
```

nom	voiture	vitesse
Sylvie	Twingo	140
Rayan	Twingo	140
Sylvie	Ferrari	280
Rayan	Clio	160
Gunter	Lada	85
NA	208	160

10.6.3.3 inner_join

Dans le cas de `inner_join(x, y)`, seules les lignes présentes à la fois dans x et y sont conservées (et si nécessaire dupliquées) dans la table résultat :

```
inner_join(personnes, voitures)
```

Joining, by = "voiture"

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160

Ici la ligne 208 est absente, ainsi que la ligne Monique, qui dans le cas d'un `left_join` avait été conservée et s'était vue attribuer une `vitesse` à NA.

10.6.3.4 full_join

Dans le cas de `full_join(x, y)`, 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 :

```
full_join(personnes, voitures)
```

Joining, by = "voiture"

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Monique	Scenic	NA
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160
NA	208	160

10.6.3.5 semi_join et anti_join

`semi_join` et `anti_join` sont des jointures *filtrantes*, c'est-à-dire qu'elles sélectionnent les lignes de `x` sans ajouter les colonnes de `y`.

Ainsi, `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 :

```
semi_join(personnes, voitures)
```

Joining, by = "voiture"

nom	voiture
Sylvie	Twingo
Sylvie	Ferrari
Gunter	Lada
Rayan	Twingo
Rayan	Clio

Un `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` :

```
anti_join(personnes, voitures)
```

Joining, by = "voiture"

nom	voiture
Monique	Scenic

10.7 Ressources

Toutes les ressources ci-dessous sont en anglais...

Le livre *R for data science*, librement accessible en ligne, contient plusieurs chapitres très complets sur la manipulation des données, notamment :

- [Data transformation](#) pour les manipulations
- [Relational data](#) pour les tables multiples

Le [site de l'extension](#) comprend une [liste des fonctions](#) et les pages d'aide associées, mais aussi une [introduction](#) au package et plusieurs articles dont un spécifiquement sur les [jointures](#).

Enfin, une “antisèche” très synthétique est également accessible depuis RStudio, en allant dans le menu *Help* puis *Cheatsheets* et *Data Transformation with dplyr*.

10.8 Exercices

On commence par charger les extensions et les données nécessaires.

```
library(tidyverse)
library(nycflights13)
data(flights)
data(airports)
data(airlines)
```

10.8.1 Les verbes de base de dplyr

Exercice 1.1

Sélectionner les lignes 100 à 105 du tableau des vols (`flights`).

```
# A tibble: 6 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>           <int>    <dbl>     <int>
1 2013     1     1      752             759      -7       955
2 2013     1     1      753             755      -2      1056
3 2013     1     1      754             759      -5      1039
4 2013     1     1      754             755      -1      1103
5 2013     1     1      758             800      -2      1053
6 2013     1     1      759             800      -1      1057
# ... with 12 more variables: sched_arr_time <int>, 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>
```

Exercice 1.2

Sélectionnez les vols du mois de juillet (variable `month`).

```
# A tibble: 29,425 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>           <int>    <dbl>     <int>
  1 2013     7     1      1               2029      212      236
  2 2013     7     1      2               2359       3      344
  3 2013     7     1     29              2245      104      151
  4 2013     7     1     43              2130      193      322
  5 2013     7     1     44              2150      174      300
  6 2013     7     1     46              2051      235      304
  7 2013     7     1     48              2001      287      308
  8 2013     7     1     58              2155      183      335
  9 2013     7     1    100              2146      194      327
 10 2013     7     1    100              2245      135      337
```

```
# ... with 29,415 more rows, and 12 more variables: sched_arr_time <int>,
#   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>
```

Sélectionnez les vols avec un retard à l'arrivée (variable `arr_delay`) compris entre 5 et 15 minutes.

```
# A tibble: 36,392 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>        <int>    <dbl>    <int>
1 2013     1     1      517          515      2       830
2 2013     1     1      554          558     -4       740
3 2013     1     1      558          600     -2       753
4 2013     1     1      558          600     -2       924
5 2013     1     1      600          600      0       837
6 2013     1     1      611          600     11       945
7 2013     1     1      623          610     13       920
8 2013     1     1      624          630     -6       840
9 2013     1     1      629          630     -1       824
10 2013    1     1      632          608     24       740
# ... with 36,382 more rows, and 12 more variables: sched_arr_time <int>,
#   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>
```

Sélectionnez les vols des compagnies Delta, United et American (codes DL, UA et AA).

```
# A tibble: 139,504 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>        <int>    <dbl>    <int>
1 2013     1     1      517          515      2       830
2 2013     1     1      533          529      4       850
3 2013     1     1      542          540      2       923
4 2013     1     1      554          600     -6       812
5 2013     1     1      554          558     -4       740
6 2013     1     1      558          600     -2       753
7 2013     1     1      558          600     -2       924
8 2013     1     1      558          600     -2       923
9 2013     1     1      559          600     -1       941
10 2013    1     1      559          600     -1       854
# ... with 139,494 more rows, and 12 more variables: sched_arr_time <int>,
```

```
#   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>
```

Exercice 1.3

Triez la table flights par retard au départ décroissant.

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>          <int>    <dbl>    <int>
1 2013     1     9      641        900    1301    1242
2 2013     6    15     1432       1935    1137    1607
3 2013     1    10     1121       1635    1126    1239
4 2013     9    20     1139       1845    1014    1457
5 2013     7    22      845       1600    1005    1044
6 2013     4    10     1100       1900     960    1342
7 2013     3    17     2321       810     911     135
8 2013     6    27      959       1900     899    1236
9 2013     7    22     2257       759     898     121
10 2013    12     5      756       1700     896    1058
# ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   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>
```

Exercice 1.4

Selectionnez les colonnes name, lat et lon de la table airports

```
# A tibble: 1,458 x 3
  name                      lat    lon
  <chr>                    <dbl>  <dbl>
1 Lansdowne Airport         41.1 -80.6
2 Moton Field Municipal Airport 32.5 -85.7
3 Schaumburg Regional      42.0 -88.1
4 Randall Airport           41.4 -74.4
5 Jekyll Island Airport     31.1 -81.4
6 Elizabethton Municipal Airport 36.4 -82.2
7 Williams County Airport   41.5 -84.5
8 Finger Lakes Regional Airport 42.9 -76.8
9 Shoestring Aviation Airfield 39.8 -76.6
10 Jefferson County Intl    48.1 -123.
# ... with 1,448 more rows
```

Sélectionnez toutes les colonnes de la table `airports` sauf les colonnes `tz` et `tzone`

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

Toujours dans la table `airports`, renommez la colonne `lat` en `latitude` et `lon` en `longitude`.

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

Exercice 1.5

Dans la table `airports`, la colonne `alt` contient l'altitude de l'aéroport en pieds. Créer une nouvelle variable `alt_m` contenant l'altitude en mètres (on convertit des pieds en mètres en les divisant par 3.2808). Sélectionner dans la table obtenue uniquement les deux colonnes `alt` et `alt_m`.

```
# A tibble: 1,458 x 2
  alt   alt_m
  <int>  <dbl>
1 1044 318.
2 264  80.5
3 801  244.
4 523  159.
5 11   3.35
6 1593 486.
7 730  223.
8 492  150.
9 1000 305.
10 108  32.9
# ... with 1,448 more rows
```

10.8.2 Enchaîner des opérations

Exercice 2.1

Réécrire le code de l'exercice précédent en utilisant le *pipe* `%>%`.

Exercice 2.2

En utilisant le *pipe*, sélectionnez les vols à destination de San Francisco (code SFO de la variable `dest`) et triez-les selon le retard au départ décroissant (variable `dep_delay`).

```
# A tibble: 13,331 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013     9    20      1139        1845     1014    1457
2 2013     7     7      2123        1030      653     17
3 2013     7     7      2059        1030      629     106
4 2013     7     6      149         1600      589     456
5 2013     7    10      133         1800      453     455
6 2013     7    10      2342        1630      432     312
7 2013     7     7      2204        1525      399     107
8 2013     7     7      2306        1630      396     250
9 2013     6    23      1833        1200      393     NA
10 2013     7    10      2232        1609      383     138
# ... with 13,321 more rows, and 12 more variables: sched_arr_time <int>,
#   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>
```

Exercice 2.3

Sélectionnez les vols des mois de septembre et octobre, conservez les colonnes `dest` et `dep_delay`, créez une nouvelle variable `retard_h` contenant le retard au départ en heures, triez selon `retard_h` par ordre décroissant et conservez uniquement les 5 premières lignes.

```
# A tibble: 5 x 3
  dest  dep_delay  retard_h
  <chr>    <dbl>     <dbl>
1 SFO      1014     16.9
2 ATL       702     11.7
3 DTW       696     11.6
4 ATL       602     10.0
5 MSP       593      9.88
```

10.8.3 group_by et summarise

Exercice 3.1

Affichez le nombre de vols par mois.

```
# A tibble: 12 x 2
  month     n
  <int> <int>
1     1 27004
2     2 24951
3     3 28834
4     4 28330
5     5 28796
6     6 28243
7     7 29425
8     8 29327
9     9 27574
10    10 28889
11    11 27268
12    12 28135
```

Triez la table résultat selon le nombre de vols croissant.

```
# A tibble: 12 x 2
  month     n
  <int> <int>
1     2 24951
2     1 27004
3    11 27268
4     9 27574
5    12 28135
6     6 28243
7     4 28330
8     5 28796
9     3 28834
10    10 28889
11    8 29327
12    7 29425
```

Exercice 3.2

Calculer la distance moyenne des vols selon l'aéroport de départ.

```
# A tibble: 3 x 2
  origin distance_moyenne
  <chr>          <dbl>
1 EWR            1057.
2 JFK            1266.
3 LGA             780.
```

Exercice 3.3

Calculer le nombre de vols à destination de Los Angeles (code LAX) pour chaque mois de l'année.

```
# A tibble: 12 x 2
  month     n
  <int> <int>
1     1 1159
2     2 1030
3     3 1178
4     4 1382
5     5 1453
6     6 1430
7     7 1500
8     8 1505
```

```

 9      9  1384
10     10  1409
11     11  1336
12     12  1408

```

Exercice 3.4

Calculer le nombre de vols selon le mois et la destination.

```

# 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
# ... with 1,103 more rows

```

Ne conserver, pour chaque mois, que la destination avec le nombre maximal de vols.

```

# A tibble: 12 x 3
# Groups:   month [12]
  month dest     n
  <int> <chr> <int>
1     1 ATL     1396
2     2 ATL     1267
3     3 ATL     1448
4     4 ATL     1490
5     5 ORD     1582
6     6 ORD     1547
7     7 ORD     1573
8     8 ORD     1604
9     9 ORD     1582
10    10 ORD    1604
11    11 ATL     1384
12    12 ATL     1463

```

Exercice 3.5

Calculer le nombre de vols selon le mois. Ajouter une colonne comportant le pourcentage de vols annuels réalisés par mois.

```
# A tibble: 12 x 3
  month     n pourcentage
  <int> <int>      <dbl>
1     1 27004      8.02
2     2 24951      7.41
3     3 28834      8.56
4     4 28330      8.41
5     5 28796      8.55
6     6 28243      8.39
7     7 29425      8.74
8     8 29327      8.71
9     9 27574      8.19
10    10 28889     8.58
11    11 27268     8.10
12    12 28135     8.35
```

Exercice 3.6

Calculer, pour chaque destination et chaque mois, le retard moyen à l'arrivée. Pour chaque mois, trier les destinations selon ce retard moyen décroissant, et (toujours pour chaque mois) ne conserver que les trois destinations avec le retard le plus important.

```
# A tibble: 36 x 3
# Groups:   month [12]
  month dest  retard_moyen
  <int> <chr>      <dbl>
1     1 TUL       68.1
2     1 OKC       57.7
3     1 CAE       55.9
4     2 DSM       48.2
5     2 TUL       33.5
6     2 GSP       32.9
7     3 DSM       60.6
8     3 CAE       46.9
9     3 PVD       44.3
10    4 CAE       71.3
# ... with 26 more rows
```

10.8.4 Jointures

Exercice 4.1

Faire la jointure de la table `airlines` sur la table `flights` à l'aide de `left_join`.

```
# A tibble: 336,776 x 20
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013     1     1      517            515        2     830
2 2013     1     1      533            529        4     850
3 2013     1     1      542            540        2     923
4 2013     1     1      544            545       -1    1004
5 2013     1     1      554            600       -6     812
6 2013     1     1      554            558       -4     740
7 2013     1     1      555            600       -5     913
8 2013     1     1      557            600       -3     709
9 2013     1     1      557            600       -3     838
10 2013    1     1      558            600       -2     753
# ... with 336,766 more rows, and 13 more variables: sched_arr_time <int>,
# 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>, name <chr>
```

Exercice 4.2

À partir de la table résultat de l'exercice précédent, calculer le retard moyen au départ pour chaque compagnie, trier selon ce retard décroissant et ne conserver que le nom de la compagnie et le retard correspondant.

```
# A tibble: 16 x 2
  name                retard_moyen
  <chr>                  <dbl>
1 Frontier Airlines Inc. 20.2
2 ExpressJet Airlines Inc. 20.0
3 Mesa Airlines Inc.     19.0
4 AirTran Airways Corporation 18.7
5 Southwest Airlines Co. 17.7
6 Endeavor Air Inc.     16.7
7 JetBlue Airways        13.0
8 Virgin America         12.9
9 SkyWest Airlines Inc.  12.6
10 United Air Lines Inc. 12.1
11 Envoy Air              10.6
12 Delta Air Lines Inc.  9.26
```

13 American Airlines Inc.	8.59
14 Alaska Airlines Inc.	5.80
15 Hawaiian Airlines Inc.	4.90
16 US Airways Inc.	3.78

Exercice 4.3

Faire la jointure de la table `airports` sur la table `flights` en utilisant comme clé le code de l'aéroport de destination.

```
# A tibble: 336,776 x 26
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>      <int>        <int>     <dbl>    <int>
1 2013     1     1       517          515      2       830
2 2013     1     1       533          529      4       850
3 2013     1     1       542          540      2       923
4 2013     1     1       544          545     -1      1004
5 2013     1     1       554          600     -6       812
6 2013     1     1       554          558     -4       740
7 2013     1     1       555          600     -5       913
8 2013     1     1       557          600     -3       709
9 2013     1     1       557          600     -3       838
10 2013    1     1       558          600     -2       753
# ... with 336,766 more rows, and 19 more variables: sched_arr_time <int>,
#   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>, name <chr>, lat <dbl>, lon <dbl>,
#   alt <int>, tz <dbl>, dst <chr>, tzone <chr>
```

À partir de cette table, afficher pour chaque mois le nom de l'aéroport de destination ayant eu le plus petit nombre de vol.

```
# A tibble: 14 x 3
# Groups:   month [12]
  month name             n
  <int> <chr>           <int>
1     1 Key West Intl    1
2     2 Jackson Hole Airport 3
3     3 Bangor Intl      2
4     4 Key West Intl    1
5     4 Myrtle Beach Intl 1
6     5 Columbia Metropolitan 9
7     6 Myrtle Beach Intl  1
```

```

8      7 La Guardia          1
9      8 South Bend Rgnl    1
10     9 South Bend Rgnl    5
11     10 Albany Intl       1
12     10 South Bend Rgnl   1
13     11 Blue Grass        1
14     12 South Bend Rgnl   1

```

Exercice 4.4

Créer une table indiquant, pour chaque trajet, le nom de l'aéroport de départ et celui de l'aéroport d'arrivée.

```

# A tibble: 336,776 x 2
  orig_name      dest_name
  <chr>          <chr>
1 Newark Liberty Intl George Bush Intercontinental
2 La Guardia      George Bush Intercontinental
3 John F Kennedy Intl Miami Intl
4 John F Kennedy Intl <NA>
5 La Guardia      Hartsfield Jackson Atlanta Intl
6 Newark Liberty Intl Chicago Ohare Intl
7 Newark Liberty Intl Fort Lauderdale Hollywood Intl
8 La Guardia      Washington Dulles Intl
9 John F Kennedy Intl Orlando Intl
10 La Guardia     Chicago Ohare Intl
# ... with 336,766 more rows

```

10.8.5 Bonus**Exercice 5.1**

Calculer le nombre de vols selon l'aéroport de destination, et fusionnez la table `airports` sur le résultat avec `left_join`. Stocker le résultat final dans un objet nommé `flights_dest`.

```

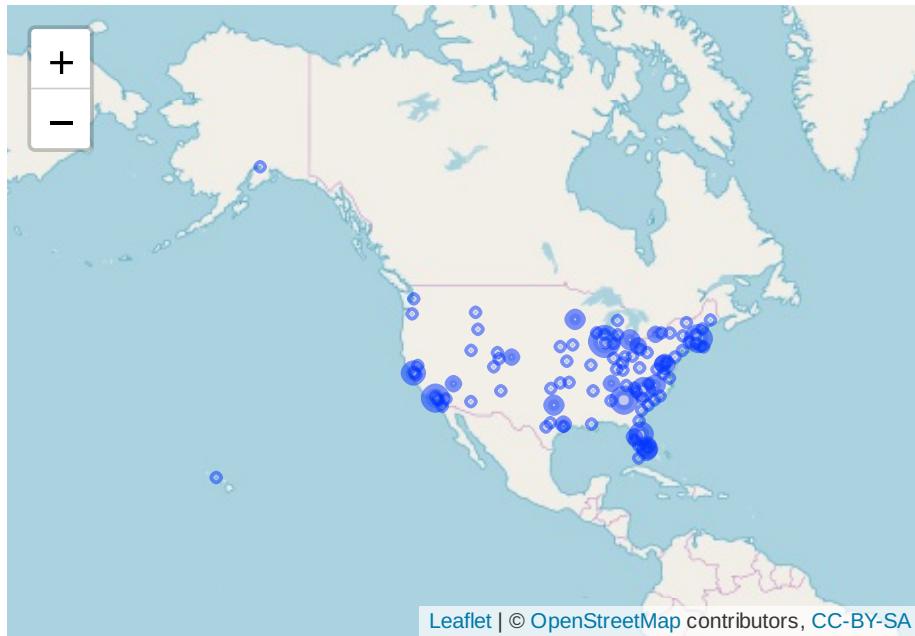
# A tibble: 105 x 9
  dest      n name                  lat    lon    alt    tz dst    tzone
  <chr> <int> <chr>                <dbl> <dbl> <int> <dbl> <chr> <chr>
1 ABQ      254 Albuquerque Inter~  35.0 -107.   5355    -7 A    America/D-
2 ACK      265 Nantucket Mem       41.3  -70.1    48    -5 A    America/N-
3 ALB      439 Albany Intl        42.7  -73.8   285    -5 A    America/N-

```

```
4 ANC      8 Ted Stevens Anchorage 61.2 -150.    152    -9 A    America/A-
5 ATL     17215 Hartsfield Jackson 33.6 -84.4   1026   -5 A    America/N-
6 AUS      2439 Austin Bergstrom 30.2 -97.7    542    -6 A    America/C-
7 AVL      275 Asheville Regional 35.4 -82.5   2165   -5 A    America/N-
8 BDL      443 Bradley Intl     41.9 -72.7    173    -5 A    America/N-
9 BGR      375 Bangor Intl      44.8 -68.8    192    -5 A    America/N-
10 BHM     297 Birmingham Intl 33.6 -86.8    644    -6 A    America/C-
# ... with 95 more rows
```

Créez une carte interactive des résultats avec leaflet et le code suivant :

```
library(leaflet)
leaflet(data = flights_dest) %>%
  addTiles %>%
  addCircles(lng=~lon, lat=~lat, radius=~n * 10, popup=~name)
```



Chapitre 11

Manipuler du texte avec **stringr**

Les fonctions de **forcats** vues précédemment permettent de modifier des modalités d'une variables qualitative globalement. Mais parfois on a besoin de manipuler le contenu même du texte d'une variable de type chaîne de caractères : combiner, rechercher, remplacer...

On va utiliser ici les fonctions de l'extension **stringr**. Celle-ci fait partie du cœur du *tidyverse*, elle est donc automatiquement chargée avec :

```
library(tidyverse)
```



stringr est en fait une interface simplifiée aux fonctions d'une autre extension, **stringi**. Si les fonctions de **stringr** ne sont pas suffisantes ou si on manipule beaucoup de chaînes de caractères, ne pas hésiter à se reporter à la documentation de **stringi**.

Dans ce qui suit on va utiliser le court tableau d'exemple **d** suivant :

```
d <- tibble(  
  nom = c("Mr Félicien Machin", "Mme Raymonde Bidule", "M. Martial Truc", "Mme Huguette Chose"),  
  adresse = c("3 rue des Fleurs", "47 ave de la Libération", "12 rue du 17 octobre 1961", "221 av  
  ville = c("Nouméa", "Marseille", "Vénissieux", "Marseille")  
)
```

nom	adresse	ville
Mr Félicien Machin	3 rue des Fleurs	Nouméa
Mme Raymonde Bidule	47 ave de la Libération	Marseille
M. Martial Truc	12 rue du 17 octobre 1961	Vénissieux
Mme Huguette Chose	221 avenue de la Libération	Marseille

11.1 Expressions régulières

Les fonctions présentées ci-dessous sont pour la plupart prévues pour fonctionner avec des *expressions régulières*. Celles-ci constituent un mini-langage, qui peut paraître assez cryptique, mais qui est très puissant pour spécifier des motifs de chaînes de caractères.

Elles permettent par exemple de sélectionner le dernier mot avant la fin d'une chaîne, l'ensemble des suites alphanumériques commençant par une majuscule, des nombres de 3 ou 4 chiffres situés en début de chaîne, et beaucoup beaucoup d'autres choses encore bien plus complexes.

Pour donner un exemple concret, l'expression régulière suivante permet de détecter une adresse de courrier électronique¹ :

```
[\w\d+.-]+@[ \w\d.-]+\.\[a-zA-Z\]{2,}
```

Par souci de simplicité, dans ce qui suit les exemples seront donnés autant que possible avec de simples chaînes, sans expression régulière. Mais si vous pensez manipuler des données textuelles, il peut être très utile de s'intéresser à cette syntaxe.

11.2 Concaténer des chaînes

La première opération de base consiste à concaténer des chaînes de caractères entre elles. On peut le faire avec la fonction **paste**.

Par exemple, si on veut concaténer l'adresse et la ville :

```
paste(d$adresse, d$ville)
```

```
[1] "3 rue des Fleurs Nouméa"
[2] "47 ave de la Libération Marseille"
[3] "12 rue du 17 octobre 1961 Vénissieux"
```

¹ Il s'agit en fait d'une version très simplifiée, la “véritable” expression permettant de tester si une adresse mail est valide fait plus de 80 lignes...

```
[4] "221 avenue de la Libération Marseille"
```

Par défaut, `paste` concatène en ajoutant un espace entre les différentes chaînes. On peut spécifier un autre séparateur avec son argument `sep` :

```
paste(d$adresse, d$ville, sep = " - ")
```

```
[1] "3 rue des Fleurs - Nouméa"  
[2] "47 ave de la Libération - Marseille"  
[3] "12 rue du 17 octobre 1961 - Vénissieux"  
[4] "221 avenue de la Libération - Marseille"
```

Il existe une variante, `paste0`, qui concatène sans mettre de séparateur, et qui est légèrement plus rapide :

```
paste0(d$adresse, d$ville)
```

```
[1] "3 rue des FleursNouméa"  
[2] "47 ave de la LibérationMarseille"  
[3] "12 rue du 17 octobre 1961Vénissieux"  
[4] "221 avenue de la LibérationMarseille"
```



À noter que `paste` et `paste0` sont des fonctions R de base. L'équivalent pour `stringr` se nomme `str_c`.

Parfois on cherche à concaténer les différents éléments d'un vecteur non pas avec ceux d'un autre vecteur, comme on l'a fait précédemment, mais *entre eux*. Dans ce cas `paste` seule ne fera rien :

```
paste(d$ville)
```

```
[1] "Nouméa"      "Marseille"   "Vénissieux" "Marseille"
```

Il faut lui ajouter un argument `collapse`, avec comme valeur la chaîne à utiliser pour concaténer les éléments :

```
paste(d$ville, collapse = ", ")
```

```
[1] "Nouméa, Marseille, Vénissieux, Marseille"
```

11.3 Convertir en majuscules / minuscules

Les fonctions `str_to_lower`, `str_to_upper` et `str_to_title` permettent respectivement de mettre en minuscules, mettre en majuscules, ou de capitaliser les éléments d'un vecteur de chaînes de caractères :

```
str_to_lower(d$nom)
```

```
[1] "mr félicien machin"  "mme raymonde bidule" "m. martial truc"
[4] "mme huguette chose"
```

```
str_to_upper(d$nom)
```

```
[1] "MR FÉLICIEN MACHIN"  "MME RAYMONDE BIDULE" "M. MARTIAL TRUC"
[4] "MME HUGUETTE CHOSE"
```

```
str_to_title(d$nom)
```

```
[1] "Mr Félicien Machin"  "Mme Raymonde Bidule" "M. Martial Truc"
[4] "Mme Huguette Chose"
```

11.4 Découper des chaînes

La fonction `str_split` permet de “découper” une chaîne de caractère en fonction d'un délimiteur. On passe la chaîne en premier argument, et le délimiteur en second :

```
str_split("un-deux-trois", "-")
```

```
[[1]]
[1] "un"     "deux"   "trois"
```

On peut appliquer la fonction à un vecteur, dans ce cas le résultat sera une liste :

```
str_split(d$nom, " ")
```

```
[[1]]
[1] "Mr"      "Félicien" "Machin"

[[2]]
[1] "Mme"     "Raymonde" "Bidule"

[[3]]
[1] "M."       "Martial"  "Truc"

[[4]]
[1] "Mme"     "Huguette" "Chose"
```

Ou un tableau (plus précisément une matrice) si on ajoute `simplify = TRUE`.

```
str_split(d$nom, " ", simplify = TRUE)
```

```
[,1] [,2]      [,3]
[1,] "Mr"  "Félicien" "Machin"
[2,] "Mme" "Raymonde" "Bidule"
[3,] "M."   "Martial"  "Truc"
[4,] "Mme"  "Huguette" "Chose"
```

Si on souhaite créer de nouvelles colonnes dans un tableau de données en découpant une colonne de type texte, on pourra utiliser la fonction `separate` de l'extension `tidyR`. Celle-ci est expliquée section [12.3.3](#).

Voici juste un exemple de son utilisation :

```
library(tidyr)
d %>% separate(nom, c("genre", "prenom", "nom"))

# A tibble: 4 x 5
  genre prenom nom     adresse           ville
  <chr> <chr>   <chr>   <chr>
1 Mr    Félicien Machin 3 rue des Fleurs Nouméa
2 Mme   Raymonde Bidule 47 ave de la Libération Marseille
3 M     Martial Truc   12 rue du 17 octobre 1961 Vénissieux
4 Mme   Huguette Chose  221 avenue de la Libération Marseille
```

11.5 Extraire des sous-chaînes par position

La fonction `str_sub` permet d'extraire des sous-chaînes par position, en indiquant simplement les positions des premier et dernier caractères :

```
str_sub(d$ville, 1, 3)

[1] "Nou" "Mar" "Vén" "Mar"
```

11.6 DéTECTer des motifs

`str_detect` permet de détecter la présence d'un motif parmi les éléments d'un vecteur. Par exemple, si on souhaite identifier toutes les adresses contenant "Libération" :

```
str_detect(d$adresse, "Libération")

[1] FALSE TRUE FALSE TRUE
```

`str_detect` renvoie un vecteur de valeurs logiques et peut donc être utilisée, par exemple, avec le verbe `filter` de `dplyr` pour extraire des sous-populations.

Une variante, `str_count`, compte le nombre d'occurrences d'une chaîne pour chaque élément d'un vecteur :

```
str_count(d$ville, "s")
```

```
[1] 0 1 2 1
```



Attention, les fonctions de `stringr` étant prévues pour fonctionner avec des expressions régulières, certains caractères n'auront pas le sens habituel dans la chaîne indiquant le motif à rechercher. Par exemple, le `.` ne sera pas un point mais le symbole représentant “n’importe quel caractère”.

La section sur les modificateurs de motifs explique comment utiliser des chaînes “classiques” au lieu d’expressions régulières.

On peut aussi utiliser `str_subset` pour ne garder d’un vecteur que les éléments correspondant au motif :

```
str_subset(d$adresse, "Libération")
```

```
[1] "47 ave de la Libération"      "221 avenue de la Libération"
```

11.7 Extraire des motifs

`str_extract` permet d’extraire les valeurs correspondant à un motif. Si on lui passe comme motif une chaîne de caractère, cela aura peu d’intérêt :

```
str_extract(d$adresse, "Libération")
```

```
[1] NA          "Libération" NA          "Libération"
```

C’est tout de suite plus intéressant si on utilise des expressions régulières. Par exemple la commande suivante permet d’isoler les numéros de rue.

```
str_extract(d$adresse, "^\\d+")
```

```
[1] "3"    "47"   "12"   "221"
```

`str_extract` ne récupère que la première occurrence du motif. Si on veut toutes les extraire on peut utiliser `str_extract_all`. Ainsi, si on veut extraire l'ensemble des nombres présents dans les adresses :

```
str_extract_all(d$adresse, "\\d+")
```

```
[[1]]
[1] "3"

[[2]]
[1] "47"

[[3]]
[1] "12"   "17"   "1961"

[[4]]
[1] "221"
```



Si on veut faire de l'extraction de groupes dans des expressions régulières (identifiés avec des parenthèses), on pourra utiliser `str_match`.

À noter que si on souhaite extraire des valeurs d'une colonne texte d'un tableau de données pour créer de nouvelles variables, on pourra utiliser la fonction `extract` de l'extension `tidyR`, décrite section 12.3.5.

Par exemple :

```
library(tidyR)
d %>% extract(adresse, "type_rue", "^(\\d+ (.*)?) ", remove = FALSE)
```

	nom	adresse	type_rue	ville
	<chr>	<chr>	<chr>	<chr>
1	Mr Félicien Machin	3 rue des Fleurs	rue	Nouméa
2	Mme Raymonde Bidule	47 ave de la Libération	ave	Marseille
3	M. Martial Truc	12 rue du 17 octobre 1961	rue	Vénissieux
4	Mme Huguette Chose	221 avenue de la Libération	avenue	Marseille

11.8 Remplacer des motifs

La fonction `str_replace` permet de remplacer une chaîne ou un motif par une autre.

Par exemple, on peut remplacer les occurrences de “Mr” par “M.” dans les noms de notre tableau :

```
str_replace(d$nom, "Mr", "M.")
```

```
[1] "M. Félicien Machin"  "Mme Raymonde Bidule" "M. Martial Truc"
[4] "Mme Huguette Chose"
```

La variante `str_replace_all` permet de spécifier plusieurs remplacements d'un coup :

```
str_replace_all(d$adresse, c("avenue"="Avenue", "ave"="Avenue", "rue"="Rue"))
```

```
[1] "3 Rue des Fleurs"           "47 Avenue de la Libération"
[3] "12 Rue du 17 octobre 1961" "221 Avenue de la Libération"
```

11.9 Modificateurs de motifs

Par défaut, les motifs passés aux fonctions comme `str_detect`, `str_extract` ou `str_replace` sont des expressions régulières classiques.

On peut spécifier qu'un motif n'est pas une expression régulière mais une chaîne de caractères normale en lui appliquant la fonction `fixed`. Par exemple, si on veut compter le nombre de points dans les noms de notre tableau, le paramétrage par défaut ne fonctionnera pas car dans une expression régulière le `.` est un symbole signifiant “n'importe quel caractère” :

```
str_count(d$nom, ".")
```

```
[1] 18 19 15 18
```

Il faut donc spécifier que notre point est bien un point avec `fixed` :

```
str_count(d$nom, fixed("."))
```

```
[1] 0 0 1 0
```

On peut aussi modifier le comportement des expressions régulières à l'aide de la fonction `regex`. On peut ainsi rendre les motifs insensibles à la casse avec `ignore_case` :

```
str_detect(d$nom, "mme")
```

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

```
str_detect(d$nom, regex("mme", ignore_case = TRUE))
```

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

On peut également permettre aux regex d'être multilignes avec l'option `multiline = TRUE`, etc.

11.10 Ressources

L'ouvrage *R for Data Science*, accessible en ligne, contient un chapitre entier sur les chaînes de caractères et les expressions régulières (en anglais).

Le [site officiel de stringr](#) contient une [liste des fonctions](#) et les pages d'aide associées, ainsi qu'un [article dédié aux expressions régulières](#).

Pour des besoins plus pointus, on pourra aussi utiliser [l'extension stringi](#) sur laquelle est elle-même basée `stringr`.

11.11 Exercices

Dans ces exercices on utilise un tableau `d`, généré par le code suivant :

```
d <- tibble(
  nom = c("M. rené Bézigue", "Mme Paulette fouchin", "Mme yvonne duluc", "M. Jean-Yves Pernoud"),
  naissance = c("18/04/1937 Vesoul", "En 1947 à Grenoble (38)", "Le 5 mars 1931 à Bar-le-Duc", "M"),
  profession = c("Ouvrier agric", "ouvrière qualifiée", "Institutrice", "Exploitant agric")
)
```

nom	naissance	profession
M. rené Bézigue	18/04/1937 Vesoul	Ouvrier agric
Mme Paulette fouchin	En 1947 à Grenoble (38)	ouvrière qualifiée
Mme yvonne duluc	Le 5 mars 1931 à Bar-le-Duc	Institutrice
M. Jean-Yves Pernoud	Marseille, juin 1938	Exploitant agric

Exercice 1

Capitalisez les noms des personnes avec `str_to_title` :

```
[1] "M. René Bézigue"      "Mme Paulette Fouchin" "Mme Yvonne Duluc"
[4] "M. Jean-Yves Pernoud"
```

Exercice 2

Dans la variable `profession`, remplacer toutes les occurrences de l'abréviation “agric” par “agricole” :

```
[1] "Ouvrier agricole"    "ouvrière qualifiée"  "Institutrice"
[4] "Exploitant agricole"
```

Exercice 3

À l'aide de `str_detect`, identifier les personnes de catégorie professionnelle “Ouvrier”. Indication : pensez au modificateur `ignore_case`.

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

Exercice 4

À l'aide de `case_when` et de `str_detect`, créer une nouvelle variable `sexé` identifiant le sexe de chaque personne en fonction de la présence de M. ou de Mme dans son nom.

```
# A tibble: 4 x 4
  nom                  sexe  naissance            profession
  <chr>                <chr> <chr>                <chr>
  1 M. rené Bézigue   Homme 18/04/1937 Vesoul  Ouvrier agric
```

```
2 Mme Paulette fouchin Femme En 1947 à Grenoble (38)    ouvrière qualifiée
3 Mme yvonne duluc      Femme Le 5 mars 1931 à Bar-le-Duc Institutrice
4 M. Jean-Yves Pernoud Homme Marseille, juin 1938     Exploitant agric
```

Exercice 5

Extraire l'année de naissance de chaque individu avec `str_extract`. Vous pouvez utiliser le regex "`\d\d\d\d`" qui permet d'identifier les nombres de quatre chiffres.

Vous devez obtenir le vecteur suivant :

```
[1] "1937" "1947" "1931" "1938"
```

À l'aide de la fonction `extract` de l'extension `tidyverse` et du regex précédent, créez une nouvelle variable `annee` dans le tableau, qui contient l'année de naissance (pour plus d'informations sur `extract`, voir la section [12.3.5](#)).

```
# A tibble: 4 x 4
  nom                naissance            annee profession
  <chr>              <chr>                <chr> <chr>
1 M. rené Bézigue   18/04/1937 Vesoul       1937 Ouvrier agric
2 Mme Paulette fouchin En 1947 à Grenoble (38) 1947 ouvrière qualifiée
3 Mme yvonne duluc      Le 5 mars 1931 à Bar-le-Duc 1931 Institutrice
4 M. Jean-Yves Pernoud Marseille, juin 1938     1938 Exploitant agric
```

Chapitre 12

Mettre en ordre avec `tidyverse`

12.1 Tidy data

Comme indiqué dans la section 6.3, les extensions du *tidyverse* comme `dplyr` ou `ggplot2` partent du principe que les données sont “bien rangées” sous forme de *tidy data*.

Prenons un exemple avec les données suivantes, qui indique la population de trois pays pour quatre années différentes :

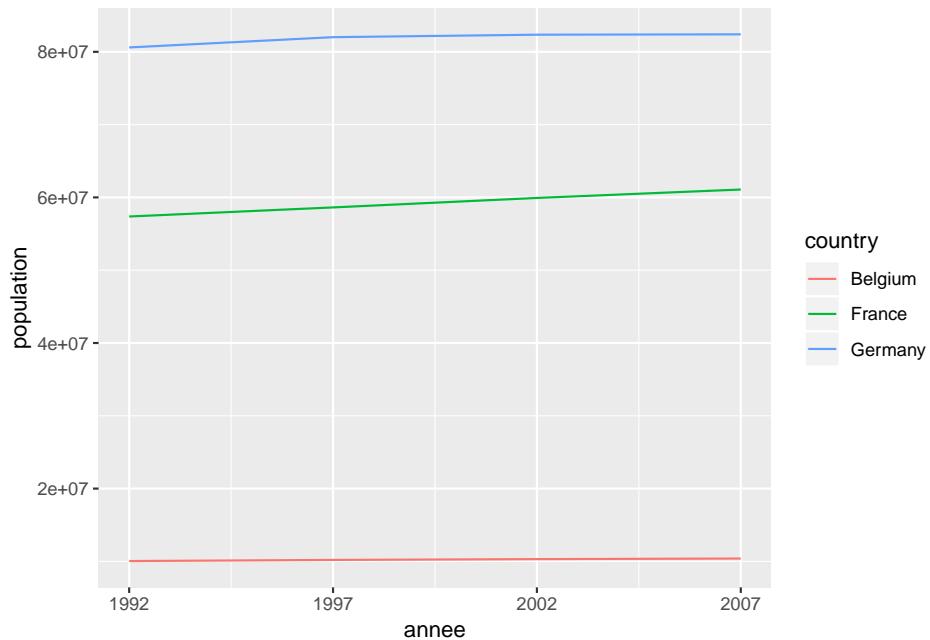
country	1992	1997	2002	2007
Belgium	10045622	10199787	10311970	10392226
France	57374179	58623428	59925035	61083916
Germany	80597764	82011073	82350671	82400996

Imaginons qu’on souhaite représenter avec `ggplot2` l’évolution de la population pour chaque pays sous forme de lignes : c’est impossible avec les données sous ce format. On a besoin d’arranger le tableau de la manière suivante :

country	annee	population
Belgium	1992	10045622
Belgium	1997	10199787
Belgium	2002	10311970
Belgium	2007	10392226
France	1992	57374179
France	1997	58623428
France	2002	59925035
France	2007	61083916
Germany	1992	80597764
Germany	1997	82011073
Germany	2002	82350671
Germany	2007	82400996

C'est seulement avec les données dans ce format qu'on peut réaliser le graphique :

```
ggplot(d) +
  geom_line(aes(x = annee, y = population, color = country)) +
  scale_x_continuous(breaks = unique(d$annee))
```



C'est la même chose pour `dplyr`, par exemple si on voulait calculer la population minimale pour chaque pays avec `summarise` :

```
d %>%
  group_by(country) %>%
  summarise(pop_min = min(population))
```

```
# A tibble: 3 x 2
  country   pop_min
  <fct>     <int>
1 Belgium 10045622
2 France  57374179
3 Germany 80597764
```

12.2 Trois règles pour des données bien rangées

Le concept de *tidy data* repose sur trois règles interdépendantes. Des données sont considérées comme *tidy* si :

1. chaque ligne correspond à une observation
2. chaque colonne correspond à une variable
3. chaque valeur est présente dans une unique case de la table ou, de manière équivalente, des unités d'observations différentes sont présentes dans des tables différentes

Ces règles ne sont pas forcément très intuitives. De plus, il y a une infinité de manières pour un tableau de données de ne pas être *tidy*.

Prenons par exemple les règles 1 et 2 et le tableau de notre premier exemple :

country	1992	1997	2002	2007
Belgium	10045622	10199787	10311970	10392226
France	57374179	58623428	59925035	61083916
Germany	80597764	82011073	82350671	82400996

Pourquoi ce tableau n'est pas *tidy* ? Parce que si on essaie d'identifier les variables mesurées dans le tableau, il y en a trois : le pays, l'année et la population. Or elles ne correspondent pas aux colonnes de la table. C'est le cas par contre pour la table transformée :

country	annee	population
Belgium	1992	10045622
Belgium	1997	10199787
Belgium	2002	10311970
Belgium	2007	10392226
France	1992	57374179
France	1997	58623428
France	2002	59925035
France	2007	61083916
Germany	1992	80597764
Germany	1997	82011073
Germany	2002	82350671
Germany	2007	82400996

On peut remarquer qu'en modifiant notre table pour satisfaire à la deuxième règle, on a aussi réglé la première : chaque ligne correspond désormais à une observation, en l'occurrence l'observation de trois pays à plusieurs moments dans le temps. Dans notre table d'origine, chaque ligne comportait en réalité quatre observations différentes.

Ce point permet d'illustrer le fait que les règles sont interdépendantes.

Autre exemple, généré depuis le jeu de données `nycflights13`, permettant cette fois d'illustrer la troisième règle :

year	month	day	dep_time	carrier	name
2013	1	1	517	UA	United Air Lines Inc.
2013	1	1	533	UA	United Air Lines Inc.
2013	1	1	542	AA	American Airlines Inc.
2013	1	1	554	UA	United Air Lines Inc.
2013	1	1	558	AA	American Airlines Inc.
2013	1	1	558	UA	United Air Lines Inc.
2013	1	1	558	UA	United Air Lines Inc.
2013	1	1	559	AA	American Airlines Inc.

Dans ce tableau on a bien une observation par ligne (un vol), et une variable par colonne. Mais on a une “infraction” à la troisième règle, qui est que chaque valeur doit être présente dans une unique case : si on regarde la colonne `name`, on a en effet une duplication de l'information concernant le nom des compagnies aériennes. Notre tableau mêle en fait deux types d'observations différents : des observations sur les vols, et des observations sur les compagnies aériennes.

Pour “arranger” ce tableau, il faut séparer les deux types d'observations en deux tables différentes :

year	month	day	dep_time	carrier
2013	1	1	517	UA
2013	1	1	533	UA
2013	1	1	542	AA
2013	1	1	554	UA
2013	1	1	558	AA
2013	1	1	558	UA
2013	1	1	558	UA
2013	1	1	559	AA

carrier	name
UA	United Air Lines Inc.
AA	American Airlines Inc.

On a désormais deux tables distinctes, l'information n'est pas dupliquée, et on peut facilement faire une jointure si on a besoin de récupérer l'information d'une table dans une autre.

12.3 Les verbes de `tidy`

L'objectif de `tidy` est de fournir des fonctions pour arranger ses données et les convertir dans un format *tidy*. Ces fonctions prennent la forme de verbes qui viennent compléter ceux de `dplyr` et s'intègrent parfaitement dans les séries de *pipes* (%>%), les *pipelines*, permettant d'enchaîner les opérations.

12.3.1 `pivot_longer` : transformer des colonnes en lignes

Prenons le tableau `d` suivant, qui liste la population de 6 pays en 2002 et 2007 :

country	2002	2007
Belgium	10311970	10392226
France	59925035	61083916
Germany	82350671	82400996
Italy	57926999	58147733
Spain	40152517	40448191
Switzerland	7361757	7554661

Dans ce tableau, une même variable (la population) est répartie sur plusieurs colonnes, chacune représentant une observation à un moment différent. On souhaite que la variable ne représente plus qu'une seule colonne, et que les observations soient réparties sur plusieurs lignes.

Pour cela on va utiliser la fonction `pivot_longer`¹ :

¹`pivot_longer` et `pivot_wider` ont été introduites dans la version 1.0 de `tidyverse`. Elles ont alors remplacé `gather` et `spread`.

```
d %>% pivot_longer(c(`2002`, `2007`))
```

```
# A tibble: 12 x 3
  country     name   value
  <fct>      <chr>  <int>
  1 Belgium   2002 10311970
  2 Belgium   2007 10392226
  3 France    2002 59925035
  4 France    2007 61083916
  5 Germany   2002 82350671
  6 Germany   2007 82400996
  7 Italy     2002 57926999
  8 Italy     2007 58147733
  9 Spain     2002 40152517
 10 Spain    2007 40448191
 11 Switzerland 2002 7361757
 12 Switzerland 2007 7554661
```

La fonction `pivot_longer` prend comme premier argument la liste des colonnes à rassembler (ici on a mis 2002 et 2007 entre *backticks* (`2002`) pour indiquer à `pivot_longer` qu'il s'agit d'un nom de colonne et pas d'un nombre). Ces colonnes peuvent être spécifiées avec la même syntaxe que celle de la fonction `select` de `dplyr`.

Par exemple, il est parfois plus rapide d'indiquer à `pivot_longer` les colonnes qu'on ne souhaite pas “rassembler”. On peut le faire avec la syntaxe suivante :

```
d %>% pivot_longer(-country)
```

```
# A tibble: 12 x 3
  country     name   value
  <fct>      <chr>  <int>
  1 Belgium   2002 10311970
  2 Belgium   2007 10392226
  3 France    2002 59925035
  4 France    2007 61083916
  5 Germany   2002 82350671
  6 Germany   2007 82400996
  7 Italy     2002 57926999
  8 Italy     2007 58147733
  9 Spain     2002 40152517
```

```

10 Spain      2007  40448191
11 Switzerland 2002  7361757
12 Switzerland 2007  7554661

```

Par défaut, les colonnes qui contiennent les noms des colonnes d'origine et leurs valeurs sont nommées `name` et `value`. Si cela ne convient pas, on peut indiquer les noms à utiliser via les arguments `names_to` et `values_to` :

```
d %>% pivot_longer(c(`2002`, `2007`), names_to = "population", values_to = "annee")
```

```

# A tibble: 12 x 3
  country   population    annee
  <fct>     <chr>        <int>
1 Belgium    2002        10311970
2 Belgium    2007        10392226
3 France     2002        59925035
4 France     2007        61083916
5 Germany    2002        82350671
6 Germany    2007        82400996
7 Italy       2002        57926999
8 Italy       2007        58147733
9 Spain       2002        40152517
10 Spain      2007        40448191
11 Switzerland 2002      7361757
12 Switzerland 2007      7554661

```

Au final, le nom de `pivot_longer` s'explique par le fait qu'on fait “pivoter” notre tableau de départ d'un format “large” (avec plus de colonnes) vers un format “long” (avec plus de lignes).

12.3.2 `pivot_wider` : transformer des lignes en colonnes

La fonction `pivot_wider` est l'inverse de `pivot_longer`.

Soit le tableau `d` suivant :

country	continent	year	variable	value
Belgium	Europe	2002	lifeExp	78.320
Belgium	Europe	2002	pop	10311970.000
Belgium	Europe	2007	lifeExp	79.441
Belgium	Europe	2007	pop	10392226.000
France	Europe	2002	lifeExp	79.590
France	Europe	2002	pop	59925035.000
France	Europe	2007	lifeExp	80.657
France	Europe	2007	pop	61083916.000

Ce tableau a le problème inverse du précédent : on a deux variables, `lifeExp` et `pop` qui, plutôt que d'être réparties en deux colonnes, sont réparties entre plusieurs lignes.

On va donc utiliser `pivot_wider` pour répartir ces lignes dans deux colonnes différentes :

```
d %>% pivot_wider(names_from = variable, values_from = value)
```

```
# A tibble: 6 x 5
  country continent year lifeExp     pop
  <fct>    <fct>   <int>   <dbl>   <dbl>
1 Belgium Europe     2002     78.3 10311970
2 Belgium Europe     2007     79.4 10392226
3 France  Europe     2002     79.6 59925035
4 France  Europe     2007     80.7 61083916
5 Germany Europe     2002     78.7 82350671
6 Germany Europe     2007     79.4 82400996
```

`pivot_wider` prend deux arguments principaux :

- `names_from` indique la colonne contenant les noms des nouvelles variables à créer
- `values_from` indique la colonne contenant les valeurs de ces variables

Il peut arriver que certaines variables soient absentes pour certaines observations. Dans ce cas l'argument `values_fill` permet de spécifier la valeur à utiliser pour ces données manquantes (par défaut les valeurs absentes sont, logiquement, indiquées par des NA).

Exemple avec le tableau `d` suivant :

country	continent	year	variable	value
Belgium	Europe	2002	lifeExp	78.320
Belgium	Europe	2002	pop	10311970.000
Belgium	Europe	2007	lifeExp	79.441
Belgium	Europe	2007	pop	10392226.000
France	Europe	2002	lifeExp	79.590
France	Europe	2002	pop	59925035.000
France	Europe	2007	lifeExp	80.657
France	Europe	2007	pop	61083916.000
Germany	Europe	2002	lifeExp	78.670
Germany	Europe	2002	pop	82350671.000
Germany	Europe	2007	lifeExp	79.406
Germany	Europe	2007	pop	82400996.000
France	Europe	2002	density	94.000

```
d %>%
  pivot_wider(names_from = variable, values_from = value)
```

```
# A tibble: 6 x 6
  country continent year lifeExp      pop density
  <chr>    <chr>   <dbl>    <dbl>    <dbl>    <dbl>
1 Belgium Europe     2002     78.3 10311970     NA
2 Belgium Europe     2007     79.4 10392226     NA
3 France  Europe     2002     79.6 59925035     94
4 France  Europe     2007     80.7 61083916     NA
5 Germany Europe     2002     78.7 82350671     NA
6 Germany Europe     2007     79.4 82400996     NA
```

```
d %>%
  pivot_wider(
  names_from = variable, values_from = value,
  values_fill = list(value = 0)
)
```

```
# A tibble: 6 x 6
  country continent year lifeExp      pop density
  <chr>    <chr>   <dbl>    <dbl>    <dbl>    <dbl>
1 Belgium Europe     2002     78.3 10311970     0
2 Belgium Europe     2007     79.4 10392226     0
3 France  Europe     2002     79.6 59925035     94
```

4	France	Europe	2007	80.7	61083916	0
5	Germany	Europe	2002	78.7	82350671	0
6	Germany	Europe	2007	79.4	82400996	0

Au final, le nom de `pivot_wider` s'explique par le fait qu'on fait “pivoter” notre tableau de départ d'un format “long” (avec plus de lignes) vers un format “large” (avec plus de colonnes).

12.3.3 `separate` : séparer une colonne en plusieurs

Parfois on a plusieurs informations réunies en une seule colonne et on souhaite les séparer. Soit le tableau d'exemple caricatural suivant, nommé `df` :

eleve	note
Félicien Machin	5/20
Raymonde Bidule	6/10
Martial Truc	87/100

`separate` permet de séparer la colonne `note` en deux nouvelles colonnes `note` et `note_sur` :

```
df %>% separate(note, c("note", "note_sur"))
```

```
# A tibble: 3 x 3
  eleve           note note_sur
  <chr>          <chr> <chr>
1 Félicien Machin 5     20
2 Raymonde Bidule 6     10
3 Martial Truc   87    100
```

`separate` prend deux arguments principaux, le nom de la colonne à séparer et un vecteur indiquant les noms des nouvelles variables à créer. Par défaut `separate` “sépare” au niveau des caractères non-alphanumérique (espace, symbole, etc.). On peut lui indiquer explicitement le caractère sur lequel séparer avec l'argument `sep` :

```
df %>% separate(eleve, c("prenom", "nom"), sep = " ")
```

```
# A tibble: 3 x 3
  prenom    nom    note
```

```
<chr>    <chr>  <chr>
1 Félicien Machin 5/20
2 Raymonde Bidule 6/10
3 Martial Truc   87/100
```

12.3.4 `unite` : regrouper plusieurs colonnes en une seule

`unite` est l'opération inverse de `separate`. Elle permet de regrouper plusieurs colonnes en une seule. Imaginons qu'on obtient le tableau `d` suivant :

code_departement	code_commune	commune	pop_tot
01	004	Ambérieu-en-Bugey	14233
01	007	Ambronay	2437
01	014	Arbent	3440
01	024	Attignat	3110
01	025	Bâgé-la-Ville	3130
01	027	Balan	2785

On souhaite reconstruire une colonne `code_insee` qui indique le code INSEE de la commune, et qui s'obtient en concaténant le code du département et celui de la commune. On peut utiliser `unite` pour cela :

```
d %>% unite(code_insee, code_departement, code_commune)
```

```
# A tibble: 6 x 3
  code_insee  commune      pop_tot
  <chr>        <chr>     <int>
1 01_004      Ambérieu-en-Bugey 14233
2 01_007      Ambronay       2437
3 01_014      Arbent         3440
4 01_024      Attignat       3110
5 01_025      Bâgé-la-Ville  3130
6 01_027      Balan          2785
```

Le résultat n'est pas idéal : par défaut `unite` ajoute un caractère `_` entre les deux valeurs concaténées, alors qu'on ne veut aucun séparateur. De plus, on souhaite conserver nos deux colonnes d'origine, qui peuvent nous être utiles. On peut résoudre ces deux problèmes à l'aide des arguments `sep` et `remove` :

```
d %>%
  unite(code_insee, code_departement, code_commune,
        sep = "", remove = FALSE)

# A tibble: 6 x 5
  code_insee code_departement code_commune commune      pop_tot
  <chr>       <chr>          <chr>        <chr>      <int>
1 01004       01              004         Ambérieu-en-Bugey 14233
2 01007       01              007         Ambronay     2437
3 01014       01              014         Arbent       3440
4 01024       01              024         Attignat    3110
5 01025       01              025         Bâgé-la-Ville 3130
6 01027       01              027         Balan       2785
```

12.3.5 `extract` : créer de nouvelles colonnes à partir d'une colonne de texte

`extract` permet de créer de nouvelles colonnes à partir de sous-chaînes d'une colonne de texte existante, identifiées par des groupes dans une expression régulière.

Par exemple, à partir du tableau suivant :

eleve	note
Félicien Machin	5/20
Raymonde Bidule	6/10
Martial Truc	87/100

On peut extraire les noms et prénoms dans deux nouvelles colonnes avec :

```
df %>% extract(eleve,
  c("prenom", "nom"),
  "^(.*)(.*)$")
```

```
# A tibble: 3 x 3
  prenom   nom    note
  <chr>    <chr>  <chr>
1 Félicien Machin 5/20
2 Raymonde Bidule 6/10
3 Martial Truc   87/100
```

On passe donc à `extract` trois arguments : la colonne d'où on doit extraire les valeurs, un vecteur avec les noms des nouvelles colonnes à créer, et une expression régulière comportant autant de groupes (identifiés par des parenthèses) que de nouvelles colonnes.

Par défaut la colonne d'origine n'est pas conservée dans la table résultat. On peut modifier ce comportement avec l'argument `remove = FALSE`. Ainsi, le code suivant extrait les initiales du prénom et du nom mais conserve la colonne d'origine :

```
df %>% extract(eleve,
  c("initial_prenom", "initial_nom"),
  "^(.)* (.)*$",
  remove = FALSE)
```

```
# A tibble: 3 x 4
  eleve      initial_prenom initial_nom note
  <chr>       <chr>          <chr>        <chr>
1 Félicien    M              5/20
2 Raymonde   R              6/10
3 Martial     T              87/100
```

12.3.6 `complete` : compléter des combinaisons de variables manquantes

Imaginons qu'on ait le tableau de résultats suivants :

eleve	matiere	note
Alain	Maths	16
Alain	Français	9
Barnabé	Maths	17
Chantal	Français	11

Les élèves Barnabé et Chantal n'ont pas de notes dans toutes les matières. Supposons que c'est parce qu'ils étaient absents et que leur note est en fait un 0. Si on veut calculer les moyennes des élèves, on doit compléter ces notes manquantes.

La fonction `complete` est prévue pour ce cas de figure : elle permet de compléter des combinaisons manquantes de valeurs de plusieurs colonnes.

On peut l'utiliser de cette manière :

```
df %>% complete(eleve, matiere)
```

```
# A tibble: 6 x 3
  eleve    matiere    note
  <chr>    <chr>     <dbl>
1 Alain    Français   9
2 Alain    Maths      16
3 Barnabé Français  NA
4 Barnabé Maths      17
5 Chantal  Français  11
6 Chantal  Maths      NA
```

On voit que les combinaisons manquante “Barnabé - Français” et “Chantal - Maths” ont bien été ajoutées par `complete`.

Par défaut les lignes insérées récupèrent des valeurs manquantes `NA` pour les colonnes restantes. On peut néanmoins choisir une autre valeur avec l’argument `fill`, qui prend la forme d’une liste nommée :

```
df %>% complete(eleve, matiere, fill = list(note = 0))
```

```
# A tibble: 6 x 3
  eleve    matiere    note
  <chr>    <chr>     <dbl>
1 Alain    Français   9
2 Alain    Maths      16
3 Barnabé Français  0
4 Barnabé Maths      17
5 Chantal  Français  11
6 Chantal  Maths      0
```

Parfois on ne souhaite pas inclure toutes les colonnes dans le calcul des combinaisons de valeurs. Par exemple, supposons qu’on rajoute dans notre tableau une colonne avec les identifiants de chaque élève :

	id	eleve	matiere	note
1001001		Alain	Maths	16
1001001		Alain	Français	9
1001002		Barnabé	Maths	17
1001003		Chantal	Français	11

Si on applique `complete` comme précédemment, le résultat n’est pas bon car il contient toutes les combinaisons de `id`, `eleve` et `matiere`.

```
df %>% complete(id, eleve, matiere)
```

```
# A tibble: 18 x 4
  id eleve   matiere   note
  <dbl> <chr>   <chr>     <dbl>
1 1001001 Alain   Français  9
2 1001001 Alain   Maths     16
3 1001001 Barnabé Français NA
4 1001001 Barnabé Maths    NA
5 1001001 Chantal Français NA
6 1001001 Chantal Maths    NA
7 1001002 Alain   Français  NA
8 1001002 Alain   Maths     NA
9 1001002 Barnabé Français NA
10 1001002 Barnabé Maths    17
11 1001002 Chantal Français NA
12 1001002 Chantal Maths    NA
13 1001003 Alain   Français  NA
14 1001003 Alain   Maths     NA
15 1001003 Barnabé Français NA
16 1001003 Barnabé Maths    NA
17 1001003 Chantal Français 11
18 1001003 Chantal Maths    NA
```

Dans ce cas, pour signifier à `complete` que `id` et `eleve` sont deux attributs d'un même individu et ne doivent pas être combinés entre eux, on doit les placer dans une fonction `nesting` :

```
df %>% complete(nesting(id, eleve), matiere)
```

```
# A tibble: 6 x 4
  id eleve   matiere   note
  <dbl> <chr>   <chr>     <dbl>
1 1001001 Alain   Français  9
2 1001001 Alain   Maths     16
3 1001002 Barnabé Français NA
4 1001002 Barnabé Maths    17
5 1001003 Chantal Français 11
6 1001003 Chantal Maths    NA
```

12.4 Ressources

Chaque jeu de données est différent, et le travail de remise en forme est souvent long et plus ou moins compliqué. On n'a donné ici que les exemples les plus simples, et c'est souvent en combinant différentes opérations qu'on finit par obtenir le résultat souhaité.

Le livre *R for data science*, librement accessible en ligne, contient [un chapitre complet](#) sur la remise en forme des données.

L'article [Tidy data](#), publié en 2014 dans le *Journal of Statistical Software*, présente de manière détaillée le concept éponyme (mais il utilise des extensions désormais obsolètes qui ont depuis été remplacées par `dplyr` et `tidyverse`).

Le site de l'extension est accessible à l'adresse : <https://tidyverse.org/> et contient une liste des fonctions et les pages d'aide associées.

Chapitre 13

Diffuser et publier avec rmarkdown

L'extension **rmarkdown** permet de générer des documents de manière dynamique en mélangeant texte mis en forme et résultats produits par du code R. Les documents générés peuvent être au format HTML, PDF, Word, et bien d'autres¹. C'est donc un outil très pratique pour l'exportation, la communication et la diffusion de résultats d'analyse.

Le présent document a lui-même été généré à partir de fichiers R Markdown².

rmarkdown ne fait pas partie du *tidyverse*, mais elle est installée et chargée par défaut par RStudio³.

Voici un exemple de document R Markdown minimal :

```
---
```

```
title: "Test R Markdown"
```

```
---
```



```
*R Markdown* permet de mélanger :
```



```
- du texte libre mis en forme
```

```
- des blocs de code R
```

¹On peut citer les formats **odt**, **rtf**, **Markdown**, etc.

²Plus précisément grâce à l'extension **bookdown** qui permet de générer des documents de type livre.

³Si vous n'utilisez pas ce dernier, l'extension peut être installée à part avec `install.packages("rmarkdown")` et chargée explicitement avec `library(rmarkdown)`.

Les blocs de code sont exécutés et leur résultat affiché, par exemple :

```
```{r}
mean(mtcars$mpg)
```
```

Graphiques

On peut également inclure des graphiques :

```
```{r}
plot(mtcars$hp, mtcars$mpg)
```
```

Ce document peut être “compilé” sous différents formats. Lors de cette étape, le texte est mis en forme, les blocs de code sont exécutés, leur résultat ajouté au document, et le tout est transformé dans un des différents formats possibles.

Voici le rendu du document précédent au format HTML :

Test R Markdown

R Markdown permet de mélanger :

- du texte libre mis en forme
- des blocs de code R

Les blocs de code sont exécutés et leur résultat affiché, par exemple :

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

Graphiques

On peut également inclure des graphiques :

```
plot(mtcars$hp, mtcars$mpg)
```

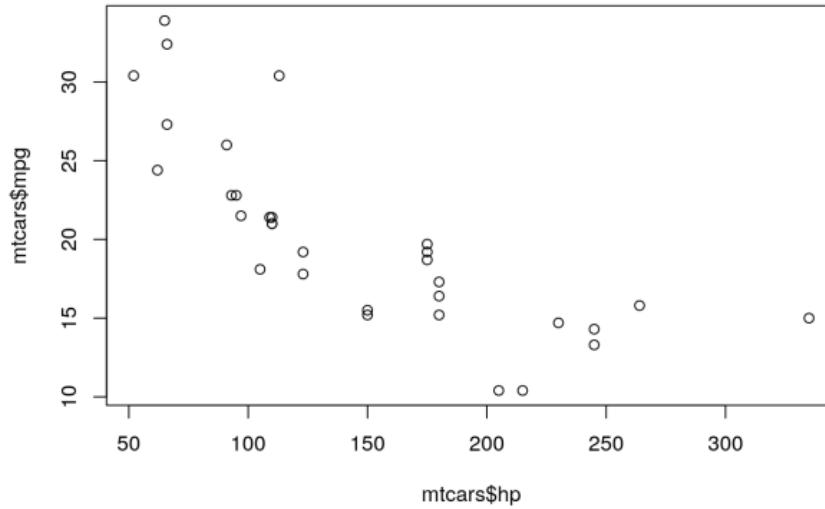


Figure 13.1: Rendu HTML

Le rendu du même document au format PDF :

Test R Markdown

R Markdown permet de mélanger :

- du texte libre mis en forme
- des blocs de code R

Les blocs de code sont exécutés et leur résultat affiché, par exemple :

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

Graphiques

On peut également inclure des graphiques :

```
plot(mtcars$hp, mtcars$mpg)
```

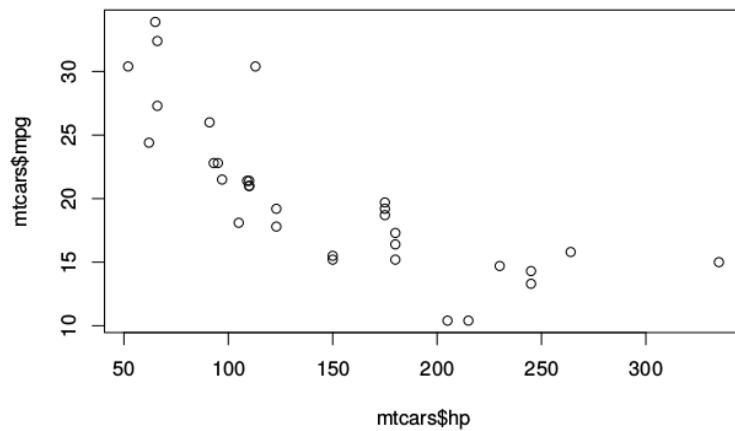


Figure 13.2: Rendu PDF

Et le rendu au format docx :

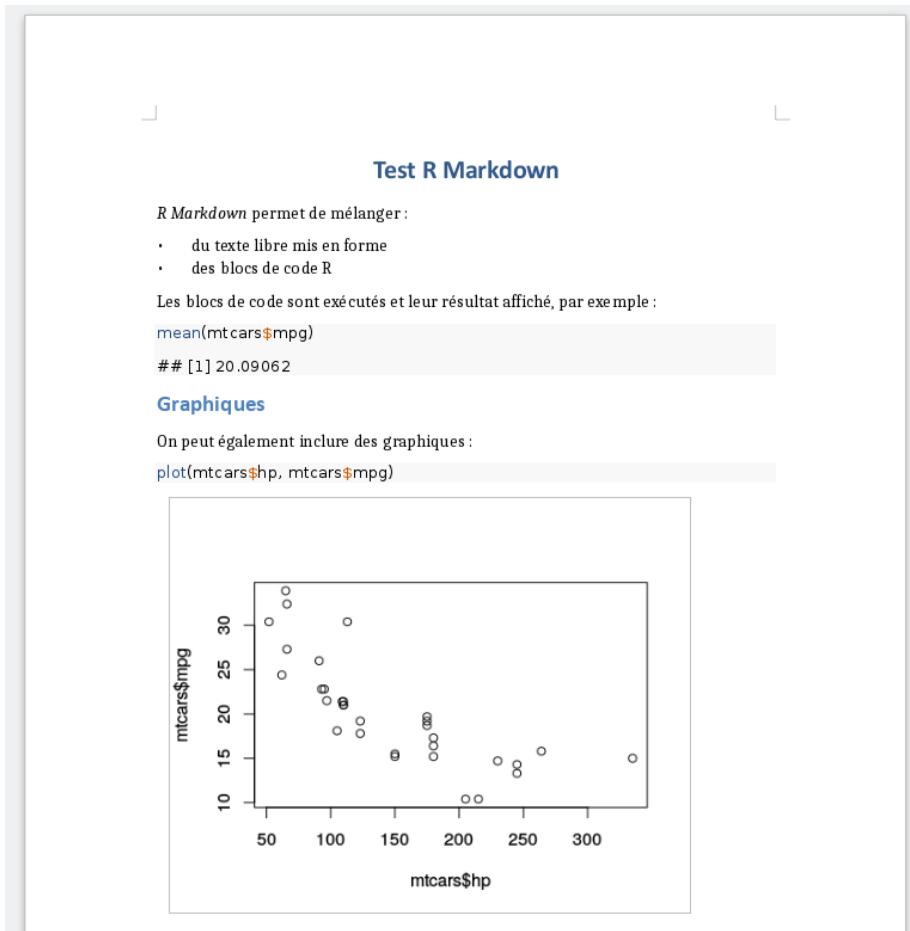


Figure 13.3: Rendu docx

Les avantages de ce système sont nombreux :

- le code et ses résultats ne sont pas séparés des commentaires qui leur sont associés
- le document final est reproductible
- le document peut être très facilement régénéré et mis à jour, par exemple si les données source ont été modifiées.

13.1 Crer un nouveau document

Un document R Markdown est un simple fichier texte enregistr avec l'extension `.Rmd`.

Sous RStudio, on peut crer un nouveau document en allant dans le menu *File* puis en choisissant *New file* puis *R Markdown*.... La bote de dialogue suivante s'affiche :

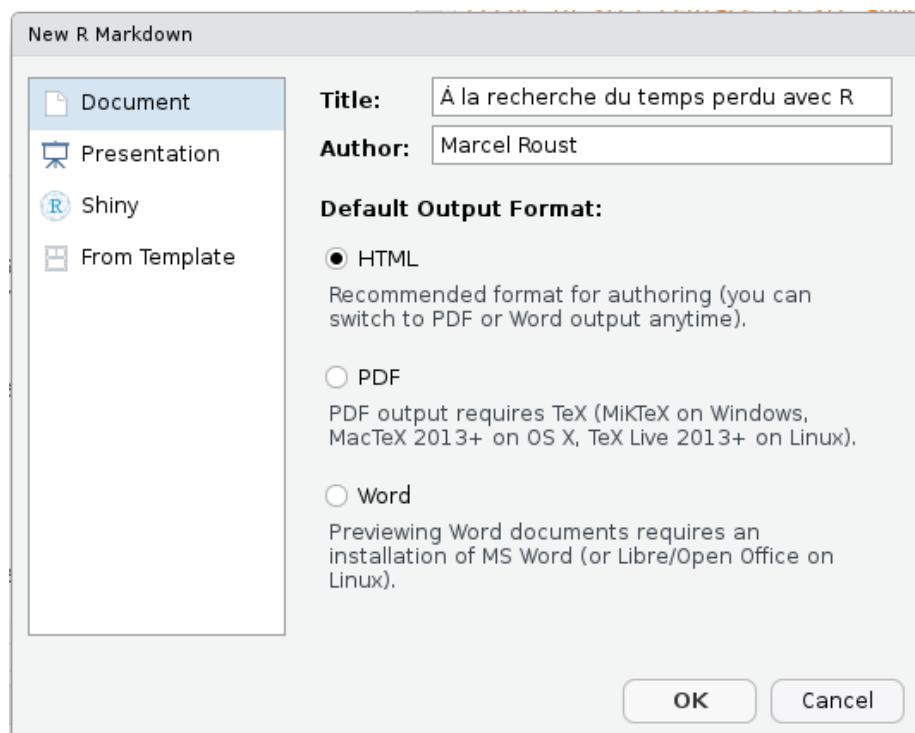


Figure 13.4: Cration d'un document R Markdown

On peut indiquer le titre, l'auteur du document ainsi que le format de sortie par dfaut (il est possible de modifier facilement ces lments par la suite). Plutt qu'un document classique, on verra section 13.6 qu'on peut aussi choisir de crer une prsentation sous forme de slides (entre *Presentation*) ou de crer un document  partir d'un modle (Entre *From Template*).

Un fichier comportant un contenu d'exemple s'affiche alors. Vous pouvez l'enregistrer o vous le souhaitez avec une extension `.Rmd`.

13.2 Éléments d'un document R Markdown

Un document R Markdown est donc un fichier texte qui ressemble à quelque chose comme ça :

```
---
```

```
title: "Titre"
author: "Prénom Nom"
date: "10 avril 2017"
output: html_document
---
```

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## Introduction

Ceci est un document RMarkdown, qui mélange :

- du texte balisé selon la syntaxe Markdown
- des bouts de code R qui seront exécutés

Le code R se présente de la manière suivante :

```{r}
summary(cars)
```

## Graphiques

On peut aussi inclure des graphiques, par exemple :

```{r}
plot(pressure)
```
```

On va décomposer les différents éléments constitutifs de ce document.

13.2.1 En-tête (préambule)

La première partie du document est son *en-tête*. Il se situe en tout début de document, et est délimité par trois tirets (---) avant et après :

```
---
title: "Titre"
author: "Prénom Nom"
date: "10 avril 2017"
output: html_document
---
```

Cet en-tête contient les métadonnées du document, comme son titre, son auteur, sa date, plus tout un tas d'options possibles qui vont permettre de configurer ou personnaliser l'ensemble du document et son rendu. Ici, par exemple, la ligne `output: html_document` indique que le document généré doit être au format HTML.

13.2.2 Texte du document

Le corps du document est constitué de texte qui suit la syntaxe *Markdown*. Un fichier Markdown est un fichier texte contenant un balisage léger qui permet de définir des niveaux de titres ou de mettre en forme le texte. Par exemple, le texte suivant :

```
Ceci est du texte avec *de l'italique* et **du gras**.

On peut définir des listes à puces :

- premier élément
- deuxième élément
```

Génèrera le texte mis en forme suivant :

Ceci est du texte avec de l'*italique* et du **gras**.

On peut définir des listes à puces :

- premier élément
- deuxième élément

On voit que des mots placés entre des astérisques sont mis en italique, des lignes qui commencent par un tiret sont transformés en liste à puce, etc.

On peut définir des titres de différents niveaux en faisant débuter une ligne par un ou plusieurs # :

```
# Titre de niveau 1
```

```
## Titre de niveau 2  
### Titre de niveau 3
```

Quand des titres ont été définis, si vous cliquez sur l'icône *Show document outline* totalement à droite de la barre d'outils associée au fichier R Markdown, une table des matières générée automatiquement à partir des titres s'affiche et vous permet de naviguer facilement dans le document :

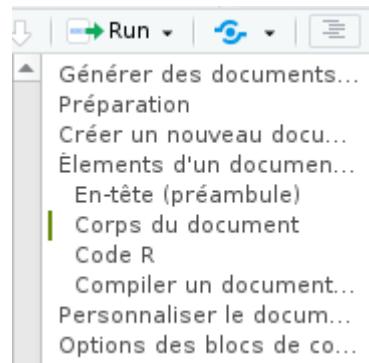


Figure 13.5: Table des matières dynamique

La syntaxe Markdown permet d'autres mises en forme, comme la possibilité d'insérer des liens ou des images. Par exemple, le code suivant :

```
[Exemple de lien](https://example.com)
```

Donnera le lien suivant :

[Exemple de lien](#)

Dans RStudio, le menu *Help* puis *Markdown quick reference* donne un aperçu plus complet de la syntaxe.

13.2.3 Blocs de code R

En plus du texte libre au format Markdown, un document R Markdown contient, comme son nom l'indique, du code R. Celui-ci est inclus dans des blocs (*chunks*) délimités par la syntaxe suivante :

```
```{r}
x <- 1:5
```
```

Comme cette suite de caractères n'est pas très simple à saisir, vous pouvez utiliser le menu *Insert* de RStudio et choisir *R*⁴, ou utiliser le raccourci clavier **Ctrl+Alt+i**.

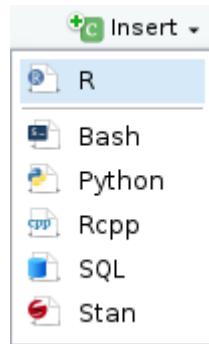


Figure 13.6: Menu d'insertion d'un bloc de code

Dans RStudio les blocs de code R sont en général affichés avec une couleur de fond légèrement différente pour les distinguer du reste du document.

Quand votre curseur se trouve dans un bloc, vous pouvez saisir le code R que vous souhaitez, l'exécuter, utiliser l'autocomplétion, exactement comme si vous vous trouviez dans un script R. Vous pouvez également exécuter l'ensemble du code contenu dans un bloc à l'aide du raccourci clavier **Ctrl+Maj+Entrée**.

Dans RStudio, par défaut, les résultats d'un bloc de code (texte, tableau ou graphique) s'affichent directement *dans* la fenêtre d'édition du document, permettant de les visualiser facilement et de les conserver le temps de la session⁵.

Lorsque le document est “compilé” au format HTML, PDF ou docx, chaque bloc est exécuté tour à tour, et le résultat inclus dans le document final, qu'il s'agisse de texte, d'un tableau ou d'un graphique. Les blocs sont liés entre eux, dans le sens où les données importées ou calculées dans un bloc sont accessibles aux blocs suivants. On peut donc aussi voir un document R Markdown comme un script R dans lequel on aurait intercalé du texte libre au format Markdown.

⁴Il est possible d'inclure dans un document R Markdown des blocs de code d'autres langages

⁵Ce comportement peut être modifié en cliquant sur l'icône d'engrenage de la barre d'outils et en choisissant *Chunk Output in Console*



À noter qu'avant chaque compilation, une nouvelle session R est lancée, ne contenant aucun objet. Les premiers blocs de code d'un document sont donc souvent utilisés pour importer des données, exécuter des recodages, etc.

13.2.4 Compiler un document (*Knit*)

On peut à tout moment compiler, ou plutôt “tricoter” (*Knit*), un document R Markdown pour obtenir et visualiser le document généré. Pour cela, il suffit de cliquer sur le bouton *Knit* et de choisir le format de sortie voulu :

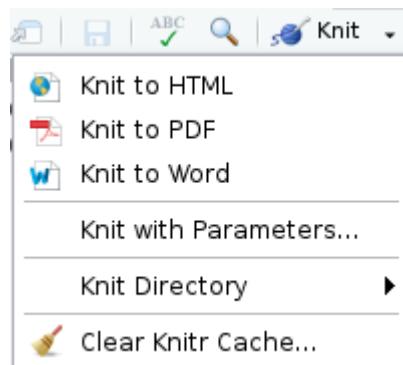


Figure 13.7: Menu *Knit*

Vous pouvez aussi utiliser le raccourci **Ctrl+Maj+K** pour compiler le document dans le dernier format utilisé.



Pour la génération du format PDF, vous devez avoir une installation fonctionnelle de LaTeX sur votre système.

Si ça n'est pas le cas, l'extension `tinytex` de Yihui Xie vise à faciliter l'installation d'une distribution LaTeX minimale quel que soit le système d'exploitation de votre machine. Pour l'utiliser il vous faut d'abord installer l'extension avec `install.packages('tinytex')`, puis lancer la commande suivante dans la console (prévoir un téléchargement d'environ 200Mo) :

```
tinytex::install_tinytex()
```

Plus d'informations sur [le site de tinytex](#).

Un onglet *R Markdown* s'ouvre dans la même zone que l'onglet *Console* et indique la progression de la compilation, ainsi que les messages d'erreur éventuels. Si tout se passe bien, Le document devrait s'afficher soit dans une fenêtre *Viewer* de RStudio (pour la sortie HTML), soit dans le logiciel par défaut de votre ordinateur.

13.3 Personnaliser le document généré

La personnalisation du document généré se fait en modifiant des options dans le préambule du document. RStudio propose néanmoins une petite interface graphique permettant de changer ces options plus facilement. Pour cela, cliquez sur l'icône en forme d'engrenage à droite du bouton *Knit* et choisissez *Output Options...*

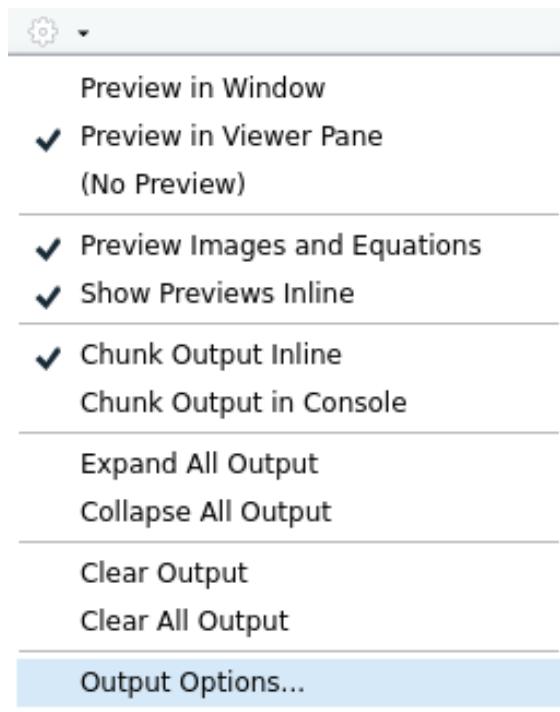


Figure 13.8: Options de sortie R Markdown

Une boîte de dialogue s'affiche vous permettant de sélectionner le format de sortie souhaité et, selon le format, différentes options :

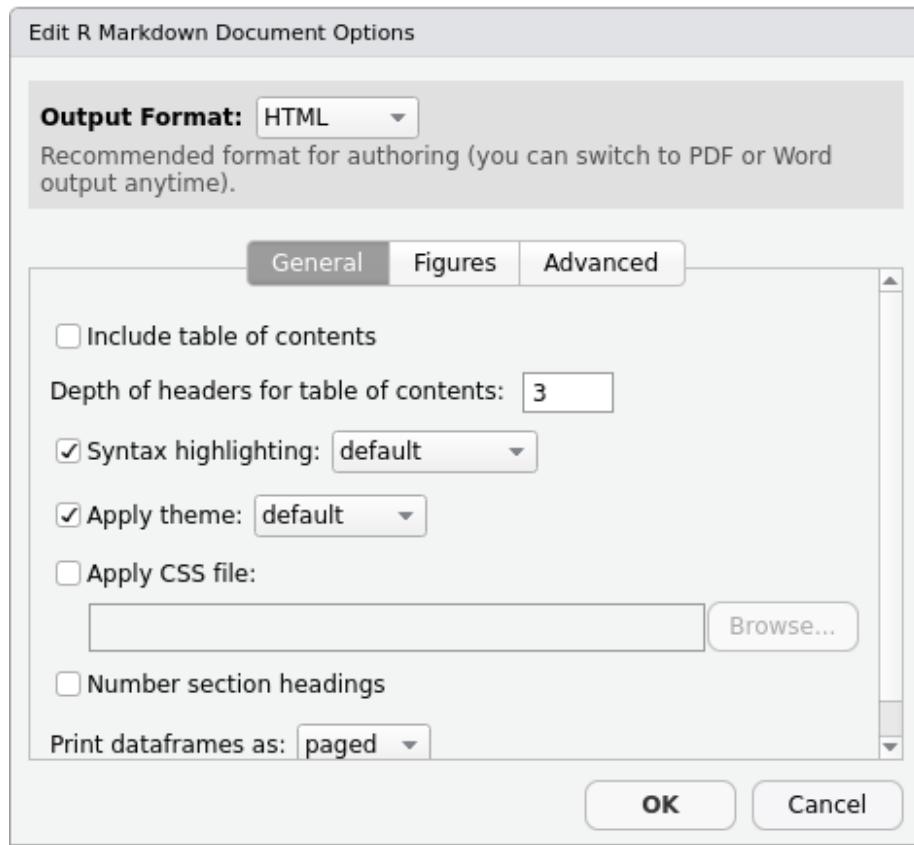


Figure 13.9: Dialogue d'options de sortie R Markdown

Pour le format HTML par exemple, l'onglet *General* vous permet de spécifier si vous voulez une table des matières, sa profondeur, les thèmes à appliquer pour le document et la coloration syntaxique des blocs R, etc. L'onglet *Figures* vous permet de changer les dimensions par défaut des graphiques générés.



Une option très intéressante pour les fichiers HTML, accessible via l'onglet *Advanced*, est l'entrée *Create standalone HTML document*. Si elle est cochée (ce qui est le cas par défaut), le document HTML généré contiendra en un seul fichier le code HTML mais aussi les images et toutes les autres ressources nécessaires à son affichage. Ceci permet de générer des fichiers (parfois assez volumineux) que vous pouvez transférer très facilement à quelqu'un par mail ou en le mettant en ligne quelque part. Si la case n'est pas cochée, les images et autres ressources sont placées dans un dossier à part.

Lorsque vous changez des options, RStudio va en fait modifier le préambule de votre document. Ainsi, si vous choisissez d'afficher une table des matières et de modifier le thème de coloration syntaxique, votre en-tête va devenir quelque chose comme :

```
---
title: "Test R Markdown"
output:
  html_document:
    highlight: kate
    toc: yes
---
```

Vous pouvez modifier les options directement en éditant le préambule.

À noter qu'il est possible de spécifier des options différentes selon les formats, par exemple :

```
---
title: "Test R Markdown"
output:
  html_document:
    highlight: kate
    toc: yes
  pdf_document:
    fig_caption: yes
    highlight: kate
---
```

La liste complète des options possibles est présente sur [le site de la documentation officielle](#) (très complet et bien fait) et sur l'antisèche et le guide de référence, accessibles depuis RStudio via le menu *Help* puis *Cheatsheets*.

13.4 Options des blocs de code R

Il est également possible de passer des options à chaque bloc de code R pour modifier son comportement.

On rappelle qu'un bloc de code se présente de la manière suivante :

```
```{r}
x <- 1:5
```
```

Les options d'un bloc de code sont à placer à l'intérieur des accolades `{r}`.

13.4.1 Nom du bloc

La première possibilité est de donner un *nom* au bloc. Celui-ci est indiqué directement après le `r` :

```
{r nom_du_bloc}
```

Il n'est pas obligatoire de nommer un bloc, mais cela peut être utile en cas d'erreur à la compilation, pour identifier le bloc ayant causé le problème. Attention, on ne peut pas avoir deux blocs avec le même nom.

13.4.2 Options

En plus d'un nom, on peut passer à un bloc une série d'options sous la forme `option = valeur`. Voici un exemple de bloc avec un nom et des options :

```
```{r mon_bloc, echo = FALSE, warning = TRUE}
x <- 1:5
```
```

Et un exemple de bloc non nommé avec des options :

```
```{r echo = FALSE, warning = FALSE}
x <- 1:5
```
```

Une des options la plus utile est l'option `echo`. Par défaut `echo` vaut `TRUE`, et le bloc de code R est inséré dans le document généré, de cette manière :

```
x <- 1:5
print(x)
```

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

Mais si on positionne l'option `echo=FALSE`, alors le code R n'est plus inséré dans le document, et seul le résultat est visible :

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

Voici une liste de quelques unes des options disponibles :

| Option | Valeurs | Description |
|---------|---------------------------------|--|
| echo | TRUE/FALSE | Afficher ou non le code R dans le document |
| eval | TRUE/FALSE | Exécuter ou non le code R à la compilation |
| include | TRUE/FALSE | Inclure ou non le code R et ses résultats dans le document |
| results | "hide"/"asis"/"asisType/dchose" | Renvoyés par le bloc de code |
| warning | TRUE/FALSE | Afficher ou non les avertissements générés par le bloc |
| message | TRUE/FALSE | Afficher ou non les messages générés par le bloc |

Il existe de nombreuses autres options décrites notamment dans [guide de référence R Markdown](#) (PDF en anglais).

13.4.3 Modifier les options

Il est possible de modifier les options manuellement en éditant l'en-tête du bloc de code, mais on peut aussi utiliser une petite interface graphique proposée par RStudio. Pour cela, il suffit de cliquer sur l'icône d'engrenage située à droite sur la ligne de l'en-tête de chaque bloc :



Figure 13.10: Menu d'options de bloc de code

Vous pouvez ensuite modifier les options les plus courantes, et cliquer sur *Apply* pour les appliquer.

13.4.4 Options globales

On peut vouloir appliquer une option à l'ensemble des blocs d'un document. Par exemple, on peut souhaiter par défaut ne pas afficher le code R de chaque bloc dans le document final.

On peut positionner une option globalement en utilisant la fonction `knitr::opts_chunk$set()`. Par exemple, insérer `knitr::opts_chunk$set(echo = FALSE)` dans un bloc de code positionnera l'option `echo = FALSE` par défaut pour tous les blocs suivants.

En général, on place toutes ces modifications globales dans un bloc spécial nommé `setup` et qui est le premier bloc du document :

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```



Par défaut RStudio exécute systématiquement le contenu du bloc `setup` avant d'exécuter celui d'un autre bloc.

Contrairement aux autres blocs de code, quand on utilise dans RStudio le menu des paramètres du bloc `setup` pour modifier ses options, celles-ci modifient non pas les options de ce bloc mais les options globales, en mettant à jour l'appel de la fonction `knitr::opts_chunk$set()`.

13.4.5 Mise en cache des résultats

Compiler un document R Markdown peut être long, car il faut à chaque fois exécuter l'ensemble des blocs de code R qui le constituent.

Pour accélérer cette opération, R Markdown utilise un système de *mise en cache* : les résultats de chaque bloc sont enregistrés dans un fichier et à la prochaine compilation, si le code et les options du bloc n'ont pas été modifiés, c'est le contenu du fichier de cache qui est utilisé, ce qui évite d'exécuter le code R.



On peut activer ou désactiver la mise en cache des résultats pour chaque bloc de code avec l'option `cache = TRUE` ou `cache = FALSE`, et on peut aussi désactiver totalement la mise en cache pour le document en ajoutant `knitr::opts_chunk$set(cache = FALSE)` dans le premier bloc `setup`.

Ce système de cache peut poser problème par exemple si les données source changent : dans ce cas les résultats de certains blocs peuvent ne pas être mis à jour s'ils sont présents en cache. Dans ce cas, on peut vider le cache du

document, ce qui forcera un recalcul de tous les blocs de code à la prochaine compilation. Pour cela, vous pouvez ouvrir le menu *Knit* et choisir *Clear Knitr Cache...* :

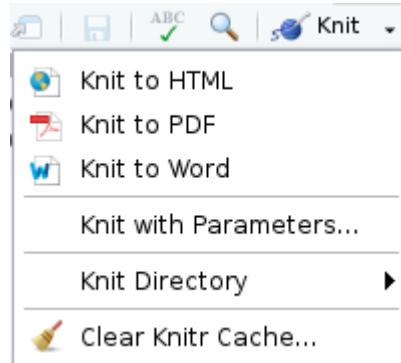


Figure 13.11: Menu *Knit*

13.5 Rendu des tableaux

13.5.1 Tableaux croisés

Par défaut, les tableaux issus de la fonction `table` sont affichés comme ils apparaissent dans la console de R, en texte brut :

```
library(questionr)
data(hdv2003)
tab <- lprop(table(hdv2003$qualif, hdv2003$sexe))
tab
```

| | Homme | Femme | Total |
|--------------------------|-------|-------|-------|
| Ouvrier spécialisé | 47.3 | 52.7 | 100.0 |
| Ouvrier qualifié | 78.4 | 21.6 | 100.0 |
| Technicien | 76.7 | 23.3 | 100.0 |
| Profession intermédiaire | 55.0 | 45.0 | 100.0 |
| Cadre | 55.8 | 44.2 | 100.0 |
| Employé | 16.2 | 83.8 | 100.0 |
| Autre | 36.2 | 63.8 | 100.0 |
| Ensemble | 44.8 | 55.2 | 100.0 |

On peut améliorer leur présentation en utilisant la fonction `kable` de l'extension

knitr. Celle-ci fournit un formatage adapté en fonction du format de sortie. On aura donc des tableaux “propres” que ce soit en HTML, PDF ou aux formats traitements de texte :

```
library(knitr)
kable(tab)
```

| | Homme | Femme | Total |
|--------------------------|----------|----------|-------|
| Ouvrier specialise | 47.29064 | 52.70936 | 100 |
| Ouvrier qualifie | 78.42466 | 21.57534 | 100 |
| Technicien | 76.74419 | 23.25581 | 100 |
| Profession intermediaire | 55.00000 | 45.00000 | 100 |
| Cadre | 55.76923 | 44.23077 | 100 |
| Employe | 16.16162 | 83.83838 | 100 |
| Autre | 36.20690 | 63.79310 | 100 |
| Ensemble | 44.82759 | 55.17241 | 100 |

Différents arguments permettent de modifier la sortie de **kable**. **digits**, par exemple, permet de spécifier le nombre de chiffres significatifs à afficher dans les colonnes de nombres :

```
kable(tab, digits = 1)
```

| | Homme | Femme | Total |
|--------------------------|-------|-------|-------|
| Ouvrier specialise | 47.3 | 52.7 | 100 |
| Ouvrier qualifie | 78.4 | 21.6 | 100 |
| Technicien | 76.7 | 23.3 | 100 |
| Profession intermediaire | 55.0 | 45.0 | 100 |
| Cadre | 55.8 | 44.2 | 100 |
| Employe | 16.2 | 83.8 | 100 |
| Autre | 36.2 | 63.8 | 100 |
| Ensemble | 44.8 | 55.2 | 100 |

13.5.2 Tableaux de données et tris à plat

En ce qui concerne les tableaux de données (tibble ou *data frame*), l'affichage HTML par défaut se contente d'un affichage texte comme dans la console, très peu lisible dès que le tableau dépasse une certaine taille.

Une alternative est d'utiliser la fonction **paged_table**, qui affiche une représentation HTML paginée du tableau :

| | id | ... | sexe | nivetud | poids | <dbl> |
|----|-----------|------------|-------------|--|--------------|-------|
| | | | | <int><int><fctr> <fctr> | | |
| 1 | 1 | 28 | Femme | Enseignement supérieur y compris technique supérieur | 2634.39822 | |
| 2 | 2 | 23 | Femme | NA | 9738.39578 | |
| 3 | 3 | 59 | Homme | Dernière année d'études primaires | 3994.10246 | |
| 4 | 4 | 34 | Homme | Enseignement supérieur y compris technique supérieur | 5731.66151 | |
| 5 | 5 | 71 | Femme | Dernière année d'études primaires | 4329.09400 | |
| 6 | 6 | 35 | Femme | Enseignement technique ou professionnel court | 8674.69938 | |
| 7 | 7 | 60 | Femme | Dernière année d'études primaires | 6165.80349 | |
| 8 | 8 | 47 | Homme | Enseignement technique ou professionnel court | 12891.64076 | |
| 9 | 9 | 20 | Femme | NA | 7808.87206 | |
| 10 | 10 | 28 | Homme | Enseignement technique ou professionnel long | 2277.16047 | |

1-10 of 2,000 rows | 1-6 of 21 columns

Previous [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) ... [200](#) [Next](#)

Figure 13.12: Rendu d'une table par `paged_table`

Une alternative est d'utiliser `kable`, comme précédemment pour les tableaux croisés, ou encore la fonction `datatable` de l'extension `DT`, qui propose encore davantage d'interactivité :

| Show 10 entries | | | | | | | Search: |
|-----------------|-----------|------------|-------------|--|--------------|-----------------------|---------|
| | id | age | sexe | nivetud | poids | occup | |
| 1 | 1 | 28 | Femme | Enseignement supérieur y compris technique supérieur | 2634.3982157 | Exerce une profession | |
| 2 | 2 | 23 | Femme | | 9738.3957759 | Etudiant, élève | |
| 3 | 3 | 59 | Homme | Dernière année d'études primaires | 3994.1024587 | Exerce une profession | |
| 4 | 4 | 34 | Homme | Enseignement supérieur y compris technique supérieur | 5731.6615081 | Exerce une profession | |
| 5 | 5 | 71 | Femme | Dernière année d'études primaires | 4329.0940022 | Retraite | |
| 6 | 6 | 35 | Femme | Enseignement technique ou professionnel court | 8674.6993828 | Exerce une profession | |
| 7 | 7 | 60 | Femme | Dernière année d'études primaires | 6165.8034861 | Au foyer | |
| 8 | 8 | 47 | Homme | Enseignement technique ou professionnel court | 12891.640759 | Exerce une profession | |
| 9 | 9 | 20 | Femme | | 7808.8720636 | Etudiant, élève | |
| 10 | 10 | 28 | Homme | Enseignement technique ou professionnel long | 2277.160471 | Exerce une profession | |

Showing 1 to 10 of 2,000 entries

Previous [1](#) [2](#) [3](#) [4](#) [5](#) ... [200](#) [Next](#)

Figure 13.13: Rendu d'une table par `DT::datatable`

Dans tous les cas il est déconseillé d'afficher de cette manière un tableau de données de très grandes dimensions, car le fichier HTML résultant contiendrait l'ensemble des données et serait donc très volumineux.



On peut définir un mode d'affichage par défaut pour tous les tableaux de données en modifiant les *Output options* du format HTML (onglet *General*, *Print dataframes as*), ou en modifiant manuellement l'option `df_print` de l'entrée `html_document` dans le préambule.

À noter que les tableaux issus de la fonction `freq` de `questionr` s'affichent comme des tableaux de données (et non comme des tableaux croisés).

13.6 Modèles de documents

On a vu ici la production de documents “classiques”, mais R Markdown permet de créer bien d'autres choses.

Le site de documentation de l'extension propose [une galerie](#) des différentes sorties possibles. On peut ainsi créer des slides, des sites Web ou même des livres entiers, comme le présent document.

13.6.1 Slides

Un usage intéressant est la création de diaporamas pour des présentations sous forme de slides. Le principe reste toujours le même : on mélange texte au format Markdown et code R, et R Markdown transforme le tout en présentations au format HTML ou PDF. En général les différents slides sont séparés au niveau de certains niveaux de titre.

Certains modèles de slides sont inclus avec R Markdown, notamment :

- `ioslides` et `Slidy` pour des présentations HTML
- `beamer` pour des présentations en PDF via `LaTeX`

Quand vous créez un nouveau document dans RStudio, ces modèles sont accessibles via l'entrée *Presentation* :

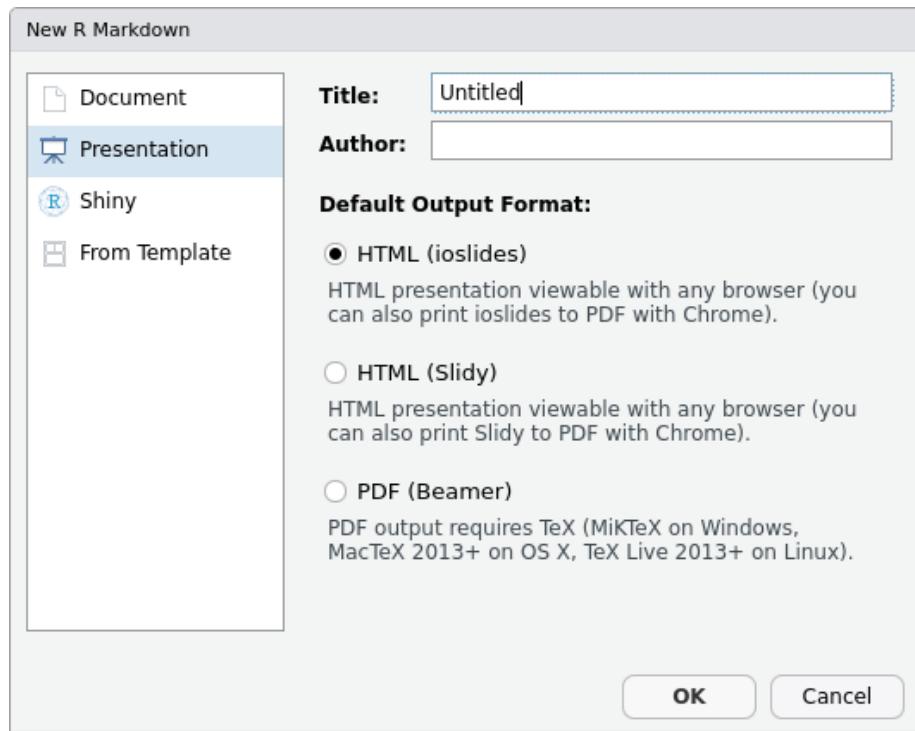


Figure 13.14: Créer une présentation R Markdown

D'autres extensions, qui doivent être installées séparément, permettent aussi des diaporamas dans des formats variés. On citera notamment :

- [xaringan](#) pour des présentations HTML basées sur [remark.js](#)
- [revealjs](#) pour des présentations HTML basées [reveal.js](#)
- [rmdshower](#) pour des diaporamas HTML basés sur [shower](#)

Une fois l'extension installée, elle propose en général un *template* de départ lorsqu'on crée un nouveau document dans RStudio. Ceux-ci sont accessibles depuis l'entrée *From Template*.

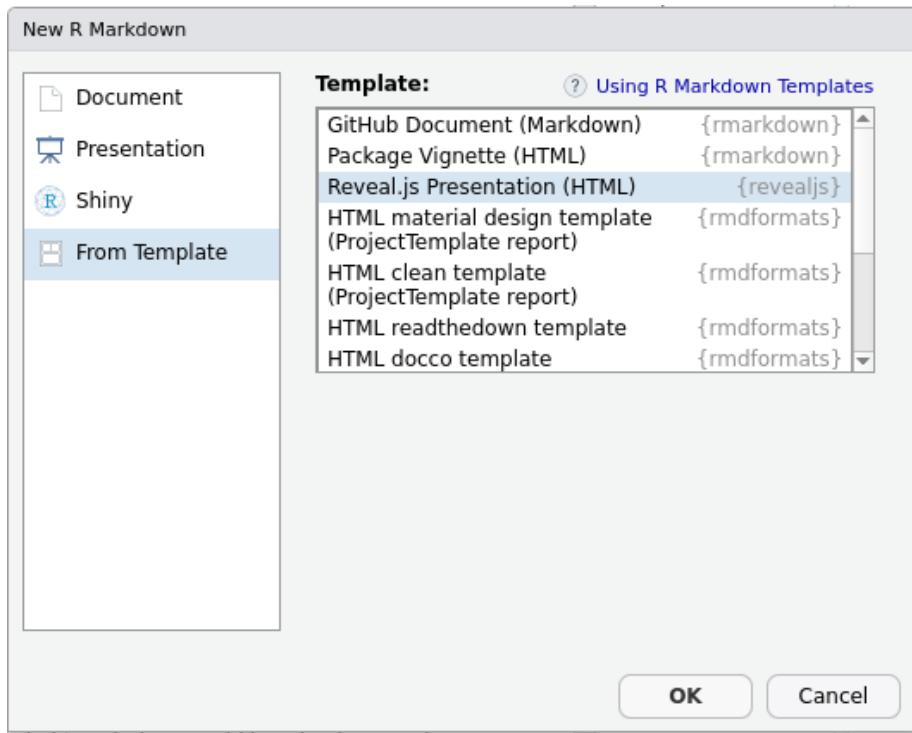


Figure 13.15: Créer une présentation à partir d'un template

13.6.2 Templates

Il existe également différents *templates* permettant de changer le format et la présentation des documents générés. Une liste de ces formats et leur documentation associée est accessible depuis la page [formats](#) de la documentation.

On notera notamment :

- des formats d'article correspondant à des publications dans différentes revues : `jss_article`, `elsevier_article`, etc.
- le format [Tufte Handouts](#) qui permet de produire des documents PDF ou HTML dans un format proche de celui utilisé par Edward Tufte pour certaines de ses publications

Enfin, l'extension `rmdformats` propose plusieurs modèles HTML adaptés notamment pour des documents longs :

The screenshot shows a website with a dark header bar containing the text "readthedown template example". Below the header, there is a sidebar with the following navigation links:

- Code and tables
- Figures
- Mathjax

The main content area has a title "Code and tables" and a section titled "Syntax highlighting". It contains a sample R code chunk:

```
library(ggplot2)
library(dplyr)

not_null <- function (v) {
  if (!is.null(v)) return(paste0v, "not null"))
}

data(iris)
tbl <- iris %>%
  group_by(Species) %>%
  summarise(Sepal.Length = mean(Sepal.Length),
            Sepal.Width = mean(Sepal.Width),
            Petal.Length = mean(Petal.Length),
            Petal.Width = mean(Petal.Width))
```

Below this, there is a section titled "Verbatim" which displays the output of the `str(iris)` command:

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

Figure 13.16: Modèle `readthedown`

html_clean template example

Code and tables

Syntax highlighting

Here is a sample code chunk, just to show that syntax highlighting works as expected.

```
library(ggplot2)
library(dplyr)

not_null <- function(v) [
  if (!is.null(v)) return(costa(v, "not null"))
]

data(iris)
tbl <- iris %>%
  group_by(Species) %>%
  summarise(Sepal.Length = mean(Sepal.Length),
            Sepal.Width = mean(Sepal.Width),
            Petal.Length = mean(Petal.Length),
            Petal.Width = mean(Petal.Length))
```

Verbatim

Here is the structure of the `iris` dataset.

```
str(iris)
```

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

Table

Here is a sample table output.

| Species | Sepal.Length | Sepal.Width | Petal.Length | Petal.Width |
|------------|--------------|-------------|--------------|-------------|
| setosa | 5.005 | 3.428 | 1.462 | 1.462 |
| versicolor | 5.938 | 2.770 | 4.260 | 4.260 |

Figure 13.17: Modèle `html_clean`

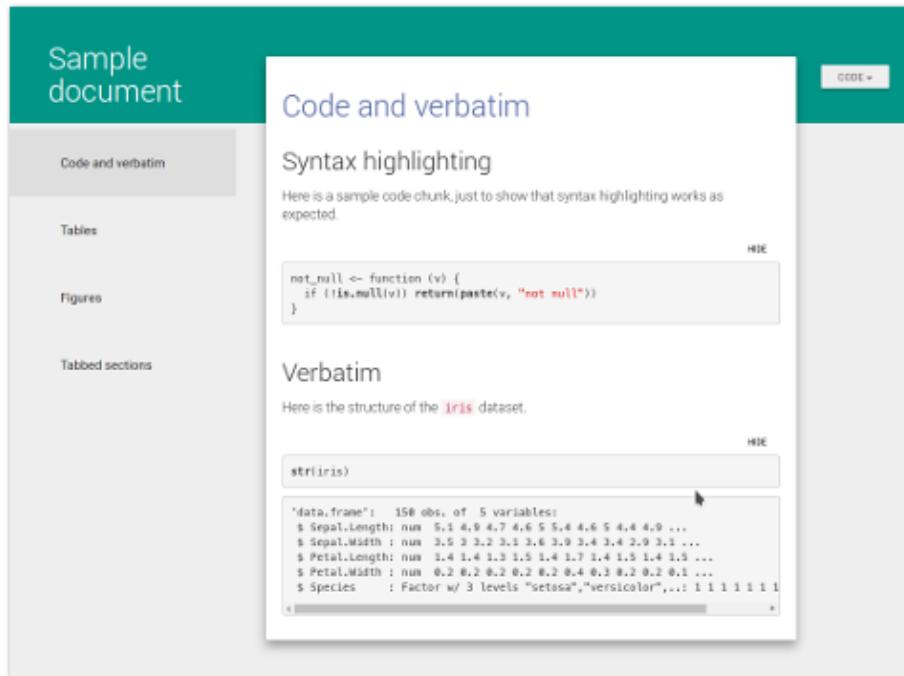


Figure 13.18: Modèle `material`

Là encore, la plupart du temps, ces modèles de documents proposent un *template* de départ lorsqu'on crée un nouveau document dans RStudio (entrée *From Template*) :

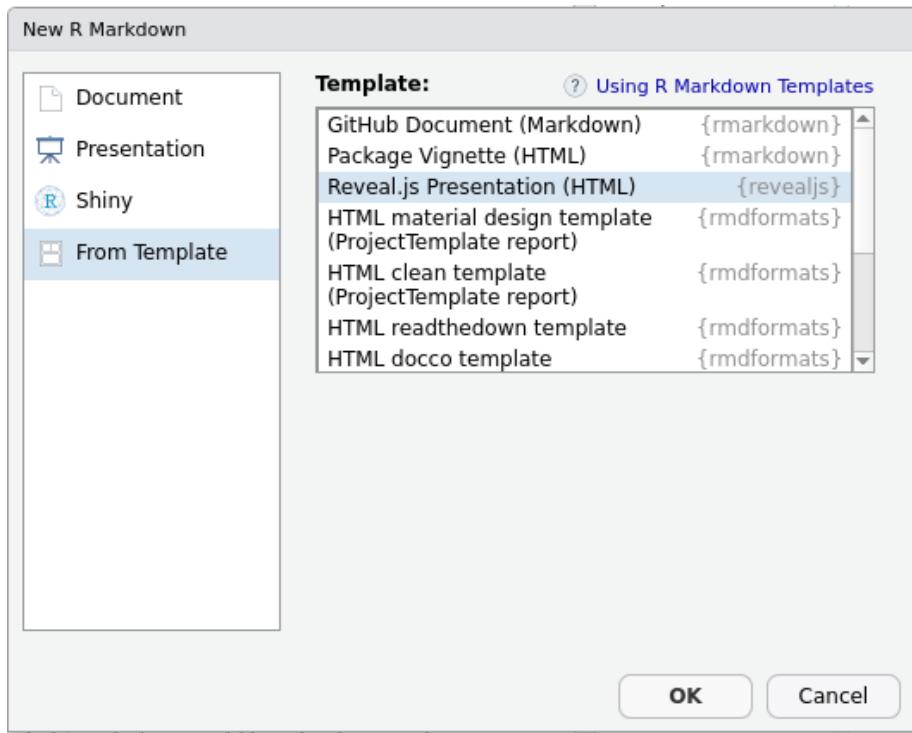


Figure 13.19: Créer un document à partir d'un template

13.7 Ressources

Les ressources suivantes sont toutes en anglais...

L'ouvrage *R for data science*, accessible en ligne, contient [un chapitre dédié à R Markdown](#).

Le [site officiel de l'extension](#) contient une documentation très complète, tant pour les débutants que pour un usage avancé.

Enfin, l'aide de RStudio (menu *Help* puis *Cheatsheets*) permet d'accéder à deux documents de synthèse : une “antisèche” synthétique (*R Markdown Cheat Sheet*) et un “guide de référence” plus complet (*R Markdown Reference Guide*).

Appendix A

Ressources

A.1 Aide

A.1.1 Aide de R et RStudio

Il est possible d'obtenir à tout moment de l'aide (en anglais) sur une fonction en tapant `help()` avec comme argument le nom de la fonction dans la console :

```
help("mean")
```

Vous pouvez aussi aller dans l'onglet *Help* de l'interface de RStudio (dans le quart de l'écran en bas à droite) et utiliser le moteur de recherche intégré.

Chaque page d'aide est très complète mais pas toujours très accessible. Elle est structurée selon différentes sections, notamment :

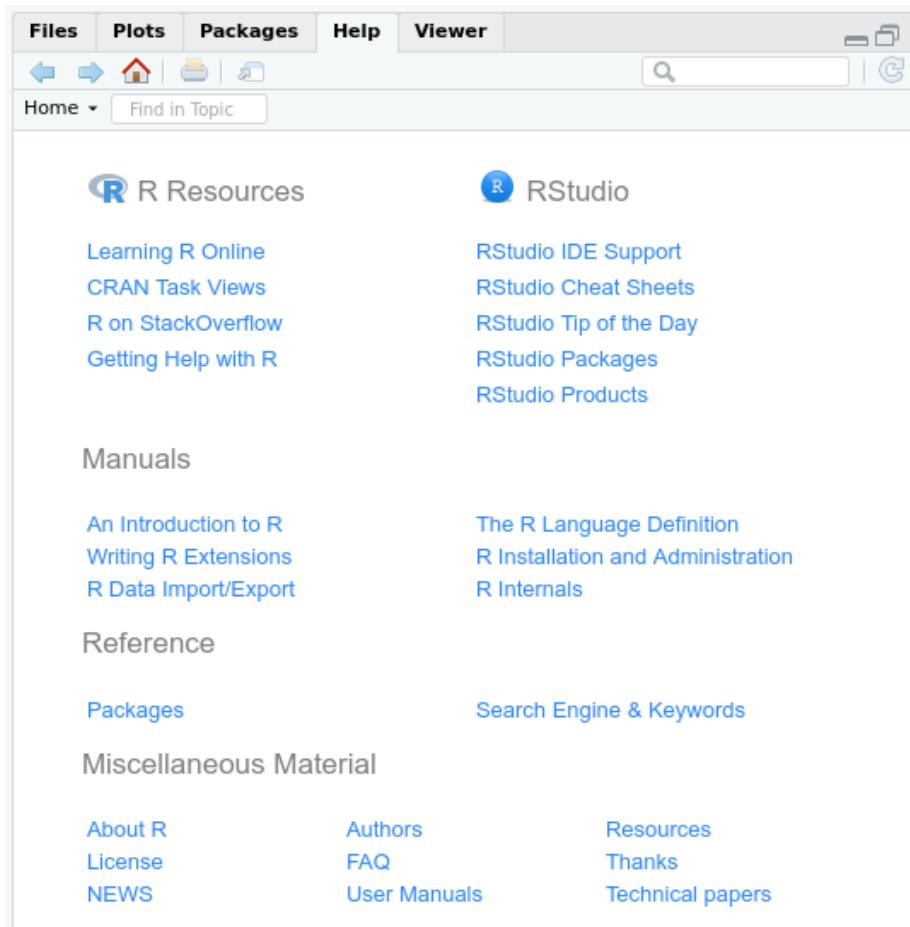
- *Description* : donne un résumé en une phrase de ce que fait la fonction
- *Usage* : indique la ou les manières de l'utiliser
- *Arguments* : détaille les arguments possibles et leur signification
- *Value* : indique la forme du résultat renvoyé par la fonction
- *Details* : apporte des précisions sur le fonctionnement de la fonction
- *See Also* : renvoie vers d'autres fonctions semblables ou liées, ce qui peut être très utile pour découvrir ou retrouver une fonction dont on a oublié le nom
- *Examples* : donne une série d'exemples d'utilisation

Les exemples d'une page d'aide peuvent être exécutés directement dans la console avec la fonction `example` :

```
example("mean")
```

```
mean> x <- c(0:10, 50)
mean> xm <- mean(x)
mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

L'onglet *Help* de RStudio permet d'afficher mais aussi de naviguer dans les pages d'aide de R et dans d'autres ressources :

Figure A.1: Onglet *Help* de Rstudio

Cette page permet d'accéder aux manuels officiels de R (section *Manuals*), qui abordent différents aspects plus ou moins techniques du langage et du logiciel, en anglais. On citera notamment les documents *An Introduction to R* et *R Data Import/Export*. Elle propose également un lien vers la FAQ officielle.

A.1.2 Aide en ligne

Plusieurs sites proposent une interface permettant de naviguer et rechercher dans l'aide de R et de l'ensemble des extensions existantes.

On notera notamment :

- [RDocumentation](#)
- [rrdr.io](#)

A.1.3 Antisèches

RStudio propose plusieurs *cheat sheets* (antisèches) en anglais qui proposent sur deux pages une synthèse compacte de fonctions et de leur usage selon différentes thématiques, notamment :

- [Manipulation des données avec dplyr](#)
- [Visualisation avec ggplot2](#)
- [Export avec RMarkdown](#)
- etc.

La liste complète est disponible en ligne :

<https://www.rstudio.com/resources/cheatsheets/>

Ou directement depuis RStudio, via le menu *Help*, puis *Cheatsheets*.

A.1.4 Où poser des questions

Outre l'aide intégrée au logiciel, il existe de nombreuses ressources en ligne, forums, listes de discussions, pour poser ses questions et échanger avec des utilisateurs de R.

A.1.4.1 Discussion instantanée

Grrr (“pour quand votre R fait Grrr”) est un groupe Slack (plateforme de discussion instantanée) francophone dédié aux échanges et à l’entraide autour de R. Il est ouvert à tous et se veut accessible aux débutants. Vous pouvez même utiliser un pseudonyme si vous préférez.

Pour rejoindre la discussion, il suffit de suivre le lien d’invitation suivant :
<https://frama.link/r-grrr>

A.1.4.2 Listes de discussion

La liste **R-soc** est une liste francophone spécialement dédiée aux utilisateurs de R en sciences sociales. Toutes les questions y sont les bienvenues, et les réponses sont en général assez rapides. Il suffit de s’y abonner pour pouvoir ensuite poster sa question :

- <https://groupes.renater.fr/sympa/subscribe/r-soc>

La liste **semin-r** est la liste de discussion du groupe des utilisateurs de R animé par le Muséum national d'Histoire naturelle. Elle est ouverte à tous et les questions y sont bienvenues :

- <https://listes.mnhn.fr/wws/subscribe/semin-r>

Il existe aussi une liste officielle anglophone baptisée **R-help**. Elle est cependant à réserver aux questions les plus pointues, et dans tous les cas il est nécessaire d'avoir en tête et de respecter les **bonnes pratiques** avant de poster sur la liste :

- <https://stat.ethz.ch/mailman/listinfo/r-help>

A.1.4.3 Sur le Web

Pour les anglophones, la ressource la plus riche concernant R est certainement le site **StackOverflow**. Sous forme de questions/réponses, il comporte un très grand nombre d'informations sur R et les réponses y sont très rapides. Avant de poster une question il est fortement recommandé de faire une recherche sur le site, car il y a de fortes chances que celle-ci ait déjà été posée :

- <https://stackoverflow.com/questions/tagged/r>

Pour les francophones, on pourra citer le forum du CIRAD, qui comporte une section *questions en cours* assez active. Là aussi, pensez à faire une recherche sur le forum avant de poser votre question :

- <http://forums.cirad.fr/logiciel-R/>

A.2 Ouvrages, blogs, MOOCs...

A.2.1 Francophones

Parmi les ressources en français, on peut citer notamment **R et espace**, manuel d'initiation à la programmation avec R appliqué à l'analyse de l'information géographique, librement téléchargeable en ligne.

La section **Contributed documentation** du site officiel de R contient également des liens vers différents documents en français, plus ou moins accessibles et plus ou moins récemment mis à jour.

Le pôle bioinformatique lyonnais (PBIL) propose depuis longtemps une somme très importante de documents, qui comprend des cours complets de statistiques utilisant R :

- <http://pbil.univ-lyon1.fr/R/>

Plusieurs blogs francophones autour de R sont également actifs, parmi lesquels :

- [ElementR](#), le blog du groupe du même nom, qui propose de nombreuses ressources sur R en général et en particulier sur la cartographie ou l'analyse de réseaux.
- [R-atique](#), blog animé par Lise Vaudor, propose régulièrement des articles intéressants et accessibles sur des méthodes d'analyse ou sur des extensions R.

Enfin, le site *France Université Numérique* propose régulièrement des sessions de cours en ligne, parmi lesquels une [Introduction à la statistique avec R](#) et un cours sur l'[Analyse des données multidimensionnelles](#).

A.2.2 Anglophones

Les ressources anglophones sont évidemment très nombreuses.

On citera essentiellement l'ouvrage en ligne [R for data science](#), très complet, et qui fournit une introduction très complète et progressive à R, et aux packages du *tidyverse*. Il existe également en version papier.

Pour aborder des aspects beaucoup plus avancés, l'ouvrage également en ligne [Advanced R](#), d'Hadley Wickham, est extrêmement bien fait et très complet.

On notera également l'existence du [R journal](#), revue en ligne consacrée à R, et qui propose régulièrement des articles sur des méthodes d'analyse, des extensions, et l'actualité du langage.

La plateforme [R-bloggers](#) agrège les contenus de plusieurs centaines de blogs parlant de R, très pratique pour suivre l'actualité de la communauté.

Enfin, sur Twitter, les échanges autour de R sont regroupés autour du *hashtag* [#rstats](#).

A.3 Extensions

A.3.1 Où trouver des extensions intéressantes ?

Il existe plusieurs milliers d'extensions pour R, et il n'est pas toujours facile de savoir laquelle choisir pour une tâche donnée.

Si un des meilleurs moyens reste le bouche à oreille, on peut aussi se reporter à la page [CRAN Task view](#) qui liste un certain nombre de domaines (classification, sciences sociales, séries temporelles...) et indique, pour chacun d'entre eux, une liste d'extensions potentiellement intéressantes accompagnées d'une courte description. On peut même installer l'ensemble des extensions d'une catégorie avec la fonction `install.views()`.

Une autre possibilité est de consulter la page [listant l'ensemble des packages existant](#). S'il n'est évidemment pas possible de passer en revue les milliers d'extensions une à une, on peut toujours effectuer une recherche dans la page avec des mots-clés correspondant aux fonctionnalités recherchées.

Un autre site intéressant est [Awesome R](#), une liste élaborée collaborativement des extensions les plus utiles ou les plus populaires classées par grandes catégories : manipulation des données, graphiques interactifs, etc.

La page [frrenchies](#) liste des packages pouvant être utiles pour des utilisateurs français (géolocalisation, traitement du langage, accès à des API...), ainsi que des ressources francophones.

Enfin, certaines extensions fournissent des “galeries” permettant de repérer ou découvrir certains *packages*. C'est notamment le cas de [htmlwidgets](#), qui propose une [galerie d'extensions proposant des graphiques interactifs](#), ou de [R Markdown](#).

A.3.2 L'extension `questionr`

`questionr` est une extension utilisée régulièrement dans ce document et comprenant quelques fonctions utiles pour l'utilisation du logiciel en sciences sociales, ainsi que différents jeux de données. Elle est développée en collaboration avec François Briatte et Joseph Larmarange.

L'installation se fait soit via le bouton *Install* de l'onglet *Packages* de RStudio, soit en utilisant la commande suivante dans la console :

```
install.packages("questionr")
```

Il est possible d'installer la version de développement à l'aide de la fonction `install_github` de l'extension `devtools` :

```
devtools:::install_github("juba/questionr")
```

`questionr` propose à la fois des fonctions, des interfaces interactives et des jeux de données d'exemple.

A.3.2.1 Fonctions et utilisation

Pour plus de détails sur la liste des fonctions de l'extension et son utilisation, on pourra se reporter au [site Web de l'extension](#), hébergé sur GitHub.

L'onglet [Reference](#) liste l'ensemble des fonctions de `questionr`, tandis que l'onglet [Articles](#) propose une [présentation des trois interfaces interactives \(Addins\)](#) visant à faciliter le recodage de certaines variables.

Ces interfaces sont également abordées dans la partie [9](#).

A.3.2.2 Jeu de données `hdv2003`

`hdv2003` est un extrait comportant 2000 individus et 20 variables provenant de l'enquête *Histoire de Vie* réalisée par l'INSEE en 2003.

L'extrait est tiré du fichier détail [mis à disposition librement](#) (ainsi que de nombreux autres) par l'INSEE. On trouvera une [documentation complète](#) à la même adresse.

Les variables retenues ont été parfois partiellement recodées. La liste des variables est la suivante :

| Variable | Description |
|----------------------------|--|
| <code>id</code> | Identifiant (numéro de ligne) |
| <code>poids</code> | Variable de pondération |
| <code>age</code> | Âge |
| <code>sexe</code> | Sexe |
| <code>nivetud</code> | Niveau d'études atteint |
| <code>occup</code> | Occupation actuelle |
| <code>qualif</code> | Qualification de l'emploi actuel |
| <code>freres.soeurs</code> | Nombre total de frères, sœurs, demi-frères et demi-sœurs |
| <code>clso</code> | Sentiment d'appartenance à une classe sociale |
| <code>relig</code> | Pratique et croyance religieuse |
| <code>trav.imp</code> | Importance accordée au travail |
| <code>trav.satisf</code> | Satisfaction ou insatisfaction au travail |
| <code>hard.rock</code> | Écoute du Hard rock ou assimilés |
| <code>lecture.bd</code> | Lecture de bandes dessinées |
| <code>peche.chasse</code> | Pêche ou chasse pour le plaisir au cours des 12 derniers mois |
| <code>cuisine</code> | Cuisine pour le plaisir au cours des 12 derniers mois |
| <code>bricol</code> | Bricolage ou mécanique pour le plaisir au cours des 12 derniers mois |
| <code>cinema</code> | Cinéma au cours des 12 derniers mois |
| <code>sport</code> | Sport ou activité physique pour le plaisir au cours des 12 derniers mois |
| <code>heures.tv</code> | Nombre moyen d'heures passées à regarder la télévision par jour |



Comme il s'agit d'un extrait du fichier, la variable de pondération n'a en toute rigueur aucune valeur statistique. Elle a été tout de même incluse à des fins "pédagogiques".

A.3.2.3 Jeu de données rp2012

rp2012 est un jeu de données issu du recensement de la population de 2012 de l'INSEE. Il comporte une petite partie des résultats pour l'ensemble des communes de plus de 2000 habitants de France métropolitaine, soit au final 5170 communes et 60 variables.

Liste de quelques variables du fichier :

| Variable | Description |
|--------------------------|---|
| <code>commune</code> | nom de la commune |
| <code>code_insee</code> | Code de la commune |
| <code>pop_tot</code> | Population total |
| <code>pop_act_15p</code> | Population active de 15 ans et plus |
| <code>log_rp</code> | Nombre de résidences principales |
| <code>agric</code> | Part des agriculteurs dans la population active |
| <code>indep</code> | Part des artisans, commerçants et chefs d'entreprises |
| <code>cadres</code> | Part des cadres |
| <code>intern</code> | Part des professions intermédiaires |
| <code>empl</code> | Part des employés |
| <code>ouvr</code> | Part des ouvriers |
| <code>chom</code> | Part des chômeurs |
| <code>etud</code> | Part des étudiants |
| <code>dipl_sup</code> | Part des diplômés du supérieur |
| <code>dipl_aucun</code> | Part des personnes sans diplôme |
| <code>proprio</code> | Part des propriétaires parmi les résidences principales |
| <code>hlm</code> | Part des logements HLM parmi les résidences principales |
| <code>locataire</code> | Part des locataires parmi les résidences principales |
| <code>maison</code> | Part des maisons parmi les résidences principales |

