

guide-R

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

Joseph Larmarange

23 avril 2024

Table des matières

Préface	3
Remerciements	4
Licence	4
I Bases du langage	5
1 Packages	6
1.1 Installation (CRAN)	6
1.2 Chargement	7
1.3 Mise à jour	8
1.4 Installation depuis GitHub	8
1.5 Le tidyverse	9
2 Vecteurs	11
2.1 Types et classes	11
2.2 Création d'un vecteur	12
2.3 Longueur d'un vecteur	15
2.4 Combiner des vecteurs	15
2.5 Vecteurs nommés	16
2.6 Indexation par position	17
2.7 Indexation par nom	18
2.8 Indexation par condition	19
2.9 Assignation par indexation	22
2.10 En résumé	24
2.11 webin-R	24
3 Listes	25
3.1 Propriétés et création	25
3.2 Indexation	28
3.3 En résumé	31
3.4 webin-R	32
4 Tableaux de données	33
4.1 Propriétés et création	33
4.2 Indexation	35

4.3	Afficher les données	38
4.4	En résumé	44
4.5	webin-R	45
5	Tibbles	46
5.1	Le concept de tidy data	46
5.2	tibbles : des tableaux de données améliorés	46
5.3	Données et tableaux imbriqués	50
6	Attributs	53
II	Manipulation de données	56
7	Le pipe	57
7.1	Le pipe natif de R : <code> ></code>	58
7.2	Le pipe du tidyverse : <code>%>%</code>	59
7.3	Vaut-il mieux utiliser <code> ></code> ou <code>%>%</code> ?	60
7.4	Accéder à un élément avec <code>purrr::pluck()</code> et <code>purrr::chuck()</code>	60
8	dplyr	63
8.1	Opérations sur les lignes	64
8.1.1	<code>filter()</code>	64
8.1.2	<code>slice()</code>	68
8.1.3	<code>arrange()</code>	69
8.1.4	<code>slice_sample()</code>	71
8.1.5	<code>distinct()</code>	72
8.2	Opérations sur les colonnes	74
8.2.1	<code>select()</code>	74
8.2.2	<code>relocate()</code>	79
8.2.3	<code>rename()</code>	79
8.2.4	<code>rename_with()</code>	80
8.2.5	<code>pull()</code>	81
8.2.6	<code>mutate()</code>	82
8.3	Opérations groupées	82
8.3.1	<code>group_by()</code>	82
8.3.2	<code>summarise()</code>	87
8.3.3	<code>count()</code>	88
8.3.4	Grouper selon plusieurs variables	89
8.4	Cheatsheet	94
8.5	webin-R	94
9	Facteurs et forcats	95
9.1	Création d'un facteur	95

9.2	Changer l'ordre des modalités	98
9.3	Modifier les modalités	102
9.4	Découper une variable numérique en classes	109
10	Combiner plusieurs variables	113
10.1	if_else()	113
10.2	case_when()	115
10.3	recode_if()	117
11	Étiquettes de variables	122
11.1	Principe	122
11.2	Manipulation sur un vecteur / une colonne	124
11.3	Manipulation sur un tableau de données	125
11.4	Préserver les étiquettes	126
12	Étiquettes de valeurs	128
12.1	La classe <code>haven_labelled</code>	128
12.2	Manipulation sur un vecteur / une colonne	129
12.3	Manipulation sur un tableau de données	132
12.4	Conversion	133
12.4.1	Quand convertir les vecteurs labellisés ?	133
12.4.2	Convertir un vecteur labellisé en facteur	135
12.4.3	Convertir un vecteur labellisé en numérique ou en texte	136
12.4.4	Conversion conditionnelle en facteurs	138
13	Valeurs manquantes	142
13.1	Valeurs manquantes étiquetées (<i>tagged NAs</i>)	143
13.1.1	Création et test	143
13.1.2	Valeurs uniques, doublons et tris	145
13.1.3	Tagged NAs et étiquettes de valeurs	146
13.1.4	Conversion en user NAs	148
13.2	Valeurs manquantes définies par l'utilisateur (<i>user NAs</i>)	149
13.2.1	Création	149
13.2.2	Tests	151
13.2.3	Conversion	152
14	Import & Export de données	155
14.1	Importer un fichier texte	155
14.1.1	Structure d'un fichier texte	155
14.1.2	Interface graphique avec RStudio	156
14.1.3	Dans un script	157
14.2	Importer un fichier Excel	157

14.3	Importer depuis des logiciels de statistique	158
14.3.1	SPSS	159
14.3.2	SAS	159
14.3.3	Stata	160
14.3.4	dBase	160
14.4	Sauver ses données	160
14.5	Export de tableaux de données	162
15	Mettre en forme des nombres	163
15.1	label_number()	164
15.2	Les autres fonctions de {scales}	166
15.2.1	label_comma()	166
15.2.2	label_percent()	166
15.2.3	label_dollar()	167
15.2.4	label_pvalue()	167
15.2.5	label_scientific()	168
15.2.6	label_bytes()	168
15.2.7	label_ordinal()	168
15.2.8	label_date(), label_date_short() & label_time()	169
15.2.9	label_wrap()	169
15.3	Les fonctions de formatage de {gtsummary}	170
15.3.1	style_number()	170
15.3.2	style_sigfig()	171
15.3.3	style_percent()	172
15.3.4	style_pvalue()	172
15.3.5	style_ratio()	173
15.4	Bonus : signif_stars() de {ggstats}	173
16	Couleurs & Palettes	174
16.1	Noms de couleur	174
16.2	Couleurs RVB et code hexadécimal	175
16.3	Palettes de couleurs	176
16.3.1	Color Brewer	176
16.3.2	Palettes de Paul Tol	178
16.3.3	Interface unifiée avec {paletteer}	181
III	Analyses	184
17	Graphiques avec ggplot2	185
17.1	Ressources	185
17.2	Les bases de ggplot2	185
17.3	Cheatsheet	190

17.4	Exploration visuelle avec <code>esquisse</code>	191
17.5	webin-R	194
17.6	Combiner plusieurs graphiques	194
18	Statistique univariée & Intervalles de confiance	201
18.1	Exploration graphique	201
18.1.1	Variable continue	201
18.1.2	Variable catégorielle	205
18.2	Tableaux et tris à plat	208
18.2.1	Thème du tableau	210
18.2.2	Étiquettes des variables	211
18.2.3	Statistiques affichées	214
18.2.4	Affichage du nom des statistiques	218
18.2.5	Forcer le type de variable	220
18.2.6	Afficher des statistiques sur plusieurs lignes (variables continues)	222
18.2.7	Mise en forme des statistiques	223
18.2.8	Données manquantes	227
18.2.9	Ajouter les effectifs observés	229
18.3	Calcul manuel	229
18.3.1	Variable continue	229
18.3.2	Variable catégorielle	231
18.4	Intervalles de confiance	232
18.4.1	Avec <code>gtsummary</code>	232
18.4.2	Calcul manuel	234
18.5	webin-R	237
19	Statistique bivariable & Tests de comparaison	238
19.1	Deux variables catégorielles	238
19.1.1	Tableau croisé avec <code>gtsummary</code>	238
19.1.2	Représentations graphiques	241
19.1.3	Calcul manuel	247
19.1.4	Test du χ^2 et dérivés	249
19.1.5	Comparaison de deux proportions	252
19.2	Une variable continue selon une variable catégorielle	255
19.2.1	Tableau comparatif avec <code>gtsummary</code>	255
19.2.2	Représentations graphiques	257
19.2.3	Calcul manuel	264
19.2.4	Tests de comparaison	265
19.2.5	Différence de deux moyennes	268
19.3	Deux variables continues	268
19.3.1	Représentations graphiques	268
19.3.2	Tester la relation entre les deux variables	275
19.4	Matrice de corrélations	276

19.5	webin-R	278
20	Échelles de Likert	279
20.1	Exemple de données	279
20.2	Tableau de fréquence	280
20.3	Représentations graphiques	284
21	Régression linéaire	289
21.1	Modèle à une seule variable explicative continue	289
21.2	Modèle à une seule variable explicative catégorielle	294
21.3	Modèle à plusieurs variables explicatives	296
22	Régression logistique binaire	299
22.1	Préparation des données	299
22.2	Statistiques descriptives	305
22.3	Calcul de la régression logistique binaire	306
22.4	Interpréter les coefficients	307
22.5	La notion d' <i>odds ratio</i>	310
22.6	Afficher les écarts-types plutôt que les intervalles de confiance	314
22.7	Afficher toutes les comparaisons (<i>pairwise contrasts</i>)	318
22.8	Identifier les variables ayant un effet significatif	321
22.9	Régressions logistiques univariées	323
22.10	Présenter l'ensemble des résultats dans un même tableau	325
22.11	webin-R	327
23	Sélection pas à pas d'un modèle	328
23.1	Données d'illustration	328
23.2	Présentation de l'AIC	330
23.3	Sélection pas à pas descendante	330
23.4	Sélection pas à pas ascendante	335
23.5	Forcer certaines variables dans le modèle réduit	338
23.6	Minimisation du BIC	339
23.7	Afficher les indicateurs de performance	340
23.8	Sélection pas à pas et valeurs manquantes	342
24	Prédictions marginales, contrastes marginaux & effets marginaux	347
24.1	Terminologie	347
24.2	Données d'illustration	348
24.3	Prédictions marginales	351
24.3.1	Prédictions marginales moyennes	351
24.3.2	Prédictions marginales à la moyenne	358
24.3.3	Variantes	365

24.4	Contrastes marginaux	367
24.4.1	Contrastes marginaux moyens	367
24.4.2	Contrastes marginaux à la moyenne	376
24.5	Pentes marginales / Effets marginaux	378
24.5.1	Pentes marginales moyennes / Effets marginaux moyens	379
24.5.2	Pentes marginales à la moyenne / Effets marginaux à la moyenne	384
24.6	Lectures complémentaires (en anglais)	384
24.7	webin-R	385
25	Contrastes (variables catégorielles)	386
25.1	Contrastes de type traitement	386
25.1.1	Exemple 1 : un modèle linéaire avec une variable catégorielle	386
25.1.2	Exemple 2 : une régression logistique avec deux variables catégorielles	389
25.1.3	Changer la modalité de référence	393
25.2	Contrastes de type somme	396
25.2.1	Exemple 1 : un modèle linéaire avec une variable catégorielle	396
25.2.2	Exemple 2 : une régression logistique avec deux variables catégorielles	399
25.3	Contrastes par différences successives	402
25.3.1	Exemple 1 : un modèle linéaire avec une variable catégorielle	402
25.3.2	Exemple 2 : une régression logistique avec deux variables catégorielles	404
25.4	Autres types de contrastes	406
25.4.1	Contrastes de type Helmert	406
25.4.2	Contrastes polynomiaux	408
25.5	Lectures additionnelles	409
26	Interactions	410
26.1	Données d'illustration	410
26.2	Modèle sans interaction	411
26.3	Définition d'une interaction	413
26.4	Significativité de l'interaction	415
26.5	Interprétation des coefficients	417
26.6	Définition alternative de l'interaction	422
26.7	Identifier les interactions pertinentes	426
26.8	Pour aller plus loin	428
26.9	webin-R	429
27	Multicolinéarité	430
27.1	Définition	430
27.2	Mesure de la colinéarité	431
27.3	La multicolinéarité est-elle toujours un problème ?	436
27.4	webin-R	438

IV	Données pondérées avec survey	439
28	Définir un plan d'échantillonnage	440
28.1	Différents types d'échantillonnage	440
28.2	Avec <code>survey::svydesign()</code>	441
28.3	Avec <code>srvyr::as_survey_design()</code>	444
28.4	webin-R	448
29	Manipulation de données pondérées	449
29.1	Utilisation de <code>{srvyr}</code>	449
29.2	Lister / Rechercher des variables	451
29.3	Extraire un sous-échantillon	452
30	Analyses uni- et bivariées pondérées	454
30.1	La fonction <code>tbl_svysummary()</code>	454
30.2	Calcul manuel avec <code>{survey}</code>	458
30.3	Intervalles de confiance et tests statistiques	460
30.4	Calcul manuel avec <code>{srvyr}</code>	462
30.5	Impact du plan d'échantillonnage	464
31	Graphiques pondérés	469
32	Régression logistique binaire pondérée	473
32.1	Données des exemples	473
32.2	Calcul de la régression logistique binaire	474
32.3	Sélection de modèle	475
32.4	Affichage des résultats	477
32.5	Prédictions marginales	479
V	Manipulation avancée	481
33	Fusion de tables	482
33.1	Jointures avec <code>dplyr</code>	482
33.1.1	Clés implicites	482
33.1.2	Clés explicites	484
33.1.3	Types de jointures	487
33.2	Jointures avec <code>merge()</code>	492
33.3	Ajouter des observations avec <code>bind_rows()</code>	494
34	Dates avec <code>lubridate</code>	497
34.1	Création de dates / de dates-heures	497
34.1.1	lors de l'import d'un fichier CSV	498
34.1.2	à partir d'une chaîne de caractères	501

34.1.3	à partir des composants	502
34.1.4	conversion	503
34.2	Manipuler les composants d'une date/date-heure	504
34.2.1	Extraire un composant	504
34.2.2	Arrondis	506
34.2.3	Modifier un composant	506
34.3	Durées, périodes, intervalles & Arithmétique	507
34.3.1	Durées (Duration)	508
34.3.2	Périodes (Period)	510
34.3.3	Intervalles (Interval)	512
34.4	Calcul d'un âge	515
34.5	Fuseaux horaires	516
34.6	Pour aller plus loin	518
35	Chaînes de texte avec stringr	519
35.1	Concaténer des chaînes	520
35.2	Convertir en majuscules / minuscules	521
35.3	Découper des chaînes	522
35.4	Extraire des sous-chaînes par position	523
35.5	Détecter des motifs	524
35.6	Expressions régulières	525
35.7	Extraire des motifs	525
35.8	Remplacer des motifs	527
35.9	Modificateurs de motifs	528
35.10	Insérer une variable dans une chaîne de caractères	529
35.11	Ressources	529
36	Réorganisation avec tidyr	530
36.1	Tidy data	530
36.2	Trois règles pour des données bien rangées	532
36.3	<code>pivot_longer()</code> : rassembler des colonnes	534
36.4	<code>pivot_wider()</code> : disperser des lignes	535
36.5	<code>separate()</code> : séparer une colonne en plusieurs colonnes	538
36.6	<code>separate_rows()</code> : séparer une colonne en plusieurs lignes	539
36.7	<code>unite()</code> : regrouper plusieurs colonnes en une seule	541
36.8	<code>extract()</code> : créer de nouvelles colonnes à partir d'une colonne de texte	542
36.9	<code>complete()</code> : compléter des combinaisons de variables manquantes	543
36.10	Ressources	547
36.11	Fichiers volumineux	547
36.12	webin-R	548
37	Conditions logiques	549
37.1	Opérateurs de comparaison	549

37.2	Comparaison et valeurs manquantes	552
37.3	Opérateurs logiques (algèbre booléenne)	554
37.3.1	Opérations logiques et Valeurs manquantes	555
37.3.2	L'opérateur %in%	555
37.4	Aggrégation	556
37.5	Programmation	557
38	Transformations multiples	558
38.1	Transformations multiples sur les colonnes	558
38.1.1	Usage de base	558
38.1.2	Fonctions multiples	563
38.1.3	Accéder à la colonne courante	563
38.1.4	pick()	565
38.2	Sélection de lignes à partir d'une sélection de colonnes	566
38.3	Transformations multiples sur les lignes	567
38.3.1	Création	567
38.3.2	Statistiques ligne par ligne	569
VI	Analyses avancées	574
39	Analyse factorielle	575
39.1	Principe général	576
39.2	Première illustration : ACM sur les loisirs	577
39.2.1	Calcul de l'ACM	578
39.2.2	Exploration graphique interactive	578
39.2.3	Représentations graphiques	580
39.2.4	Variance expliquée et valeurs propres	580
39.2.5	Contribution aux axes	582
39.2.6	Représentation des modalités dans le plan factoriel	584
39.2.7	Représentation des individus dans le plan factoriel	586
39.2.8	Récupérer les coordonnées des individus / des variables	590
39.3	Ajout de variables / d'observations additionnelles	591
39.4	Gestion des valeurs manquantes	593
39.5	webin-R	597
39.6	Lectures additionnelles	597
40	Classification ascendante hiérarchique	598
40.1	Calculer une matrice des distances	598
40.1.1	Distance de Gower	599
40.1.2	Distance du Φ^2	600
40.1.3	Illustration	601

40.2	Calcul du dendrogramme	602
40.2.1	Représentation graphique du dendrogramme	604
40.3	Découper le dendrogramme	605
40.3.1	Classes obtenues avec la distance de Gower	605
40.3.2	Classes obtenues à partir de l'ACM (distance du Φ^2)	611
40.4	Calcul de l'ACM et de la CAH avec FactoMineR	616
40.5	Caractériser la typologie	621
40.6	webin-R	628
41	Régression logistique multinomiale	629
41.1	Données d'illustration	629
41.2	Calcul du modèle multinomial	631
41.3	Affichage des résultats du modèle	632
41.4	Données pondérées	640
41.4.1	avec <code>svrepmisc::svymultinom()</code>	640
41.4.2	avec <code>svyVGAM::svy_glm()</code>	643
41.5	webin-R	646
42	Régression logistique ordinale	647
42.1	Données d'illustration	647
42.2	Calcul du modèle ordinal	649
42.2.1	avec <code>MASS::polr()</code>	649
42.2.2	Fonctions alternatives	650
42.3	Affichage des résultats du modèle	651
42.4	Données pondérées	656
42.4.1	avec <code>survey::svyolr()</code>	656
42.4.2	avec <code>svrepmisc::svymultinom()</code>	656
42.4.3	avec <code>svyVGAM::svy_glm()</code>	659
43	Modèles de comptage (Poisson & apparentés)	661
43.1	Modèle de Poisson	661
43.1.1	Préparation des données du premier exemple	661
43.1.2	Calcul & Interprétation du modèle de Poisson	664
43.1.3	Évaluation de la surdispersion	667
43.2	Modèle de quasi-Poisson	671
43.3	Modèle binomial négatif	674
43.4	Exemple avec une plus grande surdispersion	679
43.5	Modèles de comptage avec une variable binaire	683
43.6	Données pondérées et plan d'échantillonnage	690
43.7	Lectures complémentaires	692
44	Modèles d'incidence / de taux	693
44.1	Premier exemple (données individuelles, évènement unique)	693

44.2 Deuxième exemple (données agrégées)	695
44.3 Troisième exemple (données individuelles, évènement unique)	697
44.4 Lectures complémentaires	700
45 Modèles de comptage <i>zero-inflated</i> et <i>hurdle</i>	701
45.1 Données d'illustration	701
45.2 Modèles de comptage classique	702
45.3 Modèles <i>zero-inflated</i>	704
45.4 Modèles <i>hurdle</i>	710
45.5 Modèles de taux <i>zero-inflated</i> ou <i>hurdle</i>	713
45.6 Lectures complémentaires	713
46 Quel modèle choisir ?	714
 VII Pour aller plus loin	 715
47 Ressources documentaires	716
47.1 Ressources génériques	716
47.2 Analyse de réseaux	717
47.3 Analyse spatiale & Cartographie	717
47.4 Analyse textuelle	717

Préface

⚠ Guide en cours d'écriture

Ce guide est encore incomplet. Le plan prévu est visible à cette adresse : <https://github.com/larmarange/guide-R/issues/12>.

En attendant, nous vous conseillons de compléter votre lecture par le site [analyse-R](#).

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

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

De même, il ne s'agit pas d'une introduction ou d'un guide pour les utilisatrices et utilisateurs débutant·es. Si vous découvrez **R**, nous vous conseillons la lecture de *l'Introduction à R et au tidyverse* de Julien Barnier (<https://juba.github.io/tidyverse/>). Néanmoins, quelques rappels sur les bases du langage sont fournis dans la section *Bases du langage*. Une bonne compréhension de ces dernières, bien qu'un peu ardue de prime abord, permet de comprendre le sens des commandes que l'on utilise et de pleinement exploiter la puissance que **R** offre en matière de manipulation de données.

R dispose de nombreuses extensions ou packages (plus de 16 000) et il existe souvent plusieurs manières de procéder pour arriver au même résultat. En particulier, en matière de manipulation de données, on oppose¹ souvent *base R* qui repose sur les fonctions disponibles en standard dans **R**, la majorité étant fournies dans les packages `{base}`, `{utils}` ou encore `{stats}`, qui sont toujours chargés par défaut, et le `{tidyverse}` qui est une collection de packages comprenant, entre autres, `{dplyr}`, `{tibble}`, `{tidyr}`, `{forcats}` ou encore `{ggplot2}`. Il y a un débat ouvert, parfois passionné, sur le fait de privilégier l'une ou l'autre approche, et les

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

avantages et inconvénients de chacune dépendent de nombreux facteurs, comme la lisibilité du code ou bien les performances en temps de calcul. Dans ce guide, nous avons adopté un point de vue pragmatique et utiliserons, le plus souvent mais pas exclusivement, les fonctions du `{tidyverse}`, de même que nous avons privilégié d'autres packages, comme `{gtsummary}` ou `{ggstats}` par exemple pour la statistique descriptive. Cela ne signifie pas, pour chaque point abordé, qu'il s'agit de l'unique manière de procéder. Dans certains cas, il s'agit simplement de préférences personnelles.

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

En complément, on pourra également se référer aux [webin-R](#), une série de vidéos avec partage d'écran, librement accessibles sur YouTube : <https://www.youtube.com/c/webinR>.

Cette version du guide a utilisé *R version 4.3.3 (2024-02-29 ucrt)*. Ce document est généré avec [quarto](#) et le code source est disponible sur [GitHub](#). Pour toute suggestion ou correction, vous pouvez ouvrir un [ticket GitHub](#). Pour d'autres questions, vous pouvez utiliser les forums de discussion disponibles en bas de chaque page sur la version web du guide. Ce document est régulièrement mis à jour. La dernière version est consultable sur <https://larmarange.github.io/guide-R/>.

Remerciements

Ce document a bénéficié de différents apports provenant notamment de l'*Introduction à R* et de l'*Introduction à R et au tidyverse* de Julien Barnier et d'*analyse-R : introduction à l'analyse d'enquêtes avec R et RStudio*. Certains chapitres se sont appuyés sur l'ouvrage de référence *R for data science* par Hadley Wickham, Mine Çetinkaya-Rundel et Garret Grolemund, ou encore sur les [notes de cours](#) d'Ewan Gallic.

Merci donc à Julien Barnier, Julien BiauDET, François Briatte, Milan Bouchet-Valat, Mine Çetinkaya-Rundel, Ewen Gallic, Frédérique Giraud, Joël Gombin, Garret Grolemund, Mayeul Kauffmann, Christophe Lalanne, Nicolas Robette et Hadley Wickham.

Licence

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



partie I

Bases du langage

1 Packages

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

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

Il existe un très grand nombre d'extensions (plus de 16 000 à ce jour), qui sont diffusées par un réseau baptisé **CRAN** (*Comprehensive R Archive Network*).

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

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

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

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

1.1 Installation (CRAN)

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

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

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

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

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

Note

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

1.2 Chargement

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

```
library(gtsummary)
```

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

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

Important

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

1.3 Mise à jour

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

```
update.packages()
```

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

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

```
remove.packages("gtsummary")
```

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

Après une mise à jour majeure de **R**, il est souvent nécessaire de réinstaller tous les packages utilisés. De même, on peut parfois souhaiter mettre à jour uniquement les packages utilisés par un projet donné sans avoir à mettre à jour tous les autres packages présents sur son PC.

Une astuce consiste à avoir recours à la fonction `renv::dependencies()` qui examine le code du projet courant pour identifier les packages utilisés, puis à passer cette liste de packages à `remotes::install_cran()` qui installera les packages manquants ou pour lesquels une mise à jour est disponible.

Il vous suffit d'exécuter la commande ci-dessous :

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

1.4 Installation depuis GitHub

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

Dans d'autres cas de figure, la dernière version stable d'un package est disponible sur **CRAN** tandis que la version en cours de développement est, elle, disponible sur **GitHub**. Il faut

être vigilant avec les versions de développement. Parfois, elle corrige un bug ou introduit une nouvelle fonctionnalité qui n'est pas encore dans la version stable. Mais les versions de développement peuvent aussi contenir de nouveaux bugs ou des fonctionnalités instables.

⚠ Sous Windows

Pour les utilisatrices et utilisateurs sous **Windows**, il faut être conscient que le code source d'un package doit être compilé afin de pouvoir être utilisé. **CRAN** fournit une version des packages déjà compilée pour **Windows** ce qui facilite l'installation.

Par contre, lorsque l'on installe un package depuis **GitHub**, **R** ne récupère que le code source et il est donc nécessaire de compiler localement le package. Pour cela, il est nécessaire que soit installé sur le PC un outil complémentaire appelé **RTools**. Il est téléchargeable à l'adresse <https://cran.r-project.org/bin/windows/Rtools/>.

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

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

1.5 Le tidyverse

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

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

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

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

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

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

Lorsque l'on charge le package `{tidyverse}` avec `library()`, cela charge également en mémoire les principaux packages du *tidyverse*¹.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --  
v dplyr      1.1.4      v readr      2.1.5  
v forcats    1.0.0      v stringr    1.5.1  
v ggplot2    3.5.1      v tibble     3.2.1  
v lubridate  1.9.3      v tidyr      1.3.1  
v purrr      1.0.2  
-- Conflicts ----- tidyverse_conflicts() --  
x dplyr::filter() masks stats::filter()  
x dplyr::lag()     masks stats::lag()  
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```



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

¹Si on a besoin d'un autre package du *tidyverse* comme `{lubridate}`, il faudra donc le charger individuellement.

2 Vecteurs

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

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

2.1 Types et classes

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

- les nombres réels (c'est-à-dire les nombres décimaux¹), par exemple `5.23` ;
- les nombres entiers, que l'on saisi en ajoutant le suffixe `L`², par exemple `4L` ;
- les chaînes de caractères (qui correspondent à du texte), que l'on saisit avec des guillemets doubles (`"`) ou simples (`'`), par exemple `"abc"` ;
- les valeurs logiques ou valeurs booléennes, à savoir vrai ou faux, que l'on représente avec les mots `TRUE` et `FALSE` (en majuscules³).

En plus de ces types de base, il existe de nombreux autres types de vecteurs utilisés pour représenter toutes sortes de données, comme les facteurs (voir Chapitre 9) ou les dates (voir Chapitre 34).

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

¹Pour rappel, **R** étant anglophone, le caractère utilisé pour indiquer les chiffres après la virgule est le point (`.`).

²**R** utilise 32 bits pour représenter des nombres entiers, ce qui correspond en informatique à des entiers longs ou *long integers* en anglais, d'où la lettre `L` utilisée pour indiquer un nombre entier.

³On peut également utiliser les raccourcis `T` et `F`. Cependant, pour une meilleure lisibilité du code, il est préférable d'utiliser les versions longues `TRUE` et `FALSE`.

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

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

Astuce

Pour un vecteur numérique, le type est **"double"** car **R** utilise une double précision pour stocker en mémoire les nombres réels.

En interne, les facteurs sont représentés par un nombre entier auquel est attaché une étiquette, c'est pourquoi `typeof()` renvoie **"integer"**.

Quand aux dates, elles sont stockées en interne sous la forme d'un nombre réel représentant le nombre de jours depuis le 1^{er} janvier 1970, d'où le fait que `typeof()` renvoie **"double"**.

2.2 Création d'un vecteur

Pour créer un vecteur, on utilisera la fonction `c()` en lui passant la liste des valeurs à combiner⁴.

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

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

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

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

⁴La lettre `c` est un raccourci du mot anglais *combine*, puisque cette fonction permet de combiner des valeurs individuelles dans un vecteur unique.

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

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

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

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

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

```
class(x)
```

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

```
1:5
```

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

```
24:32
```

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

```
55:43
```

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

2.3 Longueur d'un vecteur

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

```
length(taille)
```

```
[1] 6
```

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

```
[1] 2
```

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

```
length(NULL)
```

```
[1] 0
```

2.4 Combiner des vecteurs

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

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

```
[1] 4
```

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

```
[1] 6
```

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

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

```
length(z)
```

```
[1] 10
```

2.5 Vecteurs nommés

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

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

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

```
sexe
```

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

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

```
names(sexe)
```

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

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

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

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

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

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

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

2.6 Indexation par position

L'indexation est l'une des fonctionnalités les plus puissantes mais aussi les plus difficiles à maîtriser de **R**. Il s'agit d'opérations permettant de sélectionner des sous-ensembles de valeurs en fonction de différents critères. Il existe trois types d'indexation : (i) l'indexation par position, (ii) l'indexation par nom et (iii) l'indexation par condition. Le principe est toujours le même : on indique entre crochets⁵ (`[]`) ce qu'on souhaite garder ou non.

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

Reprenons notre vecteur `taille` :

```
taille
```

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

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

```
taille[1]
```

```
[1] 1.88
```

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

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

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

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

⁵Pour rappel, les crochets s'obtiennent sur un clavier français de type PC en appuyant sur la touche Alt Gr et la touche (ou).

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

Si on veut le dernier élément :

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

```
[1] 1.72
```

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

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

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

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

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

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

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

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

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

2.7 Indexation par nom

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

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

```
Anna  
"f"
```

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

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

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

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

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

2.8 Indexation par condition

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

```
sexe
```

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

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

Michael	John
"h"	"h"

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

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

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

```
sexe[urbain]
```

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

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

Table 2.2: Opérateurs de comparaison

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

Voyons tout de suite un exemple :

```
poids >= 80
```

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

Que s'est-il passé ? Nous avons fourni à **R** une condition et il nous a renvoyé un vecteur logique avec autant d'éléments qu'il y a d'observations et dont la valeur est **TRUE** si la condition est remplie et **FALSE** dans les autres cas. Nous pouvons alors utiliser ce vecteur logique pour obtenir la taille des participants pesant 80 kilogrammes ou plus :

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

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

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

Table 2.3: Opérateurs logiques

Opérateur logique	Signification
&	et logique
	ou logique
!	négation logique

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

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

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

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

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

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

! Comparaison et valeur manquante

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

```
taille
```

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

```
taille > 1.8
```

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

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

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


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

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

Pour compliquer encore un peu le tout, lorsqu'on utilise une condition pour l'indexation, si la condition renvoie NA, **R** ne sélectionne pas l'élément mais retourne quand même la valeur NA. Ceci a donc des conséquences sur le résultat d'une indexation par comparaison. Par exemple si je cherche à connaître le poids des personnes mesurant 1,80 mètre ou plus :

```
taille
```

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

```
poids
```

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

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

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

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

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

```
[1] 80 75
```

2.9 Assignment par indexation

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

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

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

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

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

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

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

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

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

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

```
length(sexe)
```

```
[1] 6
```

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

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

```
length(sexe)
```

[1] 7

2.10 En résumé

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

2.11 webin-R

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

<https://youtu.be/Eh8piunoqQc>

3 Listes

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

3.1 Propriétés et création

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

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

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

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

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

```
length(l1)
```

```
[1] 2
```

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

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

```
$minuscules
```

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

```
$majuscules
```

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

```
$mois
```

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

```
length(l2)
```

```
[1] 3
```

```
names(l2)
```

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

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

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

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

```
length(l)
```

```
[1] 2
```

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

```
str(l)
```

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

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

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

```
[1] 5
```

```
str(l)
```

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

Note

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

3.2 Indexation

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

```
l
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
[1] NA
```


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

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

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

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

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

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

Maintenant, nous pouvons calculer notre moyenne :

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

```
[1] 3
```

Nous pouvons aussi utiliser l'indexation par nom.

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

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

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

```
l$mois
```

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

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

```
l$1
```

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

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

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

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

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

```
$minuscules
```

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

```
$majuscules
```

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

```
$mois
```

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

3.3 En résumé

- Les listes sont des objets unidimensionnels pouvant contenir tout type d'objet, y compris d'autres listes.
- Elles ont une longueur qu'on obtient avec `length()`.
- On crée une liste avec `list()` et on peut fusionner des listes avec `append()`.

- Tout comme les vecteurs, les listes peuvent être nommées et les noms des éléments s'obtiennent avec `base::names()`.
- Les crochets simples (`[]`) permettent de sélectionner les éléments d'une liste, en utilisant l'indexation par position, l'indexation par nom ou l'indexation par condition. Cela renvoie toujours une autre liste.
- Les doubles crochets (`[[]]`) renvoient directement le contenu d'un élément de la liste qu'on aura sélectionné par position ou par nom.
- Le symbole `$` est un raccourci pour facilement sélectionner un élément par son nom, `liste$nom` étant équivalent à `liste[["nom"]]`.

3.4 webin-R

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

<https://youtu.be/Eh8piunoqQc>

4 Tableaux de données

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

4.1 Propriétés et création

Dans **R**, les tableaux de données sont tout simplement des listes (voir Chapitre 3) avec quelques propriétés spécifiques :

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

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

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

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

```
   sexe age blond  
1    f  52 FALSE  
2    f  31  TRUE  
3    h  29  TRUE  
4    h  35 FALSE
```

```
str(df)
```

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

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

```
length(df)
```

```
[1] 3
```

```
names(df)
```

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

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

```
nrow(df)
```

```
[1] 4
```

```
ncol(df)
```

```
[1] 3
```

```
dim(df)
```

```
[1] 4 3
```

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

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

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

4.2 Indexation

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

```
df[1]
```

	sexe
Anna	f
Mary-Ann	f
Michael	h
John	h

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

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

```
df$sexe
```

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

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

Un premier exemple :

```
df
```

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

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

```
[1] 29
```

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

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

```
[1] 29
```

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

```
[1] 29
```

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

```
[1] 29
```

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

```
[1] 29
```

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

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

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

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

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

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

Avertissement

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

```
df[2, ]
```

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

```
df[, 2]
```

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

```
df[2]
```

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

Note

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

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

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

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

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

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



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

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

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

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

4.3 Afficher les données

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

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

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

```
View(hdv2003)
```

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

```
head(hdv2003)
```

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

Showing 1 to 19 of 2,000 entries

Figure 4.1: Interface View() de R RStudio

```

id age  sexe                                nivetud    poids
1  1  28  Femme Enseignement superieur y compris technique superieur 2634.398
2  2  23  Femme                                <NA> 9738.396
3  3  59  Homme                Derniere annee d'etudes primaires 3994.102
4  4  34  Homme Enseignement superieur y compris technique superieur 5731.662
5  5  71  Femme                Derniere annee d'etudes primaires 4329.094
6  6  35  Femme    Enseignement technique ou professionnel court 8674.699

      occup      qualif freres.soeurs clso
1 Exerce une profession    Employe      8 Oui
2      Etudiant, eleve      <NA>      2 Oui
3 Exerce une profession Technicien      2 Non
4 Exerce une profession Technicien      1 Non
5      Retraite    Employe      0 Oui
6 Exerce une profession    Employe      5 Non

      relig                                trav.imp    trav.satisf
1 Ni croyance ni appartenance                Peu important Insatisfaction
2 Ni croyance ni appartenance                <NA>      <NA>
3 Ni croyance ni appartenance Aussi important que le reste    Equilibre
4 Appartenance sans pratique Moins important que le reste    Satisfaction
5      Praticquant regulier                <NA>      <NA>
6 Ni croyance ni appartenance                Le plus important    Equilibre
hard.rock lecture.bd peche.chasse cuisine bricol cinema sport heures.tv

```

1	Non	Non	Non	Oui	Non	Non	Non	0
2	Non	Non	Non	Non	Non	Oui	Oui	1
3	Non	Non	Non	Non	Non	Non	Oui	0
4	Non	Non	Non	Oui	Oui	Oui	Oui	2
5	Non	Non	Non	Non	Non	Non	Non	3
6	Non	Non	Non	Non	Non	Oui	Oui	2

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

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

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

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

```
Rows: 2,000
```

```
Columns: 20
```

```
$ id      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~
$ age     <int> 28, 23, 59, 34, 71, 35, 60, 47, 20, 28, 65, 47, 63, 67, ~
$ sexe    <fct> Femme, Femme, Homme, Homme, Femme, Femme, Femme, Homme, ~
$ nivetud <fct> "Enseignement superieur y compris technique superieur", ~
$ poids   <dbl> 2634.3982, 9738.3958, 3994.1025, 5731.6615, 4329.0940, 8~
$ occup   <fct> "Exerce une profession", "Etudiant, eleve", "Exerce une ~
$ qualif  <fct> Employe, NA, Technicien, Technicien, Employe, Employe, 0~
$ freres.soeurs <int> 8, 2, 2, 1, 0, 5, 1, 5, 4, 2, 3, 4, 1, 5, 2, 3, 4, 0, 2,~
$ clso    <fct> Oui, Oui, Non, Non, Oui, Non, Oui, Non, Oui, Non, Oui, 0~
$ relig   <fct> Ni croyance ni appartenance, Ni croyance ni appartenance~
```

```

$ trav.imp      <fct> Peu important, NA, Aussi important que le reste, Moins i~
$ trav.satisf  <fct> Insatisfaction, NA, Equilibre, Satisfaction, NA, Equilib~
$ hard.rock    <fct> Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, N~
$ lecture.bd   <fct> Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, N~
$ peche.chasse <fct> Non, Non, Non, Non, Non, Non, Oui, Oui, Non, Non, Non, N~
$ cuisine      <fct> Oui, Non, Non, Oui, Non, Non, Oui, Oui, Non, Non, Oui, N~
$ bricol       <fct> Non, Non, Non, Oui, Non, Non, Non, Oui, Non, Non, Oui, O~
$ cinema       <fct> Non, Oui, Non, Oui, Non, Oui, Non, Non, Oui, Oui, Oui, N~
$ sport        <fct> Non, Oui, Oui, Oui, Non, Oui, Non, Non, Non, Oui, Non, O~
$ heures.tv    <dbl> 0.0, 1.0, 0.0, 2.0, 3.0, 2.0, 2.9, 1.0, 2.0, 2.0, 1.0, 0~

```

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

```

library(labelled)
look_for(hdv2003)

```

pos	variable	label	col_type	missing	values
1	id	-	int	0	
2	age	-	int	0	
3	sexe	-	fct	0	Homme Femme
4	nivetud	-	fct	112	N'a jamais fait d'etudes A arrete ses etudes, avant la derni~ Derniere annee d'etudes primaires 1er cycle 2eme cycle Enseignement technique ou professio~ Enseignement technique ou professio~ Enseignement superieur y compris te~
5	poids	-	dbl	0	
6	occup	-	fct	0	Exerce une profession Chomeur Etudiant, eleve Retraite Retire des affaires Au foyer Autre inactif
7	qualif	-	fct	347	Ouvrier specialise Ouvrier qualifie Technicien Profession intermediaire

					Cadre
					Employe
					Autre
8	freres.soeurs	-	int	0	
9	clso	-	fct	0	Oui
					Non
					Ne sait pas
10	relig	-	fct	0	Pratiquant regulier
					Pratiquant occasionnel
					Appartenance sans pratique
					Ni croyance ni appartenance
					Rejet
					NSP ou NVPR
11	trav.imp	-	fct	952	Le plus important
					Aussi important que le reste
					Moins important que le reste
					Peu important
12	trav.satisf	-	fct	952	Satisfaction
					Insatisfaction
					Equilibre
13	hard.rock	-	fct	0	Non
					Oui
14	lecture.bd	-	fct	0	Non
					Oui
15	peche.chasse	-	fct	0	Non
					Oui
16	cuisine	-	fct	0	Non
					Oui
17	bricol	-	fct	0	Non
					Oui
18	cinema	-	fct	0	Non
					Oui
19	sport	-	fct	0	Non
					Oui
20	heures.tv	-	dbl	5	

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

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

```
pos variable    label col_type missing values
```

```

11  trav.imp      -      fct      952      Le plus important
                                         Aussi important que le reste
                                         Moins important que le reste
                                         Peu important
12  trav.satisf  -      fct      952      Satisfaction
                                         Insatisfaction
                                         Equilibre

```

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

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

```
summary(hdv2003)
```

```

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

                                nivetud      poids
Enseignement technique ou professionnel court      :463  Min.   : 78.08
Enseignement superieur y compris technique superieur:441 1st Qu.: 2221.82
Derniere annee d'etudes primaires                    :341 Median : 4631.19
1er cycle                                             :204 Mean   : 5535.61
2eme cycle                                           :183 3rd Qu.: 7626.53
(Other)                                              :256 Max.   :31092.14
NA's                                                :112

                                occup      qualif      freres.soeurs
Exerce une profession:1049  Employe      :594  Min.   : 0.000
Chomeur                  : 134  Ouvrier qualifie      :292 1st Qu.: 1.000
Etudiant, eleve          : 94   Cadre        :260 Median : 2.000
Retraite                  : 392  Ouvrier specialise    :203 Mean   : 3.283
Retire des affaires      : 77   Profession intermediaire:160 3rd Qu.: 5.000
Au foyer                 : 171  (Other)              :144 Max.   :22.000
Autre inactif            : 83   NA's                :347

      clso      relig

```

```

Oui      : 936   Praticquant regulier      :266
Non      :1037   Praticquant occasionnel    :442
Ne sait pas: 27   Appartenance sans pratique :760
                        Ni croyance ni appartenance:399
                        Rejet                : 93
                        NSP ou NVPR          : 40

```

```

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

```

```

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

```

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

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

```

Homme Femme
899  1101

```

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

```

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

```

4.4 En résumé

- Les tableaux de données sont des listes avec des propriétés particulières :

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

4.5 webin-R

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

<https://youtu.be/Eh8piunoqQc>

5 Tibbles

5.1 Le concept de tidy data

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

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

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

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

Un chapitre dédié à `{tidyr}` (voir Chapitre 36) présente comment définir et rendre des données *tidy* avec ce package.

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

5.2 tibbles : des tableaux de données améliorés

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

Ce format est fourni est géré par l'extension du même nom (`{tibble}`), qui fait partie du cœur du *tidyverse*. La plupart des fonctions des extensions du *tidyverse* acceptent des *data.frames* en entrée, mais retournent un *tibble*.

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

- n'ont pas de noms de lignes (*rownames*)
- autorisent des noms de colonnes invalides pour les *data.frames* (espaces, caractères spéciaux, nombres...) ¹

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

- s'affichent plus intelligemment que les *data frames* : seules les premières lignes sont affichées, ainsi que quelques informations supplémentaires utiles (dimensions, types des colonnes...)
- ne font pas de *partial matching* sur les noms de colonnes 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*.

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

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

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

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

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

```
# A tibble: 32 x 11
   mpg   cyl  disp    hp  drat    wt   qsec    vs  am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21     6  160   110  3.9   2.62  16.5    0    1     4     4
2  21     6  160   110  3.9   2.88  17.0    0    1     4     4
3 22.8     4  108    93  3.85  2.32  18.6    1    1     4     1
4 21.4     6  258   110  3.08  3.22  19.4    1    0     3     1
5 18.7     8  360   175  3.15  3.44  17.0    0    0     3     2
6 18.1     6  225   105  2.76  3.46  20.2    1    0     3     1
7 14.3     8  360   245  3.21  3.57  15.8    0    0     3     4
```

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

```

 8 24.4      4 147.    62 3.69 3.19 20      1    0    4    2
 9 22.8      4 141.    95 3.92 3.15 22.9    1    0    4    2
10 19.2      6 168.   123 3.92 3.44 18.3    1    0    4    4
# i 22 more rows

```

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

```
class(d)
```

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

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

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

```

# A tibble: 32 x 12
  rowname      mpg  cyl  disp   hp  drat   wt  qsec    vs  am  gear  carb
  <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Mazda RX4    21      6  160   110  3.9   2.62  16.5    0    1    4     4
2 Mazda RX4 ~  21      6  160   110  3.9   2.88  17.0    0    1    4     4
3 Datsun 710   22.8     4  108    93  3.85  2.32  18.6    1    1    4     1
4 Hornet 4 D~  21.4     6  258   110  3.08  3.22  19.4    1    0    3     1
5 Hornet Spo~  18.7     8  360   175  3.15  3.44  17.0    0    0    3     2
6 Valiant     18.1     6  225   105  2.76  3.46  20.2    1    0    3     1
7 Duster 360  14.3     8  360   245  3.21  3.57  15.8    0    0    3     4
8 Merc 240D   24.4     4  147.    62  3.69  3.19  20      1    0    4     2
9 Merc 230    22.8     4  141.    95  3.92  3.15  22.9    1    0    4     2
10 Merc 280   19.2     6  168.   123  3.92  3.44  18.3    1    0    4     4
# i 22 more rows

```

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

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

	rowname	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
11	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
13	Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
14	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
15	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
16	Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
17	Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
18	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
19	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
20	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
21	Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
22	Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
23	AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
24	Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
25	Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
26	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
27	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
28	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
29	Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
30	Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
31	Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
32	Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

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

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

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1

Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Note

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

5.3 Données et tableaux imbriqués

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

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

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

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

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

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

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

```
reg <-
  iris |>
  group_by(Species) |>
  nest() |>
  mutate(
    model = map(
```

```

    data,
    ~ lm(Sepal.Length ~ Petal.Length + Petal.Width, data = .)
  ),
  tbl = map(model, gtsummary::tbl_regression)
)
reg

```

```

# A tibble: 3 x 4
# Groups:   Species [3]
  Species    data      model  tbl
  <fct>    <list>    <list> <list>
1 setosa   <tibble [50 x 4]> <lm>   <tbl_rgrs>
2 versicolor <tibble [50 x 4]> <lm>   <tbl_rgrs>
3 virginica <tibble [50 x 4]> <lm>   <tbl_rgrs>

```

```

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

```

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

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

6 Attributs

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

Parmi les attributs les plus courants, on retrouve notamment :

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

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

```
attributes(iris)
```

```
$names
```

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

```
$class
```

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

```
$row.names
```

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

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


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

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

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

```
class(iris)
```

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

```
names(iris)
```

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

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

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

```
$names
```

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

```
$class
```

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

```
$row.names
```

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

```
$perso
```

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

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

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

partie II

Manipulation de données

7 Le pipe

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

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

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

La moyenne est de 6,7.

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

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

La moyenne est de 6,7.

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

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

```

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

```

La moyenne est de 6,7.

C'est déjà mieux, mais toujours pas optimal.

7.1 Le pipe natif de R : `|>`

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

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

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

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

```

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

```

La moyenne est de 6,7.

Le code n'est-il pas plus lisible ?

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

7.2 Le pipe du tidyverse : %>%

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

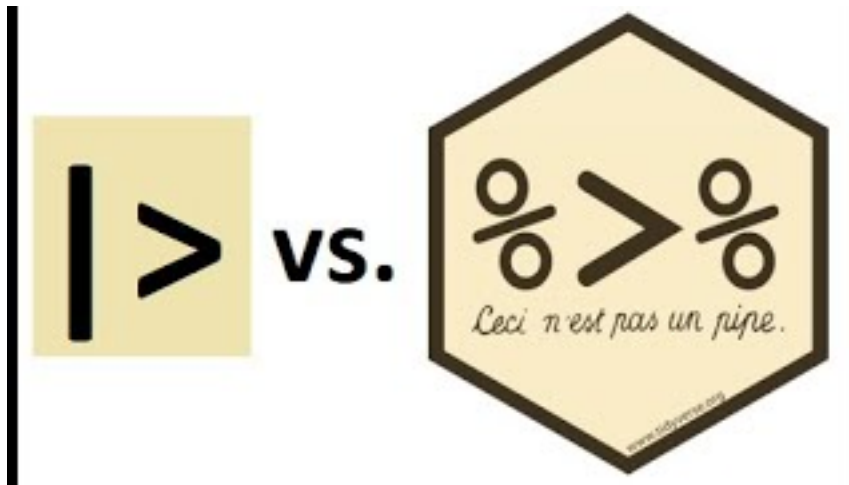
En cela, **R** s'est notamment inspiré d'un opérateur similaire introduit dès 2014 dans le *tidyverse*. Le pipe du *tidyverse* fonctionne de manière similaire. Il est implémenté dans le package `{magrittr}` qui doit donc être chargé en mémoire. Le *pipe* est également disponible lorsque l'on effectue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

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

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

La moyenne est de 6,7.

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



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

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

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

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

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

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

```
[1] 1.199333
```

Cette écriture est équivalente à :

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

```
[1] 1.199333
```

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

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

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

```
v[2]
```

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

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

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

```
[1] 3.2
```

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

```
[1] 3.2
```

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

```
[1] 3.2
```

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


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

NULL

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

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

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

NULL

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

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

8 dplyr

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

`{dplyr}` part du principe que les données sont *tidy* (chaque variable est une colonne, chaque observation est une ligne, voir Chapitre 5). Les verbes de `{dplyr}` prennent en entrée un tableau de données¹ (*data frame* ou *tibble*) et renvoient systématiquement un *tibble*.

```
library(dplyr)
```

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

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

On va charger les trois tables du jeu de données :

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

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

¹Le package `{dbplyr}` permet d'étendre les verbes de `{dplyr}` à des tables de bases de données **SQL**, `{dtplyr}` à des tableaux de données du type `{data.table}` et `{srvyr}` à des données pondérées du type `{survey}`.

8.1 Opérations sur les lignes

8.1.1 filter()

`dplyr::filter()` sélectionne des lignes d'un tableau de données selon une condition. On lui passe en paramètre un test, et seules les lignes pour lesquelles ce test renvoi TRUE (vrai) sont conservées².

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

Cela peut s'écrire plus simplement avec un pipe :

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

```
# A tibble: 27,004 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     1     517             515           2     830           819
```

²Si le test renvoie faux (FALSE) ou une valeur manquante (NA), les lignes correspondantes ne seront donc pas sélectionnées.

```

2 2013 1 1 533 529 4 850 830
3 2013 1 1 542 540 2 923 850
4 2013 1 1 544 545 -1 1004 1022
5 2013 1 1 554 600 -6 812 837
6 2013 1 1 554 558 -4 740 728
7 2013 1 1 555 600 -5 913 854
8 2013 1 1 557 600 -3 709 723
9 2013 1 1 557 600 -3 838 846
10 2013 1 1 558 600 -2 753 745
# i 26,994 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>

```

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

```

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

```

```

# A tibble: 14,919 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>   <int>         <int>      <dbl>      <int>         <int>
1 2013     1     1     611           600         11       945           931
2 2013     1     1     623           610         13       920           915
3 2013     1     1     743           730         13      1107          1100
4 2013     1     1     743           730         13      1059          1056
5 2013     1     1     851           840         11      1215          1206
6 2013     1     1     912           900         12      1241          1220
7 2013     1     1     914           900         14      1058          1043
8 2013     1     1     920           905         15      1039          1025
9 2013     1     1    1011          1001         10      1133          1128
10 2013     1     1    1112          1100         12      1440          1438
# i 14,909 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>

```

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

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

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

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

A tibble: 342 x 19

	year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>
1	2013	1	1	857	900	-3	1516	1530
2	2013	1	2	909	900	9	1525	1530
3	2013	1	3	914	900	14	1504	1530
4	2013	1	4	900	900	0	1516	1530
5	2013	1	5	858	900	-2	1519	1530
6	2013	1	6	1019	900	79	1558	1530
7	2013	1	7	1042	900	102	1620	1530
8	2013	1	8	901	900	1	1504	1530
9	2013	1	9	641	900	1301	1242	1530
10	2013	1	10	859	900	-1	1449	1530

i 332 more rows

i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
hour <dbl>, minute <dbl>, time_hour <dtm>

💡 Évaluation contextuelle

Il est important de noter que `{dplyr}` procède à une évaluation contextuelle des expressions qui lui sont passées. Ainsi, on peut indiquer directement le nom d'une variable et `{dplyr}` l'interprétera dans le contexte du tableau de données, c'est-à-dire regardera s'il existe une colonne portant ce nom dans le tableau.

Dans l'expression `flights |> filter(month == 1)`, `month` est interprété comme la colonne `month` du tableau `flights`, à savoir `flights$month`.

Il est également possible d'indiquer des objets extérieurs au tableau :

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

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

Cela fonctionne car il n'y a pas de colonne *m* dans `flights`. Dès lors, `{dplyr}` regarde s'il existe un objet *m* dans l'environnement de travail.

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

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

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     1     517           515           2     830           819
2  2013     1     1     533           529           4     850           830
3  2013     1     1     542           540           2     923           850
4  2013     1     1     544           545          -1    1004          1022
5  2013     1     1     554           600          -6     812           837
6  2013     1     1     554           558          -4     740           728
7  2013     1     1     555           600          -5     913           854
8  2013     1     1     557           600          -3     709           723
9  2013     1     1     557           600          -3     838           846
10 2013     1     1     558           600          -2     753           745
```

```
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

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

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

```
# A tibble: 28,834 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>   <int>         <int>         <dbl>     <int>         <int>
1  2013     3     1       4           2159           125       318             56
2  2013     3     1      50           2358            52       526            438
3  2013     3     1     117           2245           152       223           2354
4  2013     3     1    454            500            -6       633            648
5  2013     3     1    505            515           -10       746            810
6  2013     3     1    521            530            -9       813            827
7  2013     3     1    537            540            -3       856            850
8  2013     3     1    541            545            -4      1014           1023
9  2013     3     1    549            600           -11       639            703
10 2013     3     1    550            600           -10       747            801
# i 28,824 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

8.1.2 slice()

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

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

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

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

Si l'on veut sélectionner les 5 premières lignes :

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

```
# A tibble: 5 x 8
  faa   name          lat   lon   alt   tz dst   tzone
  <chr> <chr>          <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 04G   Lansdowne Airport  41.1 -80.6  1044   -5 A   America/New~
2 06A   Moton Field Municipal Airport  32.5 -85.7   264   -6 A   America/Chi~
3 06C   Schaumburg Regional  42.0 -88.1   801   -6 A   America/Chi~
4 06N   Randall Airport    41.4 -74.4   523   -5 A   America/New~
5 09J   Jekyll Island Airport  31.1 -81.4    11   -5 A   America/New~
```

8.1.3 arrange()

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

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

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

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



```
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

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

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

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

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

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

```
# A tibble: 336,776 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     9     641            900        1301    1242           1530
2  2013     6    15    1432           1935        1137    1607           2120
3  2013     1    10    1121           1635        1126    1239           1810
```

```

4 2013    9    20    1139          1845    1014    1457          2210
5 2013    7    22     845          1600    1005    1044          1815
6 2013    4    10    1100          1900     960    1342          2211
7 2013    3    17    2321           810     911     135          1020
8 2013    6    27     959          1900     899    1236          2226
9 2013    7    22    2257           759     898     121          1026
10 2013   12     5     756          1700     896    1058          2020
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>

```

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

```

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

```

```

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

```

8.1.4 slice_sample()

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

```

airports |>
  slice_sample(n = 5)

```

```

# A tibble: 5 x 8
  faa   name          lat   lon   alt   tz dst  tzone
  <chr> <chr>         <dbl> <dbl> <dbl> <chr> <chr> <chr>

```

	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
1	WLK	Selawik Airport	66.6	-160.	17	-9	A	America/Anc~
2	GWO	Greenwood Leflore	33.5	-90.1	162	-6	A	America/Chi~
3	ECG	Elizabeth City Cgas Rgnl	36.3	-76.2	12	-5	A	America/New~
4	DVL	Devils Lake Regional Airport	48.1	-98.9	1445	-6	A	America/Chi~
5	LMT	Klamath Falls Airport	42.2	-122.	4095	-8	A	America/Los~

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

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

```
# A tibble: 33,677 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     2    14     820             817           3    1148           1127
2  2013     7     3    1628            1606          22    1916           1909
3  2013     2     2     819             819           0    1256           1307
4  2013    11    16     743             745          -2    1037           1056
5  2013     4    25    1405            1359           6    1514           1519
6  2013     1     2    1456            1500          -4    1824           1810
7  2013     1     8     544             530          14     853            829
8  2013     6    11     810             815          -5    1012           1008
9  2013    12    25    1606            1550          16    1738           1744
10 2013     4     5    1432            1435          -3    1557           1612
# i 33,667 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>
```

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

8.1.5 distinct()

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

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

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

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

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

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

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

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

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

8.2 Opérations sur les colonnes

8.2.1 select()

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

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

```
# A tibble: 1,458 x 2
   lat   lon
   <dbl> <dbl>
1  41.1 -80.6
2  32.5 -85.7
3  42.0 -88.1
4  41.4 -74.4
5  31.1 -81.4
6  36.4 -82.2
```

```

7  41.5  -84.5
8  42.9  -76.8
9  39.8  -76.6
10 48.1 -123.
# i 1,448 more rows

```

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

```

airports |>
  select(-lat, -lon)

```

```

# A tibble: 1,458 x 6
  faa   name                alt    tz dst  tzone
  <chr> <chr>                <dbl> <dbl> <chr> <chr>
1 04G   Lansdowne Airport      1044   -5 A   America/New_York
2 06A   Moton Field Municipal Airport  264   -6 A   America/Chicago
3 06C   Schaumburg Regional      801   -6 A   America/Chicago
4 06N   Randall Airport        523   -5 A   America/New_York
5 09J   Jekyll Island Airport     11   -5 A   America/New_York
6 0A9   Elizabethton Municipal Airport 1593   -5 A   America/New_York
7 0G6   Williams County Airport   730   -5 A   America/New_York
8 0G7   Finger Lakes Regional Airport  492   -5 A   America/New_York
9 0P2   Shoestring Aviation Airfield 1000   -5 U   America/New_York
10 OS9  Jefferson County Intl     108   -8 A   America/Los_Angeles
# i 1,448 more rows

```

`dplyr::select()` comprend toute une série de fonctions facilitant la sélection de multiples colonnes. Par exemple, `dplyr::starts_with()`, `dplyr::ends_with()`, `dplyr::contains()` ou `dplyr::matches()` permettent d'exprimer des conditions sur les noms de variables :

```

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

```

```

# A tibble: 336,776 x 2
  dep_time dep_delay
  <int>      <dbl>
1     517         2
2     533         4
3     542         2
4     544        -1
5     554        -6

```

```

6      554      -4
7      555      -5
8      557      -3
9      557      -3
10     558      -2
# i 336,766 more rows

```

La syntaxe `colonne1:colonne2` permet de sélectionner toutes les colonnes situées entre *colonne1* et *colonne2* incluses³ :

```

flights |>
  select(year:day)

```

```

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

```

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

```

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

```

```

# A tibble: 336,776 x 3
   year month   day
<int> <int> <int>

```

³À 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.

```

1 2013      1      1
2 2013      1      1
3 2013      1      1
4 2013      1      1
5 2013      1      1
6 2013      1      1
7 2013      1      1
8 2013      1      1
9 2013      1      1
10 2013     1      1
# i 336,766 more rows

```

```

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

```

```

Error in `select()`:
i In argument: `all_of(c("century", "year", "month", "day"))`.
Caused by error in `all_of()`:
! Can't subset elements that don't exist.
x Element `century` doesn't exist.

```

```

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

```

```

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

```

```

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

```


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

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

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

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

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

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

8.2.2 relocate()

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

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

```
# A tibble: 1,458 x 8
   lon lat name          faa alt tz dst tzone
<dbl> <dbl> <chr>          <chr> <dbl> <dbl> <chr> <chr>
1 -80.6 41.1 Lansdowne Airport 04G 1044 -5 A America/~
2 -85.7 32.5 Moton Field Municipal Airport 06A 264 -6 A America/~
3 -88.1 42.0 Schaumburg Regional 06C 801 -6 A America/~
4 -74.4 41.4 Randall Airport 06N 523 -5 A America/~
5 -81.4 31.1 Jekyll Island Airport 09J 11 -5 A America/~
6 -82.2 36.4 Elizabethton Municipal Airport 0A9 1593 -5 A America/~
7 -84.5 41.5 Williams County Airport 0G6 730 -5 A America/~
8 -76.8 42.9 Finger Lakes Regional Airport 0G7 492 -5 A America/~
9 -76.6 39.8 Shoestring Aviation Airfield OP2 1000 -5 U America/~
10 -123. 48.1 Jefferson County Intl OS9 108 -8 A America/~
# i 1,448 more rows
```

8.2.3 rename()

Une variante de `dplyr::select()` est `dplyr::rename()`⁴, qui permet de renommer facilement des colonnes. On l'utilise en lui passant des paramètres de la forme `nouveau_nom = ancien_nom`. Ainsi, si on veut renommer les colonnes *lon* et *lat* de *airports* en *longitude* et *latitude* :

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

```
# A tibble: 1,458 x 8
   faa name          latitude longitude alt tz dst tzone
<chr> <chr>          <dbl>    <dbl> <dbl> <dbl> <chr> <chr>
1 04G Lansdowne Airport 41.1 -80.6 1044 -5 A Amer~
```

⁴Il est également possible de renommer des colonnes directement avec `select()`, avec la même syntaxe que pour `rename()`.

```

2 06A Moton Field Municipal Airpo~ 32.5 -85.7 264 -6 A Amer~
3 06C Schaumburg Regional 42.0 -88.1 801 -6 A Amer~
4 06N Randall Airport 41.4 -74.4 523 -5 A Amer~
5 09J Jekyll Island Airport 31.1 -81.4 11 -5 A Amer~
6 0A9 Elizabethton Municipal Airp~ 36.4 -82.2 1593 -5 A Amer~
7 0G6 Williams County Airport 41.5 -84.5 730 -5 A Amer~
8 0G7 Finger Lakes Regional Airpo~ 42.9 -76.8 492 -5 A Amer~
9 OP2 Shoestring Aviation Airfield 39.8 -76.6 1000 -5 U Amer~
10 OS9 Jefferson County Intl 48.1 -123. 108 -8 A Amer~
# i 1,448 more rows

```

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

```

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

```

```

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

```

8.2.4 rename_with()

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

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

```
# A tibble: 1,458 x 8
```

	FAA	NAME	LAT	LON	ALT	TZ	DST	TZONE
	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
1	04G	Lansdowne Airport	41.1	-80.6	1044	-5	A	America/~
2	06A	Moton Field Municipal Airport	32.5	-85.7	264	-6	A	America/~
3	06C	Schaumburg Regional	42.0	-88.1	801	-6	A	America/~
4	06N	Randall Airport	41.4	-74.4	523	-5	A	America/~
5	09J	Jekyll Island Airport	31.1	-81.4	11	-5	A	America/~
6	0A9	Elizabethton Municipal Airport	36.4	-82.2	1593	-5	A	America/~
7	0G6	Williams County Airport	41.5	-84.5	730	-5	A	America/~
8	0G7	Finger Lakes Regional Airport	42.9	-76.8	492	-5	A	America/~
9	0P2	Shoestring Aviation Airfield	39.8	-76.6	1000	-5	U	America/~
10	0S9	Jefferson County Intl	48.1	-123.	108	-8	A	America/~

```
# i 1,448 more rows
```

On pourra notamment utiliser les fonctions du package `snakecase` et, en particulier, `snakecase::to_snake_case()` que je recommande pour nommer de manière consistante les variables⁵.

8.2.5 pull()

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

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

```
[1] 1001.416
```

Note

`dplyr::pull()` ressemble à la fonction `purrr::chuck()` que nous avons déjà abordée (cf. Section 7.4). Cependant, `dplyr::pull()` ne fonctionne que sur des tableaux de don-

⁵Le *snake case* est une convention typographique en informatique consistant à écrire des ensembles de mots, généralement, en minuscules en les séparant par des tirets bas.

nées tandis que `purrr::chuck()` est plus générique et peut s'appliquer à tous types de listes.

8.2.6 mutate()

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

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

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

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

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

8.3 Opérations groupées

8.3.1 group_by()

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

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

```
# A tibble: 336,776 x 21
# Groups:   month [12]
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     1     517           515           2     830           819
2  2013     1     1     533           529           4     850           830
3  2013     1     1     542           540           2     923           850
4  2013     1     1     544           545          -1    1004          1022
5  2013     1     1     554           600          -6     812           837
6  2013     1     1     554           558          -4     740           728
7  2013     1     1     555           600          -5     913           854
8  2013     1     1     557           600          -3     709           723
9  2013     1     1     557           600          -3     838           846
10 2013     1     1     558           600          -2     753           745
# i 336,766 more rows
# i 13 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>, distance_km <dbl>,
#   vitesse <dbl>
```

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

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

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

```
# A tibble: 12 x 21
# Groups:   month [12]
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
1  2013     1     1     517           515           2     830           819
2  2013     2     1     456           500          -4     652           648
3  2013     3     1         4          2159        125     318           56
4  2013     4     1     454           500          -6     636           640
5  2013     5     1         9          1655        434     308          2020
6  2013     6     1         2          2359         3     341           350
```

```

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

```

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

```

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

```

```

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

```

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

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

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

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

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

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

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

```
# A tibble: 336,776 x 21
# Groups:   month [12]
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
```



```

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

```

```

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

```

```

# A tibble: 336,776 x 21
# Groups:   month [12]
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>   <int>         <int>      <dbl>      <int>         <int>
1  2013     1     9     641           900      1301      1242           1530
2  2013     1    10    1121          1635      1126      1239           1810
3  2013     1     1     848          1835       853      1001           1950
4  2013     1    13    1809           810       599      2054           1042
5  2013     1    16    1622           800       502      1911           1054
6  2013     1    23    1551           753       478      1812           1006
7  2013     1    10    1525           900       385      1713           1039
8  2013     1     1    2343          1724       379       314           1938
9  2013     1     2    2131          1512       379      2340           1741
10 2013     1     7    2021          1415       366      2332           1724
# i 336,766 more rows
# i 13 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dtm>, distance_km <dbl>,
#   vitesse <dbl>

```

8.3.2 summarise()

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

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

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

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

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

```
# A tibble: 12 x 4
  month max_delay min_delay mean_delay
  <int>   <dbl>    <dbl>    <dbl>
1     1    1301     -30     10.0
2     2     853     -33     10.8
3     3     911     -25     13.2
4     4     960     -21     13.9
5     5     878     -24     13.0
6     6    1137     -21     20.8
```

7	7	1005	-22	21.7
8	8	520	-26	12.6
9	9	1014	-24	6.72
10	10	702	-25	6.24
11	11	798	-32	5.44
12	12	896	-43	16.6

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

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

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

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

8.3.3 count()

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

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

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

8.3.4 Grouper selon plusieurs variables

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

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

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

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

```
10      7 LAX      1500
# i 1,103 more rows
```

On peut également compter selon plusieurs variables :

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

```
# A tibble: 224 x 3
  origin dest      n
  <chr>  <chr> <int>
1 JFK    LAX    11262
2 LGA    ATL    10263
3 LGA    ORD     8857
4 JFK    SFO     8204
5 LGA    CLT     6168
6 EWR    ORD     6100
7 JFK    BOS     5898
8 LGA    MIA     5781
9 JFK    MCO     5464
10 EWR    BOS     5327
# i 214 more rows
```

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

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

Au final, on obtient le code suivant :

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

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

```
# A tibble: 12 x 4
# Groups:   month [12]
  month origin dest    nb
  <int> <chr>  <chr> <int>
1     1   JFK    LAX    937
2     2   JFK    LAX    834
3     3   JFK    LAX    960
4     4   JFK    LAX    935
5     5   JFK    LAX    960
6     6   JFK    LAX    928
7     7   JFK    LAX    985
8     8   JFK    LAX    979
9     9   JFK    LAX    925
10    10   JFK    LAX    965
11    11   JFK    LAX    907
12    12   JFK    LAX    947
```

Lorsqu'on effectue un `dplyr::group_by()` suivi d'un `dplyr::summarise()`, le tableau résultat est automatiquement dégroupé *de la dernière variable de regroupement*. Ainsi le tableau généré par le code suivant est groupé par *month* et *origin*⁶ :

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

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

```
# A tibble: 2,313 x 4
# Groups:   month, origin [36]
  month origin dest    nb
  <int> <chr>  <chr> <int>
1     1   EWR    ALB     64
2     1   EWR    ATL   362
3     1   EWR    AUS     51
4     1   EWR    AVL      2
5     1   EWR    BDL     37
6     1   EWR    BNA    111
7     1   EWR    BOS   430
```

⁶Comme expliqué dans le message affiché dans la console, cela peut être contrôlé avec l'argument `.groups` de `dplyr::summarise()`, dont les options sont décrites dans l'aide de la fonction.

```

8      1 EWR    BQN      31
9      1 EWR    BTV     100
10     1 EWR    BUF     119
# i 2,303 more rows

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

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

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

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


8.4 Cheatsheet



8.5 webin-R

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

<https://youtu.be/aFvBhgmawcs>

9 Facteurs et forcats

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

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

Note

Lorsque l'on importe des données d'enquêtes, il est fréquent que les variables catégorielles sont codées sous la forme d'un code numérique (par exemple 1 pour *femme* et 2 pour *homme*) auquel est associé une *étiquette de valeur*. C'est notamment le fonctionnement usuel de logiciels tels que **SPSS**, **Stata** ou **SAS**. Les étiquettes de valeurs seront abordés dans un prochain chapitre (voir Chapitre 12).

Au moment de l'analyse (tableaux statistiques, graphiques, modèles de régression...), il sera nécessaire de transformer ces vecteurs avec étiquettes en facteurs.

9.1 Création d'un facteur

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Techniquement, les valeurs d'un facteur sont stockés de manière interne à l'aide de nombres entiers, dont la valeur représente la position de l'étiquette correspondante dans l'attribut `levels`. Ainsi, un facteur à `n` modalités sera toujours codé avec les nombre entiers allant de 1 à `n`.

```
class(f)
```

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

```
typeof(f)
```

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

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

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

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

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

9.2 Changer l'ordre des modalités

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
fct_inorder(v)
```

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

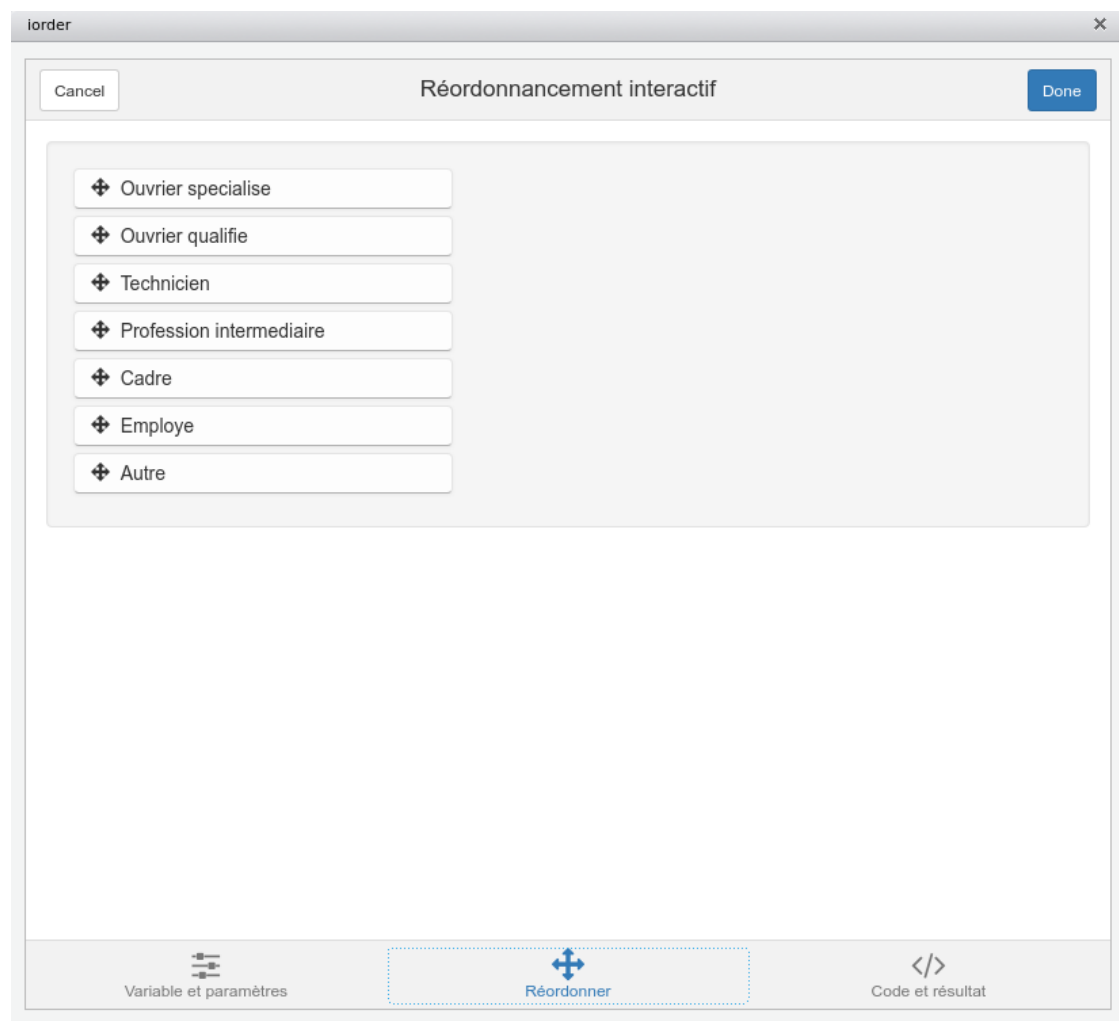
La fonction `forcats::fct_reorder()` permet de trier les modalités en fonction d'une autre variable. Par exemple, si je souhaite trier les modalités de la variable *qualif* en fonction de l'âge moyen (dans chaque modalité) :

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

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

Astuce

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



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

9.3 Modifier les modalités

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

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

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

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

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

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

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

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

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

```

    "supérieur" = "Enseignement superieur y compris technique superieur"
  )
hdv2003$instruction |>
  questionr::freq()

```

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

💡 Interface graphique

Le package `{questionr}` propose une interface graphique facilitant le recodage des modalités d'une variable qualitative. L'objectif est de permettre à la personne qui l'utilise de saisir les nouvelles valeurs dans un formulaire, et de générer ensuite le code R correspondant au recodage indiqué.

Pour utiliser cette interface, sous **RStudio** vous pouvez aller dans le menu *Addins* (présent dans la barre d'outils principale) puis choisir *Levels recoding*. Sinon, vous pouvez lancer dans la console la fonction `questionr::irec()` en lui passant comme paramètre la variable à recoder.

Cancel
Recodage interactif
Done

Ouvrier specialise →	<input type="text" value="Ouvrier specialise"/>
Ouvrier qualifie →	<input type="text" value="Ouvrier qualifie"/>
Technicien →	<input type="text" value="Technicien"/>
Profession intermediaire →	<input type="text" value="Profession intermediaire"/>
Cadre →	<input type="text" value="Cadre"/>
Employe →	<input type="text" value="Employe"/>
Autre →	<input type="text" value="Autre"/>
NA →	<input type="text" value="NA"/>

Variable et paramètres
Recodage
Code et résultat

💡 Astuce

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

<https://youtu.be/CokvTbtWdwc>

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

```
hdv2003$instruction <-
  hdv2003$nivetud |>
  fct_collapse(
```

```

"primaire" = c(
  "N'a jamais fait d'etudes",
  "A arrete ses etudes, avant la derniere annee d'etudes primaires",
  "Derniere annee d'etudes primaires"
),
"secondaire" = c(
  "1er cycle",
  "2eme cycle"
),
"technique/professionnel" = c(
  "Enseignement technique ou professionnel court",
  "Enseignement technique ou professionnel long"
),
"supérieur" = "Enseignement superieur y compris technique superieur"
)

```

Pour transformer les valeurs manquantes (NA) en une modalité explicite, on pourra avoir recours à `forcats::fct_na_value_to_level()`¹.

```

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

```

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

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

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

```

hdv2003$qualif |>
  questionr::freq()

```

¹Cette fonction s'appelait précédemment `forcats::fct_explicit_na()` et a été renommée depuis la version 1.0.0 de `{forcats}`.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
v
```

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

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

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

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

```
v
```

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

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

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

9.4 Découper une variable numérique en classes

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

On utilise pour cela la fonction `cut()` qui prend, outre la variable à découper, un certain nombre d'arguments :

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

Prenons tout de suite un exemple et tentons de découper la variable *age* en cinq classes :

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

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

Par défaut **R** nous a bien créé cinq classes d'amplitudes égales. La première classe va de 17,9 à 33,8 ans (en fait de 17 à 32), etc.

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

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

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

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

On remarque que du coup, dans notre exemple précédent, la valeur minimale, 18, est exclue de notre première classe, et qu'une observation est donc absente de ce découpage. Pour résoudre ce problème on peut soit faire commencer la première classe à 17, soit utiliser l'option `include.lowest=TRUE` :

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

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

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

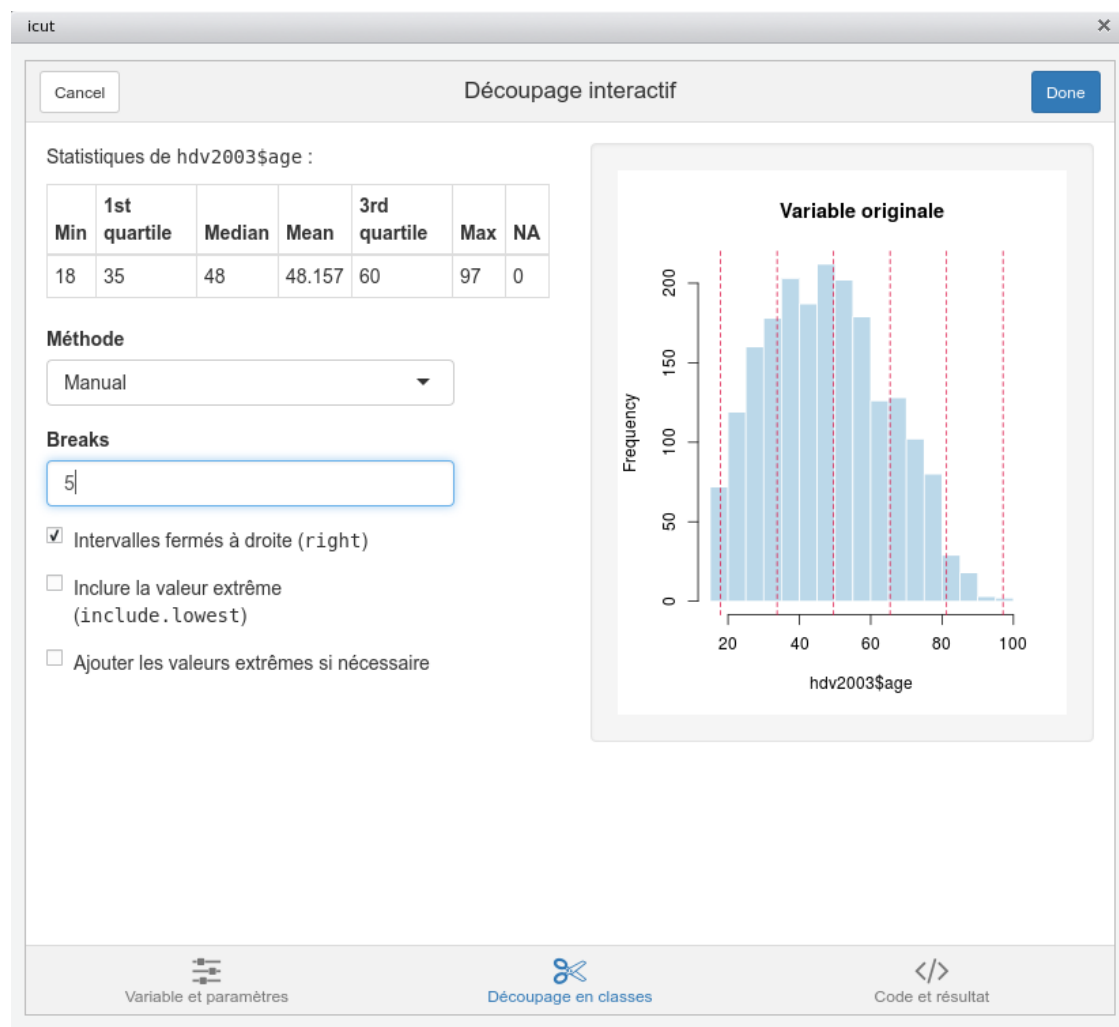
```
hdv2003 <-  
  hdv2003 |>  
  mutate(groupe_ages = cut(  
    age,  
    c(18, 20, 40, 60, 80, 97),  
    include.lowest = TRUE,  
    right = FALSE  
  ))  
hdv2003$groupe_ages |> questionr::freq()
```

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

💡 Interface graphique

Il n'est pas nécessaire de connaître toutes les options de `cut()`. Le package `{questionr}` propose là encore une interface graphique permettant de visualiser l'effet des différents paramètres et de générer le code **R** correspondant.

Pour utiliser cette interface, sous **RStudio** vous pouvez aller dans le menu *Addins* (présent dans la barre d'outils principale) puis choisir *Numeric range dividing*. Sinon, vous pouvez lancer dans la console la fonction `questionr::icut()` en lui passant comme paramètre la variable numérique à découper.



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

10 Combiner plusieurs variables

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

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

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

10.1 if_else()

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

Voici un exemple simple :

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

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

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

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

```
)
hdv2003 |>
  pull(statut) |>
  questionr::freq()
```

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

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

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

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

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

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

! `if_else()` et `ifelse()`

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

- `dplyr::if_else()` vérifie que les valeurs fournies pour `true` et celles pour `false` sont du même type et de la même classe et renvoie une erreur dans le cas contraire, là où `ifelse()` sera plus permissif ;
- si un vecteur a des attributs (cf. Chapitre 6), ils seront préservés par `dplyr::if_else()` (et pris dans le vecteur `true`), ce que ne fera pas `ifelse()` ;
- `dplyr::if_else()` propose un argument optionnel supplémentaire `missing` pour indiquer les valeurs à retourner lorsque le test renvoie `NA`.

10.2 `case_when()`

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

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

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

```
pull(statut) |>
questionr::freq()
```

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

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

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

! Important

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

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

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

Comme la condition `sexe == "Homme"` est plus générale que `sexe == "Homme" & age > 60`, cette deuxième condition n'est jamais testée ! On n'obtiendra jamais la valeur correspondante.

Pour que ce recodage fonctionne il faut donc changer l'ordre des conditions pour aller du plus spécifique au plus général :

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

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

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

10.3 recode_if()

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

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

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



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

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

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

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

```
df$pref[df$autre_details == "bleu ciel"] <- "bleu"
```

Avec `dplyr::if_else()`, on serait tenté d'écrire :

```
df |>
  mutate(pref = if_else(autre_details == "bleu ciel", "bleu", pref))
```

```
# A tibble: 5 x 2
  pref  autre_details
<chr> <chr>
1 <NA> <NA>
2 <NA> <NA>
3 bleu bleu ciel
4 <NA> <NA>
5 autre jaune
```

On obtient une erreur, car `dplyr::if_else()` exige les valeurs fournies pour `true` et `false` soient de même type. Essayons alors :

```
df |>
  mutate(pref = if_else(autre_details == "bleu ciel", factor("bleu"), pref))
```

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

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

Ici nous avons un autre problème, signalé par un message d'avertissement (*warning*) : `dplyr::if_else()` ne préserve que les attributs du vecteur passé en `true` et non ceux passés à `false`. Or l'ensemble des modalités (niveaux du facteur) de la variable *pref* n'ont pas été définis dans `factor("bleu")` et sont ainsi perdus, générant une perte de données (valeurs manquantes NA).

Pour obtenir le bon résultat, il faudrait inverser la condition :

```
df |>
  mutate(pref = if_else(
    autre_details != "bleu ciel",
    pref,
    factor("bleu")
  ))
```

```
# A tibble: 5 x 2
  pref  autre_details
<fct> <chr>
1 <NA>  <NA>
2 <NA>  <NA>
3 bleu  bleu ciel
4 <NA>  <NA>
5 autre jaune
```

Mais ce n'est toujours pas suffisant. En effet, la variable *autre_details* a des valeurs manquantes pour lesquelles le test `autre_details != "bleu ciel"` renvoie NA ce qui une fois encore génère des valeurs manquantes non souhaitées. Dès lors, il nous faut soit définir l'argument `missing` de `dplyr::if_else()`, soit être plus précis dans notre test.

```
df |>
  mutate(pref = if_else(
    autre_details != "bleu ciel",
    pref,
    factor("bleu"),
    missing = pref
  ))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

```
df |>
  mutate(pref = if_else(
    autre_details != "bleu ciel" | is.na(autre_details),
    pref,
    factor("bleu")
  ))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

Bref, on peut s'en sortir avec `dplyr::if_else()` mais ce n'est pas forcément le plus pratique dans le cas présent. La syntaxe en base **R** fonctionne très bien, mais ne peut pas être intégrée à un enchaînement d'opérations utilisant le *pipe*.

Dans ce genre de situation, on pourra être intéressé par la fonction `labelled::recode_if()` disponible dans le package `{labelled}`. Elle permet de ne modifier que certaines observations d'un vecteur en fonction d'une condition. Si la condition vaut `FALSE` ou `NA`, les observations concernées restent inchangées. Voyons comment cela s'écrit :

```
df <-
df |>
  mutate(
    pref = pref |>
      labelled::recode_if(autre_details == "bleu ciel", "bleu")
  )
df
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

C'est tout de suite plus intuitif !

11 Étiquettes de variables

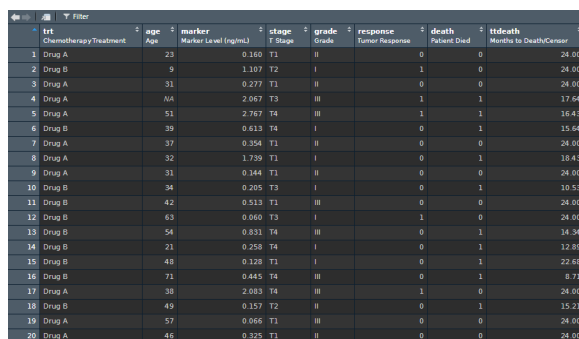
11.1 Principe

Les étiquettes de variable permettent de donner un nom long, plus explicite, aux différentes colonnes d'un tableau de données (ou encore directement à un vecteur autonome). Dans le champs des grandes enquêtes, il est fréquent de nommer les variables *q101*, *q102*, etc. pour refléter le numéro de la question et d'indiquer ce qu'elle représente (groupe d'âges, milieu de résidence...) avec une étiquette.

Un usage, introduit par le package `{haven}`, et repris depuis par de nombreux autres packages dont `{gtsummary}` que nous aborderons dans de prochains chapitres, consiste à stocker les étiquettes de variables sous la forme d'un attribut¹ `"label"` attaché au vecteur / à la colonne du tableau.

Le package `{labelled}` permet de manipuler aisément ces étiquettes de variables.

La visionneuse de données de **RStudio** sait reconnaître et afficher ces étiquettes de variable lorsqu'elles existent. Prenons pour exemple le jeu de données `gtsummary::trial` dont les colonnes ont des étiquettes de variable. La commande `View(gtsummary::trial)` permet d'ouvrir la visionneuse de données de **RStudio**. Comme on peut le constater, une étiquette de variable est bien présente sous le nom des différentes colonnes.



	treatment (Chemotherapy Treatment)	age (Age)	marker (Marker Level (ng/mL))	stage (T Stage)	grade (Grade)	response (Response, Tumor Response)	death (Patient Died)	tdeath (Months to Death/Censor)
1	Drug A	23	0.160	T1	II	0	0	24.00
2	Drug B	9	1.107	T2	I	1	0	24.00
3	Drug A	31	0.277	T1	II	0	0	24.00
4	Drug A	NA	2.667	T3	III	1	1	17.64
5	Drug A	51	2.967	T4	III	1	1	16.43
6	Drug B	39	0.613	T4	I	0	1	15.64
7	Drug A	37	0.354	T1	II	0	0	24.00
8	Drug A	32	1.739	T1	I	0	1	18.43
9	Drug A	31	0.144	T1	II	0	0	24.00
10	Drug B	34	0.205	T3	I	0	1	10.53
11	Drug B	42	0.513	T1	III	0	0	24.00
12	Drug B	63	0.060	T3	I	1	0	24.00
13	Drug B	54	0.831	T4	III	0	1	14.34
14	Drug B	21	0.258	T4	I	0	1	12.89
15	Drug B	48	0.128	T1	I	0	1	22.68
16	Drug B	71	0.445	T4	III	0	1	8.71
17	Drug A	38	2.083	T4	III	1	0	24.00
18	Drug B	49	0.157	T2	II	0	1	15.21
19	Drug A	57	0.866	T1	III	0	0	24.00
20	Drug A	44	0.935	T1	II	0	0	24.00

Figure 11.1: Présentation du tableau `gtsummary::trial` dans la visionneuse de **RStudio**

La fonction `labelled::look_for()` du package `{labelled}` permet de lister l'ensemble des variables d'un tableau de données et affiche notamment les étiquettes de variable associées.

¹Pour plus d'information sur les attributs, voir Chapitre 6.

```
library(labelled)
gtsummary::trial |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	trt	Chemotherapy Treatment	chr	0	
2	age	Age	dbl	11	
3	marker	Marker Level (ng/mL)	dbl	10	
4	stage	T Stage	fct	0	T1 T2 T3 T4
5	grade	Grade	fct	0	I II III
6	response	Tumor Response	int	7	
7	death	Patient Died	int	0	
8	ttdeath	Months to Death/Censor	dbl	0	

La fonction `labelled::look_for()` permet également de rechercher des variables en tenant compte à la fois de leur nom et de leur étiquette.

```
gtsummary::trial |>
  look_for("months")
```

pos	variable	label	col_type	missing	values
8	ttdeath	Months to Death/Censor	dbl	0	

Astuce

Comme on le voit, la fonction `labelled::look_for()` est tout à fait adaptée pour générer un dictionnaire de codification. Ses différentes options sont détaillées dans une [vignette dédiée](#). Les résultats renvoyés par `labelled::look_for()` sont récupérables dans un tableau de données que l'on pourra ainsi manipuler à sa guise.

```
gtsummary::trial |>
  look_for() |>
  dplyr::as_tibble()
```

```
# A tibble: 8 x 7
  pos variable label col_type missing levels value_labels
```

	<int>	<chr>	<chr>	<chr>	<int>	<named li>	<named list>
1	1	trt	Chemotherapy Treatment	chr	0	<NULL>	<NULL>
2	2	age	Age	dbl	11	<NULL>	<NULL>
3	3	marker	Marker Level (ng/mL)	dbl	10	<NULL>	<NULL>
4	4	stage	T Stage	fct	0	<chr [4]>	<NULL>
5	5	grade	Grade	fct	0	<chr [3]>	<NULL>
6	6	response	Tumor Response	int	7	<NULL>	<NULL>
7	7	death	Patient Died	int	0	<NULL>	<NULL>
8	8	ttdeath	Months to Death/Censor	dbl	0	<NULL>	<NULL>

11.2 Manipulation sur un vecteur / une colonne

La fonction `labelled::var_label()` permet de voir l'étiquette de variable attachée à un vecteur (renvoie `NULL` s'il n'y en a pas) mais également d'ajouter/modifier une étiquette.

Le fait d'ajouter une étiquette de variable à un vecteur ne modifie en rien son type ni sa classe. On peut associer une étiquette de variable à n'importe quel type de variable, qu'elle soit numérique, textuelle, un facteur ou encore des dates.

```
v <- c(1, 5, 2, 4, 1)
v |> var_label()
```

`NULL`

```
var_label(v) <- "Mon étiquette"
var_label(v)
```

`[1] "Mon étiquette"`

```
str(v)
```

```
num [1:5] 1 5 2 4 1
- attr(*, "label")= chr "Mon étiquette"
```

```
var_label(v) <- "Une autre étiquette"
var_label(v)
```

`[1] "Une autre étiquette"`

```
str(v)
```

```
num [1:5] 1 5 2 4 1
- attr(*, "label")= chr "Une autre étiquette"
```

Pour supprimer une étiquette, il suffit d'attribuer la valeur NULL.

```
var_label(v) <- NULL
str(v)
```

```
num [1:5] 1 5 2 4 1
```

On peut appliquer `labelled::var_label()` directement sur une colonne de tableau.

```
var_label(iris$Petal.Length) <- "Longueur du pétale"
var_label(iris$Petal.Width) <- "Largeur du pétale"
var_label(iris$Species) <- "Espèce"
iris |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	Sepal.Length	-	dbl	0	
2	Sepal.Width	-	dbl	0	
3	Petal.Length	Longueur du pétale	dbl	0	
4	Petal.Width	Largeur du pétale	dbl	0	
5	Species	Espèce	fct	0	setosa versicolor virginica

11.3 Manipulation sur un tableau de données

La fonction `labelled::set_variable_labels()` permet de manipuler les étiquettes de variable d'un tableau de données avec une syntaxe du type `{dplyr}`.


```
iris <-
  iris |>
  set_variable_labels(
    Species = NULL,
    Sepal.Length = "Longeur du sépale"
  )
iris |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	Sepal.Length	Longeur du sépale	dbl	0	
2	Sepal.Width	-	dbl	0	
3	Petal.Length	Longueur du pétale	dbl	0	
4	Petal.Width	Largeur du pétale	dbl	0	
5	Species	-	fct	0	setosa versicolor virginica

11.4 Préserver les étiquettes

Certaines fonctions de **R** ne préservent pas les attributs et risquent donc d'effacer les étiquettes de variables que l'on a définies. Un exemple est la fonction générique `subset()` qui permet de sélectionner certaines lignes remplissant une certaines conditions.

```
iris |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	Sepal.Length	Longeur du sépale	dbl	0	
2	Sepal.Width	-	dbl	0	
3	Petal.Length	Longueur du pétale	dbl	0	
4	Petal.Width	Largeur du pétale	dbl	0	
5	Species	-	fct	0	setosa versicolor virginica

```
iris |>
  subset(Species == "setosa") |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	Sepal.Length	-	dbl	0	
2	Sepal.Width	-	dbl	0	
3	Petal.Length	-	dbl	0	
4	Petal.Width	-	dbl	0	
5	Species	-	fct	0	setosa versicolor virginica

On pourra, dans ce cas précis, préférer la fonction `dplyr::filter()` qui préserve les attributs et donc les étiquettes de variables.

```
iris |>
  dplyr::filter(Species == "setosa") |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	Sepal.Length	Longueur du sépale	dbl	0	
2	Sepal.Width	-	dbl	0	
3	Petal.Length	Longueur du pétale	dbl	0	
4	Petal.Width	Largeur du pétale	dbl	0	
5	Species	-	fct	0	setosa versicolor virginica

On pourra également tirer parti de la fonction `labelled::copy_labels_from()` qui permet de copier les étiquettes d'un tableau à un autre.

```
iris |>
  subset(Species == "setosa") |>
  copy_labels_from(iris) |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	Sepal.Length	Longueur du sépale	dbl	0	
2	Sepal.Width	-	dbl	0	
3	Petal.Length	Longueur du pétale	dbl	0	
4	Petal.Width	Largeur du pétale	dbl	0	
5	Species	-	fct	0	setosa versicolor virginica

12 Étiquettes de valeurs

Dans le domaine des grandes enquêtes, il est fréquent de coder les variables catégorielles avec des codes numériques auxquels on associe certaines valeurs. Par exemple, une variable *milieu de résidence* pourrait être codée 1 pour urbain, 2 pour semi-urbain, 3 pour rural et 9 pour indiquer une donnée manquante. Une variable binaire pourrait quant à elle être codée 0 pour non et 1 pour oui. Souvent, chaque enquête définit ses propres conventions.

Les logiciels statistiques propriétaires **SPSS**, **Stata** et **SAS** ont tous les trois un système d'étiquettes de valeurs pour représenter ce type de variables catégorielles.

R n'a pas, de manière native, de système d'étiquettes de valeurs. Le format utilisé en interne pour représenter les variables catégorielles est celui des facteurs (cf. Chapitre 9). Cependant, ce dernier ne permet de contrôler comment sont associées une étiquette avec une valeur numérique précise.

12.1 La classe `haven_labelled`

Afin d'assurer une importation complète des données depuis **SPSS**, **Stata** et **SAS**, le package `{haven}` a introduit un nouveau type de vecteurs, la classe `haven_labelled`, qui permet justement de rendre compte de ces vecteurs labellisés (i.e. avec des étiquettes de valeurs). Le package `{labelled}` fournit un jeu de fonctions pour faciliter la manipulation des vecteurs labellisés.

! Important

Les vecteurs labellisés sont un format intermédiaire qui permet d'importer les données telles qu'elles ont été définies dans le fichier source. Il n'est pas destiné à être utilisé pour l'analyse statistique.

Pour la réalisation de tableaux, graphiques, modèles, **R** attend que les variables catégorielles soit codées sous formes de facteurs, et que les variables continues soient numériques. On aura donc besoin, à un moment ou à un autre, de convertir les vecteurs labellisés en facteurs ou en variables numériques classiques.

12.2 Manipulation sur un vecteur / une colonne

Pour définir des étiquettes, la fonction de base est `labelled::val_labels()`. Il est possible de définir des étiquettes de valeurs pour des vecteurs numériques, d'entiers et textuels. On indiquera les étiquettes sous la forme `étiquette = valeur`. Cette fonction s'utilise de la même manière que `labelled::var_label()` abordée au chapitre précédent (cf. Chapitre 11). Un appel simple renvoie les étiquettes de valeur associées au vecteur, NULL s'il n'y en n'a pas. Combiner avec l'opérateur d'assignation (`<-`), on peut ajouter/modifier les étiquettes de valeurs associées au vecteur.

```
library(labelled)
v <- c(1, 2, 1, 9)
v
```

```
[1] 1 2 1 9
```

```
class(v)
```

```
[1] "numeric"
```

```
val_labels(v)
```

```
NULL
```

```
val_labels(v) <- c(non = 1, oui = 2)
val_labels(v)
```

```
non oui
  1   2
```

```
v
```

```
<labelled<double>[4]>
[1] 1 2 1 9
```

```
Labels:
  value label
    1    non
    2    oui
```

```
class(v)
```

```
[1] "haven_labelled" "vctrs_vctr"      "double"
```

Comme on peut le voir avec cet exemple simple :

- l'ajout d'étiquettes de valeurs modifie la classe de l'objet (qui est maintenant un vecteur de la classe `haven_labelled`) ;
- l'objet obtenu est multi-classes, la classe `double` indiquant ici qu'il s'agit d'un vecteur numérique ;
- il n'est pas obligatoire d'associer une étiquette de valeurs à toutes les valeurs observées dans le vecteur (ici, nous n'avons pas défini d'étiquettes pour la valeur 9).

La fonction `labelled::val_label()` (notez l'absence d'un `s` à la fin du nom de la fonction) permet d'accéder / de modifier l'étiquette associée à une valeur spécifique.

```
val_label(v, 1)
```

```
[1] "non"
```

```
val_label(v, 9)
```

```
NULL
```

```
val_label(v, 9) <- "(manquant)"
val_label(v, 2) <- NULL
v
```

```
<labelled<double>[4]>
```

```
[1] 1 2 1 9
```

```
Labels:
```

value	label
1	non
9	(manquant)

Pour supprimer, toutes les étiquettes de valeurs, on attribuera `NULL` avec `labelled::val_labels()`.

```
val_labels(v) <- NULL
v
```

```
[1] 1 2 1 9
```

```
class(v)
```

```
[1] "numeric"
```

On remarquera que, lorsque toutes les étiquettes de valeurs sont supprimées, la nature de l'objet change à nouveau et il redevient un simple vecteur numérique.

Mise en garde

Il est essentiel de bien comprendre que l'ajout d'étiquettes de valeurs ne change pas fondamentalement la nature du vecteur. **Cela ne le transforme pas en variable catégorielle.** À ce stade, le vecteur n'a pas été transformé en facteur. Cela reste un vecteur numérique qui est considéré comme tel par **R**. On peut ainsi en calculer une moyenne, ce qui serait impossible avec un facteur.

```
v <- c(1, 2, 1, 2)
val_labels(v) <- c(non = 1, oui = 2)
mean(v)
```

```
[1] 1.5
```

```
f <- factor(v, levels = c(1, 2), labels = c("non", "oui"))
mean(f)
```

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

```
[1] NA
```

Les fonctions `labelled::val_labels()` et `labelled::val_label()` peuvent également être utilisées sur les colonnes d'un tableau.

```
df <- dplyr::tibble(
  x = c(1, 2, 1, 2),
  y = c(3, 9, 9, 3)
```

```
)
val_labels(df$x) <- c(non = 1, oui = 2)
val_label(df$y, 9) <- "(manquant)"
df
```

```
# A tibble: 4 x 2
  x     y
  <dbl> <lbl>
1 1 [non] 3
2 2 [oui] 9 [(manquant)]
3 1 [non] 9 [(manquant)]
4 2 [oui] 3
```

On pourra noter, que si notre tableau est un *tibble*, les étiquettes sont rendues dans la console quand on affiche le tableau.

La fonction `labelled::look_for()` est également un bon moyen d'afficher les étiquettes de valeurs.

```
df |>
  look_for()
```

```
pos variable label col_type missing values
1    x         -    dbl+lbl  0          [1] non
                                [2] oui
2    y         -    dbl+lbl  0          [9] (manquant)
```

12.3 Manipulation sur un tableau de données

{labelled} fournit 3 fonctions directement applicables sur un tableau de données : `labelled::set_value_labels()`, `labelled::add_value_labels()` et `labelled::remove_value_labels()`. La première remplace l'ensemble des étiquettes de valeurs associées à une variable, la seconde ajoute des étiquettes de valeurs (et conserve celles déjà définies), la troisième supprime les étiquettes associées à certaines valeurs spécifiques (et laisse les autres inchangées).

```
df |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	x	-	dbl+lbl	0	[1] non [2] oui
2	y	-	dbl+lbl	0	[9] (manquant)

```
df <- df |>
  set_value_labels(
    x = c(yes = 2),
    y = c("a répondu" = 3, "refus de répondre" = 9)
  )
df |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	x	-	dbl+lbl	0	[2] yes
2	y	-	dbl+lbl	0	[3] a répondu [9] refus de répondre

```
df <- df |>
  add_value_labels(
    x = c(no = 1)
  ) |>
  remove_value_labels(
    y = 9
  )
df |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	x	-	dbl+lbl	0	[2] yes [1] no
2	y	-	dbl+lbl	0	[3] a répondu

12.4 Conversion

12.4.1 Quand convertir les vecteurs labellisés ?

La classe `haven_labelled` permet d'ajouter des métadonnées aux variables sous la forme d'étiquettes de valeurs. Lorsque les données sont importées depuis **SAS**, **SPSS** ou **Stata**, cela permet notamment de conserver le codage original du fichier importé.

Mais il faut noter que ces *étiquettes de valeur* n'indiquent pas pour autant de manière systématique le type de variable (catégorielle ou continue). Les vecteurs labellisés n'ont donc pas vocation à être utilisés pour l'analyse, notamment le calcul de modèles statistiques. Ils doivent être convertis en facteurs (pour les variables catégorielles) ou en vecteurs numériques (pour les variables continues).

La question qui peut se poser est donc de choisir à quel moment cette conversion doit avoir lieu dans un processus d'analyse. On peut considérer deux approches principales.

Approche A



Approche B

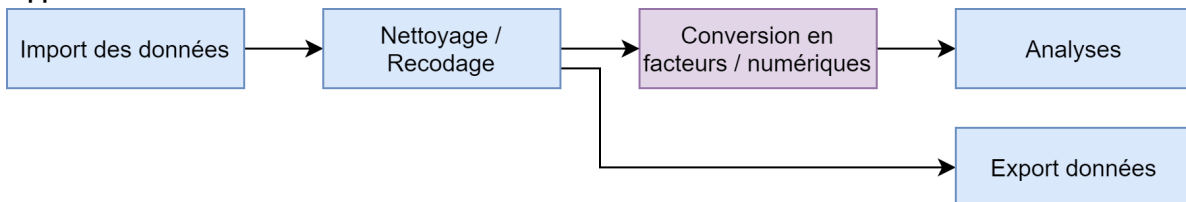


Figure 12.1: Deux approches possibles pour la conversion des étiquettes de valeurs

Dans l'**approche A**, les vecteurs labellisés sont convertis juste après l'import des données, en utilisant les fonctions `labelled::unlabelled()`, `labelled::to_factor()` ou `base::unclass()` qui sont présentées ci-après. Dès lors, toute la partie de nettoyage et de recodage des données se fera en utilisant les fonctions classiques de **R**. Si l'on n'a pas besoin de conserver le codage original, cette approche a l'avantage de s'inscrire dans le fonctionnement usuel de **R**.

Dans l'**approche B**, les vecteurs labellisés sont conservés pour l'étape de nettoyage et de recodage des données. Dans ce cas là, on pourra avoir recours aux fonctions de l'extension `{labelled}` qui facilitent la gestion des données labellisées. Cette approche est particulièrement intéressante quand (i) on veut pouvoir se référer au dictionnaire de codification fourni avec les données sources et donc on veut conserver le codage original et/ou (ii) quand les données devront faire l'objet d'un ré-export après transformation. Par contre, comme dans l'approche A, il faudra prévoir une conversion des variables labellisées au moment de l'analyse.

⚠ Avertissement

Dans tous les cas, il est recommandé d'adopter l'une ou l'autre approche, mais d'éviter de mélanger les différents types de vecteur. Une organisation rigoureuse de ses données et de son code est essentielle !

12.4.2 Convertir un vecteur labellisé en facteur

Il est très facile de convertir un vecteur labellisé en facteur à l'aide la fonction `labelled::to_factor()` du package `{labelled}`¹.

```
v <- c(1,2,9,3,3,2,NA)
val_labels(v) <- c(
  oui = 1, "peut-être" = 2,
  non = 3, "ne sait pas" = 9
)
v
```

```
<labelled<double>[7]>
[1] 1 2 9 3 3 2 NA
```

```
Labels:
  value      label
    1         oui
    2    peut-être
    3         non
    9 ne sait pas
```

```
to_factor(v)
```

```
[1] oui          peut-être   ne sait pas non          non          peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

Il possible d'indiquer si l'on souhaite, comme étiquettes du facteur, utiliser les étiquettes de valeur (par défaut), les valeurs elles-mêmes, ou bien les étiquettes de valeurs préfixées par la valeur d'origine indiquée entre crochets.

```
to_factor(v, 'l')
```

```
[1] oui          peut-être   ne sait pas non          non          peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

¹On privilégiera la fonction `labelled::to_factor()` à la fonction `haven::as_factor()` de l'extension `{haven}`, la première ayant plus de possibilités et un comportement plus consistant.

```
to_factor(v, 'v')
```

```
[1] 1 2 9 3 3 2 <NA>  
Levels: 1 2 3 9
```

```
to_factor(v, 'p')
```

```
[1] [1] oui [2] peut-être [9] ne sait pas [3] non  
[5] [3] non [2] peut-être <NA>  
Levels: [1] oui [2] peut-être [3] non [9] ne sait pas
```

Par défaut, les modalités du facteur seront triées selon l'ordre des étiquettes de valeur. Mais cela peut être modifié avec l'argument `sort_levels` si l'on préfère trier selon les valeurs ou selon l'ordre alphabétique des étiquettes.

```
to_factor(v, sort_levels = 'v')
```

```
[1] oui peut-être ne sait pas non non peut-être  
[7] <NA>  
Levels: oui peut-être non ne sait pas
```

```
to_factor(v, sort_levels = 'l')
```

```
[1] oui peut-être ne sait pas non non peut-être  
[7] <NA>  
Levels: ne sait pas non oui peut-être
```

12.4.3 Convertir un vecteur labellisé en numérique ou en texte

Pour rappel, il existe deux types de vecteurs labellisés : des vecteurs numériques labellisés (`x` dans l'exemple ci-dessous) et des vecteurs textuels labellisés (`y` dans l'exemple ci-dessous).

```
x <- c(1, 2, 9, 3, 3, 2, NA)  
val_labels(x) <- c(  
  oui = 1, "peut-être" = 2,  
  non = 3, "ne sait pas" = 9  
)  
  
y <- c("f", "f", "h", "f")  
val_labels(y) <- c(femme = "f", homme = "h")
```

Pour leur retirer leur caractère labellisé et revenir à leur classe d'origine, on peut utiliser la fonction `unclass()`.

```
unclass(x)
```

```
[1] 1 2 9 3 3 2 NA
attr("labels")
      oui    peut-être      non ne sait pas
      1         2         3         9
```

```
unclass(y)
```

```
[1] "f" "f" "h" "f"
attr("labels")
femme homme
  "f"    "h"
```

À noter que dans ce cas-là, les étiquettes sont conservées comme attributs du vecteur.

Une alternative est d'utiliser `labelled::remove_labels()` qui supprimera toutes les étiquettes, y compris les étiquettes de variable. Pour conserver les étiquettes de variables et ne supprimer que les étiquettes de valeurs, on indiquera `keep_var_label = TRUE`.

```
var_label(x) <- "Etiquette de variable"
remove_labels(x)
```

```
[1] 1 2 9 3 3 2 NA
```

```
remove_labels(x, keep_var_label = TRUE)
```

```
[1] 1 2 9 3 3 2 NA
attr("label")
[1] "Etiquette de variable"
```

```
remove_labels(y)
```

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

Dans le cas d'un vecteur numérique labellisé que l'on souhaiterait convertir en variable textuelle, on pourra utiliser `labelled::to_character()` à la place de `labelled::to_factor()` qui, comme sa grande sœur, utilisera les étiquettes de valeurs.

```
to_character(x)
```

```
[1] "oui"          "peut-être"    "ne sait pas" "non"          "non"
[6] "peut-être"    NA
attr(,"label")
[1] "Etiquette de variable"
```

12.4.4 Conversion conditionnelle en facteurs

Il n'est pas toujours possible de déterminer la nature d'une variable (continue ou catégorielle) juste à partir de la présence ou l'absence d'étiquettes de valeur. En effet, on peut utiliser des étiquettes de valeur dans le cadre d'une variable continue pour indiquer certaines valeurs spécifiques.

Une bonne pratique est de vérifier chaque variable incluse dans une analyse, une à une.

Cependant, une règle qui fonctionne dans 90% des cas est de convertir un vecteur labellisé en facteur si et seulement si toutes les valeurs observées dans le vecteur disposent d'une étiquette de valeur correspondante. C'est ce que propose la fonction `labelled::unlabelled()` qui peut même être appliqué à tout un tableau de données. Par défaut, elle fonctionne ainsi :

1. les variables non labellisées restent inchangées (variables *f* et *g* dans l'exemple ci-dessous);
2. si toutes les valeurs observées d'une variable labellisées ont une étiquette, elles sont converties en facteurs (variables *b* et *c*);
3. sinon, on leur applique `base::unclass()` (variables *a*, *d* et *e*).

```
df <- dplyr::tibble(
  a = c(1, 1, 2, 3),
  b = c(1, 1, 2, 3),
  c = c(1, 1, 2, 2),
  d = c("a", "a", "b", "c"),
  e = c(1, 9, 1, 2),
  f = 1:4,
  g = as.Date(c(
    "2020-01-01", "2020-02-01",
    "2020-03-01", "2020-04-01"
  ))
) |>
```

```

set_value_labels(
  a = c(No = 1, Yes = 2),
  b = c(No = 1, Yes = 2, DK = 3),
  c = c(No = 1, Yes = 2, DK = 3),
  d = c(No = "a", Yes = "b"),
  e = c(No = 1, Yes = 2)
)
df |> look_for()

```

pos	variable	label	col_type	missing	values
1	a	-	dbl+lbl	0	[1] No [2] Yes
2	b	-	dbl+lbl	0	[1] No [2] Yes [3] DK
3	c	-	dbl+lbl	0	[1] No [2] Yes [3] DK
4	d	-	chr+lbl	0	[a] No [b] Yes
5	e	-	dbl+lbl	0	[1] No [2] Yes
6	f	-	int	0	
7	g	-	date	0	

```

to_factor(df) |> look_for()

```

pos	variable	label	col_type	missing	values
1	a	-	fct	0	No Yes 3
2	b	-	fct	0	No Yes DK
3	c	-	fct	0	No Yes DK
4	d	-	fct	0	No Yes c
5	e	-	fct	0	No

					Yes
6	f	-	int	0	9
7	g	-	date	0	

```
unlabelled(df) |> look_for()
```

pos	variable	label	col_type	missing	values
1	a	-	dbl	0	
2	b	-	fct	0	No Yes DK
3	c	-	fct	0	No Yes DK
4	d	-	chr	0	
5	e	-	dbl	0	
6	f	-	int	0	
7	g	-	date	0	

On peut indiquer certaines options, par exemple `drop_unused_labels = TRUE` pour supprimer des facteurs créés les niveaux non observées dans les données (voir la variable *c*).

```
unlabelled(df, drop_unused_labels = TRUE) |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	a	-	dbl	0	
2	b	-	fct	0	No Yes DK
3	c	-	fct	0	No Yes
4	d	-	chr	0	
5	e	-	dbl	0	
6	f	-	int	0	
7	g	-	date	0	

```
unlabelled(df, levels = "prefixed") |>
  look_for()
```

pos	variable	label	col_type	missing	values
1	a	-	dbl	0	
2	b	-	fct	0	[1] No [2] Yes [3] DK
3	c	-	fct	0	[1] No [2] Yes [3] DK
4	d	-	chr	0	
5	e	-	dbl	0	
6	f	-	int	0	
7	g	-	date	0	

13 Valeurs manquantes

Dans **R** base, les valeurs manquantes sont indiquées par la valeurs logiques `NA` que l'on peut utiliser dans tous types de vecteurs.

Dans certains cas, par exemple dans la fonction `dplyr::if_else()` qui vérifie que les deux options sont du même type, on peut avoir besoin de spécifier une valeur manquante d'un certains types précis (numérique, entier, textuel...) ce que l'on peut faire avec les constantes `NA_real_`, `NA_integer_` ou encore `NA_character_`.

De base, il n'existe qu'un seul type de valeurs manquantes dans **R**. D'autres logiciels statistiques ont mis en place des systèmes pour distinguer plusieurs types de valeurs manquantes, ce qui peut avoir une importance dans le domaine des grandes enquêtes, par exemple pour distinguer des ne sait pas d'un refus de répondre ou d'un oubli enquêteur.

Ainsi, **Stata** et **SAS** ont un système de valeurs manquantes étiquetées ou *tagged NAs*, où les valeurs manquantes peuvent recevoir une étiquette (une lettre entre a et z). De son côté, **SPSS** permet d'indiquer, sous la forme de métadonnées, que certaines valeurs devraient être traitées comme des valeurs manquantes (par exemple que la valeur 8 correspond à des refus et que la valeur 9 correspond à des ne sait pas). Il s'agit alors de valeurs manquantes définies par l'utilisateur ou *user NAs*.

Dans tous les cas, il appartient à l'analyste de décider au cas par cas comment ces valeurs manquantes doivent être traitées. Dans le cadre d'une variable numérique, il est essentiel d'exclure ces valeurs manquantes pour le calcul de statistiques telles que la moyenne ou l'écart-type. Pour des variables catégorielles, les pourcentages peuvent être calculées sur l'ensemble de l'échantillon (les valeurs manquantes étant alors traitées comme des modalités à part entière) ou bien uniquement sur les réponses valides, en fonction du besoin de l'analyse et de ce que l'on cherche à montrer.

Afin d'éviter toute perte d'informations lors d'un import de données depuis **Stata**, **SAS** et **SPSS**, le package `{haven}` propose une implémentation sous **R** des *tagged NAs* et des *user NAs*. Le package `{labelled}` fournit quant à lui différentes fonctions pour les manipuler aisément.

```
library(labelled)
```

13.1 Valeurs manquantes étiquetées (*tagged NAs*)

13.1.1 Création et test

Les *tagged NAs* sont de véritables valeurs manquantes (NA) au sens de **R**, auxquelles a été attachées sur étiquette, une lettre unique minuscule (a-z) ou majuscule (A-Z). On peut les créer avec `labelled::tagged_na()`.

```
x <- c(1:3, tagged_na("a"), tagged_na("z"), NA)
```

Pour la plupart des fonctions de **R**, les *tagged NAs* sont juste considérées comme des valeurs manquantes régulières (*regular NAs*). Dès lors, par défaut, elles sont justes affichées à l'écran comme n'importe quelle valeur manquante et la fonction `is.na()` renvoie `TRUE`.

```
x
```

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

```
is.na(x)
```

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

Pour afficher les étiquettes associées à ces valeurs manquantes, il faut avoir recours à `labelled::na_tag()`, `labelled::print_tagged_na()` ou encore `labelled::format_tagged_na()`.

```
na_tag(x)
```

```
[1] NA NA NA "a" "z" NA
```

```
print_tagged_na(x)
```

```
[1] 1 2 3 NA(a) NA(z) NA
```

```
format_tagged_na(x)
```

```
[1] " 1" " 2" " 3" "NA(a)" "NA(z)" " NA"
```

Pour tester si une certaine valeur manquante est une *regular NA* ou une *tagged NA*, on aura recours à `labelled::is_regular_na()` et à `labelled::is_tagged_na()`.

```
is.na(x)
```

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

```
is_regular_na(x)
```

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

```
is_tagged_na(x)
```

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

Il est possible de tester une étiquette particulière en passant un deuxième argument à `labelled::is_tagged_na()`.

```
is_tagged_na(x, "a")
```

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

i Note

Il n'est possible de définir des *tagged NAs* seulement pour des vecteurs numériques (*double*). Si l'on ajoute une *tagged NA* à un vecteur d'entiers, ce vecteur sera converti en vecteur numérique. Si on l'ajoute à un vecteur textuel, la valeur manquante sera convertie en *regular NA*.

```
y <- c("a", "b", tagged_na("z"))  
y
```

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

```
is_tagged_na(y)
```

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

```
format_tagged_na(y)
```

```
Error: `x` must be a double vector
```

```
z <- c(1L, 2L, tagged_na("a"))
typeof(z)
```

```
[1] "double"
```

```
format_tagged_na(z)
```

```
[1] "      1" "      2" "NA(a)"
```

13.1.2 Valeurs uniques, doublons et tris

Par défaut, les fonctions classiques de **R** `unique()`, `duplicated()`, `ordered()` ou encore `sort()` traiteront les *tagged NAs* comme des valeurs manquantes tout ce qu'il y a de plus classique, et ne feront pas de différences entre des *tagged NAs* ayant des étiquettes différentes.

Pour traiter des *tagged NAs* ayant des étiquettes différentes comme des valeurs différentes, on aura recours aux fonctions `labelled::unique_tagged_na()`, `labelled::duplicated_tagged_na()`, `labelled::order_tagged_na()` ou encore `labelled::sort_tagged_na()`.

```
x <- c(1, 2, tagged_na("a"), 1, tagged_na("z"), 2, tagged_na("a"), NA)
x |>
  print_tagged_na()
```

```
[1]      1      2 NA(a)      1 NA(z)      2 NA(a)      NA
```

```
x |>
  unique() |>
  print_tagged_na()
```

```
[1]      1      2 NA(a)
```

```
x |>
  unique_tagged_na() |>
  print_tagged_na()
```

```
[1]      1      2 NA(a) NA(z)      NA
```

```
x |>
  duplicated()
```

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

```
x |>
  duplicated_tagged_na()
```

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

```
x |>
  sort(na.last = TRUE) |>
  print_tagged_na()
```

```
[1]      1      1      2      2 NA(a) NA(z) NA(a)      NA
```

```
x |>
  sort_tagged_na() |>
  print_tagged_na()
```

```
[1]      1      1      2      2 NA(a) NA(a) NA(z)      NA
```

13.1.3 Tagged NAs et étiquettes de valeurs

Il est tout à fait possible d'associer une étiquette de valeurs (cf. Chapitre 12) à des *tagged NAs*.

```
x <- c(
  1, 0,
  1, tagged_na("r"),
  0, tagged_na("d"),
  tagged_na("z"), NA
)
val_labels(x) <- c(
  no = 0,
  yes = 1,
  "don't know" = tagged_na("d"),
  refusal = tagged_na("r")
)
x
```

```
<labelled<double>[8]>
[1]      1      0      1 NA(r)      0 NA(d) NA(z)      NA
```

```
Labels:
  value      label
    0         no
    1         yes
NA(d) don't know
NA(r)   refusal
```

Lorsqu'un vecteur labellisé est converti en facteur avec `labelled::to_factor()`, les *tagged NAs* sont, par défaut convertis en en valeurs manquantes classiques (*regular NAs*). Il n'est pas possible de définir des *tagged NAs* pour des facteurs.

```
x |> to_factor()
```

```
[1] yes  no   yes  <NA> no   <NA> <NA> <NA>
Levels: no yes
```

L'option `explicit_tagged_na` de `labelled::to_factor()` permet de convertir les *tagged NAs* en modalités explicites du facteur.

```
x |>
  to_factor(explicit_tagged_na = TRUE)
```

```
[1] yes      no      yes      refusal    no      don't know NA(z)
[8] <NA>
Levels: no yes don't know refusal NA(z)
```

```
x |>
  to_factor(
    levels = "prefixed",
    explicit_tagged_na = TRUE
  )
```

```
[1] [1] yes      [0] no      [1] yes      [NA(r)] refusal
[5] [0] no      [NA(d)] don't know [NA(z)] NA(z)    <NA>
Levels: [0] no [1] yes [NA(d)] don't know [NA(r)] refusal [NA(z)] NA(z)
```

13.1.4 Conversion en user NAs

La fonction `labelled::tagged_na_to_user_na()` permet de convertir des *tagged NAs* en *user NAs*.

```
x |>
  tagged_na_to_user_na()
```

```
<labelled_spss<double>[8]>
[1] 1 0 1 3 0 2 4 NA
Missing range: [2, 4]
```

Labels:

value	label
0	no
1	yes
2	don't know
3	refusal
4	NA(z)

```
x |>
  tagged_na_to_user_na(user_na_start = 10)
```

```
<labelled_spss<double>[8]>
[1] 1 0 1 11 0 10 12 NA
Missing range: [10, 12]
```

Labels:

value	label
0	no
1	yes
10	don't know
11	refusal
12	NA(z)

La fonction `labelled::tagged_na_to_regular_na()` convertit les *tagged NAs* en valeurs manquantes classiques (*regular NAs*).

```
x |>
  tagged_na_to_regular_na()
```

```
<labelled<double>[8]>
[1] 1 0 1 NA 0 NA NA NA
```

```
Labels:
value label
  0      no
  1     yes
```

```
x |>
  tagged_na_to_regular_na() |>
  is_tagged_na()
```

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

13.2 Valeurs manquantes définies par l'utilisateur (*user NAs*)

Le package `{haven}` a introduit la classe `haven_labelled_spss`, une extension de la classe `haven_labelled` permettant d'indiquer des valeurs à considérer comme manquantes à la manière de **SPSS**.

! Important

Cela revient à associer à un vecteur des attributs (cf. Chapitre 6) additionnels pour indiquer des valeurs que l'utilisateur pourrait/devrait considérer comme manquante. Cependant, il ne s'agit que de métadonnées et en interne ces valeurs ne sont pas stockées sous forme de NA mais restent des valeurs valides.

Il convient de garder en mémoire que la très grande majorité des fonctions de **R** ne prendront pas en compte ces métadonnées et traiteront donc ces valeurs comme des valeurs valides. C'est donc à l'utilisateur de convertir, au besoin, ces les valeurs indiquées comme manquantes en réelles valeurs manquantes (NA).

13.2.1 Création

Il est possible d'indiquer des valeurs à considérer comme manquantes (*user NAs*) de deux manières :

- soit en indiquant une liste de valeurs individuelles avec `labelled::na_values()` (on peut indiquer `NULL` pour supprimer les déclarations existantes) ;

- soit en indiquant deux valeurs représentant une plage de valeurs à considérées comme manquantes avec `labelled::na_range()` (seront considérées comme manquantes toutes les valeurs supérieures ou égale au premier chiffre et inférieures ou égales au second chiffre¹).

```
v <- c(1, 2, 3, 9, 1, 3, 2, NA)
val_labels(v) <- c(
  faible = 1,
  fort = 3,
  "ne sait pas" = 9
)
na_values(v) <- 9
v
```

```
<labelled_spss<double>[8]>
[1] 1 2 3 9 1 3 2 NA
Missing values: 9
```

```
Labels:
value      label
    1      faible
    3      fort
    9 ne sait pas
```

```
na_values(v) <- NULL
v
```

```
<labelled<double>[8]>
[1] 1 2 3 9 1 3 2 NA
```

```
Labels:
value      label
    1      faible
    3      fort
    9 ne sait pas
```

```
na_range(v) <- c(5, Inf)
v
```

¹On peut utiliser `-Inf` et `Inf` qui représentent respectivement moins l'infini et l'infini.

```
<labelled_spss<double>[8]>
[1] 1 2 3 9 1 3 2 NA
Missing range: [5, Inf]
```

```
Labels:
  value      label
    1      faible
    3       fort
    9 ne sait pas
```

On peut noter que les *user NAs* peuvent cohabiter avec des *regular NAs* ainsi qu'avec des étiquettes de valeurs (*value labels*, cf. Chapitre 12).

Pour manipuler les variables d'un tableau de données, on peut également avoir recours à `labelled::set_na_values()` et `labelled::set_na_range()`.

```
df <-
  dplyr::tibble(
    s1 = c("M", "M", "F", "F"),
    s2 = c(1, 1, 2, 9)
  ) |>
  labelled::set_na_values(s2 = 9)
df$s2
```

```
<labelled_spss<double>[4]>
[1] 1 1 2 9
Missing values: 9
```

```
df <-
  df |>
  labelled::set_na_values(s2 = NULL)
df$s2
```

```
<labelled<double>[4]>
[1] 1 1 2 9
```

13.2.2 Tests

La fonction `is.na()` est l'une des rares fonctions de base **R** à reconnaître les *user NAs* et donc à renvoyer `TRUE` dans ce cas. Pour des tests plus spécifiques, on aura recours à `labelled::is_user_na()` et `labelled::is_regular_na()`.

```
v
```

```
<labelled_spss<double>[8]>  
[1] 1 2 3 9 1 3 2 NA  
Missing range: [5, Inf]
```

```
Labels:  
  value      label  
    1      faible  
    3       fort  
    9 ne sait pas
```

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

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

```
v |> is_user_na()
```

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

```
v |> is_regular_na()
```

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

13.2.3 Conversion

Comme dit précédemment, pour la plupart des fonctions de **R**, les *users NAs* sont toujours des valeurs valides.

```
x <- c(1:5, 11:15)  
na_range(x) <- c(10, Inf)  
x
```

```
<labelled_spss<integer>[10]>  
[1] 1 2 3 4 5 11 12 13 14 15  
Missing range: [10, Inf]
```

```
mean(x)
```

```
[1] 8
```

On aura alors recours à `labelled::user_na_to_regular_na()` pour convertir les *users NAs* en véritables valeurs manquantes avant de procéder à un calcul statistique.

```
x |>
  user_na_to_na()
```

```
<labelled<integer>[10]>
[1] 1 2 3 4 5 NA NA NA NA NA
```

```
x |>
  user_na_to_na() |>
  mean(na.rm = TRUE)
```

```
[1] 3
```

Une alternative consiste à transformer les *user NAs* en *tagged NAs* avec `labelled::user_na_to_tagged_na()`.

```
x |>
  user_na_to_tagged_na() |>
  print_tagged_na()
```

```
'x' has been converted into a double vector.
```

```
[1] 1 2 3 4 5 NA(a) NA(b) NA(c) NA(d) NA(e)
```

```
x |>
  user_na_to_tagged_na() |>
  mean(na.rm = TRUE)
```

```
'x' has been converted into a double vector.
```

```
[1] 3
```

Pour supprimer les métadonnées relatives aux *user NAs* sans les convertir en valeurs manquantes, on aura recours à `labelled::remove_user_na()`.

```
x |>
  remove_user_na()
```

```
<labelled<integer>[10]>
 [1]  1  2  3  4  5 11 12 13 14 15
```

```
x |>
  remove_user_na() |>
  mean()
```

```
[1] 8
```

Enfin, lorsque l'on convertit un vecteur labellisé en facteur avec `labelled::to_factor()`, on pourra utiliser l'argument `user_na_to_na` pour indiquer si les *users NAs* doivent être convertis ou non en valeurs manquantes classiques (NA).

```
x <- c(1, 2, 9, 2)
val_labels(x) <- c(oui = 1, non = 2, refus = 9)
na_values(x) <- 9
x |>
  to_factor(user_na_to_na = TRUE)
```

```
[1] oui  non  <NA> non
Levels: oui non
```

```
x |>
  to_factor(user_na_to_na = FALSE)
```

```
[1] oui  non  refus non
Levels: oui non refus
```

14 Import & Export de données

14.1 Importer un fichier texte

Les fichiers texte constituent un des formats les plus largement supportés par la majorité des logiciels statistiques. Presque tous permettent d'exporter des données dans un format texte, y compris les tableurs comme **Libre Office**, **Open Office** ou **Excel**.

Cependant, il existe une grande variété de format texte, qui peuvent prendre différents noms selon les outils, tels que texte tabulé ou *texte (séparateur : tabulation)*, **CSV** (pour *comma-separated value*, sachant que suivant les logiciels le séparateur peut être une virgule ou un point-virgule).

14.1.1 Structure d'un fichier texte

Dès lors, avant d'importer un fichier texte dans **R**, il est indispensable de regarder comment ce dernier est structuré. Il importe de prendre note des éléments suivants :

- La première ligne contient-elle le nom des variables ?
- Quel est le caractère séparateur entre les différentes variables (encore appelé séparateur de champs) ? Dans le cadre d'un fichier **CSV**, il aurait pu s'agir d'une virgule ou d'un point-virgule.
- Quel est le caractère utilisé pour indiquer les décimales (le séparateur décimal) ? Il s'agit en général d'un point (à l'anglo-saxonne) ou d'une virgule (à la française).
- Les valeurs textuelles sont-elles encadrées par des guillemets et, si oui, s'agit-il de guillemets simple (') ou de guillemets doubles (") ?
- Pour les variables textuelles, y a-t-il des valeurs manquantes et si oui comment sont-elles indiquées ? Par exemple, le texte **NA** est parfois utilisé.

Il ne faut pas hésiter à ouvrir le fichier avec un éditeur de texte pour le regarder de plus près.

14.1.2 Interface graphique avec RStudio

RStudio fournit une interface graphique pour faciliter l'import d'un fichier texte. Pour cela, il suffit d'aller dans le menu *File > Import Dataset* et de choisir l'option *From CSV*¹. Cette option est également disponible via l'onglet *Environment* dans le quadrant haut-droite.

Pour la suite, nous allons utiliser ce [fichier texte à titre d'exemple](#).

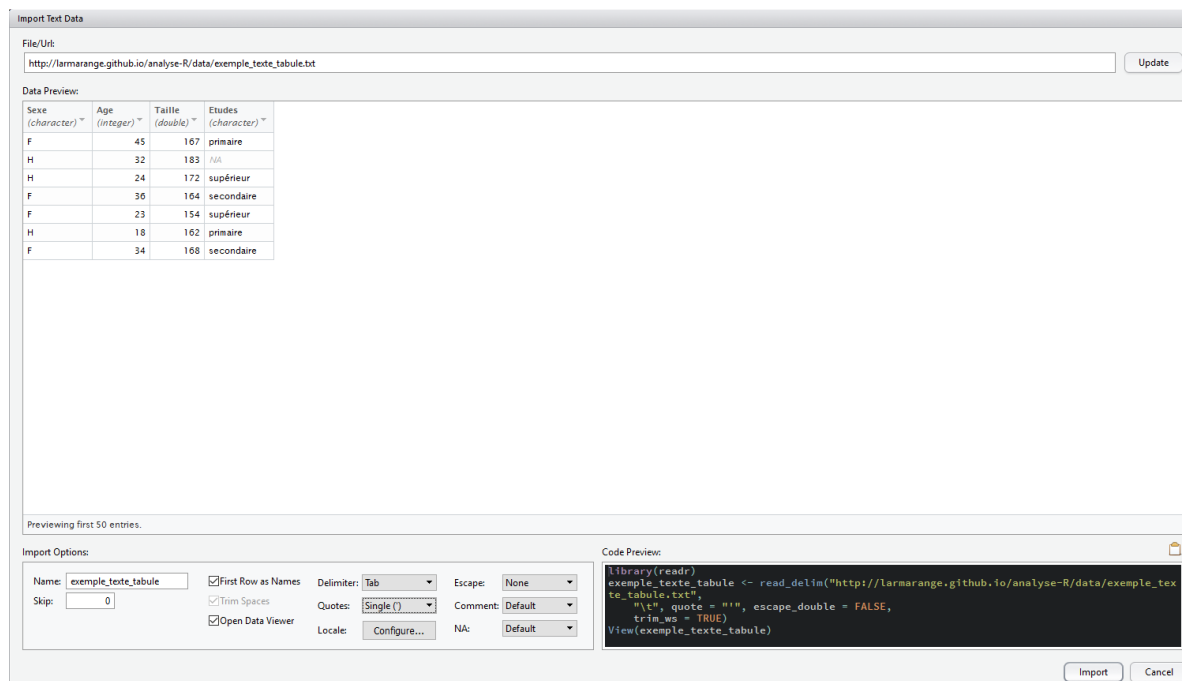


Figure 14.1: Importer un fichier texte avec RStudio

L'interface de **RStudio** vous présente sous *Import Options* les différentes options d'import disponible. La section *Data Preview* vous permet de voir en temps réel comment les données sont importées. La section *Code Preview* vous indique le code **R** correspondant à vos choix. Il n'y a plus qu'à le copier/coller dans un de vos scripts ou à cliquer sur **Import** pour l'exécuter.

Vous pourrez remarquer que **RStudio** fait appel à l'extension `{readr}` du tidyverse pour l'import des données via la fonction `readr::read_csv()`.

`{readr}` essaie de deviner le type de chacune des colonnes, en se basant sur les premières observations. En cliquant sur le nom d'une colonne, il est possible de modifier le type de la variable importée. Il est également possible d'exclure une colonne de l'import (*skip*).

¹L'option CSV fonctionne pour tous les fichiers de type texte, même si votre fichier a une autre extension, `.txt` par exemple

14.1.3 Dans un script

L'interface graphique de **RStudio** fournit le code d'import. On peut également l'adapter à ces besoins en consultant la page d'aide de `readr::read_csv()` pour plus de détails. Par exemple :

```
library(readr)
d <- read_delim(
  "http://larmarange.github.io/analyse-R/data/exemple_texte_tabule.txt",
  delim = "\t",
  quote = ""
)
```

On peut indiquer le chemin local vers un fichier (le plus courant) ou bien directement l'URL d'un fichier sur Internet.

`{readr}` propose plusieurs fonctions proches : `readr::read_delim()`, `readr::read_csv()`, `readr::read_csv2()` et `readr::read_tsv()`. Elles fonctionnent toutes de manière identique et ont les mêmes arguments. Seule différence, les valeurs par défaut de certains paramètres.

Fichiers de très grande taille

Si vous travaillez sur des données de grandes dimensions, les formats texte peuvent être lents à exporter et importer. Dans ce cas là, on pourra jeter un œil au package `{vroom}` et/ou aux fonctions `data.table::fread()` et `data.table::fwrite()`.

Dans des manuels ou des exemples en ligne, vous trouverez parfois mention des fonctions `utils::read.table()`, `utils::read.csv()`, `utils::read.csv2()`, `utils::read.delim()` ou encore `utils::read.delim2()`. Il s'agit des fonctions natives et historiques de **R** (extension `{utils}`) dédiées à l'import de fichiers textes. Elles sont similaires à celles de `{readr}` dans l'idée générale mais diffèrent dans leurs détails et les traitements effectués sur les données (pas de détection des dates par exemple). Pour plus d'information, vous pouvez vous référer à la page d'aide de ces fonctions.

14.2 Importer un fichier Excel

Une première approche pour importer des données **Excel** dans **R** consiste à les exporter depuis **Excel** dans un fichier texte (texte tabulé ou **CSV**) puis de suivre la procédure d'importation d'un fichier texte.

Une feuille **Excel** peut également être importée directement avec l'extension `{readxl}` du *tidyverse*.

La fonction `readxl::read_excel()` permet d'importer à la fois des fichiers `.xls` (**Excel** 2003 et précédents) et `.xlsx` (**Excel** 2007 et suivants).

```
library(readxl)
donnees <- read_excel("data/fichier.xlsx")
```

Une seule feuille de calculs peut être importée à la fois. On pourra préciser la feuille désirée avec `sheet` en indiquant soit le nom de la feuille, soit sa position (première, seconde, ...).

```
donnees <- read_excel("data/fichier.xlsx", sheet = 3)
donnees <- read_excel("data/fichier.xlsx", sheet = "mes_donnees")
```

On pourra préciser avec `col_names` si la première ligne contient le nom des variables.

Par défaut, `readxl::read_excel()` va essayer de deviner le type (numérique, textuelle, date) de chaque colonne. Au besoin, on pourra indiquer le type souhaité de chaque colonne avec `col_types`.

RStudio propose également pour les fichiers **Excel** un assistant d'importation, similaire à celui pour les fichiers texte, permettant de faciliter l'import.

14.3 Importer depuis des logiciels de statistique

Le package `{haven}` du *tidyverse* a été développé spécifiquement pour permettre l'importation de données depuis les formats des logiciels **Stata**, **SAS** et **SPSS**.

Il vise à offrir une importation unifiée depuis ces trois logiciels (là où le package `{foreign}` distribué en standard avec **R** adopte des conventions différentes selon le logiciel source).

Afin de ne pas perdre d'information lors de l'import, `{haven}` a introduit la notion d'étiquettes de variables (cf. Chapitre 11), une classe de vecteurs pour la gestion des étiquettes de valeurs (cf. Chapitre 12), des mécanismes pour reproduire la gestion des valeurs manquantes de ces trois logiciels (cf. Chapitre 13), mais également une gestion et un import correct des dates, dates-heures et des variables horaires (cf. le package `{hms}`).

À noter que **RStudio** intègre également une interface graphique pour l'import des fichiers **Stata**, **SAS** et **SPSS**.

14.3.1 SPSS

Les fichiers générés par **SPSS** sont de deux types : les fichiers **SPSS natifs** (extension `.sav`) et les fichiers au format **SPSS export** (extension `.por`).

Dans les deux cas, on aura recours à la fonction `haven::read_spss()` :

```
library(haven)
donnees <- read_spss("data/fichier.sav", user_na = TRUE)
```

Valeurs manquantes

Dans **SPSS**, il est possible de définir des valeurs à considérées comme manquantes ou *user NAs*, voir Chapitre 13. Par défaut, `haven::read_spss()` convertit toutes ces valeurs en `NA` lors de l'import.

Or, il est parfois important de garder les différentes valeurs originelles. Dans ce cas, on appellera `haven::read_spss()` avec l'option `user_na = TRUE`.

14.3.2 SAS

Les fichiers **SAS** se présentent en général sous deux format : format **SAS export** (extension `.xport` ou `.xpt`) ou format **SAS natif** (extension `.sas7bdat`).

Les fichiers **SAS natifs** peuvent être importées directement avec `haven::read_sas()` de l'extension `{haven}` :

```
library(haven)
donnees <- read_sas("data/fichier.sas7bdat")
```

Au besoin, on pourra préciser en deuxième argument le nom d'un fichier **SAS catalogue** (extension `.sas7bcats`) contenant les métadonnées du fichier de données.

```
library(haven)
donnees <- read_sas(
  "data/fichier.sas7bdat",
  catalog_file = "data/fichier.sas7bcats"
)
```

Note

Les fichiers au format **SAS export** peuvent être importés via la fonction `foreign::read.xport()` de l'extension `{foreign}`. Celle-ci s'utilise très simplement, en lui passant le nom du fichier en argument :

```
library(foreign)
donnees <- read.xport("data/fichier.xpt")
```

14.3.3 Stata

Pour les fichiers **Stata** (extension `.dta`), on aura recours aux fonctions `haven::read_dta()` et `haven::read_stata()` de l'extension `{haven}`. Ces deux fonctions sont identiques.

```
library(haven)
donnees <- read_dta("data/fichier.dta")
```

Important

Gestion des valeurs manquantes

Dans **Stata**, il est possible de définir plusieurs types de valeurs manquantes, qui sont notées sous la forme `.a` à `.z`. Elles sont importées par `{haven}` sous formes de *tagged NAs*, cf. Chapitre 13.

14.3.4 dBase

L'Insee et d'autres producteurs de données diffusent leurs fichiers au format **dBase** (extension `.dbf`). Ceux-ci sont directement lisibles dans **R** avec la fonction `foreign::read.dbf()` de l'extension `{foreign}`.

```
library(foreign)
donnees <- read.dbf("data/fichier.dbf")
```

14.4 Sauver ses données

R dispose également de son propre format pour sauvegarder et échanger des données. On peut sauver n'importe quel objet créé avec **R** et il est possible de sauver plusieurs objets dans

un même fichier. L'usage est d'utiliser l'extension `.RData` pour les fichiers de données **R**. La fonction à utiliser s'appelle tout simplement `save()`.

Par exemple, si l'on souhaite sauvegarder son tableau de données `d` ainsi que les objets `tailles` et `poids` dans un fichier `export.RData` :

```
save(d, tailles, poids, file = "export.RData")
```

À tout moment, il sera toujours possible de recharger ces données en mémoire à l'aide de la fonction `load()` :

```
load("export.RData")
```

Mise en garde

Si entre temps vous aviez modifié votre tableau `d`, vos modifications seront perdues. En effet, si lors du chargement de données, un objet du même nom existe en mémoire, ce dernier sera remplacé par l'objet importé.

La fonction `save.image()` est un raccourci pour sauvegarder tous les objets de la session de travail dans le fichier `.RData` (un fichier un peu étrange car il n'a pas de nom mais juste une extension). Lors de la fermeture de **RStudio**, il vous sera demandé si vous souhaitez enregistrer votre session. Si vous répondez *Oui*, c'est cette fonction `save.image()` qui sera appliquée.

```
save.image()
```

Un autre mécanisme possible est le format **RDS** de **R**. La fonction `saveRDS()` permet de sauvegarder **un et un seul** objet **R** dans un fichier.

```
saveRDS(d, file = "mes_donnees.rds")
```

Cet objet pourra ensuite être lu avec la fonction `readRDS()`. Mais au lieu d'être directement chargé dans la mémoire de l'environnement de travail, l'objet lu sera retourné par la fonction `readRDS()` et ce sera à l'utilisateur de le sauvegarder.

```
donnees <- readRDS("mes_donnees.rds")
```

14.5 Export de tableaux de données

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

- `readr::write_csv()` et `readr::write_tsv()` permettent d'exporter au format **CSV** et texte tabulé respectivement, `readr::write_delim()` offrant de multiples options pour l'export au format texte ;
- `haven::write_sas()` permet d'exporter au format **SAS** ;
- `haven::write_sav()` au format **SPSS** ;
- `haven::write_dta()` au format **Stata** ;
- `foreign::write.dbf()` au format **dBase**.

L'extension `readxl` ne fournit pas de fonction pour exporter au format **Excel**. Par contre, on pourra passer par la fonction `openxlsx::write.xlsx()` du package `{openxlsx}` ou la fonction `xlsx::write.xlsx()` de l'extension `{xlsx}`. L'intérêt de `{openxlsx}` est de ne pas dépendre de **Java** à la différence de `{xlsx}`.

15 Mettre en forme des nombres

Dans les chapitres suivants, nous aurons régulièrement besoin, pour produire des tableaux ou des figures propres, de **fonctions de formatage** qui permettent de transformer des valeurs numériques en chaînes de texte.

La fonction **R** de base est `format()` mais de nombreux autres packages proposent des variations pour faciliter cette opération. Le plus complet est probablement `{scales}` et, notamment, ses fonctions `scales::label_number()` et `scales::number()`.

Elles ont l'air très similaires et partagent un grand nombre de paramètres en commun. La différence est que `scales::number()` a besoin d'un vecteur numérique en entrée qu'elle va mettre en forme, tandis que `scales::label_number()` renvoie une fonction que l'on pourra ensuite appliquer à un vecteur numérique.

```
library(scales)
x <- c(0.0023, .123, 4.567, 874.44, 8957845)
number(x)
```

```
[1] "0.00"          "0.12"          "4.57"          "874.44"        "8 957 845.00"
```

```
f <- label_number()
f(x)
```

```
[1] "0.00"          "0.12"          "4.57"          "874.44"        "8 957 845.00"
```

```
label_number()(x)
```

```
[1] "0.00"          "0.12"          "4.57"          "874.44"        "8 957 845.00"
```

Dans de nombreux cas de figure (par exemple pour un graphique `{ggplot2}` ou un tableau `{gtsummary}`), il sera demandé de fournir une fonction de formatage, auquel cas on aura recours aux fonctions de `{scales}` préfixées par `label_*` qui permettent donc de générer une fonction personnalisée.

15.1 label_number()

`scales::label_number()` est la fonction de base de mise en forme de nombres dans `{scales}`, une majorité des autres fonctions faisant appel à `scales::label_number()` et partageant les mêmes arguments.

Le paramètre `accuracy` permet de définir le niveau d'arrondi à utiliser. Par exemple, `.1` pour afficher une seule décimale. Il est aussi possible d'indiquer un nombre qui n'est pas une puissance de 10 (par exemple `.25`). Si on n'indique rien (`NULL`), alors `scales::label_number()` essaiera de deviner un nombre de décimales pertinent en fonction des valeurs du vecteur de nombres à mettre en forme.

```
label_number(accuracy = NULL)(x)
```

```
[1] "0.00"      "0.12"      "4.57"      "874.44"    "8 957 845.00"
```

```
label_number(accuracy = .1)(x)
```

```
[1] "0.0"       "0.1"       "4.6"       "874.4"     "8 957 845.0"
```

```
label_number(accuracy = .25)(x)
```

```
[1] "0.0"       "0.0"       "4.5"       "874.5"     "8 957 845.0"
```

```
label_number(accuracy = 10)(x)
```

```
[1] "0"         "0"         "0"         "870"       "8 957 840"
```

L'option `scale` permet d'indiquer un facteur multiplicatif à appliquer avant de mettre en forme. On utilisera le plus souvent les options `prefix` et `suffix` en même temps pour indiquer les unités.

```
label_number(scale = 100, suffix = "%")(x) # pour cent
```

```
[1] "0%"        "12%"       "457%"      "87 444%"   "895 784 500%"
```

```
label_number(scale = 1000, suffix = "\u2030")(x) # pour mille
```

```
[1] "2%"      "123%"      "4 567%"      "874 440%"
[5] "8 957 845 000%"
```

```
label_number(scale = .001, suffix = " milliers", accuracy = .1)(x)
```

```
[1] "0.0 milliers"      "0.0 milliers"      "0.0 milliers"      "0.9 milliers"
[5] "8 957.8 milliers"
```

Les arguments `decimal.mark` et `big.mark` permettent de définir, respectivement, le séparateur de décimale et le séparateur de milliers. Ainsi, pour afficher des nombres à la française (virgule pour les décimales, espace pour les milliers) :

```
label_number(decimal.mark = ",", big.mark = " ")(x)
```

```
[1] "0,00"      "0,12"      "4,57"      "874,44"      "8 957 845,00"
```

Note : il est possible d'utiliser `small.interval` et `small.mark` pour ajouter des séparateurs parmi les décimales.

```
label_number(accuracy = 10^-9, small.mark = "|", small.interval = 3)(x)
```

```
[1] "0.002|300|000"      "0.123|000|000"      "4.567|000|000"
[4] "874.440|000|000"      "8 957 845.000|000|000"
```

Les options `style_positive` et `style_negative` permettent de personnaliser la manière dont les valeurs positives et négatives sont mises en forme.

```
y <- c(-1.2, -0.3, 0, 2.4, 7.2)
label_number(style_positive = "plus")(y)
```

```
[1] "-1.2" "-0.3" "0.0"  "+2.4" "+7.2"
```

```
label_number(style_negative = "parens")(y)
```

```
[1] "(1.2)" "(0.3)" "0.0"   "2.4"   "7.2"
```


L'option `scale_cut` permet d'utiliser, entre autres, les [préfixes du Système international d'unités](#) les plus proches et arrondi chaque valeur en fonction, en ajoutant la précision correspondante. Par exemple, pour des données en grammes :

```
y <- c(.000004536, .01245, 2.3456, 47589.14, 789456244)
label_number(scale_cut = cut_si("g"), accuracy = .1)(y)
```

```
[1] "4.5 µg"    "12.4 mg"   "2.3 g"     "47.6 kg"   "789.5 Mg"
```

15.2 Les autres fonctions de {scales}

15.2.1 label_comma()

`scales::label_comma()` (et `scales::comma()`) est une variante de `scales::label_number()` qui, par défaut, affiche les nombres à l'américaine, avec une virgule comme séparateur de milliers.

```
label_comma()(x)
```

```
[1] "0.00"      "0.12"      "4.57"      "874.44"    "8,957,845.00"
```

15.2.2 label_percent()

`scales::label_percent()` (et `scales::percent()`) est une variante de `scales::label_number()` qui affiche les nombres sous formes de pourcentages (les options par défaut sont `scale = 100`, `suffix = "%"`).

```
label_percent()(x)
```

```
[1] "0%"      "12%"      "457%"     "87 444%"  "895 784 500%"
```

On peut utiliser cette fonction pour afficher des résultats en pour mille (le [code Unicode](#) du symbole ‰ étant `u2030`) :

```
label_percent(scale = 1000, suffix = "\u2030")(x)
```

```
[1] "2‰"      "123‰"     "4 567‰"   "874 440‰"
[5] "8 957 845 000‰"
```

15.2.3 label_dollar()

`scales::label_dollar()` est adapté à l’affichage des valeurs monétaires.

```
label_dollar()(x)
```

```
[1] "$0"          "$0"          "$5"          "$874"        "$8,957,845"
```

```
label_dollar(prefix = "", suffix = " €", accuracy = .01, big.mark = " ")(x)
```

```
[1] "0.00 €"      "0.12 €"      "4.57 €"      "874.44 €"
[5] "8 957 845.00 €"
```

L’option `style_negative` permet d’afficher les valeurs négatives avec des parenthèses, convention utilisée dans certaines disciplines.

```
label_dollar()(c(12.5, -4, 21, -56.36))
```

```
[1] "$12.50"  "$-4.00"  "$21.00"  "$-56.36"
```

```
label_dollar(style_negative = "parens")(c(12.5, -4, 21, -56.36))
```

```
[1] "$12.50"  "($4.00)" "$21.00"  "($56.36)"
```

15.2.4 label_pvalue()

`scales::label_pvalue()` est adapté pour la mise en forme de p-valeurs.

```
label_pvalue()(c(0.000001, 0.023, 0.098, 0.60, 0.9998))
```

```
[1] "<0.001" "0.023"  "0.098"  "0.600"  ">0.999"
```

```
label_pvalue(accuracy = .01, add_p = TRUE)(c(0.000001, 0.023, 0.098, 0.60))
```

```
[1] "p<0.01" "p=0.02" "p=0.10" "p=0.60"
```

15.2.5 label_scientific()

`scales::label_scientific()` affiche les nombres dans un format scientifique (avec des puissances de 10).

```
label_scientific(unit = "g")(c(.00000145, .0034, 5, 12478, 14569787))
```

```
[1] "1.45e-06" "3.40e-03" "5.00e+00" "1.25e+04" "1.46e+07"
```

15.2.6 label_bytes()

`scales::label_bytes()` mets en forme des tailles exprimées en octets, utilisant au besoin des multiples de 1024.

```
b <- c(478, 1235468, 546578944897)
label_bytes()(b)
```

```
[1] "478 B" "1 MB" "547 GB"
```

```
label_bytes(units = "auto_binary")(b)
```

```
[1] "478 iB" "1 MiB" "509 GiB"
```

15.2.7 label_ordinal()

`scales::label_ordinal()` permet d'afficher des rangs ou nombres ordinaux. Plusieurs langues sont disponibles.

```
label_ordinal()(1:5)
```

```
[1] "1st" "2nd" "3rd" "4th" "5th"
```

```
label_ordinal(rules = ordinal_french()(1:5))
```

```
[1] "1er" "2e" "3e" "4e" "5e"
```

```
label_ordinal(rules = ordinal_french(gender = "f", plural = TRUE))(1:5)
```

```
[1] "1res" "2es" "3es" "4es" "5es"
```

15.2.8 label_date(), label_date_short() & label_time()

scales::label_date(), scales::label_date_short() et scales::label_time() peuvent être utilisées pour la mise en forme de dates.

```
label_date()(as.Date("2020-02-14"))
```

```
[1] "2020-02-14"
```

```
label_date(format = "%d/%m/%Y")(as.Date("2020-02-14"))
```

```
[1] "14/02/2020"
```

```
label_date_short()(as.Date("2020-02-14"))
```

```
[1] "14\nfévr.\n2020"
```

La mise en forme des dates est un peu complexe. Ne pas hésiter à consulter le fichier d'aide de la fonction `base::strptime()` pour plus d'informations.

15.2.9 label_wrap()

La fonction `scales::label_wrap()` est un peu différente. Elle permet d'insérer des retours à la ligne (`\n`) dans des chaînes de caractères. Elle tient compte des espaces pour identifier les mots et éviter ainsi des coupures au milieu d'un mot.

```
x <- "Ceci est un texte assez long et que l'on souhaiterait afficher sur plusieurs lignes. C  
label_wrap(80)(x)
```

```
[1] "Ceci est un texte assez long et que l'on souhaiterait afficher sur plusieurs\nlignes. C"
```

```
label_wrap(80)(x) |> message()
```

Ceci est un texte assez long et que l'on souhaiterait afficher sur plusieurs lignes. Cependant, on souhaite éviter que des coupures apparaissent au milieu d'un mot.

```
label_wrap(40)(x) |> message()
```

Ceci est un texte assez long et que
l'on souhaiterait afficher sur
plusieurs lignes. Cependant, on
souhaite éviter que des coupures
apparaissent au milieu d'un mot.

15.3 Les fonctions de formatage de {gtsummary}

Véritable couteau-suisse du statisticien, le package {gtsummary} sera largement utilisé dans les prochains chapitres pour produire des tableaux statistiques prêts à être publiés.

Ce package utilise par défaut ses propres fonctions de formatage mais, au besoin, il sera toujours possible de lui transmettre des fonctions de formatage créées avec {scales}.

15.3.1 style_number()

Fonction de base, `gtsummary::style_number()` accepte les paramètres `big.mark` (séparateur de milliers), `decimal.mark` (séparateur de décimales) et `scale` (facteur d'échelle). Le nombre de décimales se précisera quant à lui avec `digits` où l'on indiquera le nombre de décimales souhaité.

```
library(gtsummary)
x <- c(0.123, 0.9, 1.1234, 12.345, -0.123, -0.9, -1.1234, -132.345)
style_number(x, digits = 1)
```

```
[1] "0.1"      "0.9"      "1.1"      "12.3"     "-0.1"     "-0.9"     "-1.1"     "-132.3"
```

Astuce

Nous verrons dans le chapitre sur les statistiques univariées (cf. Section 18.2.1) la fonction `gtsummary::theme_gtsummary_language()` qui permet de fixer globalement le séparateur de milliers et celui des décimales, afin de changer les valeurs par défaut de l'ensemble des fonctions de formatage de `{gtsummary}`.

Il est important de noter que cela n'a aucun effet sur les fonctions de formatage de `{scales}`.

Mise en garde

`gtsummary::style_number()` est directement une fonction de formatage (comme `scales::number()`) et non une fonction qui génère une fonction de formatage (comme `scales::label::number()`).

Pour créer une fonction de formatage personnalisée, on pourra avoir recours à `purrr::partial()` qui permet d'appeler partiellement une fonction et qui renvoie une nouvelle fonction avec des paramètres par défaut personnalisés.

```
fr <- style_number |>
  purrr::partial(decimal.mark = ",", digits = 1)
fr(x)
```

```
[1] "0,1"    "0,9"    "1,1"    "12,3"   "-0,1"   "-0,9"   "-1,1"   "-132,3"
```

15.3.2 style_sigfig()

Variante de `gtsummary::style_number()`, `gtsummary::style_sigfig()` arrondi les valeurs transmises pour n'afficher qu'un nombre choisi de chiffres significatifs. Le nombre de décimales peut ainsi varier.

```
style_sigfig(x)
```

```
[1] "0.12"  "0.90"  "1.1"   "12"    "-0.12" "-0.90" "-1.1"  "-132"
```

```
style_sigfig(x, digits = 3)
```

```
[1] "0.123"  "0.900"  "1.12"   "12.3"   "-0.123" "-0.900" "-1.12"  "-132"
```

15.3.3 style_percent()

La fonction `gtsummary::style_percent()` a un fonctionnement un peu différent de celui de `scales::label_percent()`. Par défaut, le symbole % n'est pas affiché (mais paramétrable avec `symbol = TRUE`). Par défaut, une décimale est affichée pour les valeurs inférieures à 10% et aucune pour celles supérieures à 10%. Un symbole < est ajouté devant les valeurs strictement positives inférieures à 0,1%.

```
v <- c(0, 0.0001, 0.005, 0.01, 0.10, 0.45356, 0.99, 1.45)
label_percent(accuracy = .1)(v)
```

```
[1] "0.0%" "0.0%" "0.5%" "1.0%" "10.0%" "45.4%" "99.0%" "145.0%"
```

```
style_percent(v)
```

```
[1] "0" "<0.1" "0.5" "1.0" "10" "45" "99" "145"
```

```
style_percent(v, symbol = TRUE)
```

```
[1] "0%" "<0.1%" "0.5%" "1.0%" "10%" "45%" "99%" "145%"
```

```
style_percent(v, digits = 1)
```

```
[1] "0" "0.01" "0.50" "1.00" "10.0" "45.4" "99.0" "145.0"
```

15.3.4 style_pvalue()

La fonction `gtsummary::style_pvalue()` est similaire à `scales::label_pvalue()` mais adapte le nombre de décimales affichées,

```
p <- c(0.000001, 0.023, 0.098, 0.60, 0.9998)
label_pvalue()(p)
```

```
[1] "<0.001" "0.023" "0.098" "0.600" ">0.999"
```

```
style_pvalue(p)
```

```
[1] "<0.001" "0.023" "0.10" "0.6" ">0.9"
```

```
style_pvalue(p, prepend_p = TRUE)
```

```
[1] "p<0.001" "p=0.023" "p=0.10" "p=0.6" "p>0.9"
```

15.3.5 style_ratio()

Enfin, `gtsummary::style_ratio()` est adaptée à l’affichage de ratios.

```
r <- c(0.123, 0.9, 1.1234, 12.345, 101.234, -0.123, -0.9, -1.1234, -12.345, -101.234)
style_ratio(r)
```

```
[1] "0.12" "0.90" "1.12" "12.3" "101" "-0.12" "-0.90" "-1.12" "-12.3"
[10] "-101"
```

15.4 Bonus : signif_stars() de {ggstats}

La fonction `ggstats::signif_stars()` de `{ggstats}` permet d’afficher des p-valeurs sous forme d’étoiles de significativité. Par défaut, trois astérisques si $p < 0,001$, deux si $p < 0,01$, une si $p < 0,05$ et un point si $p < 0,10$. Les valeurs sont bien sûr paramétrables.

```
p <- c(0.5, 0.1, 0.05, 0.01, 0.001)
ggstats::signif_stars(p)
```

```
[1] ""      "."     "*"     "**"     "***"
```

```
ggstats::signif_stars(p, one = .15, point = NULL)
```

```
[1] ""      "*"     "*"     "**"     "***"
```

15.5

16 Couleurs & Palettes

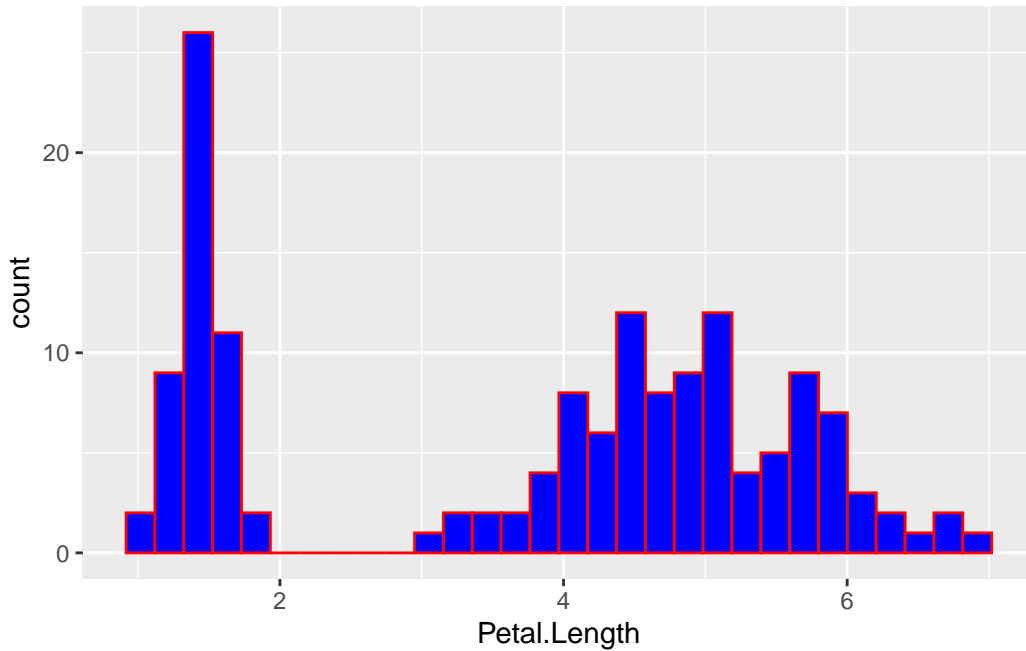
Dans les prochains chapitres, notamment lorsque nous ferons des graphiques, nous aurons besoin de spécifier à **R** les couleurs souhaitées.

Le choix d'une palette de couleurs adaptée à sa représentation graphique est également un élément essentiel avec quelques règles de base : un dégradé est adapté pour représenter une variable continue tandis que pour une variable catégorielle non ordonnée on aura recours à une palette contrastée.

16.1 Noms de couleur

Lorsque l'on doit indiquer à **R** une couleur, notamment dans les fonctions graphiques, on peut mentionner certaines couleurs en toutes lettres (en anglais) comme "red" ou "blue". La liste des couleurs reconnues par **R** est disponible sur <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf>.

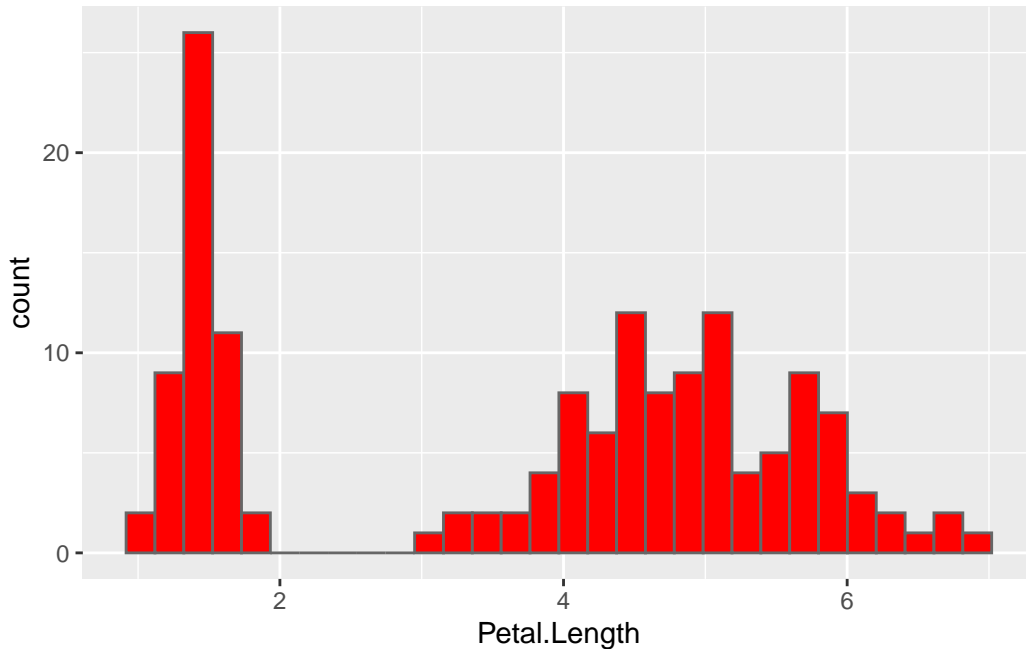
```
library(tidyverse)
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram(colour = "red", fill = "blue")
```



16.2 Couleurs RVB et code hexadécimal

En informatique, les couleurs sont usuellement codées en Rouge/Vert/Bleu (voir https://fr.wikipedia.org/wiki/Rouge_vert_bleu) et représentées par un code hexadécimal à 6 caractères (chiffres 0 à 9 et/ou lettres A à F), précédés du symbole #. Ce code est reconnu par **R**. On pourra par exemple indiquer "#FF0000" pour la couleur rouge ou "#666666" pour un gris foncé. Le code hexadécimal des différentes couleurs peut s'obtenir aisément sur internet, de nombreux sites étant consacrés aux palettes de couleurs.

```
ggplot(iris) +  
  aes(x = Petal.Length) +  
  geom_histogram(colour = "#666666", fill = "#FF0000")
```



Parfois, au lieu du code hexadécimal, les couleurs RVB sont indiquées avec trois chiffres entiers compris entre 0 et 255. La conversion en hexadécimal se fait avec la fonction `grDevices::rgb()`.

```
rgb(255, 0, 0, maxColorValue = 255)
```

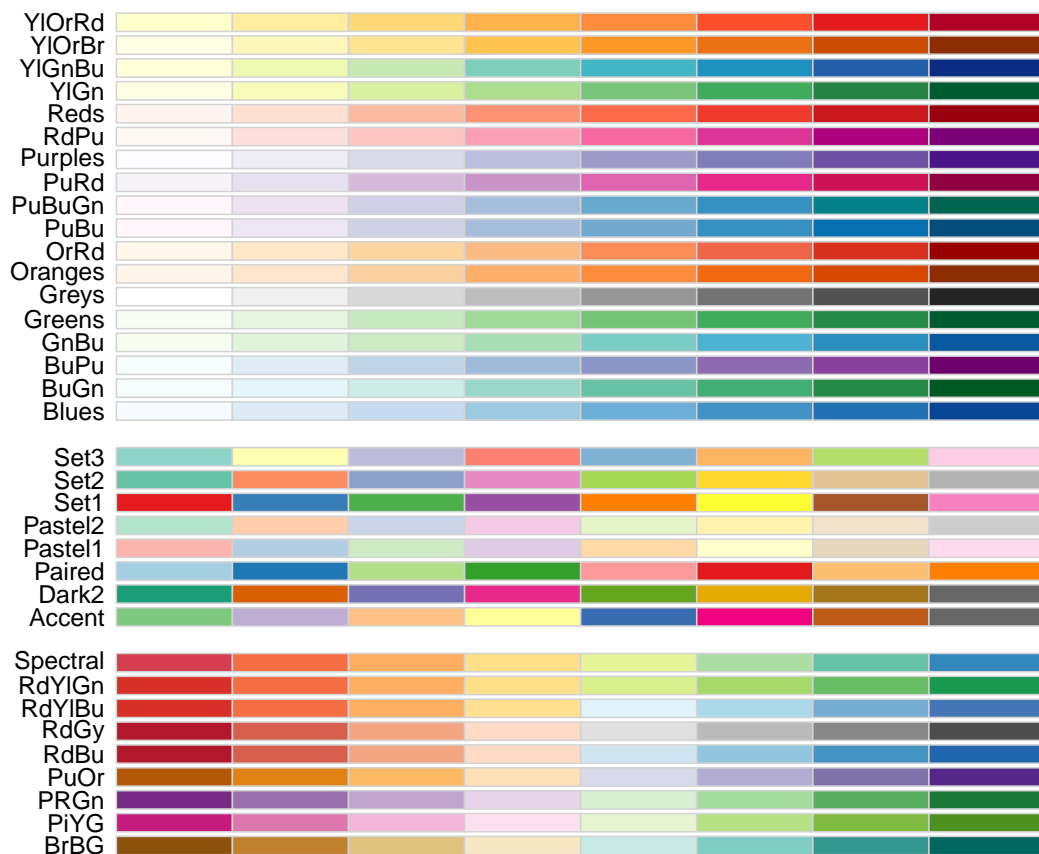
```
[1] "#FF0000"
```

16.3 Palettes de couleurs

16.3.1 Color Brewer

Le projet **Color Brewer** a développé des palettes cartographiques, à la fois séquentielles, divergentes et catégorielles, présentées en détail sur <http://colorbrewer2.org/>. Pour chaque type de palette, et en fonction du nombre de classes, est indiqué sur ce site si la palette est adaptée aux personnes souffrant de daltonisme, si elle est rendra correctement sur écran, en cas d'impression couleur et en cas d'impression en noir et blanc.

Voici un aperçu des différentes palettes disponibles :



L'extension `{RColorBrewer}` permet d'accéder à ces palettes sous **R**.

Si on utilise `{ggplot2}`, les palettes Color Brewer sont directement disponibles via les fonctions `ggplot2::scale_fill_brewer()` et `ggplot2::scale_colour_brewer()`.

🔥 Mise en garde

Les palettes Color Brewer sont seulement implémentées pour des variables catégorielles. Il est cependant possible de les utiliser avec des variables continues en les combinant

avec `ggplot2::scale_fill_gradientn()` ou `ggplot2::scale_colour_gradientn()` (en remplaçant "Set1" par le nom de la palette désirée) :

```
scale_fill_gradientn(values = RColorBrewer::brewer.pal(6, "Set1"))
```

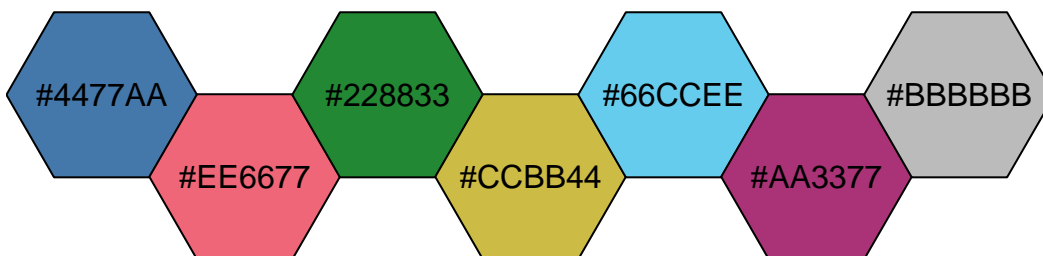
16.3.2 Palettes de Paul Tol

Le physicien Paul Tol a développé plusieurs palettes de couleurs adaptées aux personnes souffrant de déficit de perception des couleurs (daltonisme). À titre personnel, il s'agit des palettes de couleurs que j'utilise le plus fréquemment.

Le détail de ses travaux est présenté sur <https://personal.sron.nl/~pault/>.

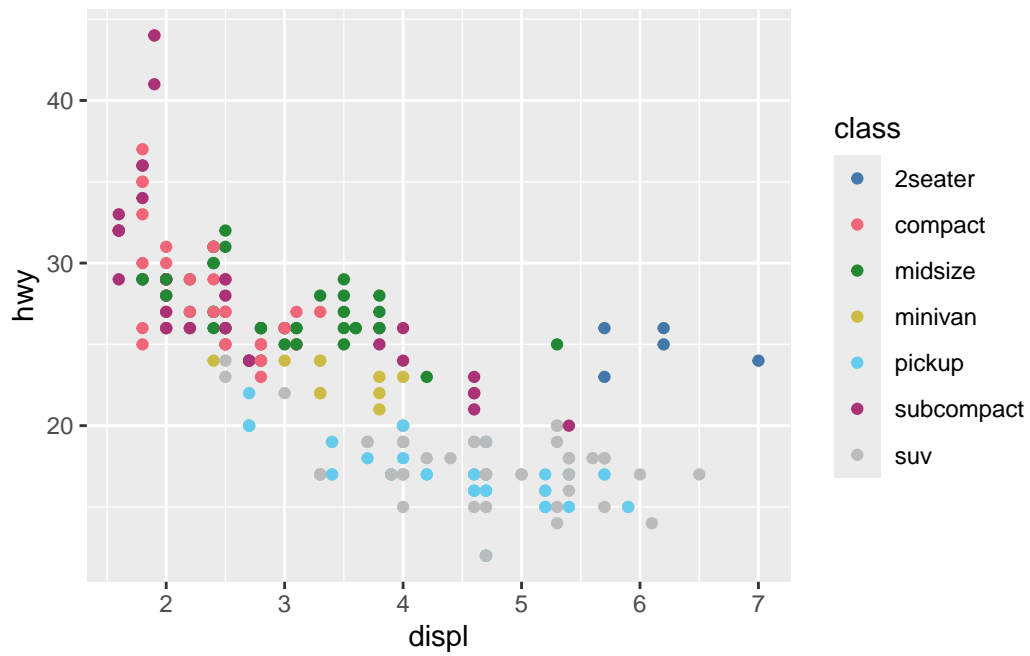
Le package `{khroma}` implémente ces palettes de couleurs proposées par Paul Tol afin de pouvoir les utiliser directement dans **R** et avec `{ggplot}`.

```
library(khroma)
plot_scheme(colour("bright")(7), colours = TRUE)
```

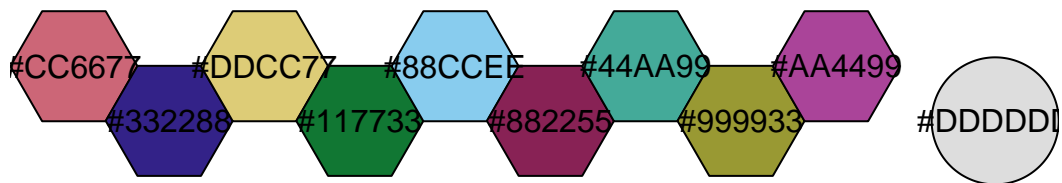


```
ggplot(mpg) +
  aes(x = displ, y = hwy, colour = class) +
```

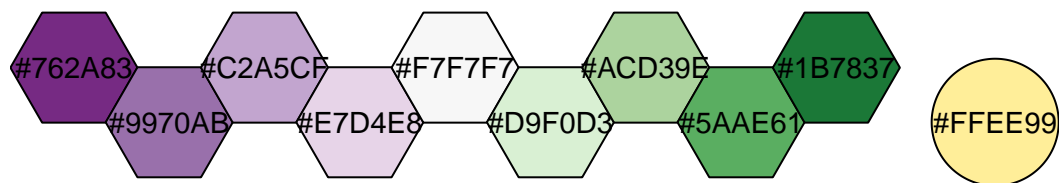
```
geom_point() +  
khroma::scale_colour_bright()
```



```
plot_scheme(colour("muted")(9), colours = TRUE)
```



```
plot_scheme(colour("PRGn")(9), colours = TRUE, size = 0.9)
```



Pour la liste complète des palettes disponibles, voir <https://packages.tesselle.org/khroma/articles/tol.html>.

16.3.3 Interface unifiée avec {paletteer}

L'extension {paletteer} vise à proposer une interface unifiée pour l'utilisation de palettes de couleurs fournies par d'autres packages (dont {khroma}, mais aussi par exemple {ggsci} qui fournit les palettes utilisées par certaines revues scientifiques). Plus de 2 500 palettes sont ainsi disponibles.

On peut afficher un aperçu des principales palettes disponibles dans {paletteer} avec la commande suivante :

```
gt::info_paletteer()
```

Pour afficher la liste complète des palettes discrètes et continues, on utilisera les commandes suivantes :

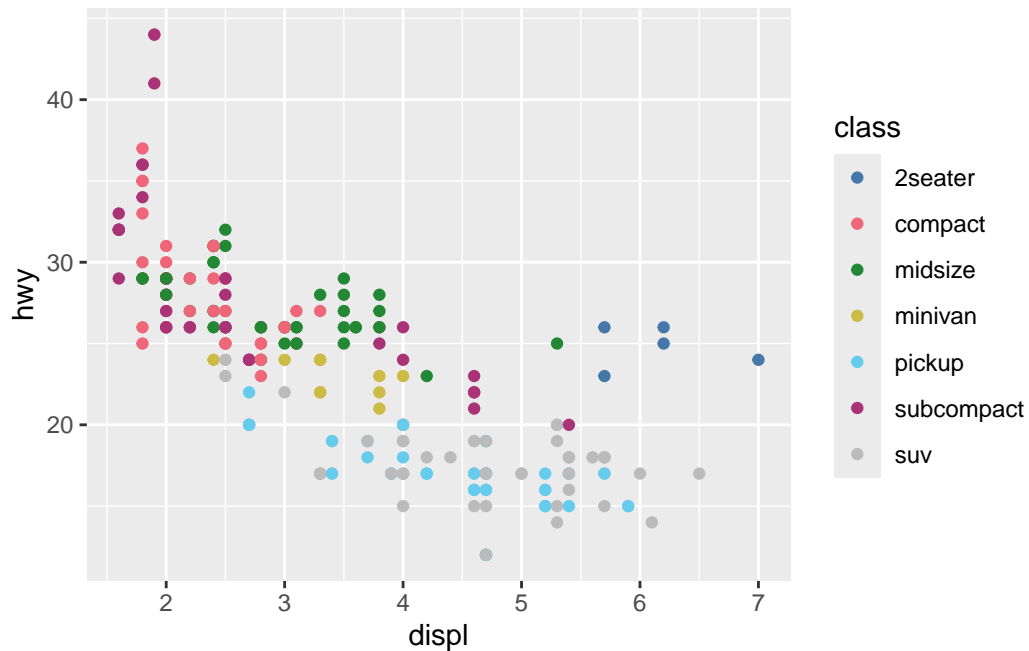
```
palettes_d_names |> View()
palettes_c_names |> View()
```

La fonction `paletteer::paletteer_d()` permet d'obtenir les codes hexadécimaux d'une palette discrète en précisant le nombre de couleurs attendues. Les fonctions `paletteer::scale_color_paletteer_d()` et `paletteer::scale_fill_paletteer_d()` permettront d'utiliser une palette donnée avec {ggplot2}.

```
library(paletteer)
paletteer_d("khroma::bright", n = 5)
```

```
<colors>
#4477AAFF #EE6677FF #228833FF #CCBB44FF #66CCEEFF
```

```
ggplot(mpg) +
  aes(x = displ, y = hwy, colour = class) +
  geom_point() +
  scale_color_paletteer_d("khroma::bright")
```

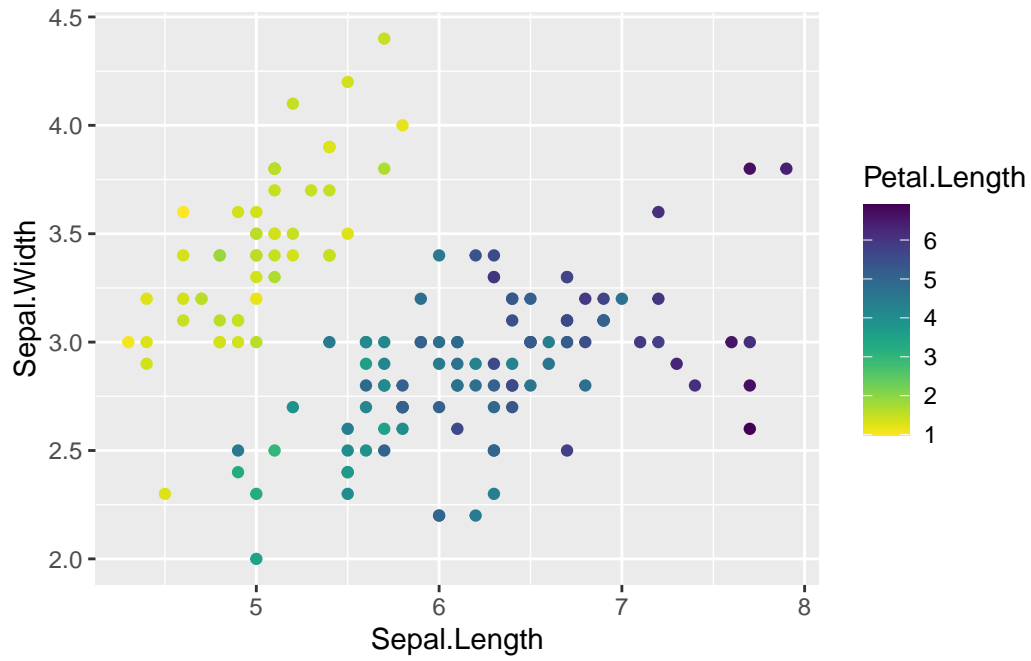



L'équivalent existe pour les palettes continues, avec `paletteer::paletteer_c()`, `paletteer::scale_color_paletteer_c()` et `paletteer::scale_fill_paletteer_c()`.

```
paletteer_c("viridis::viridis", n = 6)
```

```
<colors>
#440154FF #414487FF #2A788EFF #22A884FF #7AD151FF #FDE725FF
```

```
ggplot(iris) +
  aes(x = Sepal.Length, y = Sepal.Width, colour = Petal.Length) +
  geom_point() +
  scale_colour_paletteer_c("viridis::viridis", direction = -1)
```



partie III

Analyses

17 Graphiques avec ggplot2

Le package `{ggplot2}` fait partie intégrante du *tidyverse*. Développé par Hadley Wickham, ce package met en œuvre la grammaire graphique théorisée par Leland Wilkinson. Il devient vite indispensable lorsque l'on souhaite réaliser des graphiques un peu complexe.

17.1 Ressources

Il existe de très nombreuses ressources traitant de `{ggplot2}`.

Pour une introduction en français, on pourra se référer au chapitre [Visualiser avec ggplot2](#) de l'*Introduction à R et au tidyverse* de Julien Barnier, au chapitre [Introduction à ggplot2, la grammaire des graphiques](#) du site *analyse-R* et adapté d'une séance de cours de François Briatte, ou encore au chapitre [Graphiques](#) du cours *Logiciel R et programmation* d'Ewen Gallic.

Pour les anglophones, la référence reste encore l'ouvrage *ggplot2: Elegant Graphics for Data Analysis* d'Hadley Wickham lui-même, dont la troisième édition est librement accessible en ligne (<https://ggplot2-book.org/>). D'un point de vue pratique, l'ouvrage *R Graphics Cookbook: practical recipes for visualizing data* de Winston Chang est une mine d'informations, ouvrage là encore librement accessible en ligne (<https://r-graphics.org/>).

17.2 Les bases de ggplot2

`{ggplot2}` nécessite que les données du graphique soient sous la forme d'un tableau de données (*data.frame* ou *tibble*) au format *tidy*, c'est-à-dire avec une ligne par observation et les différentes valeurs à représenter sous forme de variables du tableau.

Tous les graphiques avec `{ggplot2}` suivent une même logique. En **premier** lieu, on appellera la fonction `ggplot2::ggplot()` en lui passant en paramètre le fichier de données.

`{ggplot2}` nomme *esthétiques* les différentes propriétés visuelles d'un graphique, à savoir l'axe des x (`x`), celui des y (`y`), la couleur des lignes (`colour`), celle de remplissage des polygones (`fill`), le type de lignes (`linetype`), la forme des points (`shape`), etc. Une représentation graphique consiste donc à représenter chacune de nos variables d'intérêt selon une esthétique

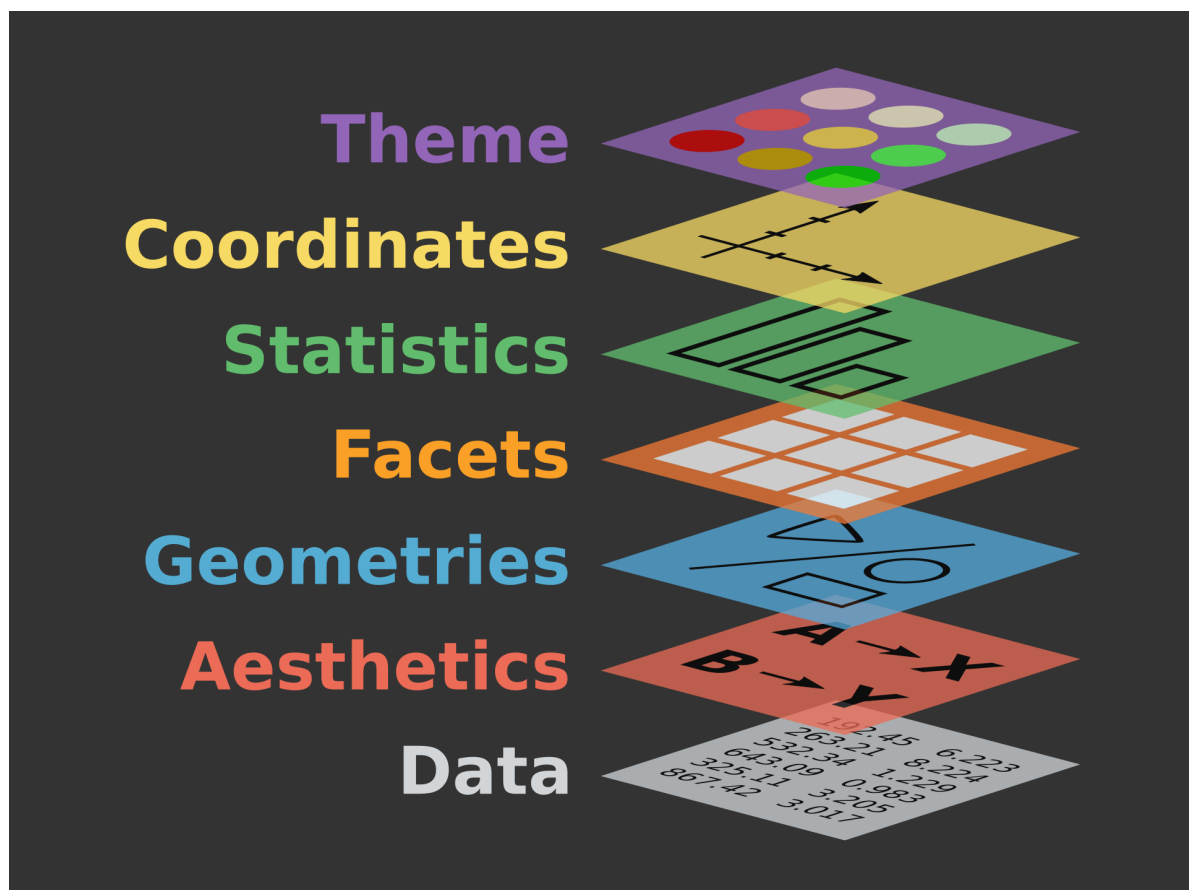


Figure 17.1: La grammaire des graphiques

donnée. En **second** lieu, on appellera donc la fonction `ggplot2::aes()` pour indiquer la correspondance entre les variables de notre fichier de données et les esthétiques du graphique.

A minima, il est nécessaire d'indiquer en **troisième** lieu une *géométrie*, autrement dit la manière dont les éléments seront représentés visuellement. À chaque géométrie correspond une fonction commençant par `geom_`, par exemple `ggplot2::geom_point()` pour dessiner des points, `ggplot2::geom_line()` pour des lignes, `ggplot2::geom_bar()` pour des barres ou encore `ggplot2::geom_area()` pour des aires. Il existe de nombreuses géométries différentes¹, chacune prenant en compte certaines esthétiques, certaines étant requises pour cette géométrie et d'autres optionnelles. La liste des esthétiques prises en compte par chaque géométrie est indiquée dans l'aide en ligne de cette dernière.

Voici un exemple minimal de graphique avec `{ggplot2}` :

```
library(ggplot2)
p <-
  ggplot(iris) +
  aes(
    x = Petal.Length,
    y = Petal.Width,
    colour = Species
  ) +
  geom_point()
p
```

¹On trouvera une liste dans la *cheat sheet* de `{ggplot2}`, voir Section 17.3.

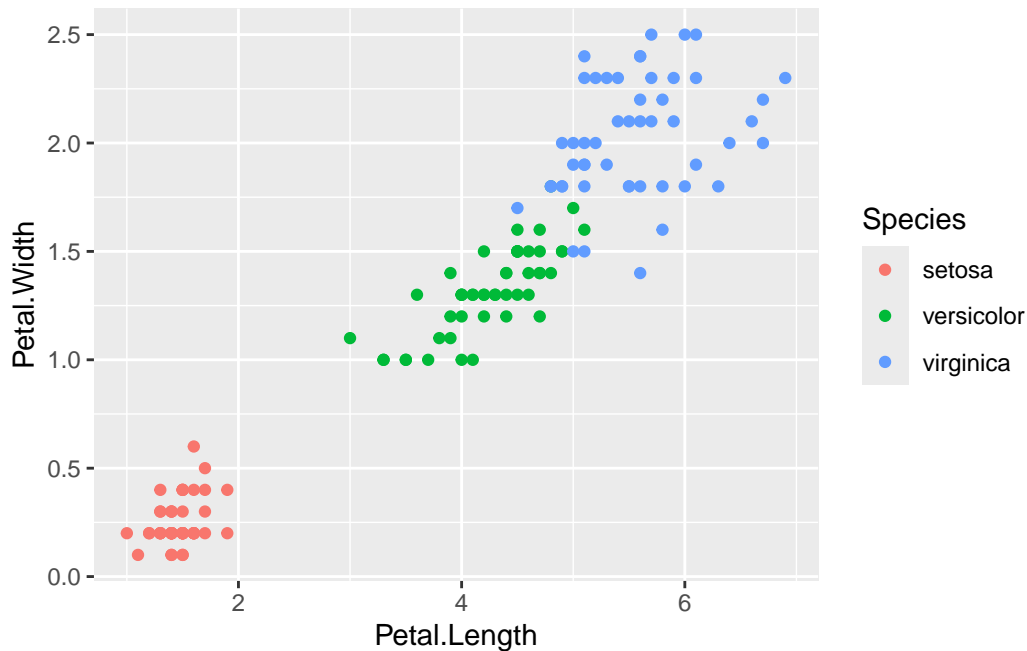


Figure 17.2: Un exemple simple de nuage de points avec `ggplot2`

! Syntaxe additive

Le développement de `{ggplot2}` a débuté avant celui du *tidyverse* et la généralisation du *pipe*. Dès lors, on ne sera pas étonné que la syntaxe de `{ggplot2}` n'ait pas recours à ce dernier mais repose sur une approche *additive*. Un graphique est dès lors initialisé avec la fonction `ggplot2::ggplot()` et l'on ajoutera successivement des éléments au graphique en appelant différentes fonctions et en utilisant l'opérateur `+`.

Il est ensuite possible de personnaliser de nombreux éléments d'un graphique et notamment :

- les *étiquettes* ou *labs* (titre, axes, légendes) avec `ggplot2::ggtitle()`, `ggplot2::xlab()`, `ggplot2::ylab()` ou encore la fonction plus générique `ggplot2::labs()` ;
- les *échelles* (*scales*) des différentes esthétiques avec les fonctions commençant par `scale_` ;
- le système de *coordonnées* avec les fonctions commençant par `coord_` ;
- les *facettes* (*facets*) avec les fonctions commençant par `facet_` ;
- la *légende* (*guides*) avec les fonctions commençant par `guide_` ;
- le *thème* du graphiques (mise en forme des différents éléments) avec `ggplot2::theme()`.

```
p +
  labs(
    x = "Longueur du pétale",
```

```

    y = "Largeur du pétale",
    colour = "Espèce"
) +
ggtitle(
  "Relation entre longueur et largeur des pétales",
  subtitle = "Jeu de données Iris"
) +
scale_x_continuous(breaks = 1:7) +
scale_y_continuous(
  labels = scales::label_number(decimal.mark = ",")
) +
coord_equal() +
facet_grid(cols = vars(Species)) +
guides(
  color = guide_legend(nrow = 2)
) +
theme_light() +
theme(
  legend.position = "bottom",
  axis.title = element_text(face = "bold")
)

```

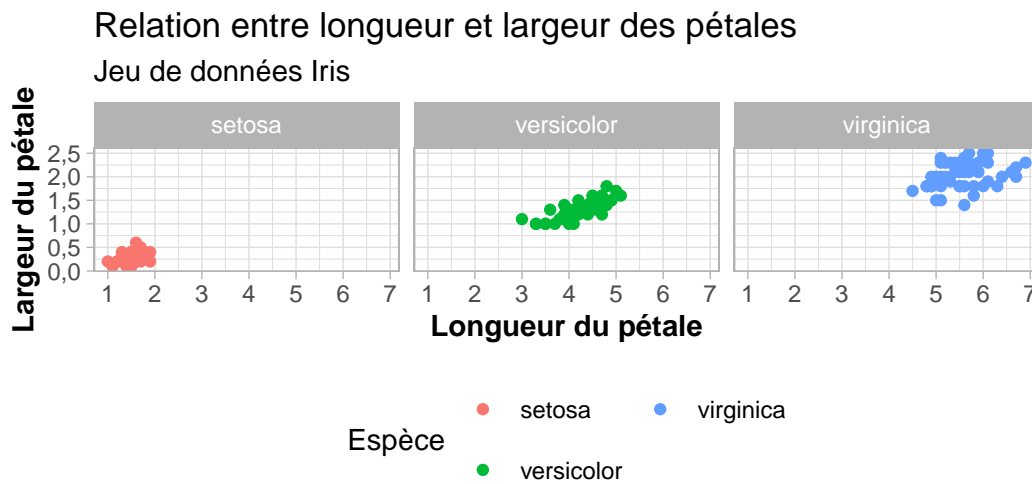



Figure 17.3: Un exemple avancé de nuage de points avec `ggplot2`

Pour visualiser chaque étape du code, vous pouvez consulter le diaporama suivant : <https://larmarange.github.io/guide-R/analyses/ressources/flipbook-ggplot2.html>

17.3 Cheatsheet



Figure 17.4: Cheatsheet `ggplot2`

17.4 Exploration visuelle avec esquisse

Le package `{esquisse}` propose un *addin* offrant une interface visuelle pour la création de graphiques `{ggplot2}`. Après installation du package, on pourra lancer `{esquisse}` directement à partir du menu *addins* de **RStudio**.

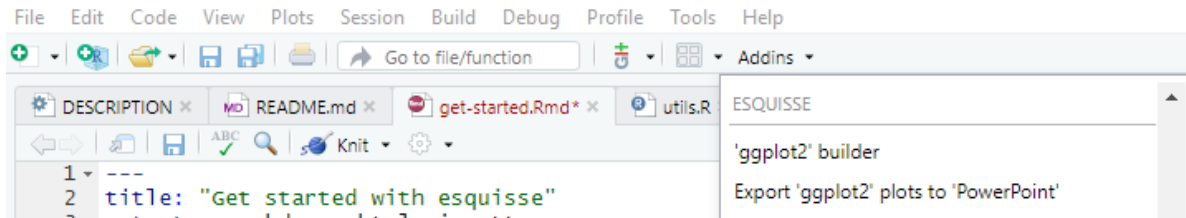


Figure 17.5: Lancement d'*esquisse* à partir du menu *Addins* de **RStudio**

Au lancement de l'*addin*, une interface permettra de choisir le tableau de données à partir duquel générer le graphique. Le plus simple est de choisir un tableau présent dans l'environnement. Mais `{esquisse}` offre aussi la possibilité d'importer des fichiers externes, voir de procéder à quelques modifications des données.

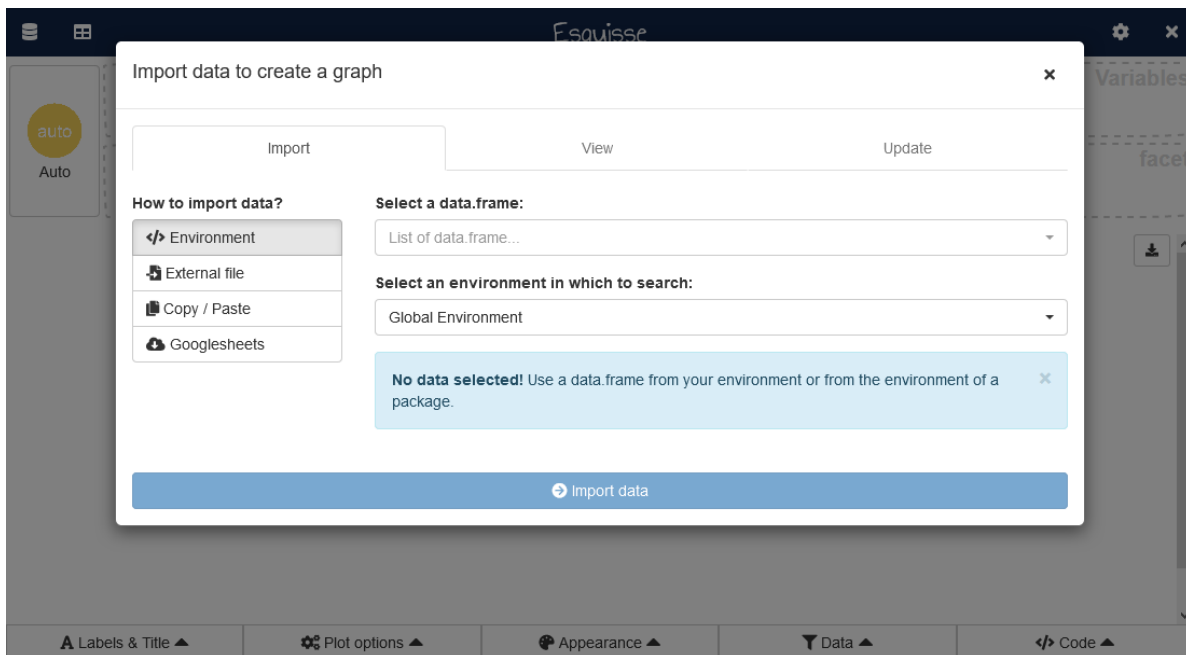


Figure 17.6: Import de données au lancement d'*esquisse*

Le principe général d'`{esquisse}` consiste à associer des variables à des esthétiques par glis-

ser/déposer². L'outil déterminera automatiquement une géométrie adaptée en fonction de la nature des variables (continues ou catégorielles). Un clic sur le nom de la géométrie en haut à gauche permet de sélectionner une autre géométrie.

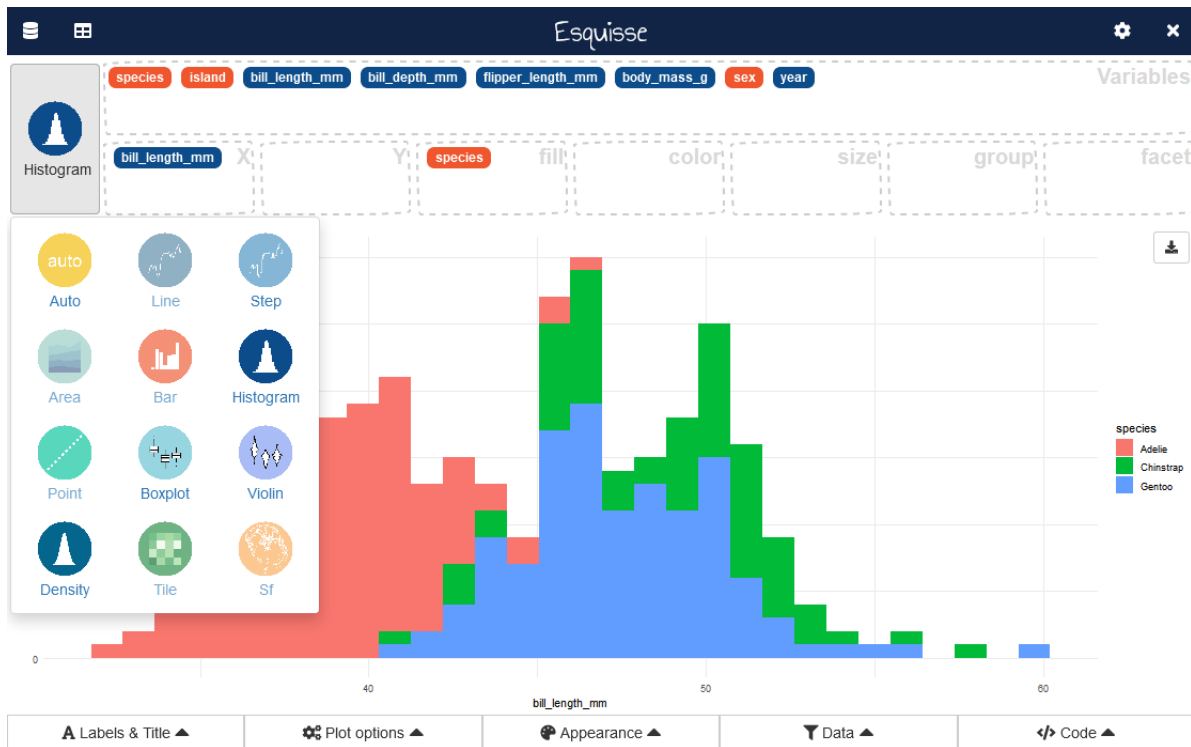


Figure 17.7: Choix d'une géométrie dans **esquisse**

Les menus situés en bas de l'écran permettent d'ajouter/modifier des étiquettes, de modifier certaines options du graphique, de modifier les échelles de couleurs et l'apparence du graphique, et de filtrer les observations incluses dans le graphique.

Le menu **Code** permet de récupérer le code correspondant au graphique afin de pouvoir le copier/coller dans un script.

`{esquisse}` offre également la possibilité d'exporter le graphique obtenu dans différents formats.

²Si une esthétique n'est pas visible à l'écran, on pourra cliquer en haut à droite sur l'icône en forme de roue dentée afin de choisir d'afficher plus d'esthétiques.

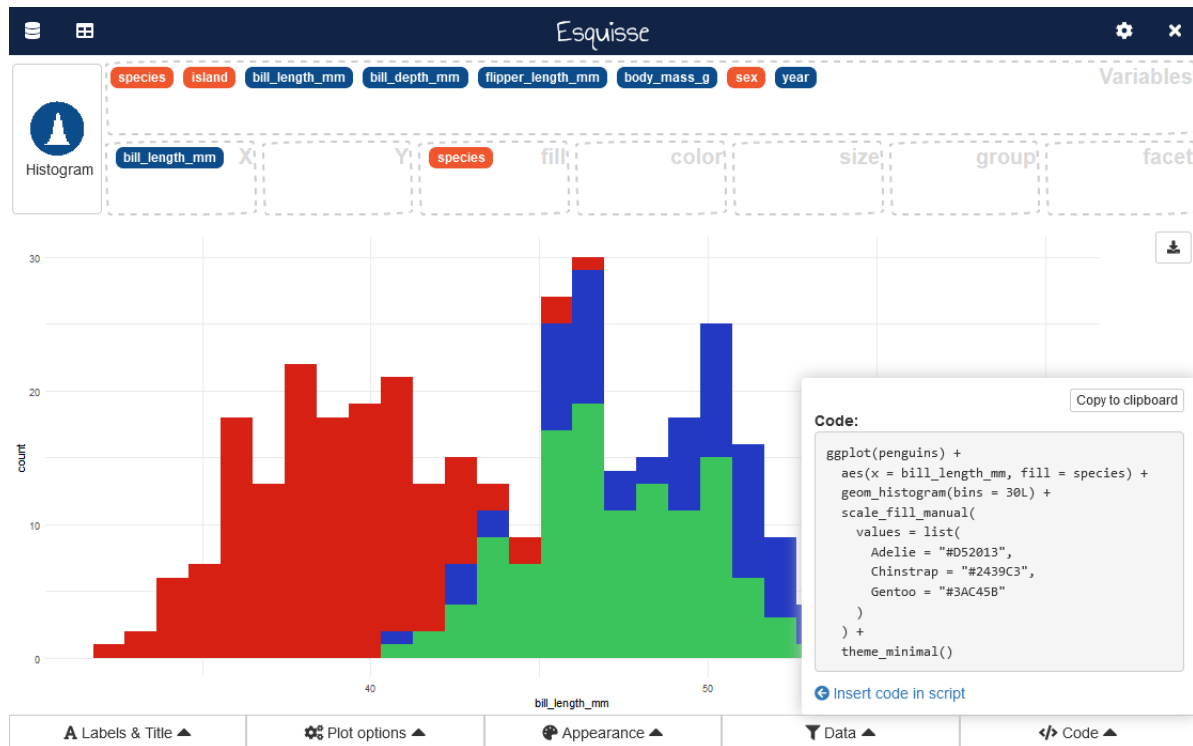


Figure 17.8: Obtenir le code du graphique obtenu avec `esquisse`

17.5 webin-R

L'utilisation d'`{esquisse}` est présentée dans le webin-R #03 (*statistiques descriptives avec gtsummary et esquisse*) sur [YouTube](#).

https://youtu.be/oEF_8GXyP5c

`{ggplot2}` est abordé plus en détails dans le webin-R #08 (*ggplot2 et la grammaire des graphiques*) sur [YouTube](#).

https://youtu.be/msnwENny_cg

17.6 Combiner plusieurs graphiques

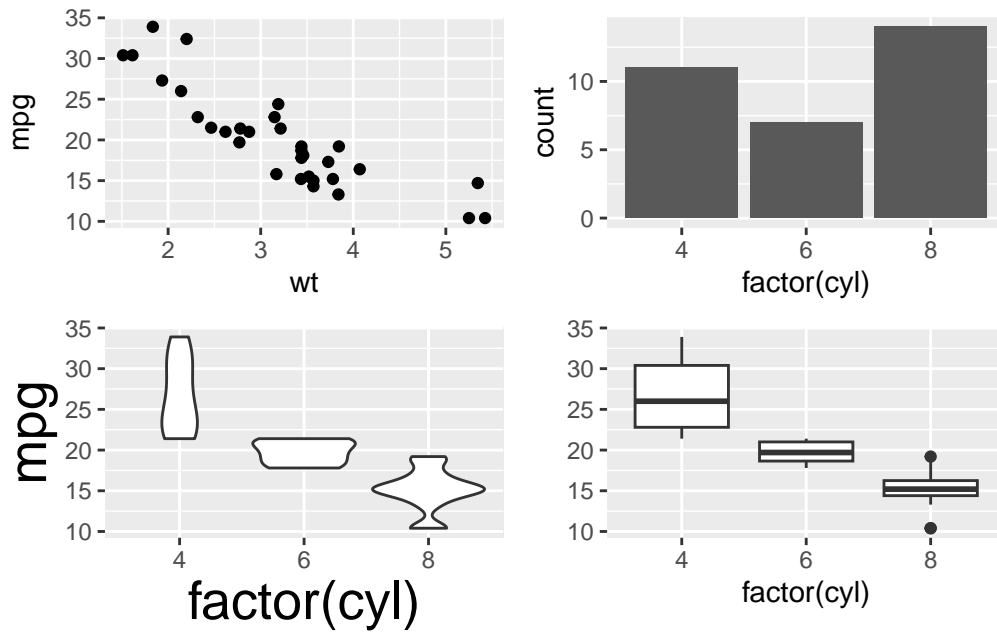
Plusieurs packages proposent des fonctions pour combiner ensemble des graphiques `{ggplot2}`, comme `{patchwork}`, `{ggpubr}`, `{egg}` ou `{cowplot}`. Ici, nous privilégierons le package `{patchwork}` car, bien qu'il ne fasse pas partie du *tidyverse*, est développé et maintenant par les mêmes auteurs que `{ggplot2}`.

Commençons par créer quelques graphiques avec `{ggplot2}`.

```
p1 <- ggplot(mtcars) +  
  aes(x = wt, y = mpg) +  
  geom_point()  
p2 <- ggplot(mtcars) +  
  aes(x = factor(cyl)) +  
  geom_bar()  
p3 <- ggplot(mtcars) +  
  aes(x = factor(cyl), y = mpg) +  
  geom_violin() +  
  theme(axis.title = element_text(size = 20))  
p4 <- ggplot(mtcars) +  
  aes(x = factor(cyl), y = mpg) +  
  geom_boxplot() +  
  ylab(NULL)
```

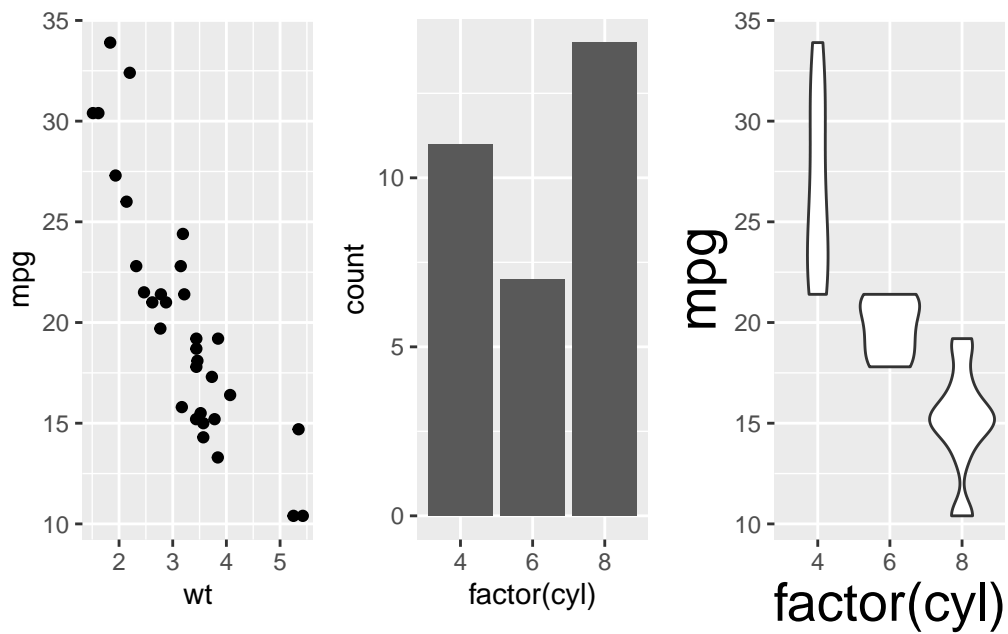
Le symbole `+` permet de combiner des graphiques entre eux. Le package `{patchwork}` déterminera le nombre de lignes et de colonnes en fonction du nombre de graphiques. On pourra noter que les axes des graphiques sont alignés les uns par rapports aux autres.

```
library(patchwork)  
p1 + p2 + p3 + p4
```

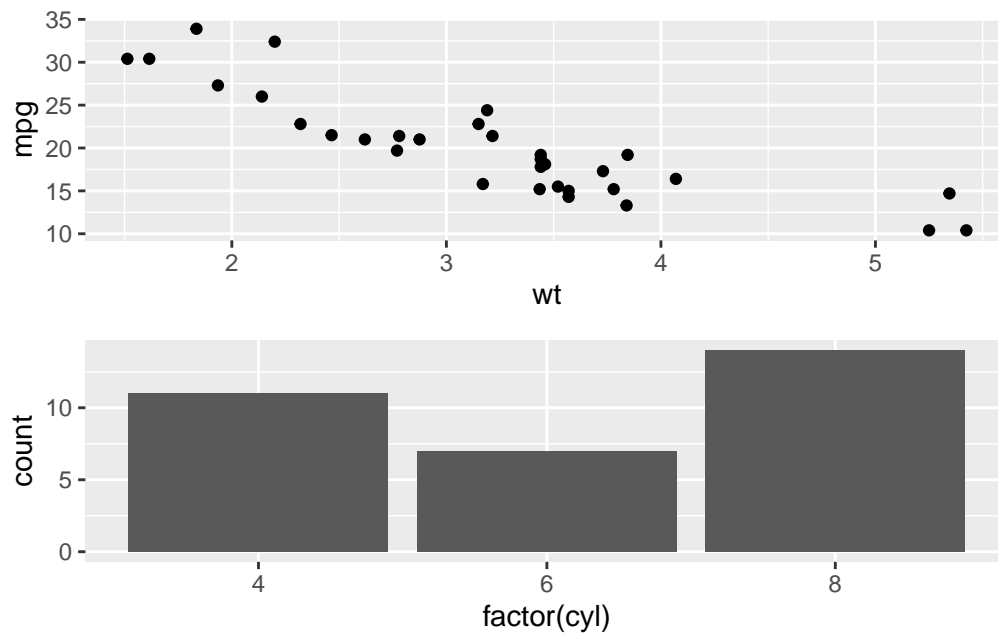


Les symboles | et / permettent d'indiquer une disposition côte à côte ou les uns au-dessus des autres.

`p1 | p2 | p3`

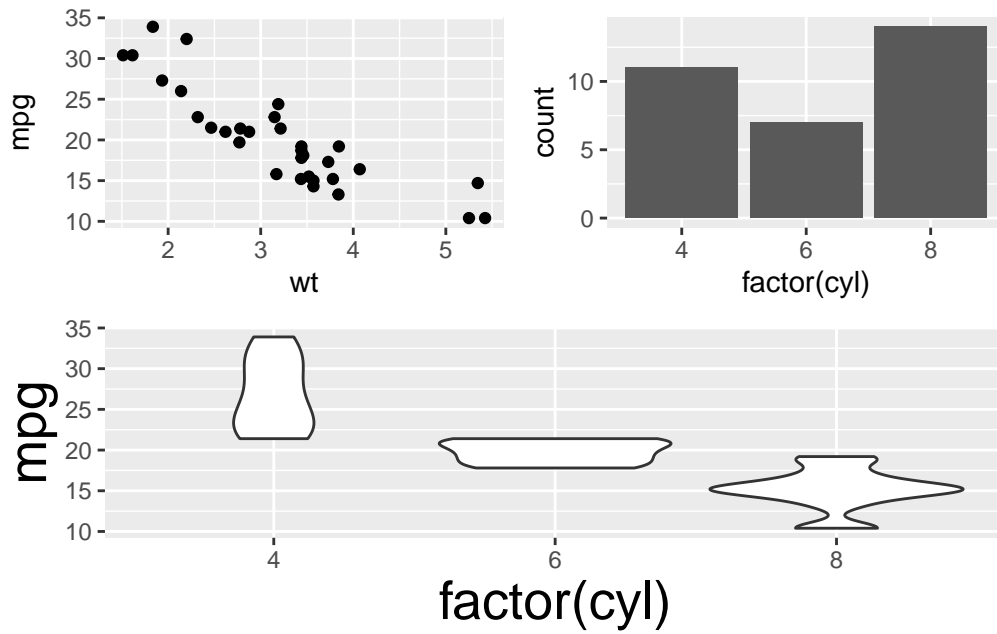


p1 / p2

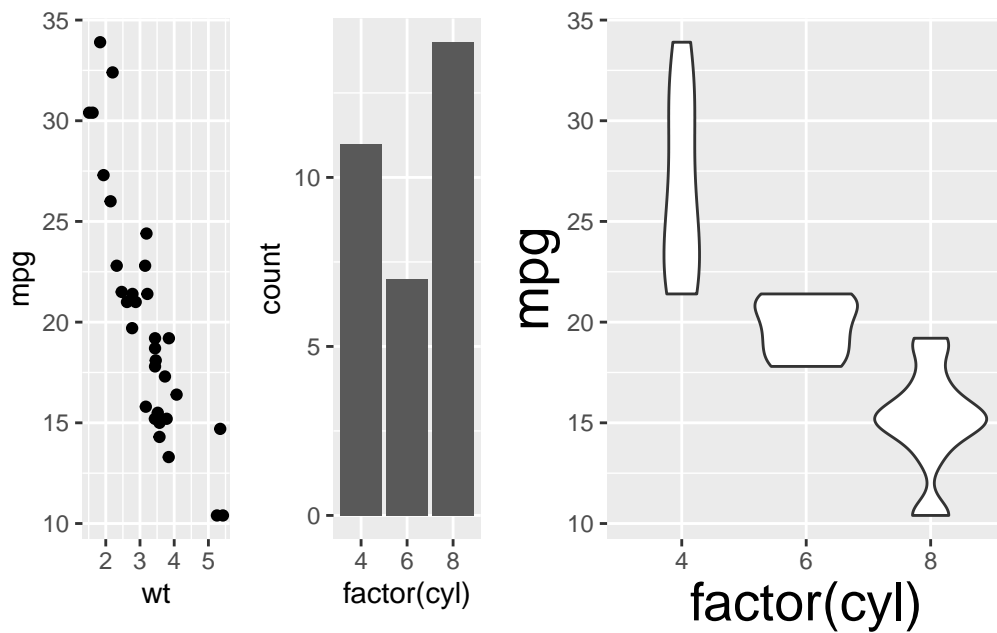


On peut utiliser les parenthèses pour indiquer des arrangements plus complexes.

(p1 + p2) / p3

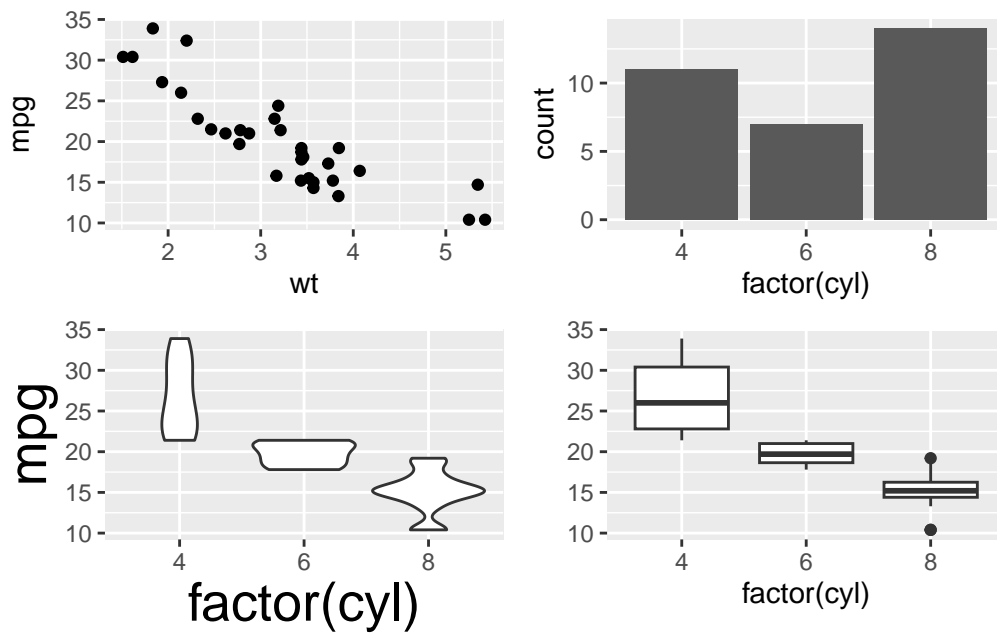


```
(p1 + p2) | p3
```



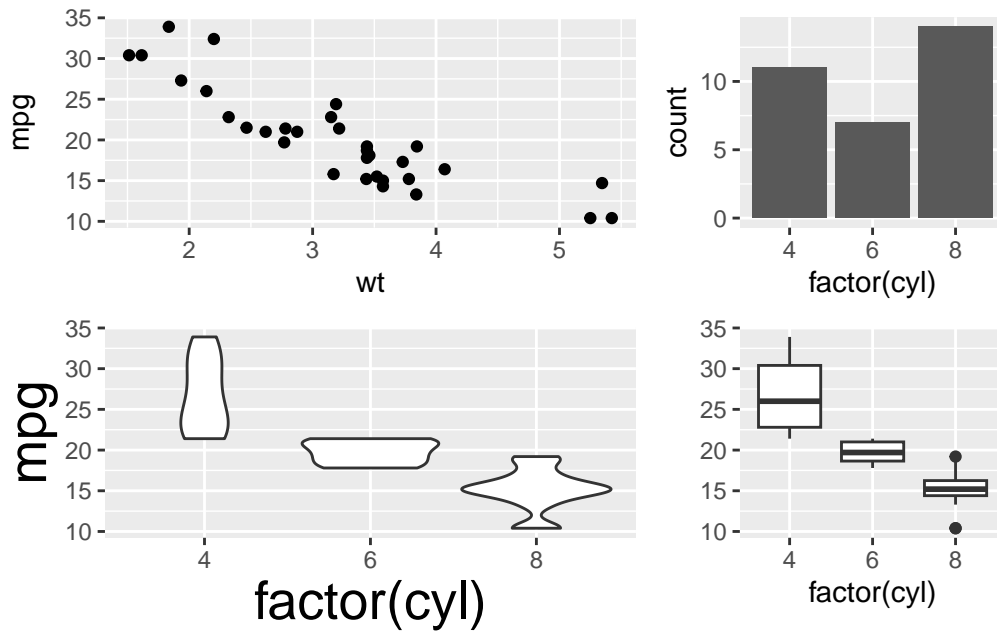
Si l'on a une liste de graphiques, on pourra appeler `patchwork::wrap_plots()`.


```
list(p1, p2, p3, p4) |>
  wrap_plots()
```



La fonction `patchwork::plot_layout()` permet de contrôler les hauteurs / largeurs relatives des lignes / colonnes.

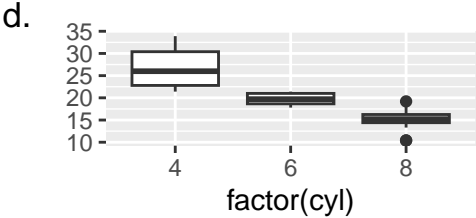
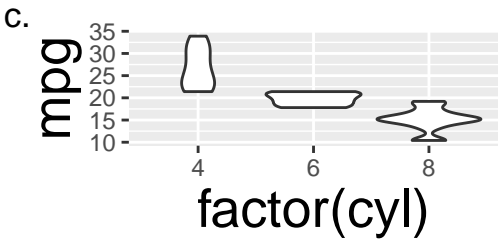
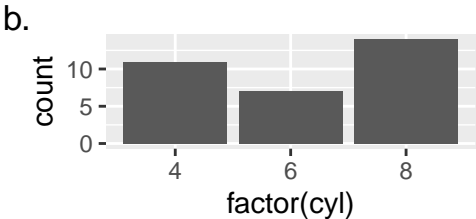
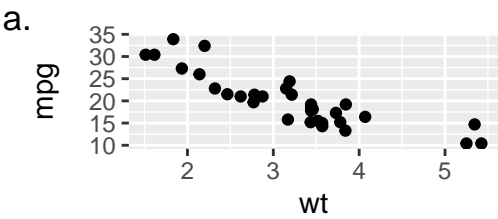
```
p1 + p2 + p3 + p4 + plot_layout(widths = c(2, 1))
```



On peut également ajouter un titre ou des étiquettes avec `patchwork::plot_annotation()`.

```
p1 + p2 + p3 + p4 +
  plot_annotation(
    title = "Titre du graphique",
    subtitle = "sous-titre",
    caption = "notes additionnelles",
    tag_levels = "a",
    tag_suffix = "."
  )
```

Titre du graphique
sous-titre



notes additionelles

18 Statistique univariée & Intervalles de confiance

On entend par statistique univariée l'étude d'une seule variable, que celle-ci soit continue (quantitative) ou catégorielle (qualitative). La statistique univariée fait partie de la statistique descriptive.

18.1 Exploration graphique

Une première approche consiste à explorer visuellement la variable d'intérêt, notamment à l'aide de l'interface proposée par `{esquisse}` (cf Section 17.4).

Nous indiquons ci-après le code correspondant aux graphiques `{ggplot2}` les plus courants.

```
library(ggplot2)
```

18.1.1 Variable continue

Un histogramme est la représentation graphique la plus commune pour représenter la distribution d'une variable, par exemple ici la longueur des pétales (variable `Petal.Length`) du fichier de données `datasets::iris`. Il s'obtient avec la géométrie `ggplot2::geom_histogram()`.

```
ggplot(iris) +  
  aes(x = Petal.Length) +  
  geom_histogram()
```

``stat_bin()` using `bins = 30`. Pick better value with `binwidth`.`

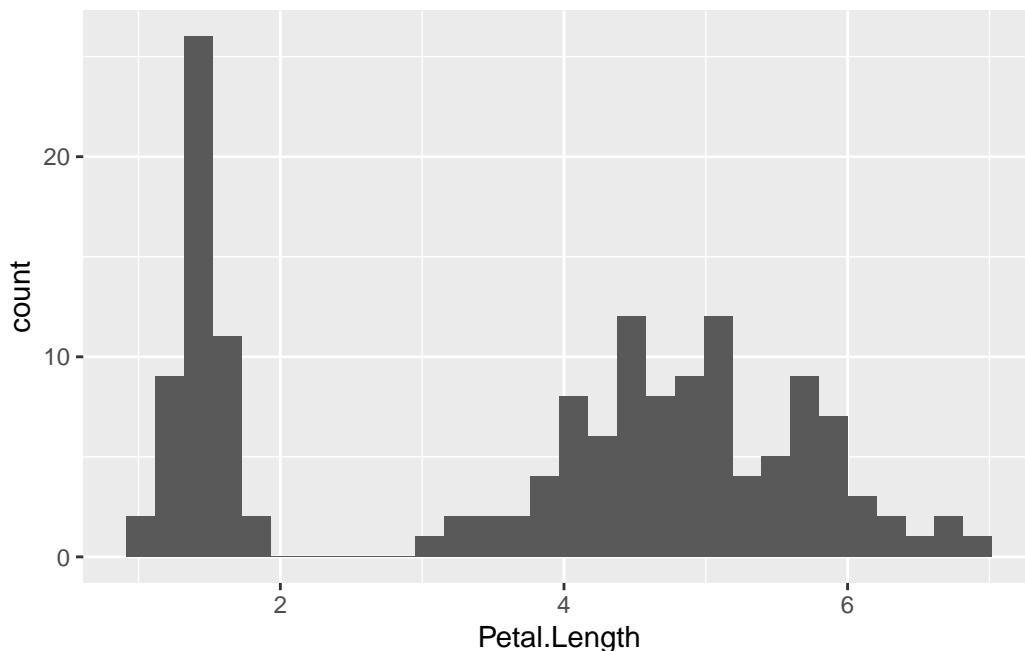


Figure 18.1: un histogramme simple

💡 Astuce

Il faut noter qu'il nous a suffi d'associer simplement la variable `Petal.Length` à l'esthétique `x`, sans avoir eu besoin d'indiquer une variable pour l'esthétique `y`.

En fait, `{ggplot2}` associe par défaut à toute géométrie une certaine statistique. Dans le cas de `ggplot2::geom_histogram()`, il s'agit de la statistique `ggplot2::stat_bin()` qui divise la variable continue en classes de même largeur et compte le nombre d'observation dans chacune. `ggplot2::stat_bin()` renvoie un certain nombre de variables calculées (la liste complète est indiquée dans la documentation dans la section *Compute variables*), dont la variable `count` qui correspond au nombre d'observations la classe. On peut associer cette variable calculée à une esthétique grâce à la fonction `ggplot2::after_stat()`, par exemple `aes(y = after_stat(count))`. Dans le cas présent, ce n'est pas nécessaire car `{ggplot2}` fait cette association automatiquement si l'on n'a pas déjà attribué une variable à l'esthétique `y`.

On peut personnaliser la couleur de remplissage des rectangles en indiquant une valeur fixe pour l'esthétique `fill` dans l'appel de `ggplot2::geom_histogram()` (et non via la fonction `ggplot2::aes()` puisqu'il ne s'agit pas d'une variable du tableau de données). L'esthétique `colour` permet de spécifier la couleur du trait des rectangles. Enfin, le paramètre `binwidth` permet de spécifier la largeur des barres.

```
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram(
    fill = "lightblue",
    colour = "black",
    binwidth = 1
  ) +
  xlab("Longeur du pétale") +
  ylab("Effectifs")
```

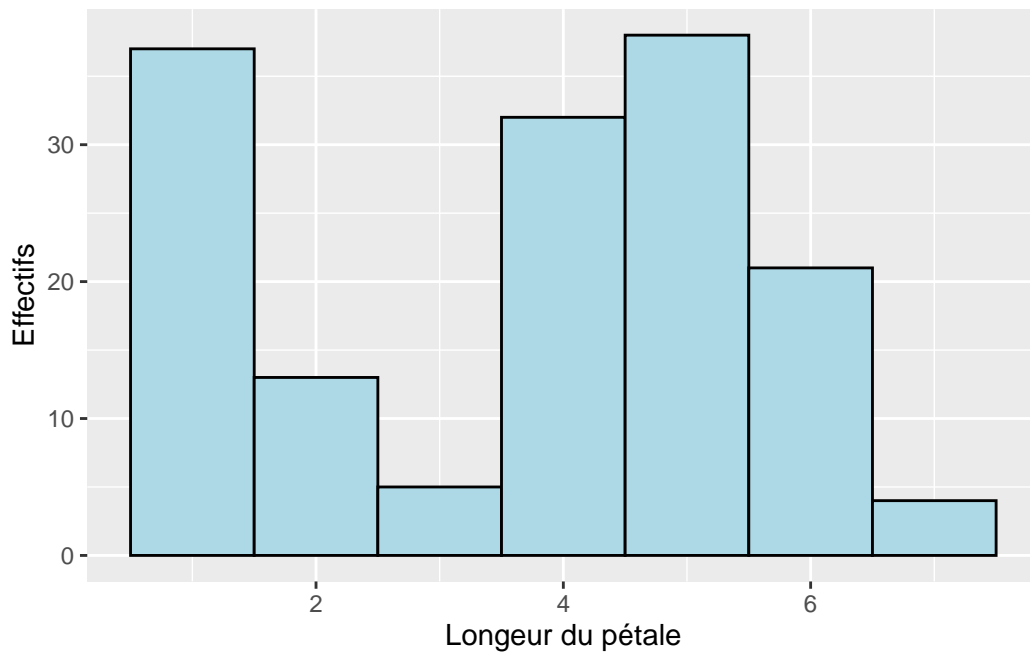


Figure 18.2: un histogramme personnalisé

On peut alternativement indiquer un nombre de classes avec `bins`.

```
ggplot(iris) +
  aes(x = Petal.Length) +
  geom_histogram(bins = 10, colour = "black")
```

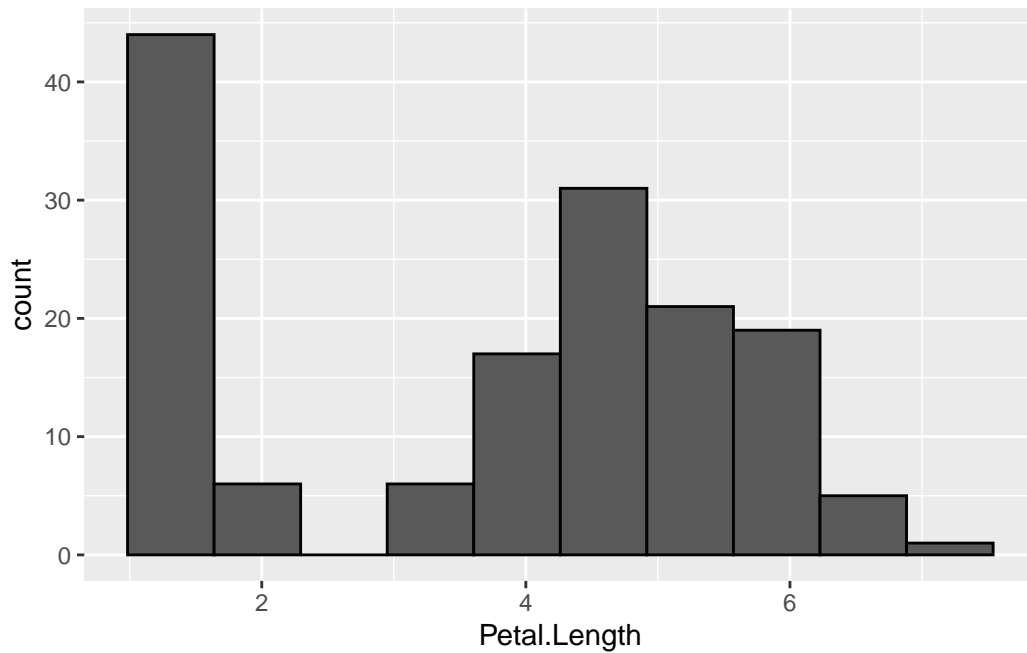


Figure 18.3: un histogramme en 10 classes

Une représentation alternative de la distribution d'une variable peut être obtenue avec une courbe de densité, dont la particularité est d'avoir une surface sous la courbe égale à 1. Une telle courbe s'obtient avec `ggplot2::geom_density()`. Le paramètre `adjust` permet d'ajuster le niveau de lissage de la courbe.

```
ggplot(iris) +  
  aes(x = Petal.Length) +  
  geom_density(adjust = .5)
```

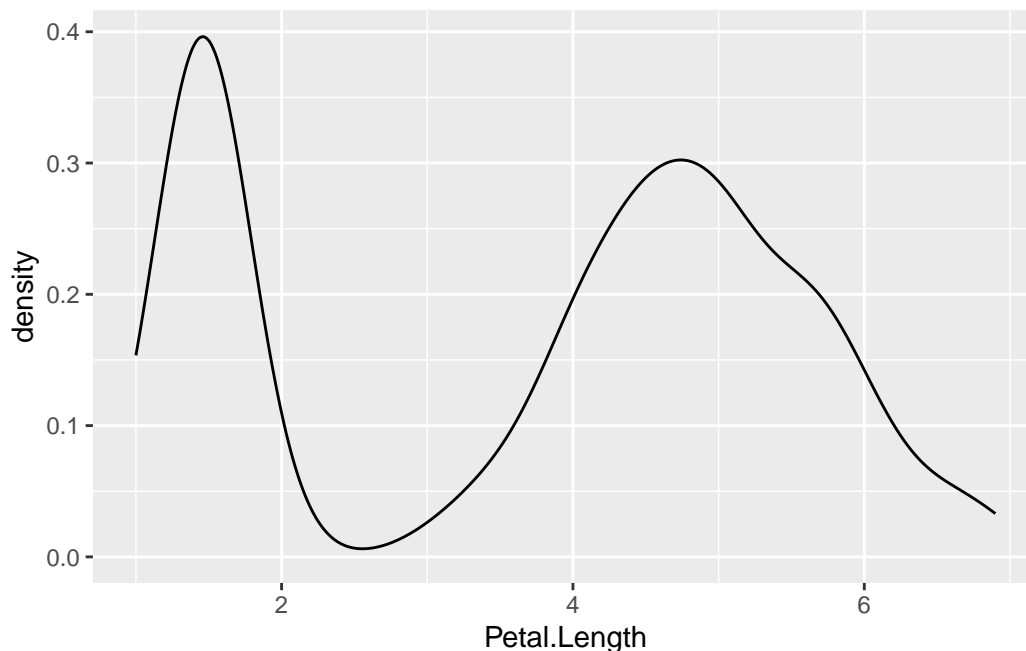


Figure 18.4: une courbe de densité

18.1.2 Variable catégorielle

Pour représenter la répartition des effectifs parmi les modalités d'une variable catégorielle, on a souvent tendance à utiliser des diagrammes en secteurs (camemberts). Or, ce type de représentation graphique est très rarement appropriée : l'œil humain préfère comparer des longueurs plutôt que des surfaces¹.

Dans certains contextes ou pour certaines présentations, on pourra éventuellement considérer un diagramme en donut, mais le plus souvent, rien ne vaut un bon vieux diagramme en barres avec `ggplot2::geom_bar()`. Prenons pour l'exemple la variable `occup` du jeu de données `hdv2003` du package `{questionr}`.

```
data("hdv2003", package = "questionr")
ggplot(hdv2003) +
  aes(x = occup) +
  geom_bar()
```

¹Voir en particulier <https://www.data-to-viz.com/caveat/pie.html> pour un exemple concret.

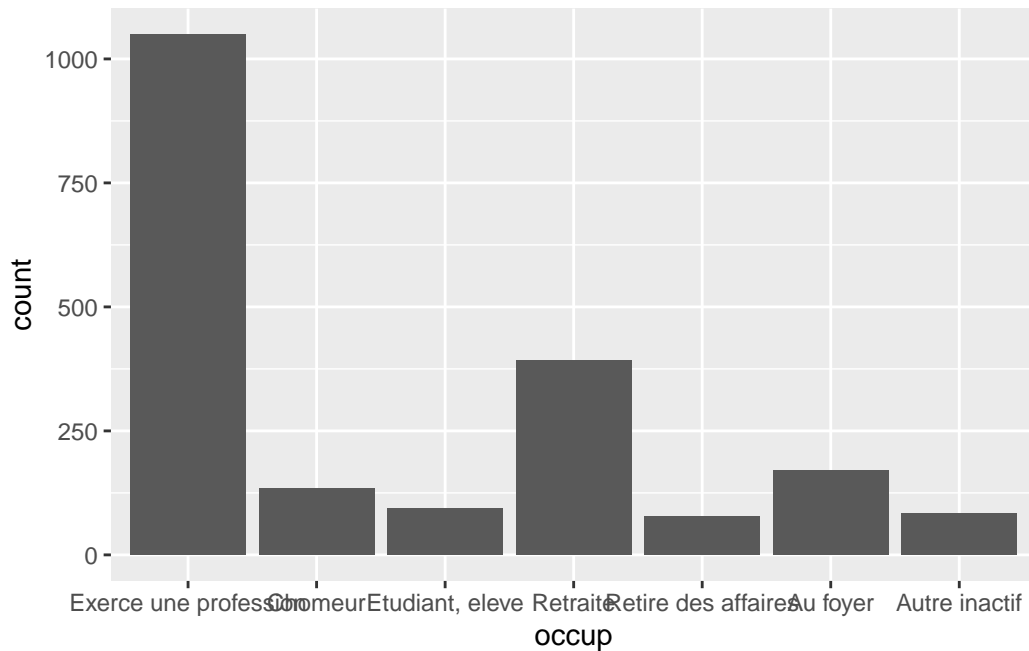


Figure 18.5: un diagramme en barres simple

💡 Astuce

Là encore, `{ggplot2}` a calculé de lui-même le nombre d'observations de chaque modalité, en utilisant cette fois la statistique `ggplot2::stat_count()`.

Si l'on souhaite représenter des pourcentages plutôt que des effectifs, le plus simple est d'avoir recours à la statistique `ggstats::stat_prop()` du package `{ggstats}`². Pour appeler cette statistique, on utilisera simplement `stat = "prop"` dans les géométries concernées.

Cette statistique, qui sera également bien utile pour des graphiques plus complexes, nécessite qu'on lui indique une esthétique `by` pour dans quels sous-groupes calculés des proportions. Ici, nous avons un seul groupe considéré et nous souhaitons des pourcentages du total. On indiquera simplement `by = 1`.

Pour formater l'axe vertical avec des pourcentages, on pourra avoir recours à la fonction `scales::label_percent()` que l'on appellera via `ggplot2::scale_y_continuous()`.

```
library(ggstats)
ggplot(hdv2003) +
  aes(x = occup, y = after_stat(prop), by = 1) +
```

²Cette statistique est également disponible via le package `{GGally}`.

```
geom_bar(stat = "prop") +
scale_y_continuous(labels = scales::label_percent())
```

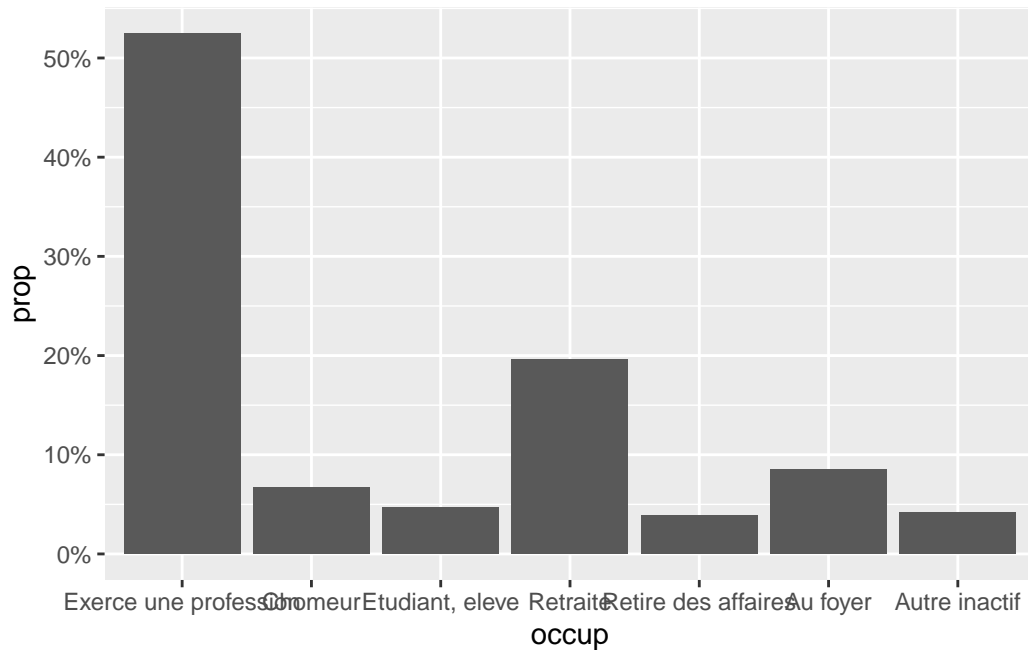


Figure 18.6: un diagramme en barres épuré

Pour une publication ou une communication, il ne faut surtout pas hésiter à **épurer** vos graphiques (*less is better!*), voire à trier les modalités en fonction de leur fréquence pour faciliter la lecture (ce qui se fait aisément avec `forcats::fct_infreq()`).

```
ggplot(hdv2003) +
  aes(x = forcats::fct_infreq(occup),
       y = after_stat(prop), by = 1) +
  geom_bar(stat = "prop",
           fill = "#4477AA", colour = "black") +
  geom_text(
    aes(label = after_stat(prop) |>
          scales::percent(accuracy = .1)),
    stat = "prop",
    nudge_y = .02
  ) +
  theme_minimal() +
  theme(
```

```

panel.grid = element_blank(),
axis.text.y = element_blank()
) +
xlab(NULL) + ylab(NULL) +
ggtitle("Occupation des personnes enquêtées")

```

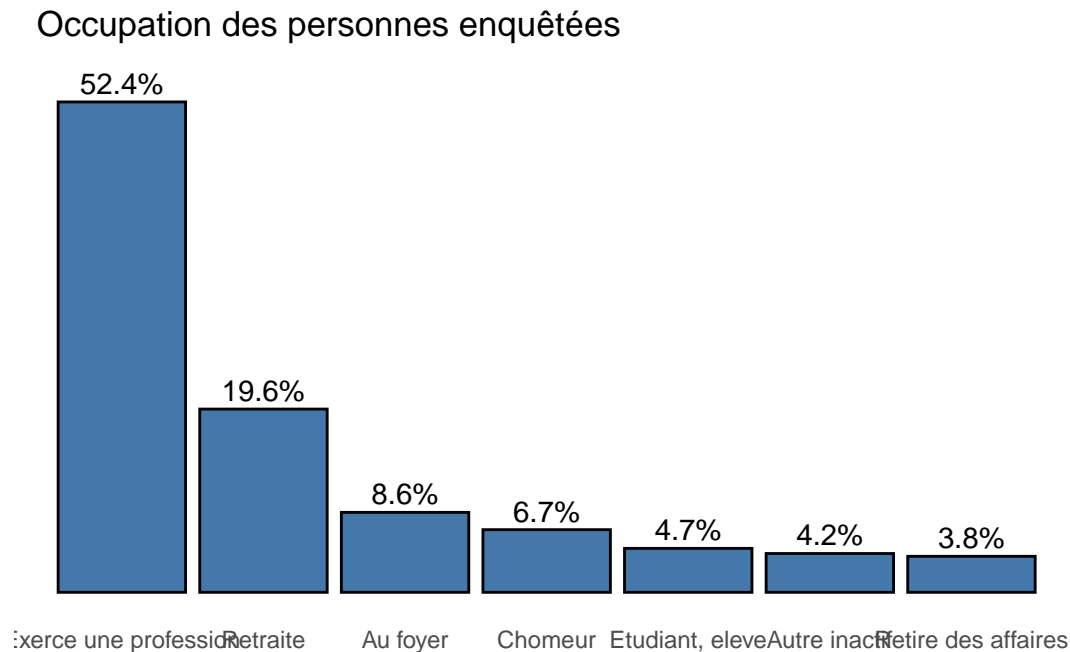


Figure 18.7: un diagramme en barres épuré

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

18.2 Tableaux et tris à plat

Le package `{gtsummary}` constitue l'une des boîtes à outils de l'analyste quantitatif, car il permet de réaliser très facilement des tableaux quasiment publiables en l'état. En matière de statistique univariées, la fonction clé est `gtsummary::tbl_summary()`.

Commençons avec un premier exemple rapide. On part d'un tableau de données et on indique, avec l'argument `include`, les variables à afficher dans le tableau statistique (si on n'indique rien, toutes les variables du tableau de données sont considérées). Il faut noter que l'argument `include` de `gtsummary::tbl_summary()` utilise la même syntaxe dite *tidy select*

que `dplyr::select()` (cf. Section 8.2.1). On peut indiquer tout autant des variables catégorielles que des variables continues.

```
library(gtsummary)
hdv2003 |>
  tbl_summary(include = c(age, occup))
```

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

Table 18.1: un tableau simple

Characteristic	N = 2,000
age	48 (35, 60)
occup	
Exerce une profession	1,049 (52%)
Chomeur	134 (6.7%)
Etudiant, eleve	94 (4.7%)
Retraite	392 (20%)
Retire des affaires	77 (3.9%)
Au foyer	171 (8.6%)
Autre inactif	83 (4.2%)

! Remarque sur les types de variables et les sélecteurs associés

`{gtsummary}` permet de réaliser des tableaux statistiques combinant plusieurs variables, l’affichage des résultats pouvant dépendre du type de variables.

Par défaut, `{gtsummary}` considère qu’une variable est **catégorielle** s’il s’agit d’un facteur, d’une variable textuelle ou d’une variable numérique ayant moins de 10 valeurs différentes.

Une variable sera considérée comme **dichotomique** (variable catégorielle à seulement deux modalités) s’il s’agit d’un vecteur logique (TRUE/FALSE), d’une variable textuelle codée **yes/no** ou d’une variable numérique codée 0/1.

Dans les autres cas, une variable numérique sera considérée comme **continue**.

Si vous utilisez des vecteurs labellisés (cf. Chapitre 12), vous devez les convertir, en amont, en facteurs ou en variables numériques. Voir l’extension `{labelled}` et les fonctions `labelled::to_factor()`, `labelled::unlabelled()` et `unclass()`.

Au besoin, il est possible de forcer le type d’une variable avec l’argument `type` de `gtsummary::tbl_summary()`.

`{gtsummary}` fournit des sélecteurs qui peuvent être utilisés dans les options des différentes fonctions, en particulier `gtsummary::all_continuous()` pour les variables continues, `gtsummary::all_dichotomous()` pour les variables dichotomiques et `gtsummary::all_categorical()` pour les variables catégorielles. Cela inclut les variables dichotomiques. Il faut utiliser `all_categorical(dichotomous = FALSE)` pour sélectionner les variables catégorielles en excluant les variables dichotomiques.

18.2.1 Thème du tableau

`{gtsummary}` fournit plusieurs fonctions préfixées `theme_gtsummary_*` permettant de modifier l’affichage par défaut des tableaux. Vous aurez noté que, par défaut, `{gtsummary}` est anglophone.

La fonction `gtsummary::theme_gtsummary_journal()` permet d’adopter les standards de certaines grandes revues scientifiques telles que *JAMA* (*Journal of the American Medical Association*), *The Lancet* ou encore le *NEJM* (*New England Journal of Medicine*).

La fonction `gtsummary::theme_gtsummary_language()` permet de modifier la langue utilisée par défaut dans les tableaux. Les options `decimal.mark` et `big.mark` permettent de définir respectivement le séparateur de décimales et le séparateur des milliers. Ainsi, pour présenter un tableau en français, on appliquera en début de script :

```
theme_gtsummary_language(  
  language = "fr",  
  decimal.mark = ",",  
  big.mark = " "  
)
```

Setting theme `language: fr`

Ce thème sera appliqué à tous les tableaux ultérieurs.

```
hdv2003 |>  
  tbl_summary(include = c(age, occup))
```

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

Table 18.2: un tableau simple en français

Caractéristique	N = 2 000
age	48 (35 – 60)
occup	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

18.2.2 Étiquettes des variables

`gtsummary`, par défaut, prends en compte les étiquettes de variables (cf. Chapitre 11), si elles existent, et sinon utilisera le nom de chaque variable dans le tableau. Pour rappel, les étiquettes de variables peuvent être manipulées avec l’extension `{labelled}` et les fonctions `labelled::var_label()` et `labelled::set_variable_labels()`.

Il est aussi possible d’utiliser l’option `label` de `gtsummary::tbl_summary()` pour indiquer des étiquettes personnalisées.

```
hdv2003 |>
  labelled::set_variable_labels(
    occup = "Occupation actuelle"
  ) |>
  tbl_summary(
    include = c(age, occup, heures.tv),
    label = list(age ~ "Âge médian")
  )
```

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

Table 18.3: un tableau étiqueté

Caractéristique	N = 2 000
Âge médian	48 (35 – 60)
Occupation actuelle	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)
heures.tv	2,00 (1,00 – 3,00)
Manquant	5

Pour modifier les modalités d’une variable catégorielle, il faut modifier en amont les niveaux du facteur correspondant.

! Remarque sur la syntaxe des options

De nombreuses options des fonctions de `{gtsummary}` peuvent s’appliquer seulement à une ou certaines variables. Pour ces options-là, `{gtsummary}` attends une formule de la forme **variables concernées ~ valeur de l’option** ou bien une liste de formules ayant cette forme.

Par exemple, pour modifier l’étiquette associée à une certaine variable, on peut utiliser l’option `label` de `gtsummary::tbl_summary()`.

```
trial |>
  tbl_summary(label = age ~ "Âge")
```

Lorsque l’on souhaite passer plusieurs options pour plusieurs variables différentes, on utilisera une `list()`.

```
trial |>
  tbl_summary(label = list(age ~ "Âge", trt ~ "Traitement"))
```

`{gtsummary}` est très flexible sur la manière d’indiquer la ou les variables concernées. Il peut s’agir du nom de la variable, d’une chaîne de caractères contenant le nom de la variable, ou d’un vecteur contenant le nom de la variable. Les syntaxes ci-dessous sont ainsi équivalentes.

```
trial |>
  tbl_summary(label = age ~ "Âge")
trial |>
  tbl_summary(label = "age" ~ "Âge")
v <- "age"
trial |>
  tbl_summary(label = v ~ "Âge")
```

Pour appliquer le même changement à plusieurs variables, plusieurs syntaxes sont acceptées pour lister plusieurs variables.

```
trial |>
  tbl_summary(label = c("age", "trt") ~ "Une même étiquette")
trial |>
  tbl_summary(label = c(age, trt) ~ "Une même étiquette")
```

Il est également possible d'utiliser la syntaxe `{tidyselect}` et les sélecteurs de `{tidyselect}` comme `tidyselect::everything()`, `tidyselect::starts_with()`, `tidyselect::contains()` ou `tidyselect::all_of()`. Ces différents sélecteurs peuvent être combinés au sein d'un `c()`.

```
trial |>
  tbl_summary(
    label = everything() ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = starts_with("a") ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = c(everything(), -age, -trt) ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = age:trt ~ "Une même étiquette"
  )
```

Bien sûr, il est possible d'utiliser les sélecteurs propres à `{gtsummary}`.


```
trial |>
  tbl_summary(
    label = all_continuous() ~ "Une même étiquette"
  )
trial |>
  tbl_summary(
    label = list(
      all_continuous() ~ "Variable continue",
      all_dichotomous() ~ "Variable dichotomique",
      all_categorical(dichotomous = FALSE) ~ "Variable catégorielle"
    )
  )
```

Enfin, si l'on ne précise rien à gauche du ~, ce sera considéré comme équivalent à `everything()`. Les deux syntaxes ci-dessous sont donc équivalentes.

```
trial |>
  tbl_summary(label = ~ "Une même étiquette")
trial |>
  tbl_summary(
    label = everything() ~ "Une même étiquette"
  )
```

18.2.3 Statistiques affichées

Le paramètre `statistic` permet de sélectionner les statistiques à afficher pour chaque variable. On indiquera une chaîne de caractères dont les différentes statistiques seront indiquées entre accolades (`{}`).

Pour une **variable continue**, on pourra utiliser `{median}` pour la médiane, `{mean}` pour la moyenne, `{sd}` pour l'écart type, `{var}` pour la variance, `{min}` pour le minimum, `{max}` pour le maximum, ou encore `{p##}` (en remplaçant `##` par un nombre entier entre 00 et 100) pour le percentile correspondant (par exemple `p25` et `p75` pour le premier et le troisième quartile). Utilisez `gtsummary::all_continuous()` pour sélectionner toutes les variables continues.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    statistic =
      all_continuous() ~ "Moy. : {mean} [min-max : {min} - {max}]"
  )
```

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

Table 18.4: statistiques personnalisées pour une variable continue

Caractéristique	N = 2 000
age	Moy. : 48 [min-max : 18 - 97]
heures.tv	Moy. : 2,25 [min-max : 0,00 - 12,00]
Manquant	5

Il est possible d'afficher des statistiques différentes pour chaque variable.

```
hdv2003 |>
tbl_summary(
  include = c(age, heures.tv),
  statistic = list(
    age ~ "Méd. : {median} [{p25} - {p75}]",
    heures.tv ~ "Moy. : {mean} ({sd})"
  )
)
```

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

Table 18.5: statistiques personnalisées pour une variable continue (2)

Caractéristique	N = 2 000
age	Méd. : 48 [35 - 60]
heures.tv	Moy. : 2,25 (1,78)
Manquant	5

Pour les variables continues, il est également possible d'indiquer le nom d'une fonction personnalisée qui prends un vecteur et renvoie une valeur résumée. Par exemple, pour afficher la moyenne des carrés :

```

moy_carres <- function(x) {
  mean(x^2, na.rm = TRUE)
}
hdv2003 |>
  tbl_summary(
    include = heures.tv,
    statistic = ~ "MC : {moy_carres}"
  )

```

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

Table 18.6: statiques personnalisées pour une variable continue (3)

Caractéristique	N = 2 000
heures.tv	MC : 8,20
Manquant	5

Pour une **variable catégorielle**, les statistiques possibles sont `{n}` le nombre d'observations, `{N}` le nombre total d'observations, et `{p}` le pourcentage correspondant. Utilisez `gtsummary::all_categorical()` pour sélectionner toutes les variables catégorielles.

```

hdv2003 |>
  tbl_summary(
    include = occup,
    statistic = all_categorical() ~ "{p} % ({n}/{N})"
  )

```

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

Table 18.7: statiques personnalisées pour une variable catégorielle

Caractéristique	N = 2 000
occup	
Exerce une profession	52 % (1 049/2 000)

Table 18.7: statiques personnalisées pour une variable catégorielle

Caractéristique	N = 2 000
Chomeur	6,7 % (134/2 000)
Etudiant, eleve	4,7 % (94/2 000)
Retraite	20 % (392/2 000)
Retire des affaires	3,9 % (77/2 000)
Au foyer	8,6 % (171/2 000)
Autre inactif	4,2 % (83/2 000)

Il est possible, pour une variable catégorielle, de trier les modalités de la plus fréquente à la moins fréquente avec le paramètre `sort`.

```
hdv2003 |>
  tbl_summary(
    include = occup,
    sort = all_categorical() ~ "frequency"
  )
```

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

Table 18.8: variable catégorielle triée par fréquence

Caractéristique	N = 2 000
occup	
Exerce une profession	1 049 (52%)
Retraite	392 (20%)
Au foyer	171 (8,6%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Autre inactif	83 (4,2%)
Retire des affaires	77 (3,9%)

Pour toutes les variables (catégorielles et continues), les statistiques suivantes sont également disponibles :

- `{N_obs}` le nombre total d'observations,

- `{N_miss}` le nombre d'observations manquantes (NA),
- `{N_nonmiss}` le nombre d'observations non manquantes,
- `{p_miss}` le pourcentage d'observations manquantes (i.e. `N_miss / N_obs`) et
- `{p_nonmiss}` le pourcentage d'observations non manquantes (i.e. `N_nonmiss / N_obs`).

18.2.4 Affichage du nom des statistiques

Lorsque l'on affiche de multiples statistiques, la liste des statistiques est regroupée dans une note de tableau qui peut vite devenir un peu confuse.

```
tbl <- hdv2003 |>
tbl_summary(
  include = c(age, heures.tv, occup),
  statistic = list(
    age ~ "{mean} ({sd})",
    heures.tv ~ "{median} [{p25} - {p75}]"
  )
)
tbl
```

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

Table 18.9: tableau par défaut

Caractéristique	N = 2 000
age	48 (17)
heures.tv	2,00 [1,00 - 3,00]
Manquant	5
occup	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

La fonction `gtsummary::add_stat_label()` permet d'indiquer le type de statistique à côté du nom des variables ou bien dans une colonne dédiée, plutôt qu'en note de tableau.

```
tbl |>
  add_stat_label()
```

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

Table 18.10: ajout du nom des statistiques

Caractéristique	N = 2 000
age, Moyenne (ET)	48 (17)
heures.tv, Médiane [EI]	2,00 [1,00 - 3,00]
Manquant	5
occup, n (%)	
Exerce une profession	1 049 (52%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (20%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

```
tbl |>
  add_stat_label(location = "column")
```

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

Table 18.11: ajout du nom des statistiques dans une colonne séparée

Caractéristique	Statistique	N = 2 000
age	Moyenne (ET)	48 (17)
heures.tv	Médiane [EI]	2,00 [1,00 - 3,00]
Manquant	n	5
occup		
Exerce une profession	n (%)	1 049 (52%)
Chomeur	n (%)	134 (6,7%)

Table 18.11: ajout du nom des statistiques dans une colonne séparée

Caractéristique	Statistique	N = 2 000
Etudiant, eleve	n (%)	94 (4,7%)
Retraite	n (%)	392 (20%)
Retire des affaires	n (%)	77 (3,9%)
Au foyer	n (%)	171 (8,6%)
Autre inactif	n (%)	83 (4,2%)

18.2.5 Forcer le type de variable

Comme évoqué plus haut, `gtsummary` détermine automatiquement le type de chaque variable. Par défaut, la variable `age` du tableau de données `trial` est traitée comme variable continue, `death` comme dichotomique (seule la valeur 1 est affichée) et `grade` comme variable catégorielle.

```
trial |>
  tbl_summary(
    include = c(grade, age, death)
  )
```

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

Table 18.12: types de variable par défaut

Caractéristique	N = 200
Grade	
I	68 (34%)
II	68 (34%)
III	64 (32%)
Age	47 (38 – 57)
Manquant	11
Patient Died	112 (56%)

Il est cependant possible de forcer un certain type avec l'argument `type`. Précision : lorsque l'on force une variable en dichotomique, il faut indiquer avec `value` la valeur à afficher (les autres sont alors masquées).

```
trial |>
  tbl_summary(
    include = c(grade, death),
    type = list(
      grade ~ "dichotomous",
      death ~ "categorical"
    ),
    value = grade ~ "III",
    label = grade ~ "Grade III"
  )
```

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

Table 18.13: types de variable personnalisés

Caractéristique	N = 200
Grade III	64 (32%)
Patient Died	
0	88 (44%)
1	112 (56%)

Il ne faut pas oublier que, par défaut, `{gtsummary}` traite les variables quantitatives avec moins de 10 valeurs comme des variables catégorielles. Prenons un exemple :

```
trial$alea <- sample(1:4, size = nrow(trial), replace = TRUE)
#| label: tbl-types-default-alea
#| tbl-cap: traitement par défaut d'une variable numérique à 4 valeurs uniques
trial |>
  tbl_summary(
    include = alea
  )
```

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

Caractéristique	N = 200
alea	
1	38 (19%)
2	54 (27%)
3	57 (29%)
4	51 (26%)

On pourra forcer le traitement de cette variable comme continue.

```
trial |>
  tbl_summary(
    include = alea,
    type = alea ~ "continuous"
  )
```

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

Table 18.15: forcer le traitement continu d'une variable numérique à 4 valeurs uniques

Caractéristique	N = 200
alea	3 (2 – 4)

18.2.6 Afficher des statistiques sur plusieurs lignes (variables continues)

Pour les variables continues, `{gtsummary}` a introduit un type de variable `"continuous2"`, qui doit être attribué manuellement via `type`, et qui permet d'afficher plusieurs lignes de statistiques (en indiquant plusieurs chaînes de caractères dans `statistic`). À noter le sélecteur dédié `gtsummary::all_continuous2()`.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    type = age ~ "continuous2",
    statistic =
      all_continuous2() ~ c(
        "{median} ({p25} – {p75})",

```

```

    "{mean} ({sd})",
    "{min} - {max}"
  )
)

```

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

Table 18.16: des statistiques sur plusieurs lignes (variables continues)

Caractéristique	N = 2 000
age	
Médiane (EI)	48 (35 - 60)
Moyenne (ET)	48 (17)
Étendue	18 - 97
heures.tv	2,00 (1,00 – 3,00)
Manquant	5

18.2.7 Mise en forme des statistiques

L'argument `digits` permet de spécifier comment mettre en forme les différentes statistiques. Le plus simple est d'indiquer le nombre de décimales à afficher. Il est important de tenir compte que plusieurs statistiques peuvent être affichées pour une même variable. On peut alors indiquer une valeur différente pour chaque statistique.

```

hdv2003 |>
  tbl_summary(
    include = c(age, occup),
    digits = list(
      all_continuous() ~ 1,
      all_categorical() ~ c(0, 1)
    )
  )

```

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

Table 18.17: personnalisation du nombre de décimales

Caractéristique	N = 2 000
age	48,0 (35,0 – 60,0)
occup	
Exerce une profession	1 049 (52,5%)
Chomeur	134 (6,7%)
Etudiant, eleve	94 (4,7%)
Retraite	392 (19,6%)
Retire des affaires	77 (3,9%)
Au foyer	171 (8,6%)
Autre inactif	83 (4,2%)

Au lieu d'un nombre de décimales, on peut indiquer plutôt une fonction à appliquer pour mettre en forme le résultat. Par exemple, `{gtsummary}` fournit les fonctions suivantes : `gtsummary::style_number()` pour les nombres de manière générale, `gtsummary::style_percent()` pour les pourcentages (les valeurs sont multipliées par 100, mais le symbole % n'est pas ajouté), `gtsummary::style_pvalue()` pour les p-valeurs, `gtsummary::style_sigfig()` qui n'affiche, par défaut, que deux chiffres significatifs, ou encore `gtsummary::style_ratio()` qui est une variante de `gtsummary::`style_sigfig()` pour les ratios (comme les *odds ratios*) que l'on compare à 1.

Il faut bien noter que ce qui est attendu par `digits`, c'est une fonction et non le résultat d'une fonction. On indiquera donc le nom de la fonction sans parenthèse, comme dans l'exemple ci-dessous (même si pas forcément pertinent ;-)).

```
hdv2003 |>
  tbl_summary(
    include = age,
    digits =
      all_continuous() ~ c(style_percent, style_sigfig, style_ratio)
  )
```

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

Table 18.18: personnalisation de la mise en forme des nombres

Caractéristique	N = 2 000
age	4 800 (35 – 60,0)

Comme `digits` s'attend à recevoir une fonction (et non le résultat) d'une fonction, on ne peut pas passer directement des arguments aux fonctions `style_*`() de `{gtsummary}`. Pour cela il faut créer une fonction à la levée :

```
trial |>
  tbl_summary(
    include = marker,
    statistic = ~ "{mean} pour 100",
    digits = ~ function(x){style_percent(x, digits = 1)}
  )
```

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

Table 18.19: passer une fonction personnalisée à `digits` (syntaxe 1)

Caractéristique	N = 200
Marker Level (ng/mL)	91,6 pour 100
Manquant	10

Depuis **R 4.1**, il existe une syntaxe raccourcie équivalente, avec le symbole `\` à la place de `function`.

```
trial |>
  tbl_summary(
    include = marker,
    statistic = ~ "{mean} pour 100",
    digits = ~ \(x){style_percent(x, digits = 1)}
  )
```

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

Table 18.20: passer une fonction personnalisée à digits (syntaxe 2)

Caractéristique	N = 200
Marker Level (ng/mL)	91,6 pour 100
Manquant	10

Une syntaxe alternative consiste à avoir recours à la fonction `purrr::partial()` qui permet d'appeler partiellement une fonction et de renvoyer une nouvelle fonction.

```
trial |>
  tbl_summary(
    include = marker,
    statistic = ~ "{mean} pour 100",
    digits = ~ purrr::partial(style_percent, digits = 1)
  )
```

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

Table 18.21: passer une fonction personnalisée à digits (syntaxe 3)

Caractéristique	N = 200
Marker Level (ng/mL)	91,6 pour 100
Manquant	10

À noter dans l'exemple précédent que les fonctions `style_*`() de `{gtsummary}` tiennent compte du thème défini (ici la virgule comme séparateur de décimale).

Pour une mise en forme plus avancée des nombres, il faut se tourner vers l'extension `{scales}` et ses diverses fonctions de mise en forme comme `scales::label_number()` ou `scales::label_percent()`.

ATTENTION : les fonctions de `{scales}` n'héritent pas des paramètres du thème `{gtsummary}` actif. Il faut donc personnaliser le séparateur de décimal dans l'appel à la fonction.

```
trial |>
  tbl_summary(
    include = marker,
    statistic = ~ "{mean}",
    digits = ~ scales::label_number(
      accuracy = .01,
      suffix = " ng/mL",
      decimal.mark = ",",
    )
  )
```

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

Table 18.22: passer une fonction personnalisée à `digits` (syntaxe 4)

Caractéristique	N = 200
Marker Level (ng/mL)	0,92 ng/mL
Manquant	10

18.2.8 Données manquantes

Le paramètre `missing` permet de s'indiquer s'il faut afficher le nombre d'observations manquantes (c'est-à-dire égales à NA) : `"ifany"` (valeur par défaut) affiche ce nombre seulement s'il y en a, `"no"` masque ce nombre et `"always"` force l'affichage de ce nombre même s'il n'y pas de valeur manquante. Le paramètre `missing_text` permet de personnaliser le texte affiché.

```
hdv2003 |>
  tbl_summary(
    include = c(age, heures.tv),
    missing = "always",
    missing_text = "Nbre observations manquantes"
  )
```

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

Table 18.23: forcer l’affichage des valeurs manquantes

Caractéristique	N = 2 000
age	48 (35 – 60)
Nbre observations manquantes	0
heures.tv	2,00 (1,00 – 3,00)
Nbre observations manquantes	5

Il est à noter, pour les variables catégorielles, que les valeurs manquantes ne sont jamais pris en compte pour le calcul des pourcentages. Pour les inclure dans le calcul, il faut les transformer en valeurs explicites, par exemple avec `forcats::fct_na_value_to_level()` de `{forcats}`.

```
hdv2003 |>
  dplyr::mutate(
    trav.imp.explicit = trav.imp |>
      forcats::fct_na_value_to_level("(non renseigné)")
  ) |>
  tbl_summary(
    include = c(trav.imp, trav.imp.explicit)
  )
```

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

Table 18.24: valeurs manquantes explicites (variable catégorielle)

Caractéristique	N = 2 000
trav.imp	
Le plus important	29 (2,8%)
Aussi important que le reste	259 (25%)
Moins important que le reste	708 (68%)
Peu important	52 (5,0%)
Manquant	952
trav.imp.explicit	
Le plus important	29 (1,5%)
Aussi important que le reste	259 (13%)
Moins important que le reste	708 (35%)
Peu important	52 (2,6%)
(non renseigné)	952 (48%)

Table 18.24: valeurs manquantes explicites (variable catégorielle)

Caractéristique	N = 2 000
-----------------	-----------

18.2.9 Ajouter les effectifs observés

Lorsque l'on masque les manquants, il peut être pertinent d'ajouter une colonne avec les effectifs observés pour chaque variable à l'aide de la fonction `gtsummary::add_n()`.

```
hdv2003 |>
  tbl_summary(
    include = c(heures.tv, trav.imp),
    missing = "no"
  ) |>
  add_n()
```

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

Table 18.25: ajouter une colonne avec les effectifs observés

Caractéristique	N	N = 2 000
heures.tv	1 995	2,00 (1,00 – 3,00)
trav.imp	1 048	
Le plus important		29 (2,8%)
Aussi important que le reste		259 (25%)
Moins important que le reste		708 (68%)
Peu important		52 (5,0%)

18.3 Calcul manuel

18.3.1 Variable continue

R fournit de base toutes les fonctions nécessaires pour le calcul des différentes statistiques descriptives :

- `mean()` pour la moyenne

- `sd()` pour l'écart-type
- `min()` et `max()` pour le minimum et le maximum
- `range()` pour l'étendue
- `median()` pour la médiane

Si la variable contient des valeurs manquantes (NA), ces fonctions renverront une valeur manquante, sauf si on leur précise `na.rm = TRUE`.

```
hdv2003$heures.tv |> mean()
```

```
[1] NA
```

```
hdv2003$heures.tv |> mean(na.rm = TRUE)
```

```
[1] 2.246566
```

```
hdv2003$heures.tv |> sd(na.rm = TRUE)
```

```
[1] 1.775853
```

```
hdv2003$heures.tv |> min(na.rm = TRUE)
```

```
[1] 0
```

```
hdv2003$heures.tv |> max(na.rm = TRUE)
```

```
[1] 12
```

```
hdv2003$heures.tv |> range(na.rm = TRUE)
```

```
[1] 0 12
```

```
hdv2003$heures.tv |> median(na.rm = TRUE)
```

```
[1] 2
```

La fonction `quantile()` permet de calculer tous types de quantiles.

```
hdv2003$heures.tv |> quantile(na.rm = TRUE)
```

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

```
hdv2003$heures.tv |>
quantile(
  probs = c(.2, .4, .6, .8),
  na.rm = TRUE
)
```

```
20% 40% 60% 80%
1    2    2    3
```

La fonction `summary()` renvoie la plupart de ces indicateurs en une seule fois, ainsi que le nombre de valeurs manquantes.

```
hdv2003$heures.tv |> summary()
```

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

18.3.2 Variable catégorielle

Les fonctions de base pour le calcul d'un tri à plat sont les fonctions `table()` et `xtabs()`. Leur syntaxe est quelque peu différente. On passe un vecteur entier à `table()` alors que la syntaxe de `xtabs()` se rapproche de celle d'un modèle linéaire : on décrit le tableau attendu à l'aide d'une formule et on indique le tableau de données. Les deux fonctions renvoient le même résultat.

```
tbl <- hdv2003$trav.imp |> table()
tbl <- xtabs(~ trav.imp, data = hdv2003)
tbl <- hdv2003 |> xtabs(~ trav.imp, data = _)
tbl
```

```
trav.imp
      Le plus important Aussi important que le reste
                29                                259
Moins important que le reste                Peu important
                708                                52
```

Comme on le voit, il s'agit du tableau brut des effectifs, sans les valeurs manquantes, et pas vraiment lisible dans la console de **R**.

Pour calculer les proportions, on appliquera `prop.table()` sur la table des effectifs bruts.

```
prop.table(tbl)
```

```
trav.imp
      Le plus important Aussi important que le reste
      0.02767176                                0.24713740
Moins important que le reste                Peu important
      0.67557252                                0.04961832
```

Pour la réalisation rapide d'un tri à plat, on pourra donc préférer la fonction `questionr::freq()` qui affiche également le nombre de valeurs manquantes et les pourcentages, en un seul appel.

```
hdv2003$trav.imp |>
  questionr::freq(total = TRUE)
```

	n	%	val%
Le plus important	29	1.5	2.8
Aussi important que le reste	259	13.0	24.7
Moins important que le reste	708	35.4	67.6
Peu important	52	2.6	5.0
NA	952	47.6	NA
Total	2000	100.0	100.0

18.4 Intervalles de confiance

18.4.1 Avec gtsummary

La fonction `gtsummary::add_ci()` permet d'ajouter des intervalles de confiance à un tableau créé avec `gtsummary::tbl_summary()`.

Avertissement

Par défaut, pour les **variables continues**, `gtsummary::tbl_summary()` affiche la médiane tandis que `gtsummary::add_ci()` calcule l'intervalle de confiance d'une moyenne ! Il faut donc :

- soit afficher la moyenne dans `gtsummary::tbl_summary()` à l'aide du paramètre

- soit calculer les intervalles de confiance d'une médiane (méthode "wilcox.test") via le paramètre `method` de `gtsummary::add_ci()`.

```
hdv2003 |>
tbl_summary(
  include = c(age, heures.tv, trav.imp),
  statistic = age ~ "{mean} ({sd})"
) |>
add_ci(
  method = heures.tv ~ "wilcox.test"
)
```

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

Table 18.26: ajouter les intervalles de confiance

Caractéristique	N = 2 000	95% CI
age	48 (17)	47, 49
heures.tv	2,00 (1,00 – 3,00)	2,5, 2,5
Manquant	5	
trav.imp		
Le plus important	29 (2,8%)	1,9%, 4,0%
Aussi important que le reste	259 (25%)	22%, 27%
Moins important que le reste	708 (68%)	65%, 70%
Peu important	52 (5,0%)	3,8%, 6,5%
Manquant	952	

L'argument `statistic` permet de personnaliser la présentation de l'intervalle ; `conf.level` de changer le niveau de confiance et `style_fun` de modifier la mise en forme des nombres de l'intervalle.

```
hdv2003 |>
tbl_summary(
  include = c(age, heures.tv),
  statistic = ~ "{mean}"
) |>
```

```
add_ci(
  statistic = ~ "entre {conf.low} et {conf.high}",
  conf.level = .9,
  style_fun = ~ purrr::partial(style_number, digits = 1)
)
```

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

Table 18.27: des intervalles de confiance personnalisés

Caractéristique	N = 2 000	90% CI
age	48	entre 47,5 et 48,8
heures.tv	2,25	entre 2,2 et 2,3
Manquant	5	

18.4.2 Calcul manuel

Le calcul de l'intervalle de confiance d'une **moyenne** s'effectue avec la fonction `t.test()`.

```
hdv2003$age |> t.test()
```

One Sample t-test

```
data: hdv2003$age
t = 127.12, df = 1999, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 47.41406 48.89994
sample estimates:
mean of x
 48.157
```

Le résultat renvoyé est une liste contenant de multiples informations.

```
hdv2003$age |> t.test() |> str()
```

```
List of 10
 $ statistic   : Named num 127
  ..- attr(*, "names")= chr "t"
 $ parameter   : Named num 1999
  ..- attr(*, "names")= chr "df"
 $ p.value     : num 0
 $ conf.int    : num [1:2] 47.4 48.9
  ..- attr(*, "conf.level")= num 0.95
 $ estimate    : Named num 48.2
  ..- attr(*, "names")= chr "mean of x"
 $ null.value  : Named num 0
  ..- attr(*, "names")= chr "mean"
 $ stderr      : num 0.379
 $ alternative: chr "two.sided"
 $ method      : chr "One Sample t-test"
 $ data.name   : chr "hdv2003$age"
 - attr(*, "class")= chr "htest"
```

Si l'on a besoin d'accéder spécifiquement à l'intervalle de confiance calculé :

```
hdv2003$age |> t.test() |> purrr::pluck("conf.int")
```

```
[1] 47.41406 48.89994
attr(,"conf.level")
[1] 0.95
```

Pour celui d'une **médiane**, on utilisera `wilcox.test()` en précisant `conf.int = TRUE`.

```
hdv2003$age |> wilcox.test(conf.int = TRUE)
```

Wilcoxon signed rank test with continuity correction

```
data: hdv2003$age
V = 2001000, p-value < 2.2e-16
alternative hypothesis: true location is not equal to 0
95 percent confidence interval:
 47.00001 48.50007
```

```
sample estimates:
(pseudo)median
      47.99996
```

```
hdv2003$age |>
  wilcox.test(conf.int = TRUE) |>
  purrr::pluck("conf.int")
```

```
[1] 47.00001 48.50007
attr("conf.level")
[1] 0.95
```

Pour une **proportion**, on utilisera `prop.test()` en lui transmettant le nombre de succès et le nombre d'observations, qu'il faudra donc avoir calculé au préalable. On peut également passer une table à deux entrées avec le nombre de succès puis le nombre d'échecs.

Ainsi, pour obtenir l'intervalle de confiance de la proportion des enquêtés qui considèrent leur travail comme *peu important*, en tenant compte des valeurs manquantes, le plus simple est d'effectuer le code suivant³ :

```
xtabs(~ I(hdv2003$trav.imp == "Peu important"), data = hdv2003) |>
  rev() |>
  prop.test()
```

1-sample proportions test with continuity correction

```
data:  rev(xtabs(~I(hdv2003$trav.imp == "Peu important"), data = hdv2003)), null probability
X-squared = 848.52, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.03762112 0.06502346
sample estimates:
      p
0.04961832
```

Par défaut, `prop.test()` produit un intervalle de confiance bilatéral en utilisant la méthode de Wilson avec correction de continuité. Pour plus d'information sur les différentes manières de calculer l'intervalle de confiance d'une proportion, on pourra se référer à ce [billet de blog](#).

³Notez l'utilisation de `rev()` pour inverser le tableau créé avec `xtabs()` afin que le nombre de succès (`TRUE`) soit indiqués avant le nombre d'échecs (`FALSE`).

💡 Astuce

Comme on le voit, il n'est pas aisé, avec les fonctions de **R base** de calculer les intervalles de confiance pour toutes les modalités d'une variable catégorielle.

On pourra éventuellement avoir recours à la petite fonction suivante qui réalise le tri à plat d'une variable catégorielle, calcule les proportions et leurs intervalles de confiance.

```
prop_ci <- function(x, conf.level = .95, correct = TRUE) {  
  tbl <- as.data.frame(table(x), responseName = "n")  
  tbl$N <- sum(tbl$n)  
  tbl$prop <- tbl$n / tbl$N  
  tbl$conf.low <- NA_real_  
  tbl$conf.high <- NA_real_  
  for (i in 1:nrow(tbl)) {  
    test <- prop.test(  
      x = tbl$n[i],  
      n = tbl$N[i],  
      conf.level = conf.level,  
      correct = correct  
    )  
    tbl$conf.low[i] <- test$conf.int[1]  
    tbl$conf.high[i] <- test$conf.int[2]  
  }  
  tbl  
}  
prop_ci(hdv2003$trav.imp)
```

	x	n	N	prop	conf.low	conf.high
1	Le plus important	29	1048	0.02767176	0.01894147	0.04001505
2	Aussi important que le reste	259	1048	0.24713740	0.22151849	0.27463695
3	Moins important que le reste	708	1048	0.67557252	0.64614566	0.70369541
4	Peu important	52	1048	0.04961832	0.03762112	0.06502346

18.5 webin-R

La statistique univariée est présentée dans le webin-R #03 (*statistiques descriptives avec gt-summary et esquisse*) sur [YouTube](https://www.youtube.com/watch?v=0EF_8GXyP5c).

https://youtu.be/oEF_8GXyP5c

19 Statistique bivariée & Tests de comparaison

19.1 Deux variables catégorielles

19.1.1 Tableau croisé avec gtsummary

Pour regarder le lien entre deux variables catégorielles, l'approche la plus fréquente consiste à réaliser un *tableau croisé*, ce qui s'obtient très facilement avec l'argument `by` de la fonction `gtsummary::tbl_summary()` que nous avons déjà abordée dans le chapitre sur la statistique univariée (cf. Section 18.2).

Prenons pour exemple le jeu de données `gtsummary::trial` et croisons les variables `stage` et `grade`. On indique à `by` la variable à représenter en colonnes et à `include` celle à représenter en lignes.

```
library(gtsummary)
```

```
#BlackLivesMatter
```

```
theme_gtsummary_language("fr", decimal.mark = ',')
```

```
Setting theme `language: fr`
```

```
trial |>
  tbl_summary(
    include = stage,
    by = grade
  )
```

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

Table 19.1: un tableau croisé avec des pourcentages en colonne

Caractéristique	I, N = 68	II, N = 68	III, N = 64
T Stage			
T1	17 (25%)	23 (34%)	13 (20%)
T2	18 (26%)	17 (25%)	19 (30%)
T3	18 (26%)	11 (16%)	14 (22%)
T4	15 (22%)	17 (25%)	18 (28%)

Par défaut, les pourcentages affichés correspondent à des pourcentages en colonne. On peut demander des pourcentages en ligne avec `percent = "row"` ou des pourcentages du total avec `percent = "cell"`.

Il est possible de passer plusieurs variables à `include` mais une seule variable peut être transmise à `by`. La fonction `gtsummary::add_overall()` permet d'ajouter une colonne totale. Comme pour un tri à plat, on peut personnaliser les statistiques affichées avec `statistic`.

```
library(gtsummary)
trial |>
  tbl_summary(
    include = c(stage, trt),
    by = grade,
    statistic = ~ "{p}% ({n}/{N})",
    percent = "row"
  ) |>
  add_overall(last = TRUE)
```

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

Table 19.2: un tableau croisé avec des pourcentages en ligne

Caractéristique	I, N = 68	II, N = 68	III, N = 64	Total, N = 200
T Stage				
T1	32% (17/53)	43% (23/53)	25% (13/53)	100% (53/53)
T2	33% (18/54)	31% (17/54)	35% (19/54)	100% (54/54)
T3	42% (18/43)	26% (11/43)	33% (14/43)	100% (43/43)
T4	30% (15/50)	34% (17/50)	36% (18/50)	100% (50/50)

Table 19.2: un tableau croisé avec des pourcentages en ligne

Caractéristique	I, N = 68	II, N = 68	III, N = 64	Total, N = 200
Chemotherapy Treatment				
Drug A	36% (35/98)	33% (32/98)	32% (31/98)	100% (98/98)
Drug B	32% (33/102)	35% (36/102)	32% (33/102)	100% (102/102)

! Important

Choisissez bien votre type de pourcentages (en lignes ou en colonnes). Si d'un point de vue purement statistique, ils permettent tous deux de décrire la relation entre les deux variables, ils ne correspondent au même *story telling*. Tout dépend donc du message que vous souhaitez faire passer, de l'histoire que vous souhaitez raconter.

`gtsummary::tbl_summary()` est bien adaptée dans le cadre d'une analyse de facteurs afin de représenter un *outcome* donné avec `by` et une liste de facteurs avec `include`.

Lorsque l'on ne croise que deux variables et que l'on souhaite un affichage un peu plus traditionnel d'un tableau croisé, on peut utiliser `gtsummary::tbl_cross()` à laquelle on transmettra une et une seule variable à `row` et une et une seule variable à `col`. Pour afficher des pourcentages, il faudra indiquer le type de pourcentages voulus avec `percent`.

```
trial |>
  tbl_cross(
    row = stage,
    col = grade,
    percent = "row"
  )
```

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

Table 19.3: un tableau croisé avec `tbl_cross()`

	I	II	III	Total
T Stage				
T1	17 (32%)	23 (43%)	13 (25%)	53 (100%)
T2	18 (33%)	17 (31%)	19 (35%)	54 (100%)

Table 19.3: un tableau croisé avec `tbl_cross()`

	I	II	III	Total
T3	18 (42%)	11 (26%)	14 (33%)	43 (100%)
T4	15 (30%)	17 (34%)	18 (36%)	50 (100%)
Total	68 (34%)	68 (34%)	64 (32%)	200 (100%)

19.1.2 Représentations graphiques

La représentation graphique la plus commune pour le croisement de deux variables catégorielles est le diagramme en barres, que l'on réalise avec la géométrie `ggplot2::geom_bar()` et en utilisant les esthétiques `x` et `fill` pour représenter les deux variables.

```
library(ggplot2)
ggplot(trial) +
  aes(x = stage, fill = grade) +
  geom_bar() +
  labs(x = "T Stage", fill = "Grade", y = "Effectifs")
```

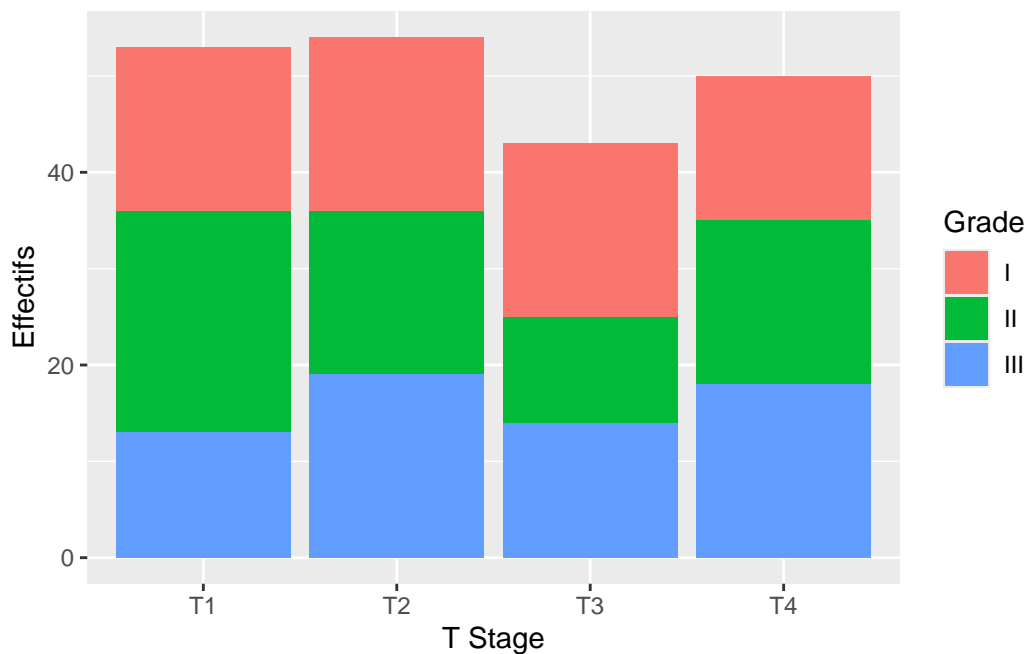


Figure 19.1: un graphique en barres croisant deux variables

On peut modifier la position des barres avec le paramètre `position`.

```
library(ggplot2)
ggplot(trial) +
  aes(x = stage, fill = grade) +
  geom_bar(position = "dodge") +
  labs(x = "T Stage", fill = "Grade", y = "Effectifs")
```

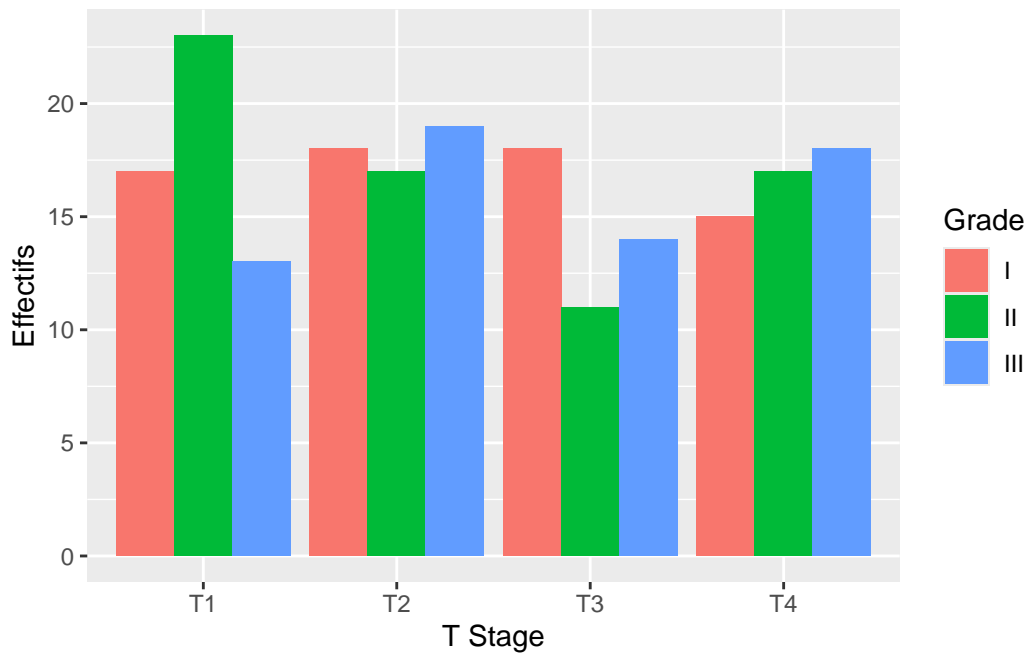


Figure 19.2: un graphique avec des barres côte à côte

Pour des barres cumulées, on aura recours à `position = "fill"`. Pour que les étiquettes de l'axe des `y` soient représentées sous forme de pourcentages (i.e. 25% au lieu de 0.25), on aura recours à la fonction `scales::percent()` qui sera transmise à `ggplot2::scale_y_continuous()`.

```
library(ggplot2)
ggplot(trial) +
  aes(x = stage, fill = grade) +
  geom_bar(position = "fill") +
  labs(x = "T Stage", fill = "Grade", y = "Proportion") +
  scale_y_continuous(labels = scales::percent)
```

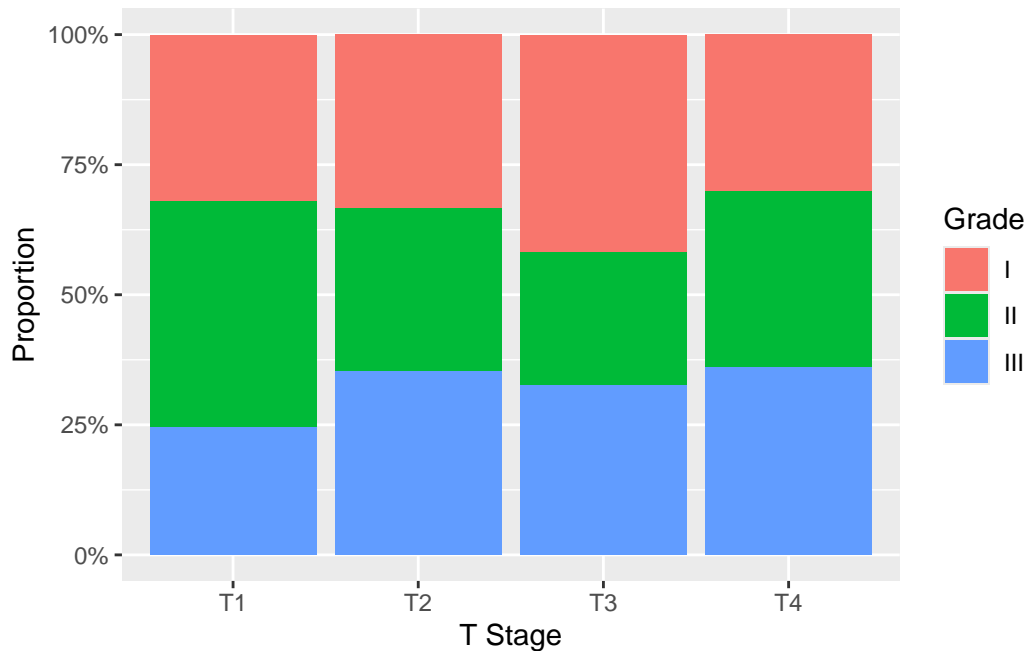


Figure 19.3: un graphique en barres cumulées

💡 Ajouter des étiquettes sur un diagramme en barres

Il est facile d'ajouter des étiquettes en ayant recours à `ggplot2::geom_text()`, à condition de lui passer les bons paramètres.

Tout d'abord, il faudra préciser `stat = "count"` pour indiquer que l'on souhaite utiliser la statistique `ggplot2::stat_count()` qui est celle utilisé par défaut par `ggplot2::geom_bar()`. C'est elle qui permet de compter le nombre d'observations.

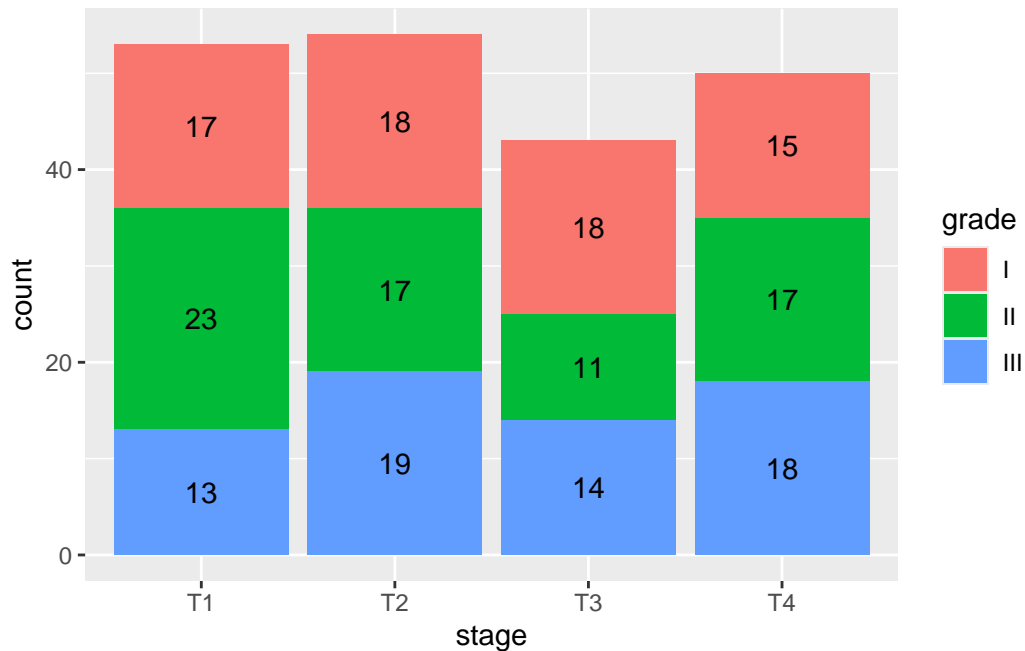
Il faut ensuite utiliser l'esthétique `label` pour indiquer ce que l'on souhaite afficher comme étiquettes. La fonction `after_stat(count)` permet d'accéder à la variable `count` calculée par `ggplot2::stat_count()`.

Enfin, il faut indiquer la position verticale avec `ggplot2::position_stack()`. En précisant un ajustement de vertical de 0.5, on indique que l'on souhaite positionner l'étiquette au milieu.

```

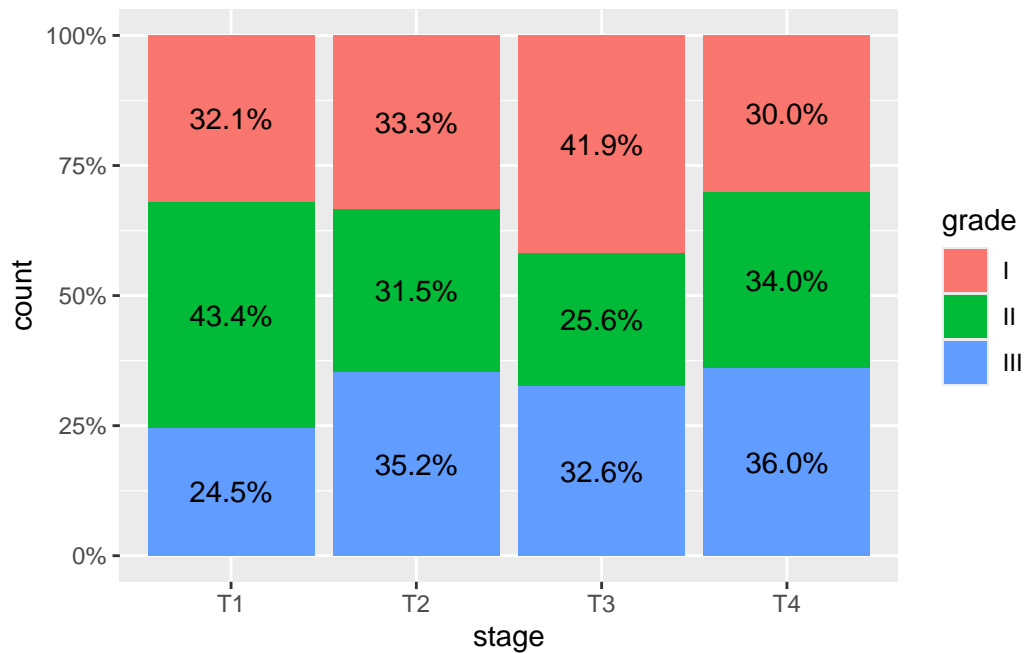
ggplot(trial) +
  aes(
    x = stage, fill = grade,
    label = after_stat(count)
  ) +
  geom_bar() +
  geom_text(
    stat = "count",
    position = position_stack(.5)
  )

```



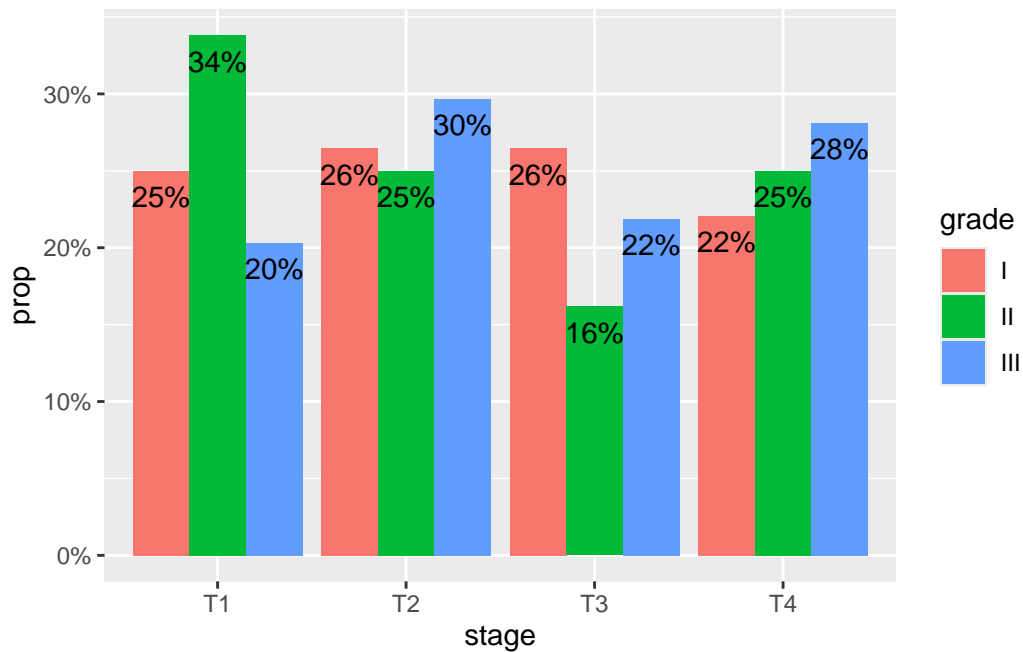
Pour un graphique en barres cumulées, on peut utiliser de manière similaire `ggplot2::position_fill()`. On ne peut afficher directement les proportions avec `ggplot2::stat_count()`. Cependant, nous pouvons avoir recours à `ggstats::stat_prop()`, déjà évoquée dans le chapitre sur la statistique univariée (cf. Section 18.1.2) et dont le dénominateur doit être précisé via l'esthétique *by*.

```
library(ggstats)
ggplot(trial) +
  aes(
    x = stage,
    fill = grade,
    by = stage,
    label = scales::percent(after_stat(prop), accuracy = .1)
  ) +
  geom_bar(position = "fill") +
  geom_text(
    stat = "prop",
    position = position_fill(.5)
  ) +
  scale_y_continuous(labels = scales::percent)
```



On peut aussi comparer facilement deux distributions, ici la proportion de chaque niveau de qualification au sein chaque sexe.

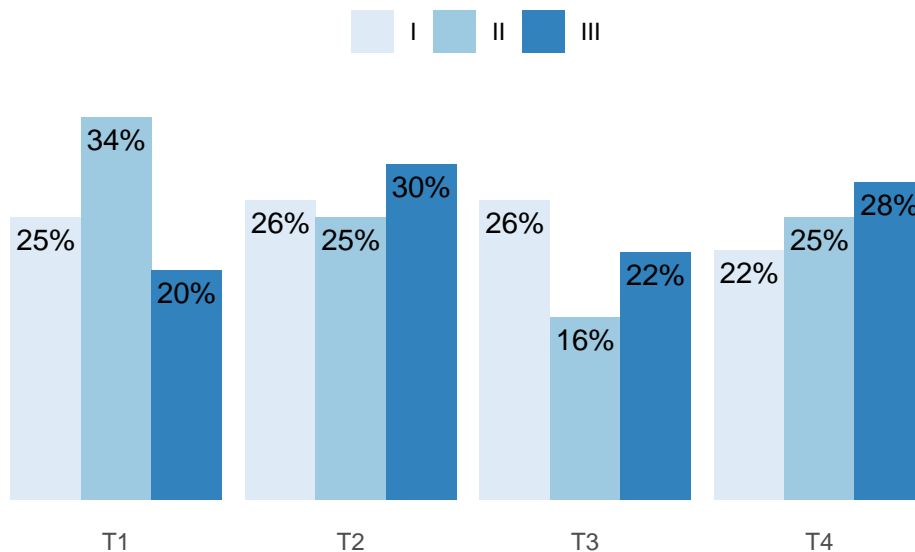

```
p <- ggplot(trial) +
  aes(
    x = stage,
    y = after_stat(prop),
    fill = grade,
    by = grade,
    label = scales::percent(after_stat(prop), accuracy = 1)
  ) +
  geom_bar(
    stat = "prop",
    position = position_dodge(.9)
  ) +
  geom_text(
    aes(y = after_stat(prop) - 0.01),
    stat = "prop",
    position = position_dodge(.9),
    vjust = "top"
  ) +
  scale_y_continuous(labels = scales::percent)
p
```



Il est possible d'alléger le graphique en retirant des éléments superflus.

```
p +
  theme_light() +
  xlab("") +
  ylab("") +
  labs(fill = "") +
  ggtitle("Distribution selon le niveau, par grade") +
  theme(
    panel.grid = element_blank(),
    panel.border = element_blank(),
    axis.text.y = element_blank(),
    axis.ticks = element_blank(),
    legend.position = "top"
  ) +
  scale_fill_brewer()
```

Distribution selon le niveau, par grade



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

19.1.3 Calcul manuel

Les deux fonctions de base permettant le calcul d'un tri à plat sont `table()` et `xtabs()` (cf. Section 18.3.2). Ces mêmes fonctions permettent le calcul du tri croisé de deux variables (ou

plus). Pour `table()`, on passera les deux vecteurs à croisés, tandis que pour `xtabs()` on décrira le tableau attendu à l'aide d'une formule.

```
table(trial$stage, trial$grade)
```

	I	II	III
T1	17	23	13
T2	18	17	19
T3	18	11	14
T4	15	17	18

```
tab <- xtabs(~ stage + grade, data = trial)
tab
```

	grade		
stage	I	II	III
T1	17	23	13
T2	18	17	19
T3	18	11	14
T4	15	17	18

Le tableau obtenu est basique et ne contient que les effectifs. La fonction `addmargins()` permet d'ajouter les totaux par ligne et par colonne.

```
tab |> addmargins()
```

	grade			
stage	I	II	III	Sum
T1	17	23	13	53
T2	18	17	19	54
T3	18	11	14	43
T4	15	17	18	50
Sum	68	68	64	200

Pour le calcul des pourcentages, le plus simple est d'avoir recours au package `{questionr}` qui fournit les fonctions `questionr::cprop()`, `questionr::rprop()` et `questionr::prop()` qui permettent de calculer, respectivement, les pourcentages en colonne, en ligne et totaux.

```
questionr::cprop(tab)
```

stage	grade			Ensemble
	I	II	III	
T1	25.0	33.8	20.3	26.5
T2	26.5	25.0	29.7	27.0
T3	26.5	16.2	21.9	21.5
T4	22.1	25.0	28.1	25.0
Total	100.0	100.0	100.0	100.0

```
questionr::rprop(tab)
```

stage	grade			Total
	I	II	III	
T1	32.1	43.4	24.5	100.0
T2	33.3	31.5	35.2	100.0
T3	41.9	25.6	32.6	100.0
T4	30.0	34.0	36.0	100.0
Ensemble	34.0	34.0	32.0	100.0

```
questionr::prop(tab)
```

stage	grade			Total
	I	II	III	
T1	8.5	11.5	6.5	26.5
T2	9.0	8.5	9.5	27.0
T3	9.0	5.5	7.0	21.5
T4	7.5	8.5	9.0	25.0
Total	34.0	34.0	32.0	100.0

19.1.4 Test du Chi² et dérivés

Dans le cadre d'un tableau croisé, on peut tester l'existence d'un lien entre les modalités de deux variables, avec le très classique test du Chi² (parfois écrit χ^2 ou Chi²). Pour une présentation plus détaillée du test, on pourra se référer à ce [cours de Julien Barnier](#).

Le test du Chi² peut se calculer très facilement avec la fonction `chisq.test()` appliquée au tableau obtenu avec `table()` ou `xtabs()`.

```
tab <- xtabs(~ stage + grade, data = trial)
tab
```

```
      grade
stage  I  II III
T1    17  23  13
T2    18  17  19
T3    18  11  14
T4    15  17  18
```

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

Pearson's Chi-squared test

```
data:  tab
X-squared = 4.8049, df = 6, p-value = 0.5691
```

Si l'on est adepte de `{gtsummary}`, il suffit d'appliquer `gtsummary::add_p()` au tableau produit avec `gtsummary::tbl_summary()`.

```
trial |>
  tbl_summary(
    include = stage,
    by = grade
  ) |>
  add_p()
```

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

Table 19.4: un tableau croisé avec test du khi²

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
T Stage				0,6
T1	17 (25%)	23 (34%)	13 (20%)	
T2	18 (26%)	17 (25%)	19 (30%)	

Table 19.4: un tableau croisé avec test du khi²

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
T3	18 (26%)	11 (16%)	14 (22%)	
T4	15 (22%)	17 (25%)	18 (28%)	

Dans notre exemple, les deux variables *stage* et *grade* ne sont clairement pas corrélées.

Un test alternatif est le test exact de Fisher. Il s'obtient aisément avec `fisher.test()` ou bien en le spécifiant via l'argument `test` de `gtsummary::add_p()`.

```
tab <- xtabs(~ stage + grade, data = trial)
fisher.test(tab)
```

Fisher's Exact Test for Count Data

```
data: tab
p-value = 0.5801
alternative hypothesis: two.sided
```

```
trial |>
  tbl_summary(
    include = stage,
    by = grade
  ) |>
  add_p(test = all_categorical() ~ "fisher.test")
```

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

Table 19.5: un tableau croisé avec test exact de Fisher

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
T Stage				0,6
T1	17 (25%)	23 (34%)	13 (20%)	
T2	18 (26%)	17 (25%)	19 (30%)	
T3	18 (26%)	11 (16%)	14 (22%)	

Table 19.5: un tableau croisé avec test exact de Fisher

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
T4	15 (22%)	17 (25%)	18 (28%)	

i Note

Formellement, le test de Fisher suppose que les marges du tableau (totaux lignes et colonnes) sont fixées, puisqu'il repose sur une loi hypergéométrique, et donc celui-ci se prête plus au cas des situations expérimentales (plans d'expérience, essais cliniques) qu'au cas des données tirées d'études observationnelles.

En pratique, le test du χ^2 étant assez robuste quant aux déviations par rapport aux hypothèses d'applications du test (effectifs théoriques supérieurs ou égaux à 5), le test de Fisher présente en général peu d'intérêt dans le cas de l'analyse des tableaux de contingence.

19.1.5 Comparaison de deux proportions

Pour comparer deux proportions, la fonction de base est `prop.test()` à laquelle on passera un tableau à 2×2 dimensions.

```
tab <- xtabs(~ I(stage == "T1") + trt, data = trial)
tab |> questionr::cprop()
```

```

      trt
I(stage == "T1") Drug A Drug B Ensemble
      FALSE   71.4    75.5    73.5
      TRUE    28.6    24.5    26.5
      Total 100.0   100.0   100.0
```

```
tab |> prop.test()
```

2-sample test for equality of proportions with continuity correction

```
data:  tab
X-squared = 0.24047, df = 1, p-value = 0.6239
alternative hypothesis: two.sided
95 percent confidence interval:
```

```
-0.2217278 0.1175050
sample estimates:
  prop 1    prop 2 
0.4761905 0.5283019
```

Il est également envisageable d'avoir recours à un test exact de Fisher. Dans le cas d'un tableau à 2×2 dimensions, le test exact de Fisher ne teste pas si les deux proportions sont différents, mais plutôt si leur *odds ratio* (qui est d'ailleurs renvoyé par la fonction) est différent de 1.

```
fisher.test(tab)
```

Fisher's Exact Test for Count Data

```
data:  tab
p-value = 0.5263
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.4115109 1.5973635
sample estimates:
odds ratio
 0.8125409
```

Mais le plus simple reste encore d'avoir recours à `{gtsummary}` et à sa fonction `gtsummary::add_difference()` que l'on peut appliquer à un tableau où le paramètre `by` n'a que deux modalités. Pour la différence de proportions, il faut que les variables transmises à `include` soit dichotomiques.

```
trial |>
  tbl_summary(
    by = trt,
    include = response
  ) |>
  add_difference()
```

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

Table 19.6: différence entre deux proportions

Caractéristique	Drug A, N = 98	Drug B, N = 102	Difference	95% IC	p-valeur
Tumor Response	28 (29%)	33 (34%)	-4,2%	-18% – 9,9%	0,6
Manquant	3	4			

Attention : si l'on passe une variable catégorielle à trois modalités ou plus, c'est la différence des moyennes standardisées (globale pour la variable) qui sera calculée et non la différence des proportions dans chaque groupe.

```
trial |>
  tbl_summary(
    by = trt,
    include = grade
  ) |>
  add_difference()
```

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

Table 19.7: différence moyenne standardisée

Caractéristique	Drug A, N = 98	Drug B, N = 102	Difference	95% IC
Grade			0,07	-0,20 – 0,35
I	35 (36%)	33 (32%)		
II	32 (33%)	36 (35%)		
III	31 (32%)	33 (32%)		

Pour calculer la différence des proportions pour chaque modalité de *grade*, il est nécessaire de transformer, en amont, la variable catégorielle *grade* en trois variables dichotomiques (de type oui/non, une par modalité), ce qui peut se faire facilement avec la fonction `fastDummies::dummy_cols()` de l'extension `{fastDummies}`.

```
trial |>
  fastDummies::dummy_cols("grade") |>
  tbl_summary()
```

```
by = trt,
include = starts_with("grade_"),
digits = ~ c(0, 1)
) |>
add_difference()
```

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

Table 19.8: différence entre proportions avec création de variables dichotomiques

Caractéristique	Drug A, N = 98	Drug B, N = 102	Difference	95% IC	p-valeur
grade_I	35 (35,7%)	33 (32,4%)	3,4%	-11% – 17%	0,7
grade_II	32 (32,7%)	36 (35,3%)	-2,6%	-17% – 11%	0,8
grade_III	31 (31,6%)	33 (32,4%)	-0,72%	-14% – 13%	>0,9

19.2 Une variable continue selon une variable catégorielle

19.2.1 Tableau comparatif avec `gtsummary`

Dans le chapitre sur la statistique univariée (cf. Section 18.2), nous avons abordé comment afficher les statistiques descriptives d'une variable continue avec `gtsummary::tbl_summary()`. Pour comparer une variable continue selon plusieurs groupes définis par une variable catégorielle, il suffit d'utiliser le paramètre `by` :

```
trial |>
tbl_summary(
  include = age,
  by = grade
)
```

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

Table 19.9: âge médian et intervalle interquartile selon le grade

Caractéristique	I, N = 68	II, N = 68	III, N = 64
Age	47 (37 – 56)	49 (37 – 57)	47 (38 – 58)
Manquant	2	6	3

La fonction `gtsummary::add_overall()` permet d'ajouter une colonne total et `gtsummary::modify_spanning_header()` peut-être utilisé pour ajouter un en-tête de colonne.

```
trial |>
  tbl_summary(
    include = age,
    by = grade
  ) |>
  add_overall(last = TRUE) |>
  modify_spanning_header(
    all_stat_cols(stat_0 = FALSE) ~ "**Grade**"
  )
```

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

Table 19.10: âge médian et intervalle interquartile selon le grade

Caractéristique	I, N = 68	II, N = 68	III, N = 64	Total, N = 200
Age	47 (37 – 56)	49 (37 – 57)	47 (38 – 58)	47 (38 – 57)
Manquant	2	6	3	11

Comme pour un tri à plat, on peut personnaliser les statistiques à afficher avec `statistic`.

```
trial |>
  tbl_summary(
    include = age,
    by = grade,
    statistic = all_continuous() ~ "{mean} ({sd})",
    digits = all_continuous() ~ c(1, 1)
  ) |>
  add_overall(last = TRUE)
```

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

Table 19.11: âge moyen et écart-type selon le grade

Caractéristique	I, N = 68	II, N = 68	III, N = 64	Total, N = 200
Age	46,2 (15,2)	47,5 (13,7)	48,1 (14,1)	47,2 (14,3)
Manquant	2	6	3	11

19.2.2 Représentations graphiques

La moyenne ou la médiane sont des indicateurs centraux et ne suffisent pas à rendre compte des différences de distribution d'une variable continue entre plusieurs sous-groupes.

Une représentation usuelle pour comparer deux distributions consiste à avoir recours à des boîtes à moustaches que l'on obtient avec `ggplot2::geom_boxplot()`.

```
ggplot(trial) +  
  aes(x = grade, y = age) +  
  geom_boxplot(fill = "lightblue") +  
  theme_light()
```

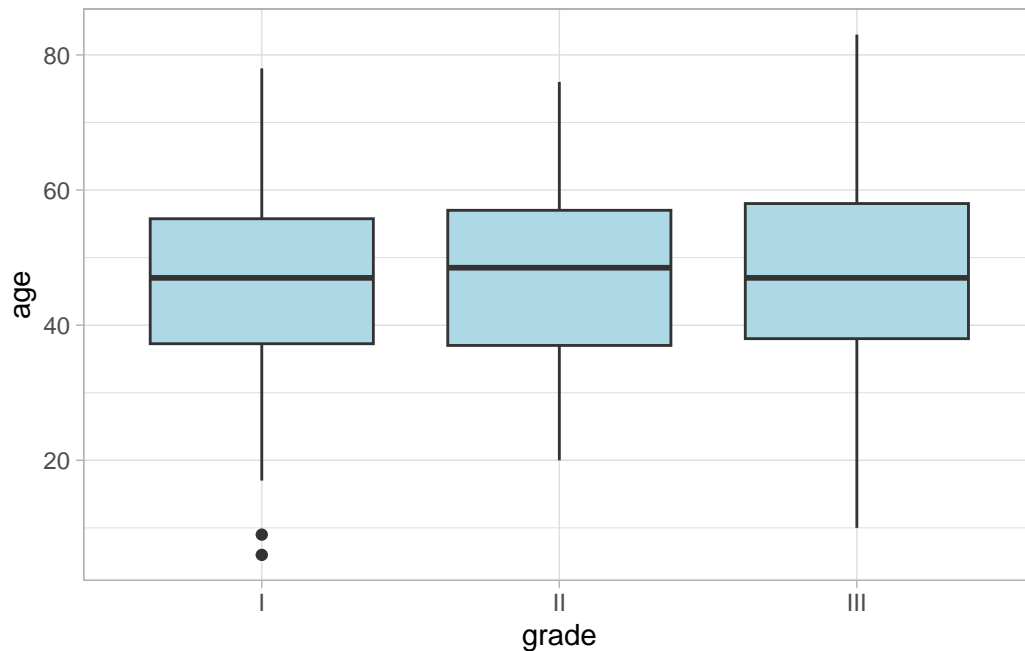


Figure 19.4: boîtes à moustache

💡 Astuce

Le trait central représente la médiane, le rectangle est délimité par le premier et le troisième quartiles (i.e. le 25^e et le 75^e percentiles). Les traits verticaux vont jusqu'aux extrêmes (minimum et maximum) ou jusqu'à 1,5 fois l'intervalle interquartile. Si des points sont situés à plus d'1,5 fois l'intervalle interquartile au-dessus du 3^e quartile ou en-dessous du 1^{er} quartile, ils sont considérés comme des valeurs atypiques et représentés par un point. Dans l'exemple précédent, c'est le cas des deux plus petites valeurs observées pour le grade I.

Alternativement, on peut utiliser un graphique en violons qui représentent des courbes de densité dessinées en miroir.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_violin(fill = "lightblue") +
  theme_light()
```

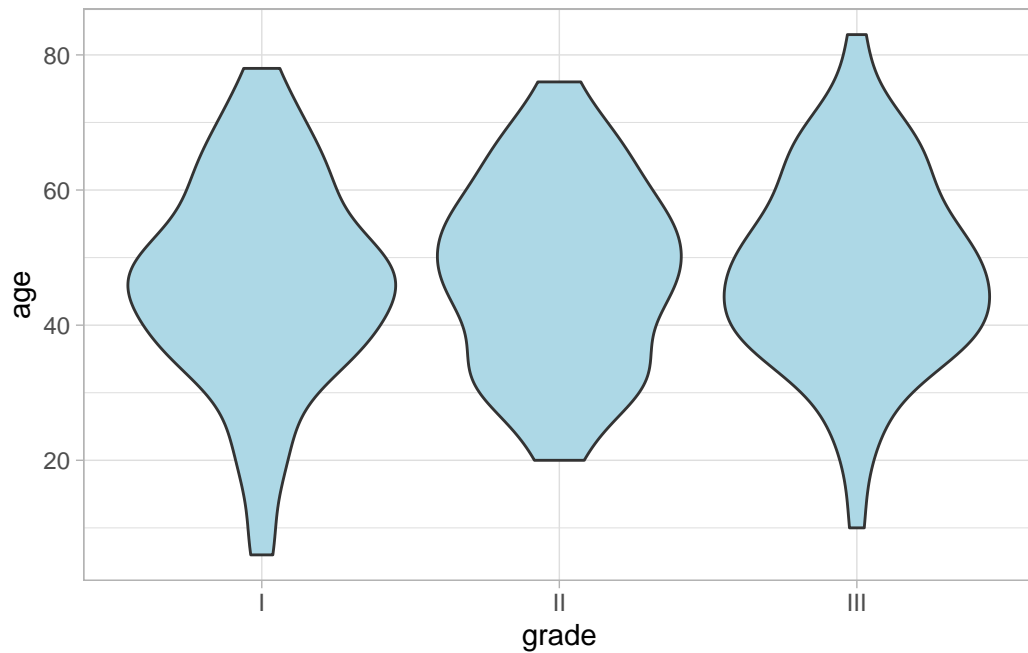


Figure 19.5: graphique en violons

Il est toujours possible de représenter les observations individuelles sous la forme d'un nuage de points. Le paramètre `alpha` permet de rendre les points transparents afin de mieux visualiser les superpositions de points.

```
ggplot(trial) +  
  aes(x = grade, y = age) +  
  geom_point(alpha = .25, colour = "blue") +  
  theme_light()
```

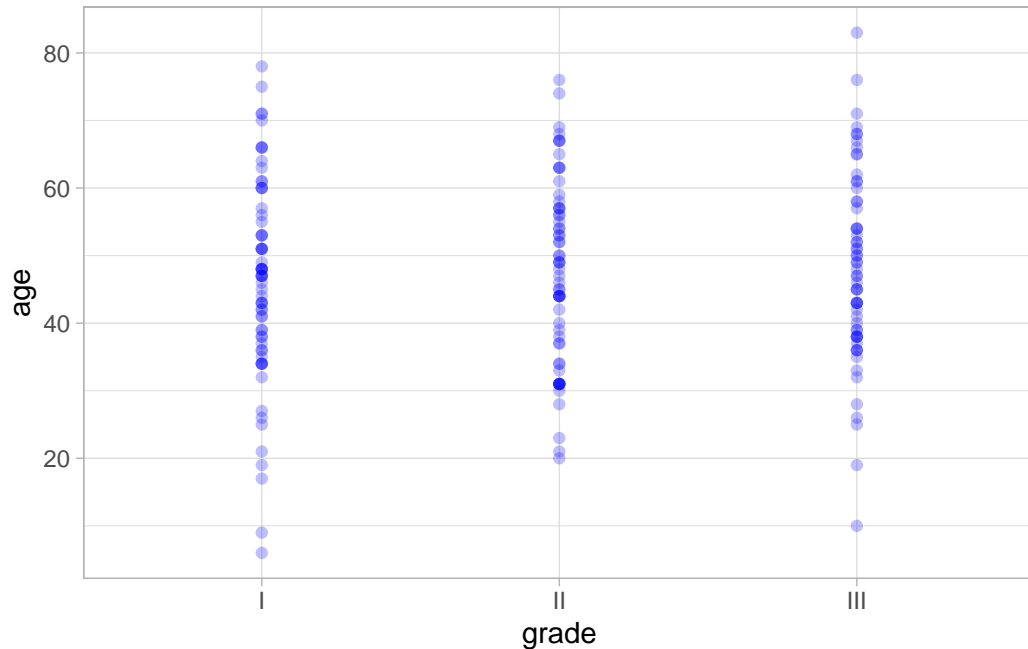


Figure 19.6: un nuage de points avec une variable continue et une variable catégorielle

Comme la variable *grade* est catégorielle, tous les points d'une même modalité sont représentées sur une même ligne. La représentation peut être améliorée en ajoutant un décalage aléatoire sur l'axe horizontal. Cela s'obtient avec `ggplot2::position_jitter()` en précisant `height = 0` pour ne pas ajouter de décalage vertical et `width = .2` pour décaler horizontalement les points entre -20% et +20%.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_point(
    alpha = .25,
    colour = "blue",
    position = position_jitter(height = 0, width = .2)
  ) +
  theme_light()
```

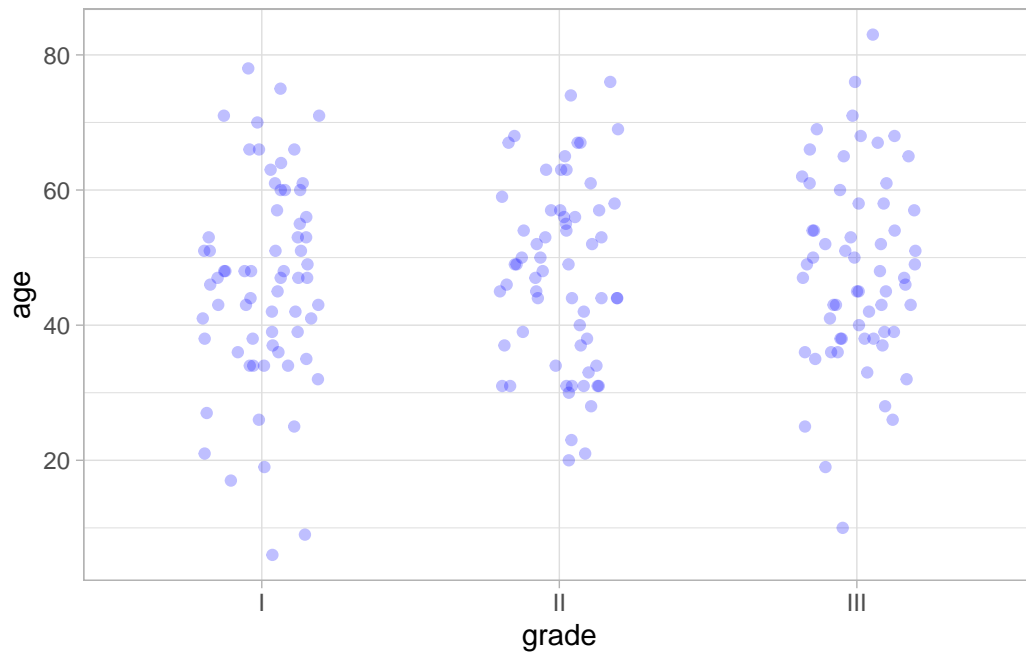


Figure 19.7: un nuage de points avec une variable continue et une variable catégorielle et avec un décalage horizontal aléatoire

La statistique `ggstats::stat_weighted_mean()` de `{ggstats}` permet de calculer à la volée la moyenne du nuage de points.

```
ggplot(trial) +
  aes(x = grade, y = age) +
  geom_point(stat = "weighted_mean", colour = "blue") +
  theme_light()
```

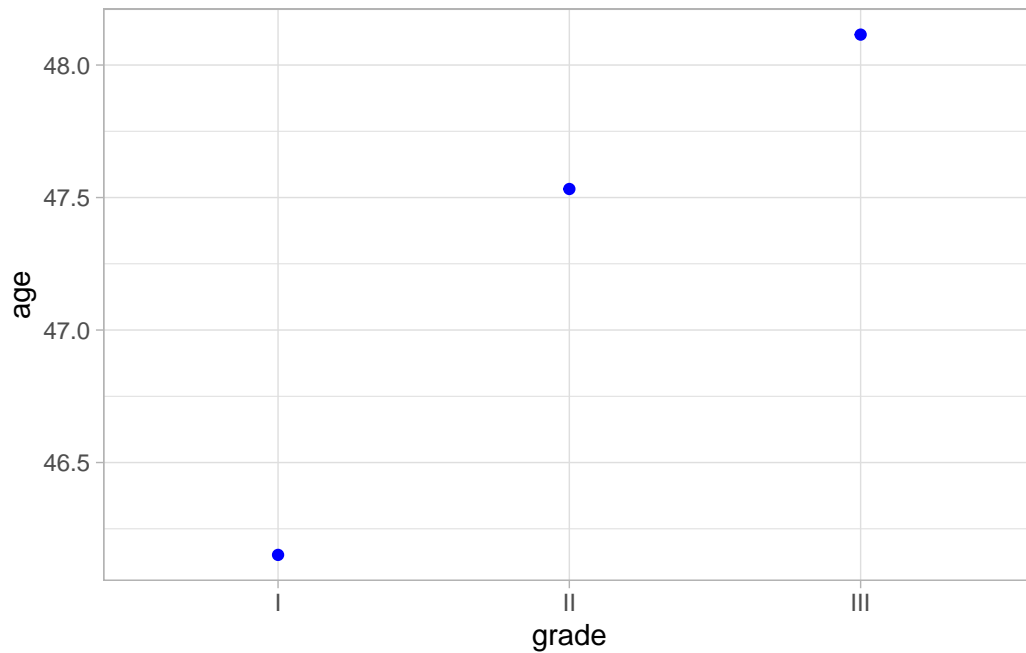



Figure 19.8: âge moyen selon le grade

Cela peut être utile pour effectuer des comparaisons multiples.

```
ggplot(trial) +  
  aes(x = grade, y = age, colour = stage, group = stage) +  
  geom_line(stat = "weighted_mean") +  
  geom_point(stat = "weighted_mean") +  
  facet_grid(cols = vars(trt)) +  
  theme_light()
```

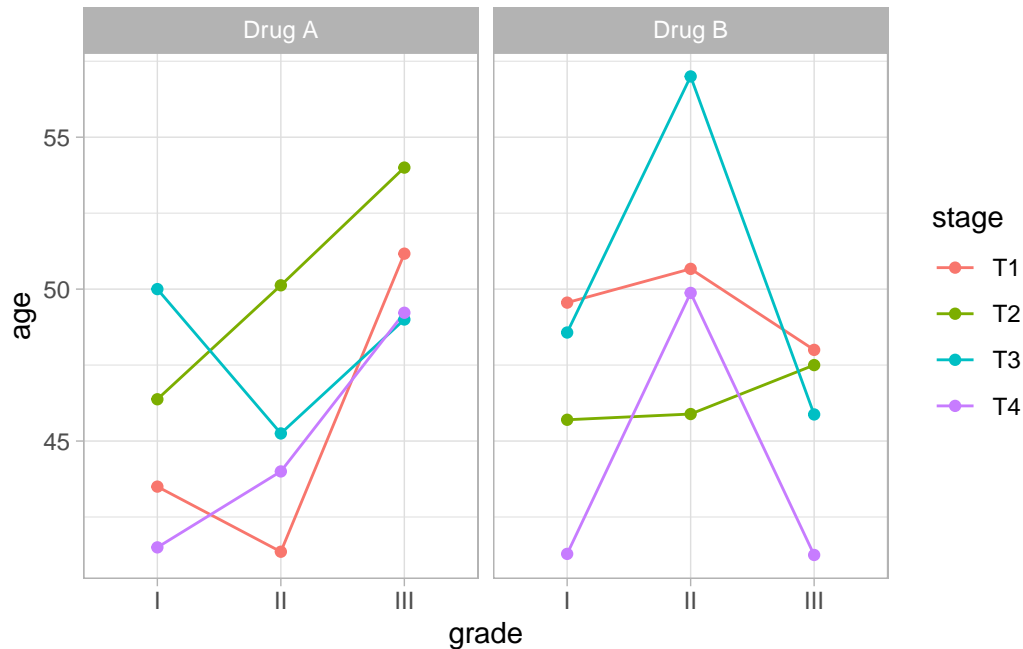
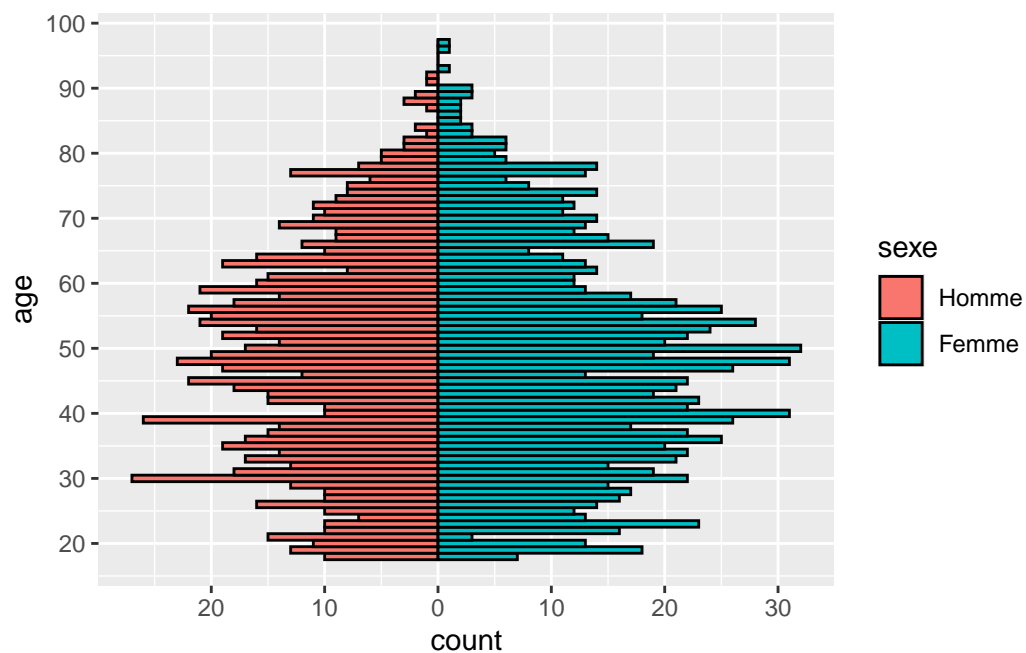


Figure 19.9: âge moyen selon le grade, par traitement et état d'avancement de la maladie

💡 Pyramide des âges

Il est possible de réaliser assez facilement une pyramide des âges en combinant un histogramme avec `position_likert_count()` fournie par le package `{ggstats}`. Nous allons pour illustrer cela prendre le jeu de données `hdv2003` fourni par le package `{questionr}`.

```
data("hdv2003", package = "questionr")
library(ggplot2)
library(ggstats)
ggplot(hdv2003) +
  aes(fill = sexe, y = age) +
  geom_histogram(
    position = "likert_count",
    binwidth = 1,
    color = "black"
  ) +
  scale_x_continuous(label = label_number_abs()) +
  scale_y_continuous(breaks = 1:10 * 10)
```



19.2.3 Calcul manuel

Le plus simple pour calculer des indicateurs par sous-groupe est d'avoir recours à `dplyr::summarise()` avec `dplyr::group_by()`.

```
library(dplyr)
trial |>
  group_by(grade) |>
  summarise(
    age_moy = mean(age, na.rm = TRUE),
    age_med = median(age, na.rm = TRUE)
  )
```

```
# A tibble: 3 x 3
  grade age_moy age_med
<fct>   <dbl>   <dbl>
1 I      46.2     47
2 II     47.5    48.5
3 III    48.1     47
```

En base **R**, on peut avoir recours à `tapply()`. On lui indique d'abord le vecteur sur lequel on souhaite réaliser le calcul, puis un facteur qui indiquera les sous-groupes, puis une fonction qui sera appliquée à chaque sous-groupe et enfin, optionnellement, des arguments additionnels qui seront transmis à cette fonction.

```
tapply(trial$age, trial$grade, mean, na.rm = TRUE)
```

```
      I      II      III
46.15152 47.53226 48.11475
```

19.2.4 Tests de comparaison

Pour comparer des moyennes ou des médianes, le plus facile est encore d'avoir recours à `{gtsummary}` et sa fonction `gtsummary::add_p()`.

```
trial |>
  tbl_summary(
    include = age,
    by = grade
  ) |>
  add_p()
```

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

Table 19.12: test de comparaison sur la somme des rangs

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
Age	47 (37 – 56)	49 (37 – 57)	47 (38 – 58)	0,8
Manquant	2	6	3	

Par défaut, pour les **variables continues**, un test de Kruskal-Wallis calculé avec la fonction `stats::kruskal.test()` est utilisé lorsqu'il y a trois groupes ou plus, et un test de Wilcoxon-Mann-Whitney calculé avec `stats::wilcox.test()` (test de comparaison des rangs) lorsqu'il n'y a que deux groupes. Au sens strict, il ne s'agit pas de tests de comparaison des médianes

mais de tests sur la somme des rangs¹. En pratique, ces tests sont appropriés lorsque l'on présente les médianes et les intervalles inter-quartiles.

Si l'on affiche des moyennes, il serait plus juste d'utiliser un test *t de Student* (test de comparaison des moyennes) calculé avec `stats::t.test()`, valable seulement si l'on compare deux moyennes. Pour tester si trois moyennes ou plus sont égales, on aura plutôt recours à `stats::oneway.test()`.

On peut indiquer à `gtsummary::add_p()` le test à utiliser avec le paramètre `test`.

```
trial |>
  tbl_summary(
    include = age,
    by = grade,
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
  add_p(
    test = all_continuous() ~ "oneway.test"
  )
```

Multiple parameters; naming those columns num.df, den.df

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

Table 19.13: test de comparaison des moyennes

Caractéristique	I, N = 68	II, N = 68	III, N = 64	p-valeur
Age	46 (15)	48 (14)	48 (14)	0,7
Manquant	2	6	3	

! Précision statistique

Classiquement, le test t de Student présuppose l'égalité des variances entre les deux sous-groupes, ce qui permet de former une estimation commune de la variance des deux

¹Si l'on a besoin spécifiquement d'un test de comparaison des médianes, il existe le **test de Brown-Mood** disponible dans le package `{coin}` avec la fonction `coin::median_test()`. Attention, il ne faut pas confondre ce test avec le **test de dispersion de Mood** implémenté dans la fonction `stats::mood.test()`.

échantillons (on parle de pooled variance), qui revient à une moyenne pondérée des variances estimées à partir des deux échantillons. Pour tester l'égalité des variances de deux échantillons, on peut utiliser `stats::var.test()`.

Dans le cas où l'on souhaite relaxer cette hypothèse d'égalité des variances, le test de Welch ou la correction de Satterthwaite reposent sur l'idée que l'on utilise les deux estimations de variance séparément, suivie d'une approximation des degrés de liberté pour la somme de ces deux variances.

Par défaut, la fonction `stats::t.test()` réalise un test de Welch. Pour un test classique de Student, il faut lui préciser `var.equal = TRUE`.

De manière similaire, `stats::oneway.test()` ne présuppose pas, par défaut, l'égalité des variances et généralise donc le test de Welch au cas à trois modalités ou plus. Cependant, on peut là encore indiquer `var.equal = TRUE`, auquel cas une analyse de variance (ANOVA) classique sera réalisée, que l'on peut aussi obtenir avec `stats::aov()`.

Il est possible d'indiquer à `gtsummary::add_p()` des arguments additionnels à passer à la fonction utilisée pour réaliser le test :

```
trial |>
  tbl_summary(
    include = age,
    by = trt,
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
  add_p(
    test = all_continuous() ~ "t.test",
    test.args = all_continuous() ~ list(var.equal = TRUE)
  )
```

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

Caractéristique	Drug A, N = 98	Drug B, N = 102	p-valeur
Age	47 (15)	47 (14)	0,8
Manquant	7	4	

19.2.5 Différence de deux moyennes

La fonction `gtsummary::add_difference()` permet, pour une variable continue et si la variable catégorielle spécifiée via `by` n'a que deux modalités, de calculer la différence des deux moyennes, l'intervalle de confiance de cette différence et test si cette différence est significativement différente de 0 avec `stats::t.test()`.

```
trial |>
  tbl_summary(
    include = age,
    by = trt,
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
  add_difference()
```

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

Table 19.15: différence de deux moyennes

Caractéristique	Drug A, N = 98	Drug B, N = 102	Difference	95% IC	p-valeur
Age	47 (15)	47 (14)	-0,44	-4,6 – 3,7	0,8
Manquant	7	4			

19.3 Deux variables continues

19.3.1 Représentations graphiques

La comparaison de deux variables continues se fait en premier lieu graphique, en représentant, via un nuage de points, l'ensemble des couples de valeurs. Notez ici l'application d'un niveau de transparence (`alpha`) afin de faciliter la lecture des points superposés.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  theme_light()
```

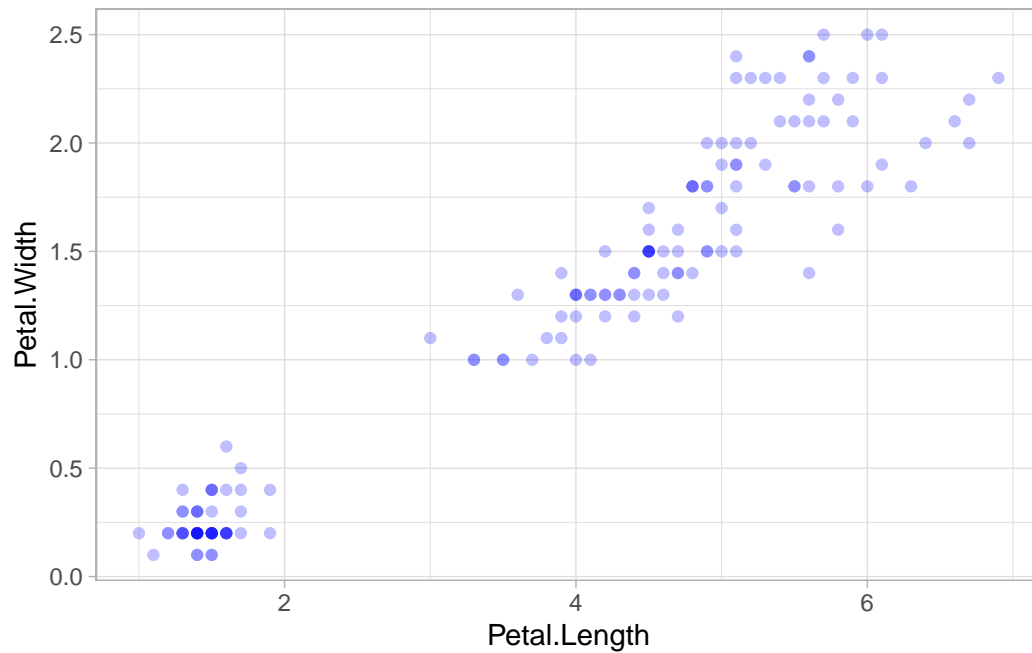


Figure 19.10: nuage de points

La géométrie `ggplot2::geom_smooth()` permet d'ajouter une courbe de tendance au graphique, avec son intervalle de confiance. Par défaut, il s'agit d'une régression polynomiale locale obtenue avec `stats::loess()`.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth() +
  geom_point(colour = "blue", alpha = .25) +
  theme_light()
```

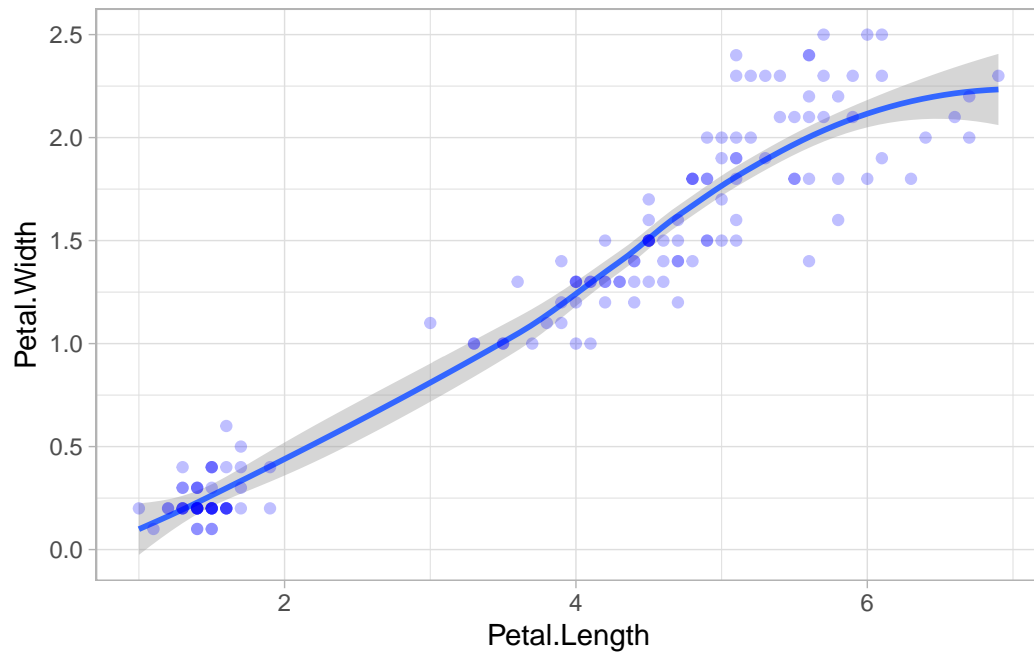



Figure 19.11: nuage de points avec une courbe de tendance

Pour afficher plutôt la droite de régression linéaire entre les deux variables, on précisera `method = "lm"`.

```
ggplot(iris) +  
  aes(x = Petal.Length, y = Petal.Width) +  
  geom_smooth(method = "lm") +  
  geom_point(colour = "blue", alpha = .25) +  
  theme_light()
```

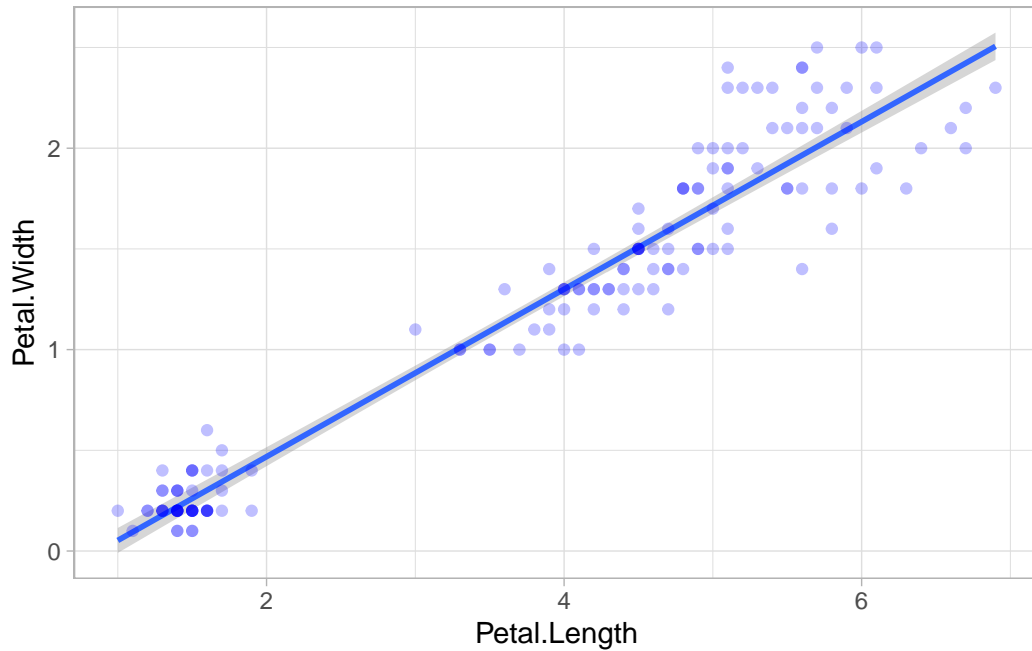


Figure 19.12: nuage de points avec droite de régression linéaire

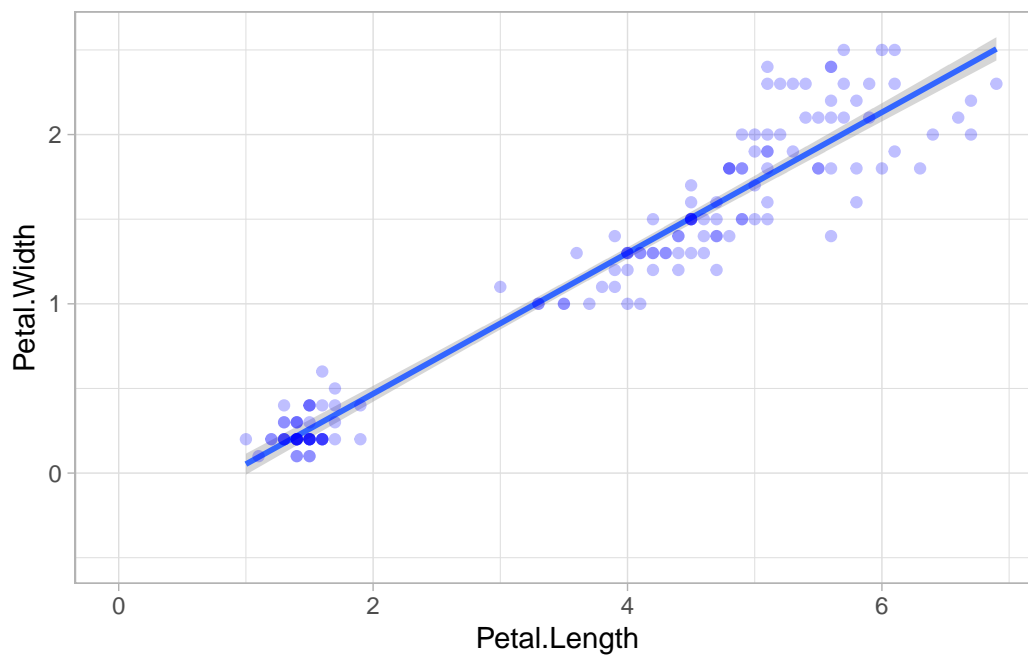
💡 Astuce pour afficher l'intercept

Supposons que nous souhaitions montrer l'endroit où la droite de régression coupe l'axe des ordonnées (soit le point sur l'axe y pour $x = 0$).

Nous pouvons étendre la surface du graphique avec `ggplot2::expand_limits()`. Cependant, cela n'étend pas pour autant la droite de régression.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth(method = "lm") +
  geom_point(colour = "blue", alpha = .25) +
  theme_light() +
  expand_limits(x = 0, y = -0.5)
```

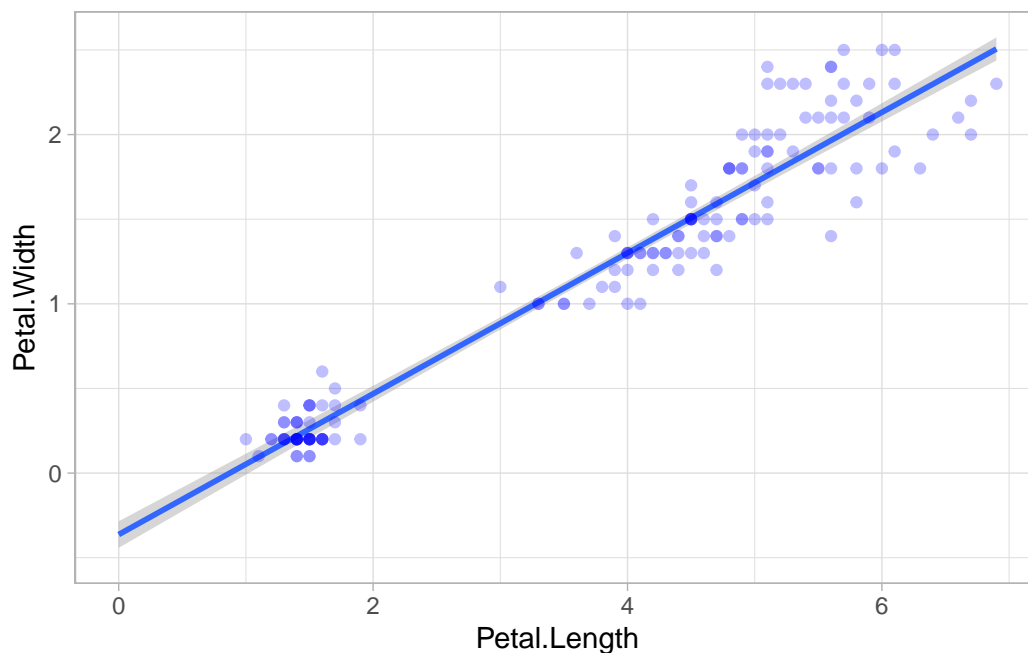
```
`geom_smooth()` using formula = 'y ~ x'
```



Une solution simple consiste à utiliser l'option `fullrange = TRUE` dans `ggplot2::geom_smooth()` pour étendre la droite de régression à l'ensemble du graphique.

```
ggplot(iris) +  
  aes(x = Petal.Length, y = Petal.Width) +  
  geom_smooth(method = "lm", fullrange = TRUE) +  
  geom_point(colour = "blue", alpha = .25) +  
  theme_light() +  
  expand_limits(x = 0, y = -0.5)
```

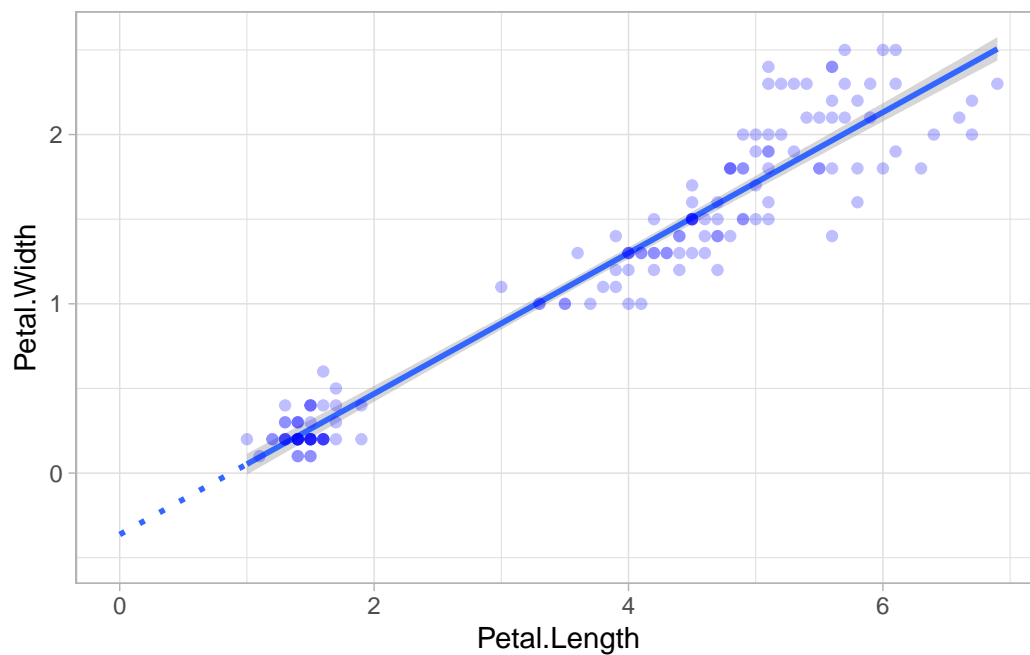
```
`geom_smooth()` using formula = 'y ~ x'
```



On peut contrôler plus finement la partie de droite à afficher avec l'argument `xseq` (liste des valeurs pour lesquelles on prédit et affiche le lissage). On peut coupler deux appels à `ggplot2::geom_smooth()` pour afficher l'extension de la droite vers la gauche en pointillés. L'option `se = FALSE` permet de ne pas calculer d'intervalles de confiance pour ce second appel.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_smooth(
    method = "lm",
    xseq = seq(0, 1, by = .1),
    linetype = "dotted",
    se = FALSE
  ) +
  geom_smooth(method = "lm") +
  geom_point(colour = "blue", alpha = .25) +
  theme_light() +
  expand_limits(x = 0, y = -0.5)
```

```
`geom_smooth()` using formula = 'y ~ x'
`geom_smooth()` using formula = 'y ~ x'
```



La géométrie `ggplot2::geom_rug()` permet d'afficher une représentation synthétique de la densité de chaque variable sur les deux axes.

```
ggplot(iris) +  
  aes(x = Petal.Length, y = Petal.Width) +  
  geom_smooth(method = "lm") +  
  geom_point(colour = "blue", alpha = .25) +  
  geom_rug() +  
  theme_light()
```

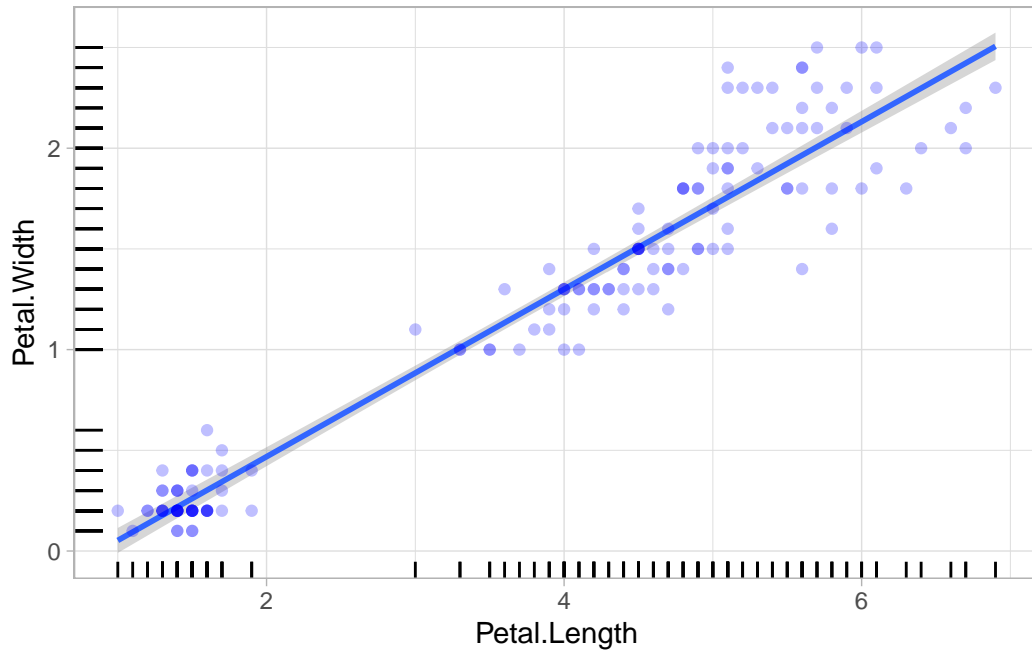


Figure 19.13: nuage de points avec représentation synthétique des densités marginales

19.3.2 Tester la relation entre les deux variables

Si l'on a besoin de calculer le coefficient de corrélation de Pearson entre deux variables, on aura recours à `stats::cor()`.

```
cor(iris$Petal.Length, iris$Petal.Width)
```

```
[1] 0.9628654
```

Pour aller plus loin, on peut calculer une régression linéaire entre les deux variables avec `stats::lm()`.

```
m <- lm(Petal.Length ~ Petal.Width, data = iris)
summary(m)
```

Call:

```
lm(formula = Petal.Length ~ Petal.Width, data = iris)
```

```

Residuals:
      Min       1Q   Median       3Q      Max
-1.33542 -0.30347 -0.02955  0.25776  1.39453

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.08356     0.07297   14.85  <2e-16 ***
Petal.Width  2.22994     0.05140   43.39  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4782 on 148 degrees of freedom
Multiple R-squared:  0.9271,    Adjusted R-squared:  0.9266
F-statistic: 1882 on 1 and 148 DF,  p-value: < 2.2e-16

```

Les résultats montrent une corrélation positive et significative entre les deux variables.

Pour une présentation propre des résultats de la régression linéaire, on utilisera `gtsummary::tbl_regression()`. La fonction `gtsummary::add_glance_source_note()` permet d'ajouter différentes statistiques en notes du tableau de résultats.

```

m |>
  tbl_regression() |>
  add_glance_source_note()

```

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

Caractéristique	Beta	95% IC	p-valeur
Petal.Width	2,2	2,1 – 2,3	<0,001

19.4 Matrice de corrélations

Le package `{GGally}` et sa fonction `GGally::ggpairs()` permettent de représenter facilement une matrice de corrélation entre plusieurs variables, tant quantitatives que qualitatives.

```
library(GGally)
ggpairs(iris)
```

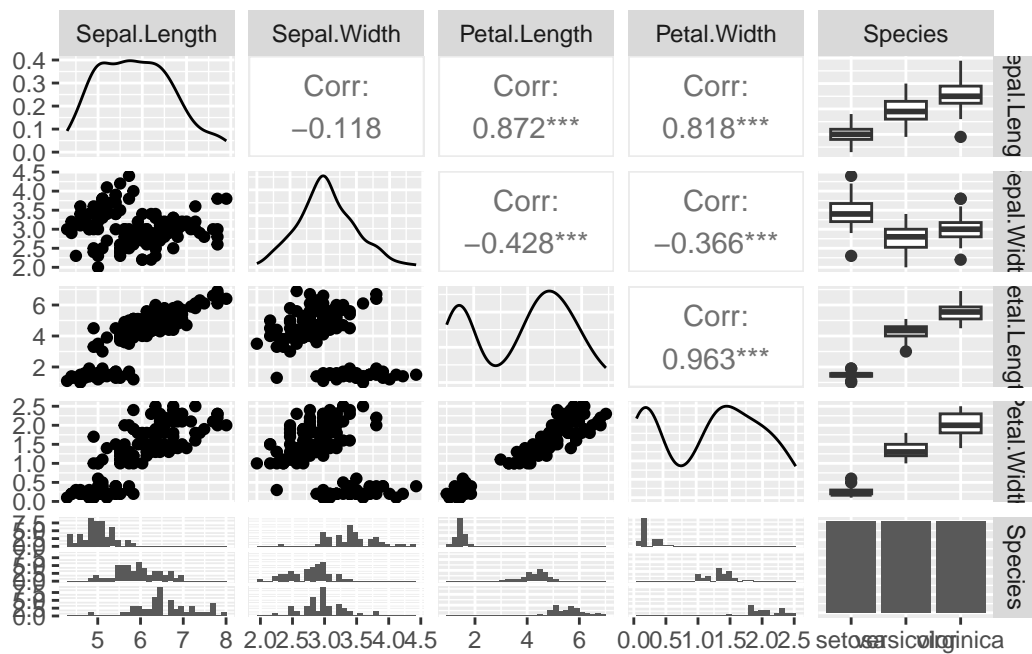


Figure 19.14: une matrice de corrélation avec `ggpairs()`

`GGally::ggpairs()` et sa petite sœur `GGally::ggduo()` offrent de nombreuses options de personnalisation qui sont détaillées sur le [site dédié du package](#).

```
ggpairs(trial, mapping = aes(colour = trt))
```




Figure 19.15: un second exemple de matrice de corrélation

19.5 webin-R

La statistique univariée est présentée dans le webin-R #03 (*statistiques descriptives avec gt-summary et esquisse*) sur [YouTube](https://youtu.be/oEF_8GXyP5c).

https://youtu.be/oEF_8GXyP5c

20 Échelles de Likert

Les échelles de Likert tirent leur nom du psychologue américain Rensis Likert qui les a développées. Elles sont le plus souvent utilisées pour des variables d'opinion. Elles sont codées sous forme de variable catégorielle et chaque item est codé selon une graduation comprenant en général cinq ou sept choix de réponse, par exemple : Tout à fait d'accord, D'accord, Ni en désaccord ni d'accord, Pas d'accord, Pas du tout d'accord.

Pour les échelles à nombre impair de choix, le niveau central permet d'exprimer une absence d'avis, ce qui rend inutile une modalité « Ne sait pas ». Les échelles à nombre pair de modalités voient l'omission de la modalité neutre et sont dites « à choix forcé ».

20.1 Exemple de données

Générons un jeu de données qui nous servira pour les différents exemples.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr      1.1.4      v readr      2.1.5
v forcats    1.0.0      v stringr    1.5.1
v ggplot2    3.5.1      v tibble     3.2.1
v lubridate  1.9.3      v tidyr      1.3.1
v purrr      1.0.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
library(labelled)
niveaux <- c(
  "Pas du tout d'accord",
  "Plutôt pas d'accord",
  "Ni d'accord, ni pas d'accord",
  "Plutôt d'accord",
```

```

  "Tout à fait d'accord"
)
set.seed(42)
df <-
  tibble(
    groupe = sample(c("A", "B"), 150, replace = TRUE),
    q1 = sample(niveaux, 150, replace = TRUE),
    q2 = sample(niveaux, 150, replace = TRUE, prob = 5:1),
    q3 = sample(niveaux, 150, replace = TRUE, prob = 1:5),
    q4 = sample(niveaux, 150, replace = TRUE, prob = 1:5),
    q5 = sample(c(niveaux, NA), 150, replace = TRUE),
    q6 = sample(niveaux, 150, replace = TRUE, prob = c(1, 0, 1, 1, 0))
  ) |>
  mutate(across(q1:q6, ~ factor(.x, levels = niveaux))) |>
  set_variable_labels(
    q1 = "Première question",
    q2 = "Seconde question",
    q3 = "Troisième question",
    q4 = "Quatrième question",
    q5 = "Cinquième question",
    q6 = "Sixième question"
  )

```

20.2 Tableau de fréquence

On peut tout à fait réaliser un tableau de fréquence classique avec `gtsummary::tbl_summary()`.

```

library(gtsummary)
df |>
  tbl_summary(include = q1:q6)

```

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

Characteristic	N = 150
Première question	
Pas du tout d'accord	39 (26%)

Characteristic	N = 150
Plutôt pas d'accord	32 (21%)
Ni d'accord, ni pas d'accord	25 (17%)
Plutôt d'accord	30 (20%)
Tout à fait d'accord	24 (16%)
Seconde question	
Pas du tout d'accord	56 (37%)
Plutôt pas d'accord	44 (29%)
Ni d'accord, ni pas d'accord	19 (13%)
Plutôt d'accord	26 (17%)
Tout à fait d'accord	5 (3.3%)
Troisième question	
Pas du tout d'accord	8 (5.3%)
Plutôt pas d'accord	17 (11%)
Ni d'accord, ni pas d'accord	29 (19%)
Plutôt d'accord	43 (29%)
Tout à fait d'accord	53 (35%)
Quatrième question	
Pas du tout d'accord	11 (7.3%)
Plutôt pas d'accord	19 (13%)
Ni d'accord, ni pas d'accord	31 (21%)
Plutôt d'accord	40 (27%)
Tout à fait d'accord	49 (33%)
Cinquième question	
Pas du tout d'accord	33 (26%)
Plutôt pas d'accord	25 (20%)
Ni d'accord, ni pas d'accord	28 (22%)
Plutôt d'accord	25 (20%)
Tout à fait d'accord	16 (13%)
Unknown	23
Sixième question	
Pas du tout d'accord	50 (33%)
Plutôt pas d'accord	0 (0%)
Ni d'accord, ni pas d'accord	50 (33%)
Plutôt d'accord	50 (33%)
Tout à fait d'accord	0 (0%)

Cependant, cela produit un tableau inutilement long, d'autant plus que les variables $q1$ à $q6$ ont les mêmes modalités de réponse. Le package `{bstfun}` propose une fonction expérimentale `bstfun::tbl_likert()` offrant un affichage plus compact.

! Important

Ce package n'est pas disponible sur **CRAN** : il est donc nécessaire de l'installer depuis **GitHub** avec la commande suivante : `remotes::install_github("MSKCC-Epi-Bio/bstfun")`. Si vous êtes sous **Windows**, il vous faudra probablement installer au préalable **RTools** qui peut être téléchargé à l'adresse <https://cran.r-project.org/bin/windows/Rtools/>.

```
library(bstfun)
```

Attachement du package : 'bstfun'

L'objet suivant est masqué depuis 'package:gtsummary':

```
trial
```

```
df |>
  tbl_likert(
    include = q1:q6
  )
```

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

	Pas du tout d'accord	Plutôt pas d'accord	Ni d'accord, ni pas d'accord	Plutôt d'accord	Tout à fait d'accord
Première question	39 (26%)	32 (21%)	25 (17%)	30 (20%)	24 (16%)
Seconde question	56 (37%)	44 (29%)	19 (13%)	26 (17%)	5 (3.3%)
Troisième question	8 (5.3%)	17 (11%)	29 (19%)	43 (29%)	53 (35%)
Quatrième question	11 (7.3%)	19 (13%)	31 (21%)	40 (27%)	49 (33%)
Cinquième question	33 (26%)	25 (20%)	28 (22%)	25 (20%)	16 (13%)

	Pas du tout d'accord	Plutôt pas d'accord	Ni d'accord, ni pas d'accord	Plutôt d'accord	Tout à fait d'accord
Sixième question	50 (33%)	0 (0%)	50 (33%)	50 (33%)	0 (0%)

On peut utiliser `add_n()` pour ajouter les effectifs totaux et `add_continuous_stat()` pour ajouter une statistique continue en traitant la variable comme un score (ici nous allons attribuer les valeurs -2, -1, 0, +1 et +2).

```
df |>
  tbl_likert(
    include = q1:q6,
    statistic = ~ "{p}%"
  ) |>
  add_n() |>
  add_continuous_stat(score_values = -2:2)
```

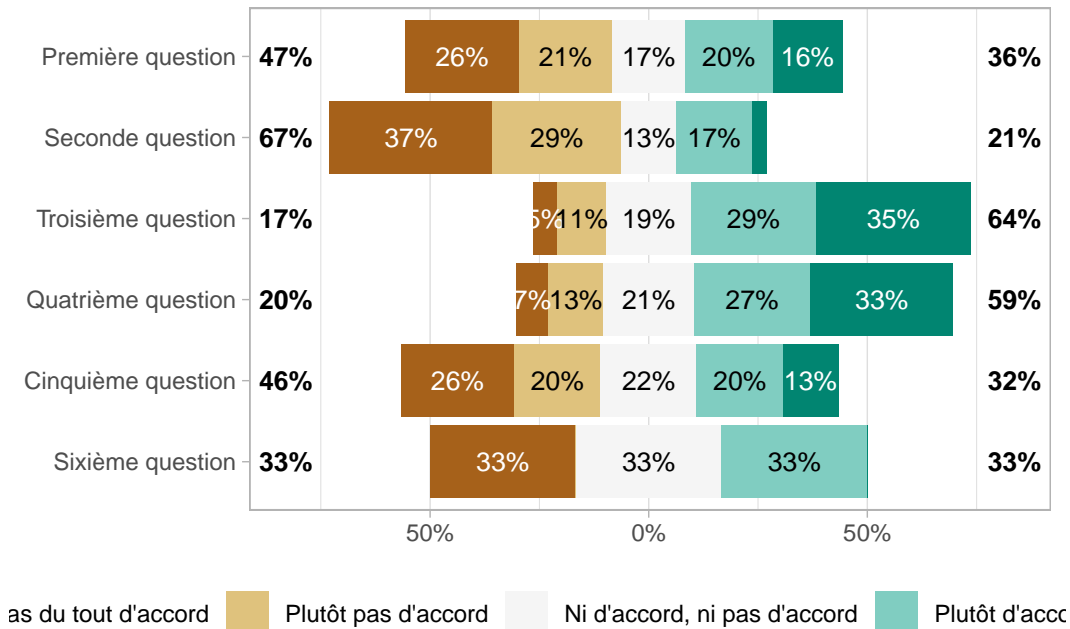
Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html>
To suppress this message, include ``message = FALSE`` in code chunk header.

		Pas du tout d'accord	Plutôt pas d'accord	Ni d'accord, ni pas d'accord	Plutôt d'accord	Tout à fait d'accord	Mean
Première question	150	26%	21%	17%	20%	16%	- 0.21
Seconde question	150	37%	29%	13%	17%	3.3%	- 0.80
Troisième question	150	5.3%	11%	19%	29%	35%	0.77
Quatrième question	150	7.3%	13%	21%	27%	33%	0.65
Cinquième question	127	26%	20%	22%	20%	13%	- 0.27
Sixième question	150	33%	0%	33%	33%	0%	- 0.33

20.3 Représentations graphiques

À partir de sa version 0.3.0, le package `{ggstats}` propose une fonction `ggstats::gglikert()` pour représenter des données de Likert sous forme d'un diagramme en barre centré sur la modalité centrale.

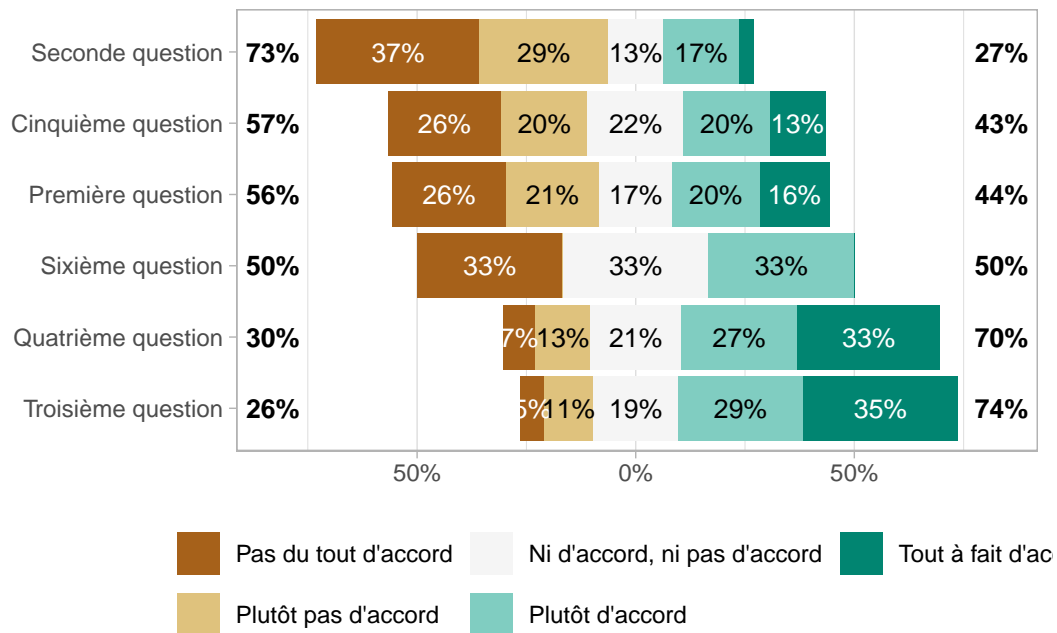
```
library(ggstats)
gglikert(df, include = q1:q6)
```



Par défaut, les pourcentages totaux ne prennent pas en compte la modalité centrale (lorsque le nombre de modalité est impair). On peut inclure la modalité centrale avec `totals_include_center = TRUE`, auquel cas la modalité centrale seront comptabilisée pour moitié de chaque côté. Le paramètre `sort` permet de trier les modalités (voir l'aide de `ggstats::gglikert()` pour plus de détails sur les différentes méthodes de tri).

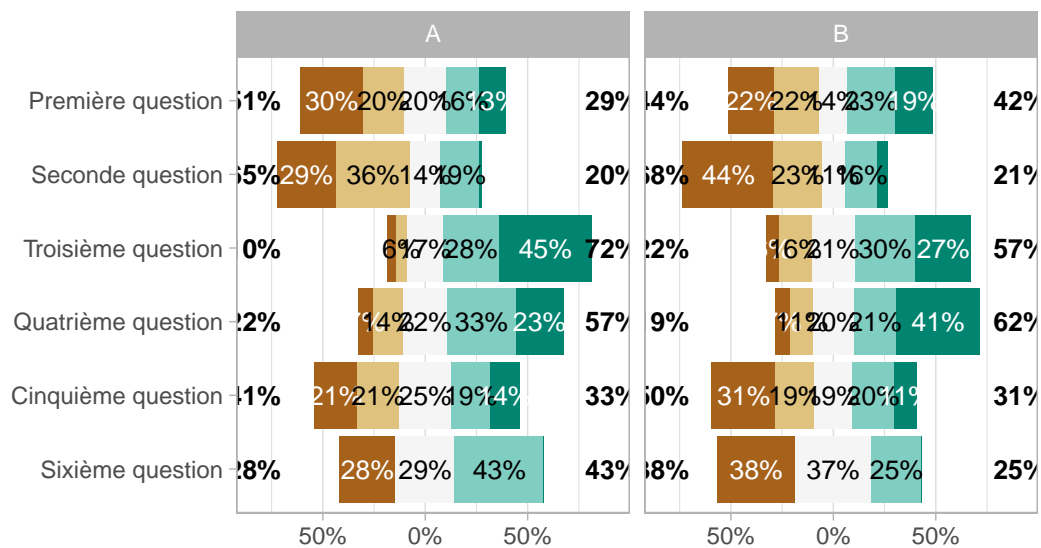
```
df |>
  gglikert(
    include = q1:q6,
    totals_include_center = TRUE,
    sort = "ascending"
  ) +
  guides(
```

```
fill = guide_legend(nrow = 2)
)
```



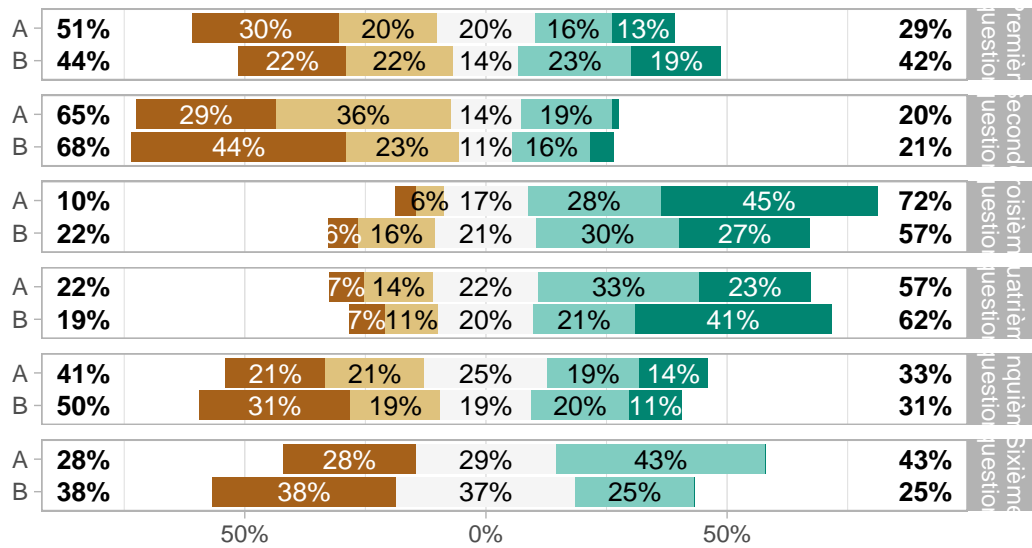
Il est possible de séparer les résultats par sous-groupe avec des facettes.

```
df |>
  gglikert(
    include = q1:q6,
    facet_cols = vars(groupe)
  )
```

as du tout d'accord Plutôt pas d'accord Ni d'accord, ni pas d'accord Plutôt d'accord

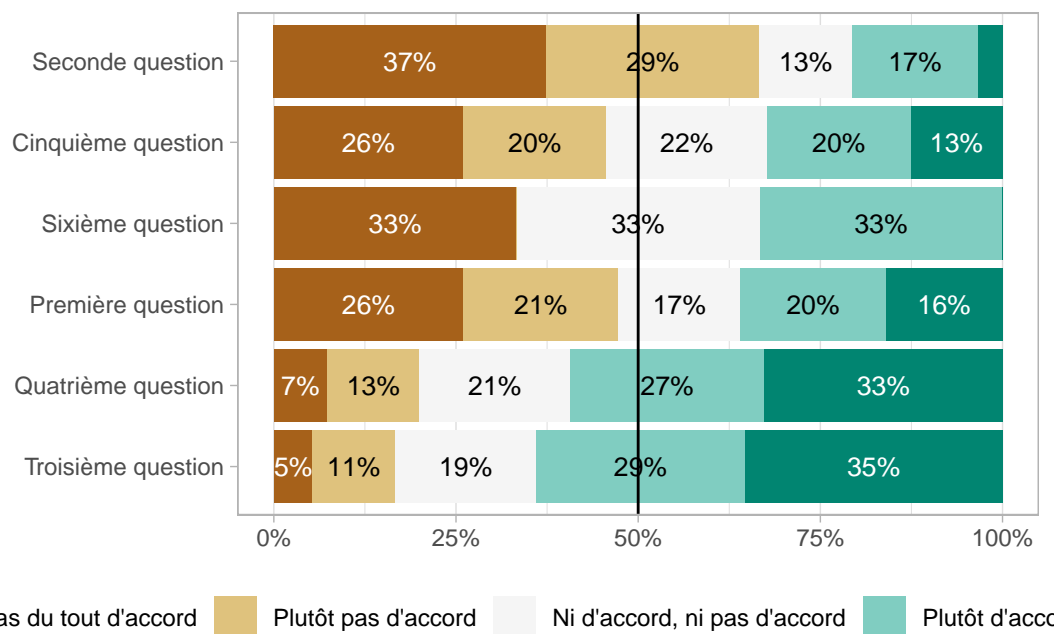
```
df |>
  gglikert(
    include = q1:q6,
    y = "groupe",
    facet_rows = vars(.question),
    facet_label_wrap = 15
  )
```



accord Plutôt pas d'accord Ni d'accord, ni pas d'accord Plutôt d'accord Tout

Une représentation alternative consiste à réaliser un graphique en barres classiques, ce que l'on peut aisément obtenir avec `ggstats::gglikert_stacked()`.

```
df |>
  gglikert_stacked(
    include = q1:q6,
    sort = "ascending",
    add_median_line = TRUE
  )
```



21 Régression linéaire

Un modèle de régression linéaire est un modèle de régression qui cherche à établir une relation linéaire entre une **variable continue**, dite expliquée, et une ou plusieurs variables, dites explicatives.

21.1 Modèle à une seule variable explicative continue

Nous avons déjà abordé très rapidement la régression linéaire dans le chapitre sur la *statistique bivariée* (cf. Section 19.3).

Reprenons le même exemple à partir du jeu de données `iris` qui comporte les caractéristiques de 150 fleurs de trois espèces différentes d'iris. Nous cherchons dans un premier temps à explorer la relation entre la largeur (*Petal.Width*) et la longueur des pétales (*Petal.Length*). Représentons cette relation sous la forme d'un nuage de points.

```
library(tidyverse)
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  labs(x = "Longueur", y = "Largeur") +
  theme_light()
```

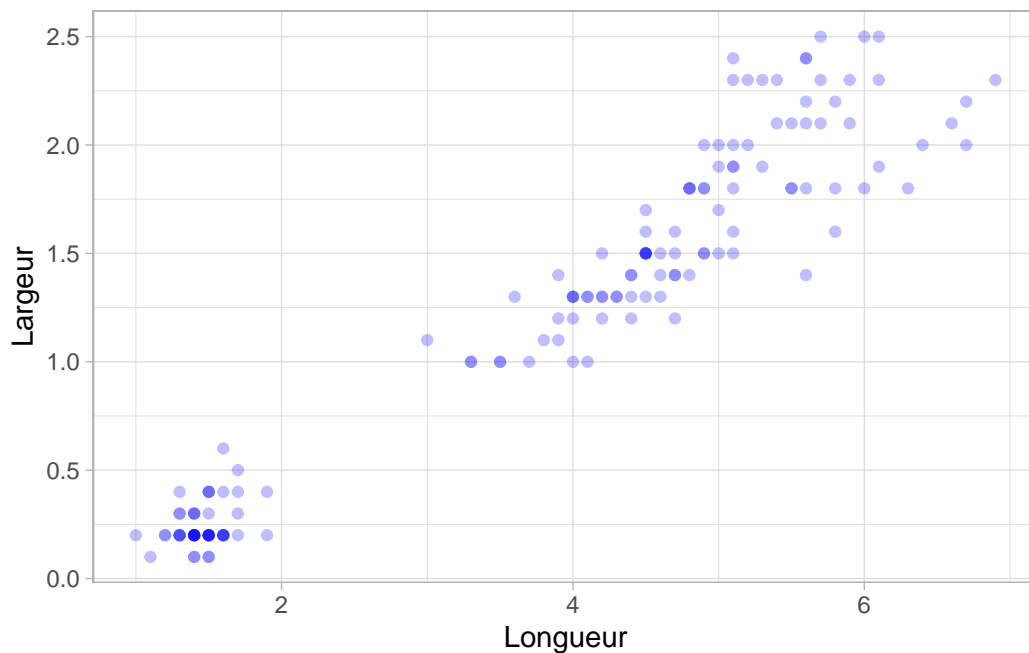


Figure 21.1: Relation entre la largeur et la longueur des pétales (nuage de points)

Il semble bien qu'il y a une **relation linéaire** entre ces deux variables, c'est-à-dire que la relation entre ces deux variables peut être représentée sous la forme d'une droite. Pour cela, on va rechercher la droite telle que la distance entre les points observés et la droite soit la plus petite possible. Cette droite peut être représentée graphique avec `ggplot2::geom_smooth()` et l'option `method = "lm"` :

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  geom_smooth(method = "lm") +
  labs(x = "Longueur", y = "Largeur") +
  theme_light()
```

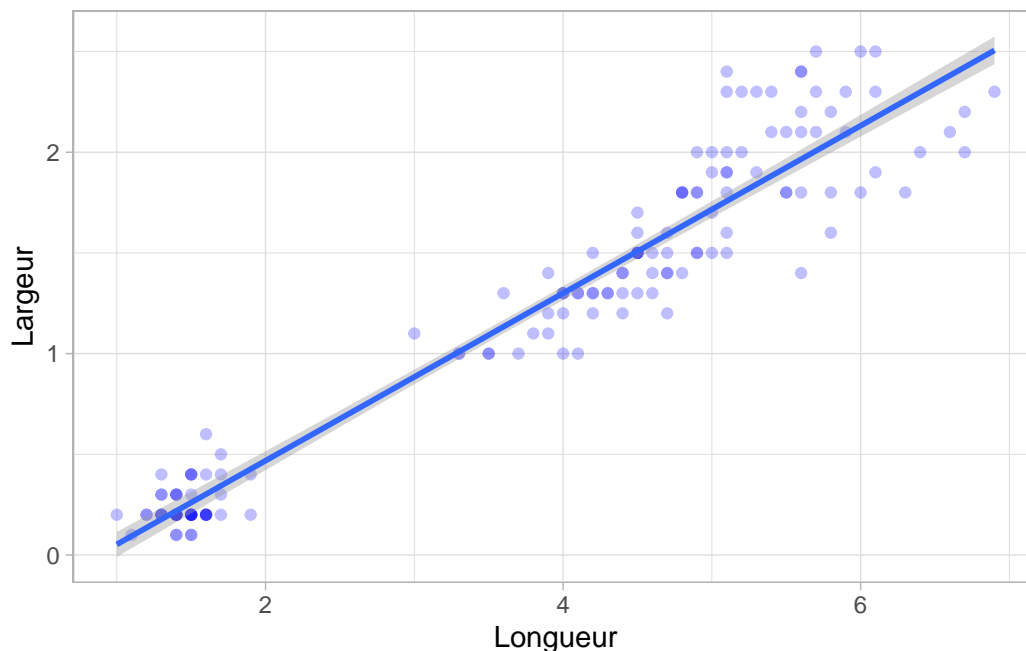


Figure 21.2: Relation linéaire entre la largeur et la longueur des pétales

La fonction de base pour calculer une régression linéaire est la fonction `stats::lm()`. On doit en premier lieu spécifier le modèle à l'aide d'une **formule** : on indique la variable à expliquer dans la partie gauche de la formule et la variable explicative dans la partie droite, les deux parties étant séparées par un tilde¹ (`~`).

Dans le cas présent, la variable *Petal.Width* fait office de variable à expliquer et *Petal.Length* de variable explicative. Le modèle s'écrit donc `Petal.Width ~ Petal.Length`.

```
mod <- lm(Petal.Width ~ Petal.Length, data = iris)
mod
```

Call:

```
lm(formula = Petal.Width ~ Petal.Length, data = iris)
```

Coefficients:

(Intercept)	Petal.Length
-0.3631	0.4158

¹Avec un clavier français, sous Windows, le caractère tilde s'obtient en pressant simultanément les touches Alt Gr et 7.

Le résultat comporte deux coefficients. Le premier, d'une valeur de 0,4158, est associé à la variable *Petal.Length* et indique la pente de la courbe (on parle de *slope* en anglais). Le second, d'une valeur de $-0,3631$, représente l'ordonnée à l'origine (*intercept* en anglais), c'est-à-dire la valeur estimée de *Petal.Width* lorsque *Petal.Length* vaut 0. Nous pouvons rendre cela plus visible en élargissant notre graphique.

```
ggplot(iris) +
  aes(x = Petal.Length, y = Petal.Width) +
  geom_point(colour = "blue", alpha = .25) +
  geom_abline(
    intercept = mod$coefficients[1],
    slope = mod$coefficients[2],
    linewidth = 1,
    colour = "red"
  ) +
  geom_vline(xintercept = 0, linewidth = 1, linetype = "dotted") +
  labs(x = "Longueur", y = "Largeur") +
  expand_limits(x = 0, y = -1) +
  theme_light()
```

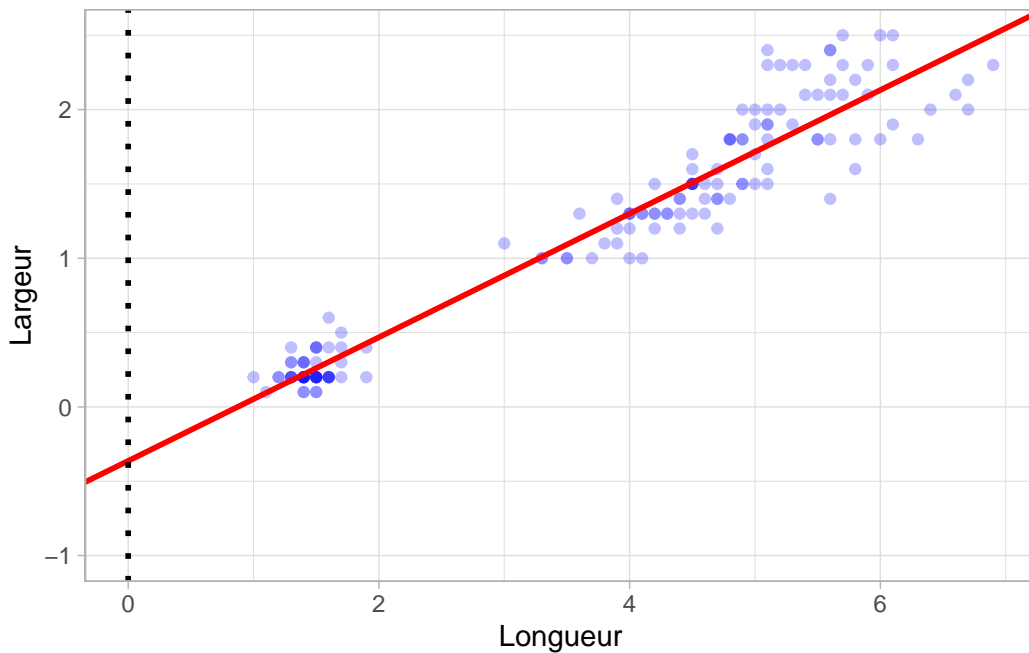


Figure 21.3: Relation linéaire entre la largeur et la longueur des pétales (représentation graphique de l'intercept)

Le modèle linéaire calculé estime donc que la relation entre nos deux variables peut s'écrire sous la forme suivante :

$$Petal.Width = 0,4158 \cdot Petal.Length - 0,3631$$

Le package `{gtsummary}` fournit `gtsummary::tbl_regression()`, une fonction bien pratique pour produire un tableau propre avec les coefficients du modèle, leur intervalle de confiance à 95% et leur p-valeurs². On précisera `intercept = TRUE` pour forcer l'affichage de l'*intercept* qui est masqué par défaut.

```
library(gtsummary)
mod %>%
  tbl_regression(intercept = TRUE)
```

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

Table 21.1: un tableau mis en forme des coefficients du modèle

Characteristic	Beta	95% CI	p-value
(Intercept)	-0.36	-0.44, -0.28	<0.001
Petal.Length	0.42	0.40, 0.43	<0.001

Les p-valeurs calculées nous indique si le coefficient est statistiquement différent de 0. En effet, pour la variable explicative, cela nous indique si la relation est statistiquement significative. Le signe du coefficient (positif ou négatif) nous indique le sens de la relation.

Astuce

Dans certains cas, si l'on suppose que la relation entre les deux variables est proportionnelle, on peut souhaiter calculer un modèle sans *intercept*. Par défaut, **R** ajoute un *intercept* à ses modèles. Pour forcer le calcul d'un modèle sans *intercept*, on ajoutera `- 1` à la formule définissant le modèle.

²Si l'on a besoin de ces informations sous la forme d'un tableau de données classique, on pourra se référer à `broom.helpers::tidy_plus_plus()`, utilisée de manière sous-jacente par `gtsummary::tbl_regression()`, ainsi qu'à la méthode `broom::tidy()`. Ces fonctions sont génériques et peut être utilisées avec une très grande variété de modèles.


```
lm(Petal.Width ~ Petal.Length - 1, data = iris)
```

Call:

```
lm(formula = Petal.Width ~ Petal.Length - 1, data = iris)
```

Coefficients:

```
Petal.Length  
0.3365
```

21.2 Modèle à une seule variable explicative catégorielle

Si dans un modèle linéaire la variable à expliquer est nécessairement continue, il est possible de définir une variable explicative catégorielle. Prenons la variable *Species*.

```
library(labelled)  
iris %>% look_for("Species")
```

```
pos variable label col_type missing values  
5 Species - fct 0 setosa  
versicolor  
virginica
```

Il s'agit d'un facteur à trois modalités. Par défaut, la première valeur du facteur (ici *setosa*) va servir de modalité de référence.

```
mod <- lm(Petal.Width ~ Species, data = iris)  
mod
```

Call:

```
lm(formula = Petal.Width ~ Species, data = iris)
```

Coefficients:

```
(Intercept) Speciesversicolor Speciesvirginica  
0.246 1.080 1.780
```

```
mod %>%
  tbl_regression(intercept = TRUE)
```

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

Table 21.2: régression linéaire avec une variable explicative catégorielle

Characteristic	Beta	95% CI	p-value
(Intercept)	0.25	0.19, 0.30	<0.001
Species			
setosa	—	—	
versicolor	1.1	1.0, 1.2	<0.001
virginica	1.8	1.7, 1.9	<0.001

Dans ce cas de figure, l'*intercept* représente la situation à la référence, donc pour l'espèce *setosa*.

Calculons les moyennes par espèce :

```
iris %>%
  group_by(Species) %>%
  summarise(mean(Petal.Width))
```

```
# A tibble: 3 x 2
  Species   `mean(Petal.Width)`
  <fct>           <dbl>
1 setosa           0.246
2 versicolor       1.33
3 virginica        2.03
```

Comme on le voit, l'*intercept* nous indique donc la moyenne observée pour l'espèce de référence (0,246).

Le coefficient associé à *versicolor* correspond à la différence par rapport à la référence (ici +1,080). Comme vous pouvez le constater, il s'agit de la différence entre la moyenne observée pour *versicolor* (1,326) et celle de la référence *setosa* (0,246) : $1,326 - 0,246 = 1,080$.

Ce coefficient est significativement différent de 0 ($p < 0,001$), indiquant que la largeur des pétales diffère significativement entre les deux espèces.

Astuce

Lorsque l'on calcule le même modèle sans *intercept*, les coefficients s'interprètent un différemment :

```
lm(Petal.Width ~ Species - 1, data = iris)
```

Call:

```
lm(formula = Petal.Width ~ Species - 1, data = iris)
```

Coefficients:

Speciessetosa	Speciesversicolor	Speciesvirginica
0.246	1.326	2.026

En l'absence d'*intercept*, trois coefficients sont calculés et il n'y a plus ici de modalité de référence. Chaque coefficient représente donc la moyenne observée pour chaque modalité. On appelle **contrastes** les différentes manières de coder des variables catégorielles dans un modèle. Nous y reviendrons plus en détail dans un chapitre dédié (cf. Chapitre 25).

21.3 Modèle à plusieurs variables explicatives

Un des intérêts de la régression linéaire est de pouvoir estimer un modèle multivarié, c'est-à-dire avec plusieurs variables explicatives. Pour cela, on listera les différentes variables explicatives dans la partie droite de la formule, séparées par le symbole +.

```
mod <- lm(  
  Petal.Width ~ Petal.Length + Sepal.Width + Sepal.Length + Species,  
  data = iris  
)  
mod
```

Call:

```
lm(formula = Petal.Width ~ Petal.Length + Sepal.Width + Sepal.Length +  
  Species, data = iris)
```

Coefficients:

(Intercept)	Petal.Length	Sepal.Width	Sepal.Length
-0.47314	0.24220	0.24220	-0.09293

Speciesversicolor	Speciesvirginica
0.64811	1.04637

```
mod %>%
  tbl_regression(intercept = TRUE)
```

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

Table 21.3: régression linéaire avec plusieurs variables explicatives

Characteristic	Beta	95% CI	p-value
(Intercept)	-0.47	-0.82, -0.12	0.008
Petal.Length	0.24	0.15, 0.34	<0.001
Sepal.Width	0.24	0.15, 0.34	<0.001
Sepal.Length	-0.09	-0.18, 0.00	0.039
Species			
setosa	—	—	
versicolor	0.65	0.40, 0.89	<0.001
virginica	1.0	0.72, 1.4	<0.001

Ce type de modèle permet d'estimer l'effet de chaque variable explicative, toutes choses égales par ailleurs. Dans le cas présent, on s'aperçoit que la largeur des pétales diffère significativement selon les espèces, est fortement corrélée positivement à la longueur du pétale et la largeur du sépale et qu'il y a, lorsque l'on ajuste sur l'ensemble des autres variables, une relation négative (faiblement significative) avec la longueur du sépale.

Lorsque le nombre de coefficients est élevé, une représentation graphique est souvent plus facile à lire qu'un tableau. On parle alors de graphique en forêt ou *forest plot* en anglais. Rien de plus facile ! Il suffit d'avoir recours à `ggstats::ggcoef_model()`.

```
library(ggstats)
ggcoef_model(mod)
```

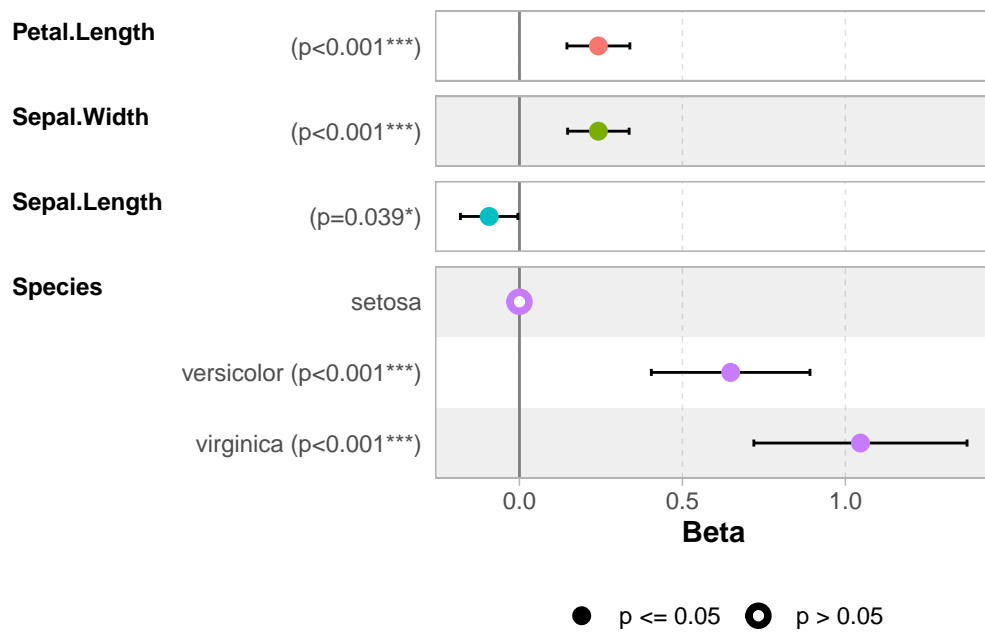


Figure 21.4: un graphique en forêt des coefficients du modèle

22 Régression logistique binaire

Dans le chapitre précédent (Chapitre 21), nous avons abordé la régression linéaire pour les variables continues. Pour analyser une variable catégorielle, il est nécessaire d'avoir recours à d'autres types de modèles. Les modèles linéaires généralisés (*generalized linear models* ou *GLM* en anglais) sont une généralisation de la régression linéaire grâce à l'utilisation d'une fonction de **lien** utilisée pour transformer la variable à expliquer et pouvoir ainsi retomber sur un modèle linéaire classique. Il existe de multiples fonctions de lien correspondant à tout autant de modèles. Par exemple, on pourra utiliser un modèle de Poisson pour une variable entière positive ou un modèle d'incidence.

Pour une variable binaire (c'est-à-dire du type oui / non ou vrai / faux), les modèles les plus courants utilisent les fonctions de lien *probit* ou *logit*, cette dernière fonction correspondent à la **régression logistique binaire** (modèle *logit*). Comme pour la régression linéaire, les variables explicatives peuvent être continues et/ou catégorielles.

22.1 Préparation des données

Dans ce chapitre, nous allons encore une fois utiliser les données de l'enquête *Histoire de vie*, fournies avec l'extension `{questionr}`.

```
data(hdv2003, package = "questionr")
d <- hdv2003
```

À titre d'exemple, nous allons étudier l'effet de l'âge, du sexe, du niveau d'étude, de la pratique religieuse et du nombre moyen d'heures passées à regarder la télévision par jour sur la probabilité de pratiquer un sport.

En premier lieu, il importe de vérifier, par exemple avec `labelled::look_for()`, que notre variable d'intérêt (ici *sport*) est correctement codée, c'est-à-dire que la première modalité correspondent à la référence (soit ne pas avoir vécu l'évènement d'intérêt) et que la seconde modalité corresponde au fait d'avoir vécu l'évènement.

```
library(labelled)
d |> look_for("sport")
```

```
pos variable label col_type missing values
19 sport      -      fct      0      Non
                                Oui
```

Dans notre exemple, la modalité Non est déjà la première modalité. Il n'y a donc pas besoin de modifier notre variable.

Il faut également la présence éventuelle de données manquantes (NA)¹. Les observations concernées seront tout simplement exclues du modèle lors de son calcul. Ce n'est pas notre cas ici.

Astuce

Alternativement, on pourra aussi coder notre variable à expliquer sous forme booléenne (FALSE / TRUE) ou numériquement en 0/1.

Il est possible d'indiquer un facteur à plus de deux modalités. Dans une telle situation, **R** considérera que tous les modalités, sauf la modalité de référence, est une réalisation de la variable d'intérêt. Cela serait correct, par exemple, si notre variable sport était codée ainsi : Non, Oui, de temps en temps, Oui, régulièrement. Cependant, afin d'éviter tout risque d'erreur ou de mauvaise interprétation, il est vivement conseillé de recoder au préalable sa variable d'intérêt en un facteur à deux modalités.

La notion de **modalité de référence** s'applique également aux variables explicatives catégorielles. En effet, dans un modèle, tous les coefficients sont calculés par rapport à la modalité de référence (cf. Section 21.2). Il importe donc de choisir une modalité de référence qui fasse sens afin de faciliter l'interprétation. Par ailleurs, ce choix doit dépendre de la manière dont on souhaite présenter les résultats (le *data storytelling* est essentiel). De manière générale on évitera de choisir comme référence une modalité peu représentée dans l'échantillon ou bien une modalité correspondant à une situation atypique.

Prenons l'exemple de la variable *sexe*. Souhaite-t-on connaître l'effet d'être une femme par rapport au fait d'être un homme ou bien l'effet d'être un homme par rapport au fait d'être une femme ? Si l'on opte pour le second, alors notre modalité de référence sera le sexe féminin. Comme est codée cette variable ?

```
d |> look_for("sexe")
```

```
pos variable label col_type missing values
3  sexe      -      fct      0      Homme
                                Femme
```

¹Pour visualiser le nombre de données manquantes (NA) de l'ensemble des variables d'un tableau, on pourra avoir recours à `questionr::freq.na()`.

La modalité Femme s'avère ne pas être la première modalité. Nous devons appliquer la fonction `forcats::fct_relevel()` ou la fonction `stats::relevel()` :

```
library(tidyverse)
d <- d |>
  mutate(sexe = sexe |> fct_relevel("Femme"))
```

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

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

Données labellisées

Si l'on utilise des données labellisées (voir Chapitre 12), nos variables catégorielles seront stockées sous la forme d'un vecteur numérique avec des étiquettes. Il sera donc nécessaire de convertir ces variables en facteurs, tout simplement avec `labelled::to_factor()` ou `labelled::unlabelled()`.

Les variables *age* et *heures.tv* sont des variables quantitatives. Il importe de vérifier qu'elles sont bien enregistrées en tant que variables numériques. En effet, il arrive parfois que dans le fichier source les variables quantitatives soient renseignées sous forme de valeur textuelle et non sous forme numérique.

```
d |> look_for("age", "heures")
```

pos	variable	label	col_type	missing	values
2	age	-	int	0	
20	heures.tv	-	dbl	5	

Nos deux variables sont bien renseignées sous forme numérique (respectivement des entiers et des nombres décimaux).

Cependant, l'effet de l'âge est rarement linéaire. Un exemple trivial est par exemple le fait d'occuper un emploi qui sera moins fréquent aux jeunes âges et aux âges élevés. Dès lors, on pourra transformer la variable *age* en groupe d'âges (et donc en variable catégorielle) avec la fonction `cut()` (cf. Section 9.4) :


```
d <- d |>
  mutate(
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      )
  )
d$groupe_ages |> questionr::freq()
```

	n	%	val%
18-24 ans	169	8.5	8.5
25-44 ans	706	35.3	35.3
45-64 ans	745	37.2	37.2
65 ans et plus	380	19.0	19.0

Jetons maintenant un œil à la variable *nivetud* :

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

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

En premier lieu, cette variable est détaillée en pas moins de huit modalités dont certaines sont peu représentées (seulement 39 individus soit 2 % n'ont jamais fait d'études par exemple). Afin d'améliorer notre modèle logistique, il peut être pertinent de regrouper certaines modalités (cf. Section 9.3) :

```
d <- d |>
  mutate(
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
      )
  )
d$etudes |> questionr::freq()
```

	n	%	val%
Primaire	466	23.3	24.7
Secondaire	387	19.4	20.5
Technique / Professionnel	594	29.7	31.5
Supérieur	441	22.0	23.4
NA	112	5.6	NA

Notre variable comporte également 112 individus avec une valeur manquante. Si nous conservons cette valeur manquante, ces 112 individus seront, par défaut, exclus de l'analyse. Ces valeurs manquantes n'étant pas négligeable (5,6 %), nous pouvons également faire le choix de considérer ces valeurs manquantes comme une modalité supplémentaire. Auquel cas, nous utiliserons la fonction `forcats::fct_na_value_to_level()` :

```
d$etudes <- d$etudes |>
  fct_na_value_to_level("Non documenté")
```

Enfin, pour améliorer les différentes sorties (tableaux et figures), nous allons ajouter des étiquettes de variables (cf. Chapitre 11) avec `labelled::set_variable_labels()`.

```
d <- d |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
  )
```

```

    relig = "Rapport à la religion",
    heures.tv = "Heures de télévision / jour"
)

```

i Code récapitulatif (préparation des données)

```

data(hdv2003, package = "questionr")
d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
      ) |>
      fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    relig = "Rapport à la religion",
    heures.tv = "Heures de télévision / jour"
  )

```

22.2 Statistiques descriptives

Avant toute analyse multivariée, il est toujours bon de procéder à une analyse descriptive bivariée simple, tout simplement avec `gtsummary::tbl_summary()`. Ajoutons quelques tests de comparaison avec `gtsummary::add_p()`. Petite astuce : `gtsummary::modify_spanning_header()` permet de rajouter un en-tête sur plusieurs colonnes.

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",", big.mark = " ")
```

```
d |>
  tbl_summary(
    by = sport,
    include = c(sexe, groupe_ages, etudes, relig, heures.tv)
  ) |>
  add_overall(last = TRUE) |>
  add_p() |>
  bold_labels() |>
  modify_spanning_header(
    update = all_stat_cols() ~ "**Pratique un sport ?**"
  )
```

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

Table 22.1: Pratique d'un sport selon différentes variables explicatives (analyse bivariée)

Caractéristique	Non, N = 1 277	Oui, N = 723 000	Total, N = 2 000	p-valeur
Sexe				<0,001
Femme	747 (58%)	354 (49%)	1 101 (55%)	
Homme	530 (42%)	369 (51%)	899 (45%)	
Groupe d'âges				<0,001
18-24 ans	58 (4,5%)	111 (15%)	169 (8,5%)	
25-44 ans	359 (28%)	347 (48%)	706 (35%)	
45-64 ans	541 (42%)	204 (28%)	745 (37%)	
65 ans et plus	319 (25%)	61 (8,4%)	380 (19%)	
Niveau d'études				<0,001
Primaire	416 (33%)	50 (6,9%)	466 (23%)	

Table 22.1: Pratique d'un sport selon différentes variables explicatives (analyse bivariée)

Caractéristique	Non, N = 1 277	Oui, N = 723	Total, N = 2 000	p-valeur
Secondaire	270 (21%)	117 (16%)	387 (19%)	0,14
Technique / Professionnel	378 (30%)	216 (30%)	594 (30%)	
Supérieur	186 (15%)	255 (35%)	441 (22%)	
Non documenté	27 (2,1%)	85 (12%)	112 (5,6%)	
Rapport à la religion				
Pratiquant regulier	182 (14%)	84 (12%)	266 (13%)	<0,001
Pratiquant occasionnel	295 (23%)	147 (20%)	442 (22%)	
Appartenance sans pratique	473 (37%)	287 (40%)	760 (38%)	
Ni croyance ni appartenance	239 (19%)	160 (22%)	399 (20%)	
Rejet	60 (4,7%)	33 (4,6%)	93 (4,7%)	
NSP ou NVPR	28 (2,2%)	12 (1,7%)	40 (2,0%)	<0,001
Heures de télévision / jour	2,00 (1,00 – 3,00)	2,00 (1,00 – 3,00)	2,00 (1,00 – 3,00)	
Manquant	2	3	5	

22.3 Calcul de la régression logistique binaire

La spécification d'une régression logistique se fait avec `stats::glm()` et est très similaire à celle d'une régression linéaire simple (cf. Section 21.3) : on indique la variable à expliquer suivie d'un tilde (~) puis des variables explicatives séparées par un plus (+)². Il faut indiquer à `glm()` la famille du modèle souhaité : on indiquera simplement `family = binomial` pour un modèle *logit*³.

```
mod <- glm(
  sport ~ sexe + groupe_ages + etudes + relig + heures.tv,
  family = binomial,
  data = d
)
```

Pour afficher les résultats du modèle, le plus simple est d'avoir recours à `gtsummary::tbl_regression()`.

²Il est possible de spécifier des modèles plus complexes, notamment avec des effets d'interaction, qui seront aborder plus loin (cf. Chapitre 26).

³Pour un modèle *probit*, on indiquera `family = binomial("probit")`.

```
mod |>
  tbl_regression(intercept = TRUE) |>
  bold_labels()
```

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

Table 22.2: Facteurs associés à la pratique d'un sport (régression logistique binaire)

Caractéristique	log(OR)	95% IC	p-valeur
(Intercept)	-0,80	-1,4 – -0,17	0,014
Sexe			
Femme	—	—	
Homme	0,44	0,23 – 0,65	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	-0,42	-0,87 – 0,03	0,065
45-64 ans	-1,1	-1,6 – -0,62	<0,001
65 ans et plus	-1,4	-1,9 – -0,85	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	0,95	0,57 – 1,3	<0,001
Technique / Professionnel	1,0	0,68 – 1,4	<0,001
Supérieur	1,9	1,5 – 2,3	<0,001
Non documenté	2,2	1,5 – 2,8	<0,001
Rapport à la religion			
Pratiquant régulier	—	—	
Pratiquant occasionnel	-0,02	-0,39 – 0,35	>0,9
Appartenance sans pratique	-0,01	-0,35 – 0,34	>0,9
Ni croyance ni appartenance	-0,22	-0,59 – 0,16	0,3
Rejet	-0,38	-0,95 – 0,17	0,2
NSP ou NVPR	-0,08	-0,92 – 0,70	0,8
Heures de télévision / jour	-0,12	-0,19 – -0,06	<0,001

22.4 Interpréter les coefficients

L'*intercept* traduit la situation à la référence (i.e. toutes les variables catégorielles à leur modalité de référence et les variables continues à 0), après transformation selon la fonction de lien (i.e. après la transformation *logit*).

Illustrons cela. Supposons donc une personne de sexe féminin, âgée entre 18 et 24 ans, de niveau d'étude primaire, pratiquante régulière et ne regardant pas la télévision (situation de référence). Seul l'*intercept* s'applique dans le modèle, et donc le modèle prédit que sa probabilité de faire du sport est de $-0,80$ selon l'échelle *logit*. Retraduisons cela en probabilité classique avec la fonction *logit inverse*.

```
logit_inverse <- binomial("logit") |> purrr::pluck("linkinv")
logit_inverse(-0.80)
```

```
[1] 0.3100255
```

Selon le modèle, la probabilité que cette personne fasse du sport est donc de 31%.

Prenons maintenant une personne identique mais de sexe masculin. Nous devons donc considérer, en plus de l'*intercept*, le coefficient associé à la modalité Homme. Sa probabilité de faire du sport est donc :

```
logit_inverse(-0.80 + 0.44)
```

```
[1] 0.4109596
```

Le coefficient associé à Homme est donc un modificateur par rapport à la situation de référence.

Enfin, considérons que cette dernière personne regarde également la télévision 2 heures en moyenne par jour. Nous devons alors considérer le coefficient associé à la variable *heures.tv* et, comme il s'agit d'une variable continue, multiplier ce coefficient par 2, car le coefficient représente le changement pour un incrément de 1 unité.

```
logit_inverse(-0.80 + 0.44 + 2 * -0.12)
```

```
[1] 0.3543437
```

Il est crucial de bien comprendre comment dans quels cas et comment chaque coefficient est utilisé par le modèle.

Le package `{breakdown}` permet de mieux visualiser notre dernier exemple.

```

individu3 <- d[1, ]
individu3$sexe[1] <- "Homme"
individu3$groupe_ages[1] <- "18-24 ans"
individu3$etudes[1] <- "Primaire"
individu3$relig[1] <- "Pratiquant regulier"
individu3$heures.tv[1] <- 2

```

```

library(breakDown)
logit <- function(x) exp(x) / (1 + exp(x))
plot(
  broken(mod, individu3, predict.function = betas),
  trans = logit
) +
  scale_y_continuous(
    labels = scales::label_percent(),
    breaks = 0:5/5,
    limits = c(0, 1)
  )

```

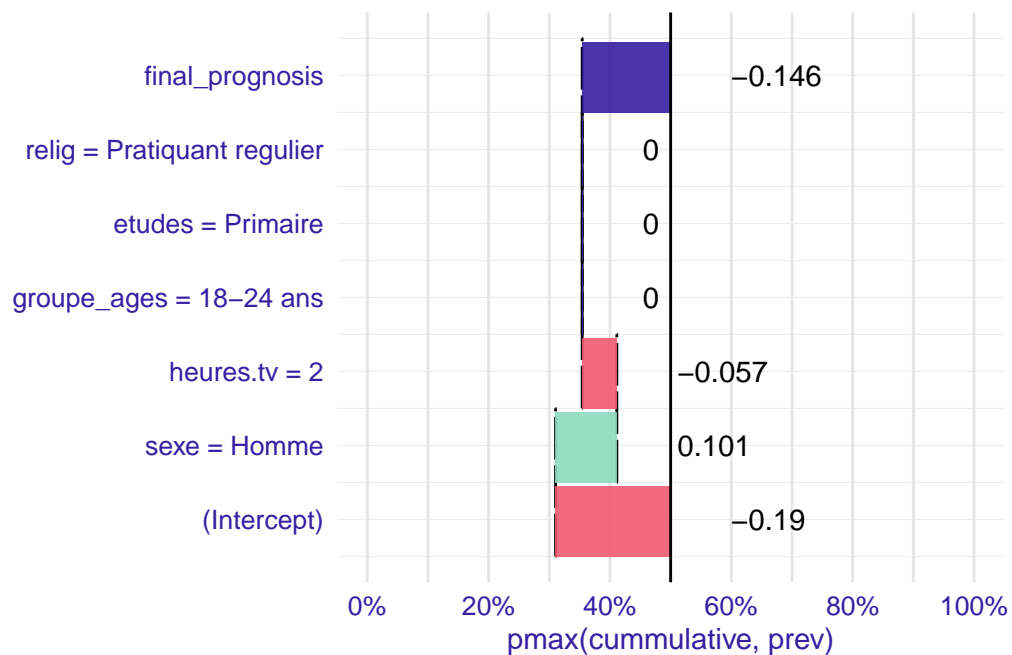


Figure 22.1: Décomposition de la probabilité de faire du sport de l'individu 3

22.5 La notion d'*odds ratio*

L'un des intérêts de la régression logistique *logit* réside dans le fait que l'exponentiel des coefficients correspond à des *odds ratio* ou rapport des côtes en français.

Astuce

Pour comprendre la notion de côte (*odd* en anglais), on peut se référer aux paris sportifs. Par exemple, lorsque les trois quarts des parieurs parient que le cheval A va remporter la course, on dit alors que ce cheval à une côte de trois contre un (trois personnes parient qu'il va gagner contre une personne qu'il va perdre). Prenons un autre cheval : si les deux tiers pensent que le cheval B va perdre (donc un tiers pense qu'il va gagner), on dira alors que sa côte est de un contre deux (une personne pense qu'il va gagner contre deux qu'il va perdre).

Si l'on connaît la proportion ou probabilité p d'avoir vécu ou de vivre un évènement donné (ici gagner la course), la côte (l'*odd*) s'obtient avec la formule suivante : $p/(1-p)$. La côte du cheval A est bien $0,75/(1-0,75) = 0,75/0,25 = 3$ est celle du cheval B $(1/3)/(2/3) = 1/2 = 0,5$.

Pour comparer deux côtes (par exemple pour savoir si le cheval A a une probabilité plus élevée de remporter la course que le cheval B, selon les parieurs), on calculera tout simplement le rapport des côtes ou *odds ratio* (OR) : $OR_{A/B} = Odds_A/Odds_B = 3/0,5 = 6$.

Ce calcul peut se faire facilement dans **R** avec la fonction `questionr::odds.ratio()`.

```
questionr::odds.ratio(.75, 1/3)
```

```
[1] 6
```

L'*odds ratio* est donc égal à 1 si les deux côtes sont identiques, est supérieur à 1 si le cheval A a une probabilité supérieure à celle du cheval B, et inférieur à 1 si c'est probabilité est inférieure.

On le voit, par construction, l'*odds ratio* de B par rapport à A est l'inverse de celui de A par rapport à B : $OR_{B/A} = 1/OR_{A/B}$.

Pour afficher les *odds ratio* il suffit d'indiquer `exponentiate = TRUE` à `gtsummary::tbl_regression()`.

```
mod |>
tbl_regression(exponentiate = TRUE) |>
bold_labels()
```

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>

To suppress this message, include ``message = FALSE`` in code chunk header.

Table 22.3: Facteurs associés à la pratique d'un sport (*odds ratios*)

Caractéristique	OR	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	1,55	1,26 – 1,91	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	0,66	0,42 – 1,03	0,065
45-64 ans	0,34	0,21 – 0,54	<0,001
65 ans et plus	0,25	0,15 – 0,43	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	2,59	1,77 – 3,83	<0,001
Technique / Professionnel	2,86	1,98 – 4,17	<0,001
Supérieur	6,63	4,55 – 9,80	<0,001
Non documenté	8,59	4,53 – 16,6	<0,001
Rapport à la religion			
Pratiquant regulier	—	—	
Pratiquant occasionnel	0,98	0,68 – 1,42	>0,9
Appartenance sans pratique	0,99	0,71 – 1,40	>0,9
Ni croyance ni appartenance	0,81	0,55 – 1,18	0,3
Rejet	0,68	0,39 – 1,19	0,2
NSP ou NVPR	0,92	0,40 – 2,02	0,8
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

Pour une représentation visuelle, graphique en forêt ou *forest plot* en anglais, on aura tout simplement recours à `ggstats::ggcoef_model()`.

```
mod |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

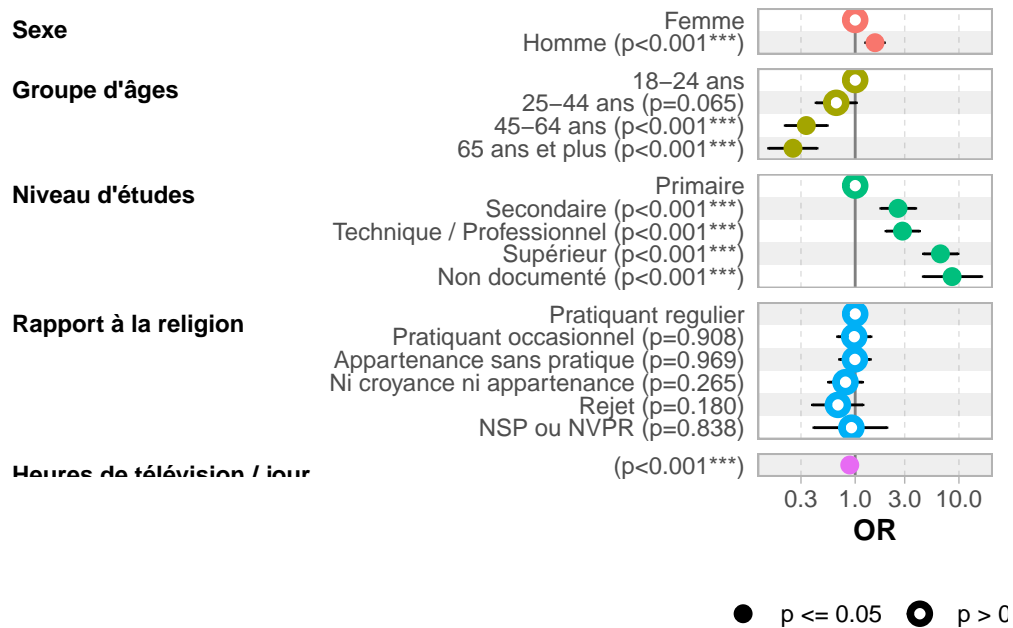


Figure 22.2: Facteurs associés à la pratique d'un sport (*forest plot*)

On pourra alternativement préférer `ggstats::ggcoef_table()`⁴ qui affiche un tableau des coefficients à la droite du *forest plot*.

```
mod |>
  ggstats::ggcoef_table(exponentiate = TRUE)
```

⁴`ggstats::ggcoef_table()` est disponible à partir de la version 0.4.0 de `{ggstats}`.

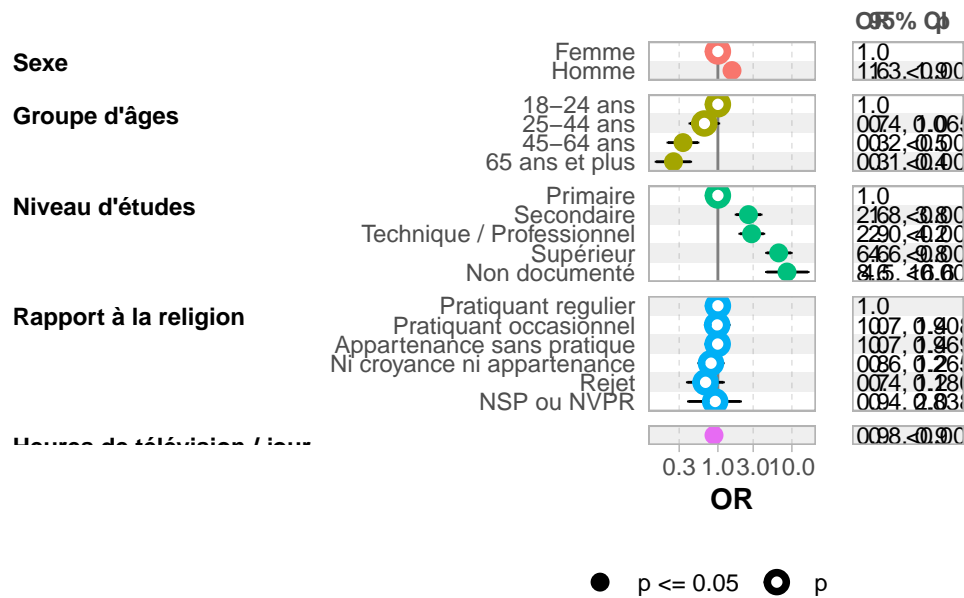


Figure 22.3: Facteurs associés à la pratique d'un sport (*forest plot* avec table des coefficients)

Note

Lorsque l'on réalise un *forest plot* de coefficients exponentialisés tels que des *odds ratios*, une bonne pratique consiste à utiliser une échelle logarithmique. En effet, l'inverse d'un *odds ratio* de 2 est 0,5. Avec une échelle logarithmique, la distance entre 0,5 et 1 est égale à celle entre 1 et 2. Sur la figure précédente, vous pourrez noter que `ggstats::ggcoef_model()` applique automatiquement une échelle logarithmique lorsque `exponentiate = TRUE`.

Quelques références : [Forest Plots: Linear or Logarithmic Scale?](#) ou encore [Graphing Ratio Measures on Forest Plot](#).

Mise en garde

En rédigeant les résultats de la régression, il faudra être vigilant à ne pas confondre les *odds ratios* avec des *prevalence ratios*. Avec un *odds ratio* de 1,55, il serait tentant d'écrire que les hommes ont une probabilité 55% supérieure de pratique un sport que les femmes (toutes choses égales par ailleurs). Une telle formulation correspond à un *prevalence ratio* (rapport des prévalences en français) ou *risk ratio* (rapport des risques), à savoir diviser la probabilité de faire du sport des hommes par celle des femmes, $p_{\text{hommes}}/p_{\text{femmes}}$. Or, cela ne correspond pas à la formule de l'*odds ratio*, à savoir

$$(p_{\text{hommes}}/(1 - p_{\text{hommes}}))/(p_{\text{femmes}}/(1 - p_{\text{femmes}})).$$

Lorsque le phénomène étudié est rare et donc que les probabilités sont faibles (inférieures à quelques pour-cents), alors il est vrai que les *odds ratio* sont approximativement égaux aux *prevalence ratios*. Mais ceci n'est plus du tout vrai pour des phénomènes plus fréquents.

22.6 Afficher les écarts-types plutôt que les intervalles de confiance

La manière de présenter les résultats d'un modèle de régression varie selon les disciplines, les champs thématiques et les revues. Si en sociologie et épidémiologie on aurait plutôt tendance à afficher les *odds ratio* avec leur intervalle de confiance, il est fréquent en économétrie de présenter plutôt les coefficients bruts et leurs écarts-types (*standard deviation* ou *sd* en anglais). De même, plutôt que d'ajouter une colonne avec les p valeurs, un usage consiste à afficher des étoiles de significativité à la droite des coefficients significatifs.

Pour cela, on pourra personnaliser le tableau produit avec `gtsummary::tbl_regression()`, notamment avec `gtsummary::add_significance_stars()` pour l'ajout des étoiles de significativité, ainsi que `gtsummary::modify_column_hide()` et `gtsummary::modify_column_unhide()` pour afficher / masquer les colonnes du tableau produit⁵.

```
mod |>
  tbl_regression() |>
  add_significance_stars() |>
  modify_column_hide(c("ci", "p.value")) |>
  modify_column_unhide("std.error") |>
  bold_labels()
```

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

Table 22.4: Présentation économétrique des facteurs associés à la pratique d'un sport

Caractéristique	log(OR)	ET
Sexe		
Femme	—	—
Homme	0,44***	0,106
Groupe d'âges		

⁵La liste des colonnes disponibles peut être obtenues avec `mod |> tbl_regression() |> purrr::pluck("table_body") |> colnames()`.

Table 22.4: Présentation économétrique des facteurs associés à la pratique d'un sport

Caractéristique	log(OR)	ET
18-24 ans	—	—
25-44 ans	-0,42	0,228
45-64 ans	-1,1***	0,238
65 ans et plus	-1,4***	0,274
Niveau d'études		
Primaire	—	—
Secondaire	0,95***	0,197
Technique / Professionnel	1,0***	0,190
Supérieur	1,9***	0,195
Non documenté	2,2***	0,330
Rapport à la religion		
Pratiquant regulier	—	—
Pratiquant occasionnel	-0,02	0,189
Appartenance sans pratique	-0,01	0,175
Ni croyance ni appartenance	-0,22	0,193
Rejet	-0,38	0,286
NSP ou NVPR	-0,08	0,411
Heures de télévision / jour	-0,12***	0,034

Les économistes pourraient préférer le package `{modelsummary}` à `{gtsummary}`. Ces deux packages ont un objectif similaire (la production de tableaux statistiques) mais abordent cet objectif avec des approches différentes. Il faut noter que `modelsummary::modelsummary()` n'affiche pas les modalités de référence, ni les étiquettes de variable.

```
mod |> modelsummary::modelsummary(stars = TRUE)
```

La fonction `modelsummary::modelplot()` permet d'afficher un graphique des coefficients.

Table 22.5: Présentation des facteurs associés à la pratique d'un sport avec modelsummary()

	(1)
(Intercept)	−0.798*
	(0.324)
sexeHomme	0.440***
	(0.106)
groupe_ages25-44 ans	−0.420+
	(0.228)
groupe_ages45-64 ans	−1.085***
	(0.238)
groupe_ages65 ans et plus	−1.381***
	(0.274)
etudesSecondaire	0.951***
	(0.197)
etudesTechnique / Professionnel	1.049***
	(0.190)
etudesSupérieur	1.892***
	(0.195)
etudesNon documenté	2.150***
	(0.330)
religPratiquant occasionnel	−0.022
	(0.189)
religAppartenance sans pratique	−0.007
	(0.175)
religNi croyance ni appartenance	−0.215
	(0.193)
religRejet	−0.384
	(0.286)
religNSP ou NVPR	−0.084
	(0.411)
heures.tv	−0.121***
	(0.034)
Num.Obs.	1995
AIC	2236.2
BIC	2320.1
Log.Lik.	−1103.086
F	21.691
RMSE	0.43
+ p < 0.1, * p < 0.05, ** p < 0.01, *** p < 0.001	

```
mod |> modelsummary::modelplot()
```

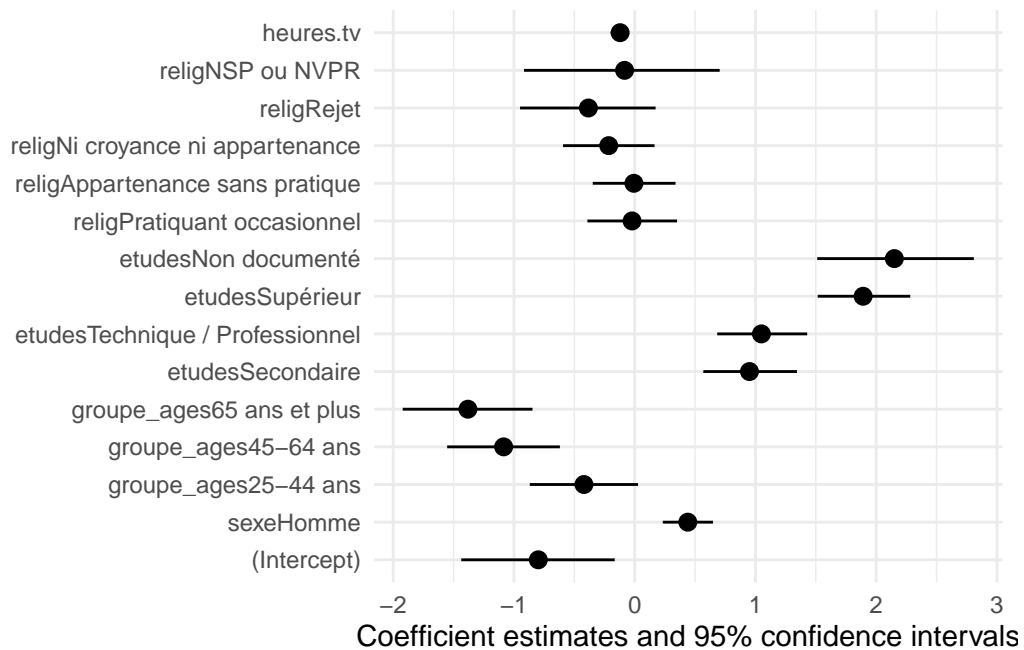


Figure 22.4: Facteurs associés à la pratique d'un sport avec modelplot()

ATTENTION : si l'on affiche les *odds ratio* avec `exponentiate = TRUE`, `modelsummary::modelplot()` conserve par défaut une échelle linéaire. On sera donc vigilant à appliquer `ggplot2::scale_x_log10()` manuellement pour utiliser une échelle logarithmique.

```
mod |>
  modelsummary::modelplot(exponentiate = TRUE) +
  ggplot2::scale_x_log10()
```

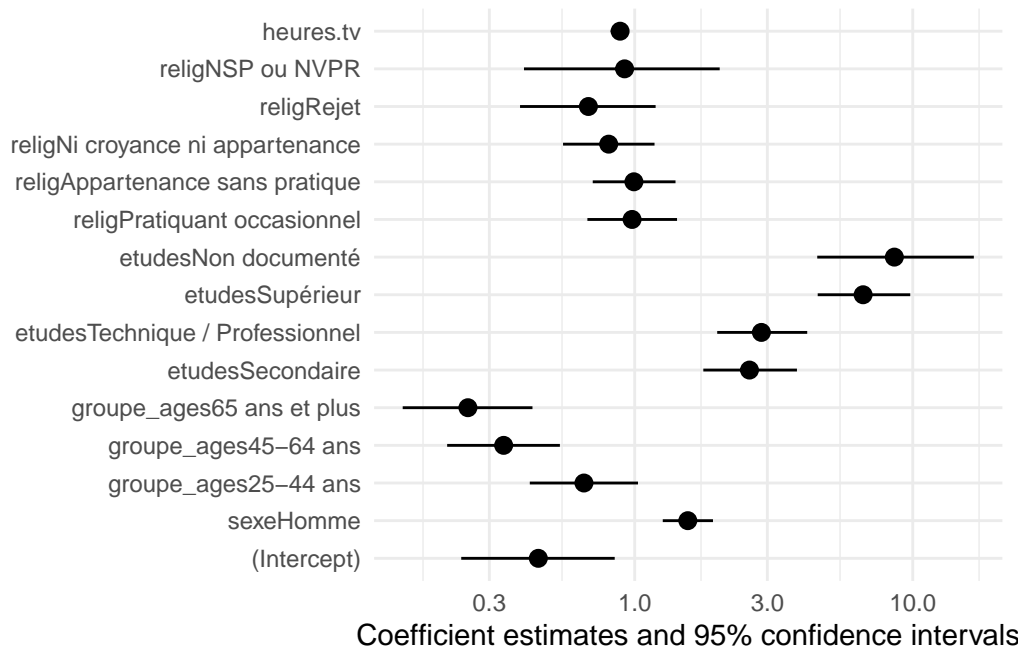



Figure 22.5: Odds Ratios associés à la pratique d'un sport avec modelplot()

22.7 Afficher toutes les comparaisons (*pairwise contrasts*)

Dans le tableau des résultats (Table 22.3), pour les variables catégorielles, il importe de bien garder en mémoire que chaque *odds ratio* doit se comparer à la valeur de référence. Ainsi, les *odds ratios* affichés pour chaque classe d'âges correspondent à une comparaison avec la classe d'âges de références, les 18-24 ans. La p-valeur associée nous indique quant à elle si cet *odds ratio* est significativement de 1, donc si cette classe d'âges données se comporte différemment de celle de référence.

Mais cela ne nous dit nullement si les 65 ans et plus diffèrent des 45-64 ans. Il est tout à fait possible de recalculer l'*odds ratio* correspondant en rapport les *odds ratio* à la référence : $OR_{65+/45-64} = OR_{65+/18-24} / OR_{45-64/18-24}$.

Le package `{emmeans}` et sa fonction `emmeans::emmeans()` permettent de recalculer toutes les combinaisons d'*odds ratio* (on parle alors de *pairwise contrasts*) ainsi que leur intervalle de confiance et la p-valeur correspondante.

On peut ajouter facilement⁶ cela au tableau produit avec `gtsummary::tbl_regression()` en ajoutant l'option `add_pairwise_contrasts = TRUE`.

⁶Cela nécessite néanmoins au minimum la version 1.11.0 du package `{broom.helpers}` et la version 1.6.3 de `{gtsummary}`.

```
mod |>
  tbl_regression(
    exponentiate = TRUE,
    add_pairwise_contrasts = TRUE
  ) |>
  bold_labels()
```

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

Table 22.6: Facteurs associés à la pratique d'un sport (*pairwise contrasts*)

Caractéristique	OR	95% IC	p-valeur
Sexe			
Homme / Femme	1,55	1,26 – 1,91	<0,001
Groupe d'âges			
(25-44 ans) / (18-24 ans)	0,66	0,37 – 1,18	0,3
(45-64 ans) / (18-24 ans)	0,34	0,18 – 0,62	<0,001
(45-64 ans) / (25-44 ans)	0,51	0,38 – 0,70	<0,001
65 ans et plus / (18-24 ans)	0,25	0,12 – 0,51	<0,001
65 ans et plus / (25-44 ans)	0,38	0,24 – 0,61	<0,001
65 ans et plus / (45-64 ans)	0,74	0,47 – 1,17	0,3
Niveau d'études			
Secondaire / Primaire	2,59	1,51 – 4,43	<0,001
(Technique / Professionnel) / Primaire	2,86	1,70 – 4,79	<0,001
(Technique / Professionnel) / Secondaire	1,10	0,74 – 1,64	>0,9
Supérieur / Primaire	6,63	3,89 – 11,3	<0,001
Supérieur / Secondaire	2,56	1,69 – 3,88	<0,001
Supérieur / (Technique / Professionnel)	2,32	1,61 – 3,36	<0,001
Non documenté / Primaire	8,59	3,49 – 21,1	<0,001
Non documenté / Secondaire	3,32	1,46 – 7,53	<0,001
Non documenté / (Technique / Professionnel)	3,01	1,38 – 6,56	0,001
Non documenté / Supérieur	1,30	0,58 – 2,90	>0,9
Rapport à la religion			
Pratiquant occasionnel / Pratiquant regulier	0,98	0,57 – 1,68	>0,9
Appartenance sans pratique / Pratiquant regulier	0,99	0,60 – 1,63	>0,9
Appartenance sans pratique / Pratiquant occasionnel	1,02	0,68 – 1,52	>0,9
Ni croyance ni appartenance / Pratiquant regulier	0,81	0,47 – 1,40	0,9

Table 22.6: Facteurs associés à la pratique d'un sport (*pairwise contrasts*)

Caractéristique	OR	95% IC	p-valeur
Ni croyance ni appartenance / Pratiquant occasionnel	0,82	0,52 – 1,31	0,8
Ni croyance ni appartenance / Appartenance sans pratique	0,81	0,54 – 1,21	0,7
Rejet / Pratiquant regulier	0,68	0,30 – 1,54	0,8
Rejet / Pratiquant occasionnel	0,70	0,33 – 1,49	0,8
Rejet / Appartenance sans pratique	0,69	0,33 – 1,41	0,7
Rejet / Ni croyance ni appartenance	0,85	0,40 – 1,79	>0,9
NSP ou NVPR / Pratiquant regulier	0,92	0,29 – 2,97	>0,9
NSP ou NVPR / Pratiquant occasionnel	0,94	0,30 – 2,92	>0,9
NSP ou NVPR / Appartenance sans pratique	0,93	0,30 – 2,82	>0,9
NSP ou NVPR / Ni croyance ni appartenance	1,14	0,37 – 3,55	>0,9
NSP ou NVPR / Rejet	1,35	0,37 – 4,88	>0,9
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

De même, on peut visualiser les coefficients avec la même option dans `ggstats::ggcoef_model()`⁷. On peut d'ailleurs choisir les variables concernées avec l'argument `pairwise_variables`.

```
mod |>
  ggstats::ggcoef_model(
    exponentiate = TRUE,
    add_pairwise_contrasts = TRUE,
    pairwise_variables = c("groupe_ages", "etudes")
  )
```

⁷Cela nécessite néanmoins au minimum la version 1.11.0 du package `{broom.helpers}` et la version 0.2.0 de `{ggstats}`.

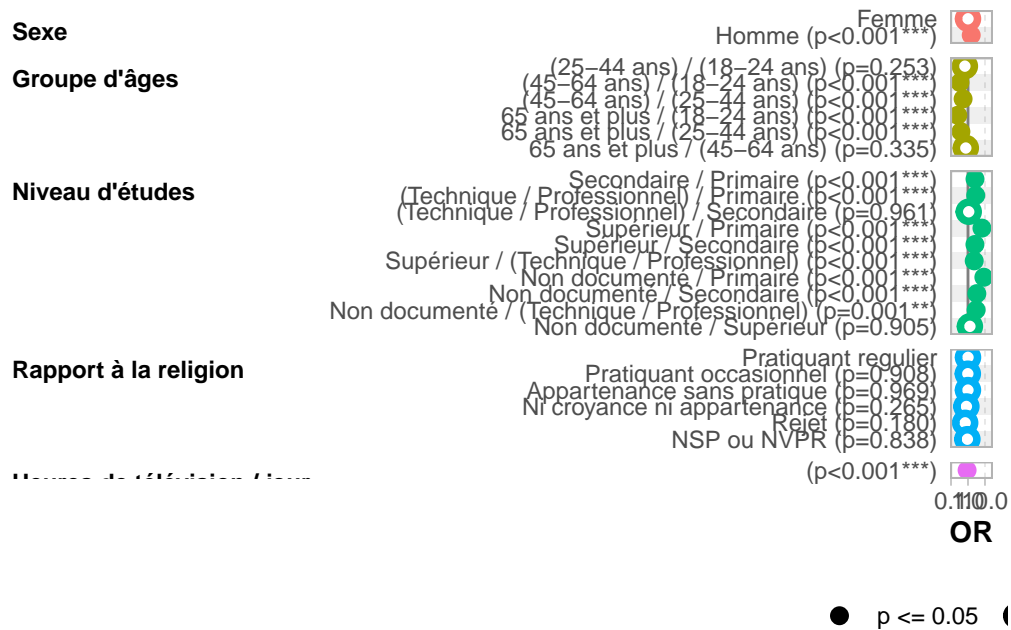


Figure 22.6: Facteurs associés à la pratique d'un sport (*pairwise contrasts*)

22.8 Identifier les variables ayant un effet significatif

Pour les variables catégorielles à trois modalités ou plus, les p-valeurs associées aux *odds ratios* nous indique si un *odd ratio* est significativement différent de 1, par rapport à la modalité de référence. Mais cela n'indique pas si globalement une variable a un effet significatif sur le modèle. Pour tester l'effet global d'une variable, on peut avoir recours à la fonction `car::Anova()`. Cette dernière va tour à tour supprimer chaque variable du modèle et réaliser une analyse de variance (ANOVA) pour voir si la variance change significativement.

```
car::Anova(mod)
```

Analysis of Deviance Table (Type II tests)

Response: sport

	LR	Chisq	Df	Pr(>Chisq)
sexe	17.309	1	3.176e-05	***
groupe_ages	52.803	3	2.020e-11	***
etudes	123.826	4	< 2.2e-16	***
relig	4.232	5	0.5165401	

```
heures.tv      13.438  1  0.0002465 ***
```

```
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Ainsi, dans le cas présent, la suppression de la variable *relig* ne modifie significativement pas le modèle, indiquant l'absence d'effet de cette variable.

Si l'on a recours à `gtsummary::tbl_regression()`, on peut facilement ajouter les p-valeurs globales avec `gtsummary::add_global_p()`⁸.

```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels() |>
  add_global_p()
```

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

Table 22.7: Ajout des p-valeurs globales

Caractéristique	OR	95% IC	p-valeur
Sexe			<0,001
Femme	—	—	
Homme	1,55	1,26 – 1,91	
Groupe d'âges			<0,001
18-24 ans	—	—	
25-44 ans	0,66	0,42 – 1,03	
45-64 ans	0,34	0,21 – 0,54	
65 ans et plus	0,25	0,15 – 0,43	
Niveau d'études			<0,001
Primaire	—	—	
Secondaire	2,59	1,77 – 3,83	
Technique / Professionnel	2,86	1,98 – 4,17	
Supérieur	6,63	4,55 – 9,80	
Non documenté	8,59	4,53 – 16,6	
Rapport à la religion			0,5
Pratiquant regulier	—	—	

⁸Si l'on veut conserver les p-valeurs individuelles associées à chaque *odds ratio*, on ajoutera l'option `keep = TRUE`.

Table 22.7: Ajout des p-valeurs globales

Caractéristique	OR	95% IC	p-valeur
Pratiquant occasionnel	0,98	0,68 – 1,42	
Appartenance sans pratique	0,99	0,71 – 1,40	
Ni croyance ni appartenance	0,81	0,55 – 1,18	
Rejet	0,68	0,39 – 1,19	
NSP ou NVPR	0,92	0,40 – 2,02	
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

i Note

Concernant le test réalisé dans le cadre d'une Anova, il existe trois tests différents que l'on présente comme le type 1, le type 2 et le type 3 (ou I, II et III). Pour une explication sur ces différents types, on pourra se référer (en anglais) à <https://mcfromnz.wordpress.com/2011/03/02/anova-type-iiii-ss-explained/> ou encore http://md.psych.bio.uni-goettingen.de/mv/unit/lm_cat/lm_cat_unbal_ss_explained.html. Le type I n'est pas recommandé dans le cas présent car il dépend de l'ordre dans lequel les différentes variables sont testées.

Lorsqu'il n'y a pas d'interaction dans un modèle, le type II serait à privilégier car plus puissant (nous aborderons les interactions dans un prochain chapitre, cf. Chapitre 26).

En présence d'interactions, il est conseillé d'avoir plutôt recours au type III. Cependant, en toute rigueur, pour utiliser le type III, il faut que les variables catégorielles soient codées en utilisant un contraste dont la somme est nulle (un contraste de type somme ou polynomial). Or, par défaut, les variables catégorielles sont codées avec un contraste de type traitement (nous aborderons les différents types de contrastes plus tard, cf. Chapitre 25).

Par défaut, `car::Anova()` utilise le type II et `gtsummary::add_global_p()` le type III. Dans les deux cas, il est possible de préciser le type de test avec `type = "II"` ou `type = "III"`.

Dans le cas de notre exemple, un modèle simple sans interaction, le type de test ne change pas les résultats.

22.9 Régressions logistiques univariées

Les usages varient selon les disciplines et les revues scientifiques, mais il n'est pas rare de présenter, avant le modèle logistique multivarié, une succession de modèles logistiques univariés (i.e. avec une seule variable explicative à la fois) afin de présenter les *odds ratios* et leur intervalle de confiance et p-valeur associés avant l'ajustement multiniveau.

Afin d'éviter le code fastidieux consistant à réaliser chaque modèle un par un (par exemple `glm(sport ~ sexe, family = binomial, data = d)`) puis à en fusionner les résultats, on pourra tirer partie de `gtsummary::tbl_uvregression()` qui permet de réaliser toutes ces régressions individuelles en une fois et de les présenter dans un tableau synthétique.

```
d |>
tbl_uvregression(
  y = sport,
  include = c(sexe, groupe_ages, etudes, relig, heures.tv),
  method = glm,
  method.args = list(family = binomial),
  exponentiate = TRUE
) |>
bold_labels()
```

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

Table 22.8: Régressions logistiques univariées

Caractéristique	N	OR	95% IC	p-valeur
Sexe	2 000			
Femme		—	—	
Homme		1,47	1,22 – 1,77	<0,001
Rapport à la religion	2 000			
Pratiquant regulier		—	—	
Pratiquant occasionnel		1,08	0,78 – 1,50	0,6
Appartenance sans pratique		1,31	0,98 – 1,78	0,071
Ni croyance ni appartenance		1,45	1,05 – 2,02	0,026
Rejet		1,19	0,72 – 1,95	0,5
NSP ou NVPR		0,93	0,44 – 1,88	0,8
Heures de télévision / jour	1 995	0,79	0,74 – 0,84	<0,001
Groupe d'âges	2 000			
18-24 ans		—	—	
25-44 ans		0,51	0,35 – 0,71	<0,001
45-64 ans		0,20	0,14 – 0,28	<0,001
65 ans et plus		0,10	0,07 – 0,15	<0,001
Niveau d'études	2 000			
Primaire		—	—	
Secondaire		3,61	2,52 – 5,23	<0,001

Table 22.8: Régressions logistiques univariées

Caractéristique	N	OR	95% IC	p-valeur
Technique / Professionnel		4,75	3,42 – 6,72	<0,001
Supérieur		11,4	8,11 – 16,3	<0,001
Non documenté		26,2	15,7 – 44,9	<0,001

22.10 Présenter l'ensemble des résultats dans un même tableau

La fonction `gtsummary::tbl_merge()` permet de fusionner plusieurs tableaux (en tenant compte du nom des variables) et donc de présenter les différents résultats de l'analyse descriptive, univariée et multivariée dans un seul et même tableau.

```
tbl_desc <-
  d |>
  tbl_summary(
    by = sport,
    include = c(sexe, groupe_ages, etudes, relig, heures.tv),
    statistic = all_categorical() ~ "{p}% ({n}/{N})",
    percent = "row",
    digits = all_categorical() ~ c(1, 0, 0)
  ) |>
  modify_column_hide("stat_1") |>
  modify_header("stat_2" ~ "**Pratique d'un sport**")

tbl_uni <-
  d |>
  tbl_uvregression(
    y = sport,
    include = c(sexe, groupe_ages, etudes, relig, heures.tv),
    method = glm,
    method.args = list(family = binomial),
    exponentiate = TRUE
  ) |>
  modify_column_hide("stat_n")

tbl_multi <-
  mod |>
  tbl_regression(exponentiate = TRUE)
```



```
list(tbl_desc, tbl_uni, tbl_multi) |>
  tbl_merge(
    tab_spanner = c(
      NA,
      "**Régressions univariées**",
      "**Régression multivariée**"
    )
  ) |>
  bold_labels()
```

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

Table 22.9: tableau synthétique de l'analyse

Caractéristique	Pratique d'un sport	OR	95% IC	p- valeur	OR	95% IC	p- valeur
Sexe							
Femme	32,2% (354/101)	—	—		—	—	
Homme	41,0% (369/899)	1,47	1,22 – 1,77	<0,001	1,55	1,26 – 1,91	<0,001
Groupe d'âges							
18-24 ans	65,7% (111/169)	—	—		—	—	
25-44 ans	49,2% (347/706)	0,51	0,35 – 0,71	<0,001	0,66	0,42 – 1,03	0,065
45-64 ans	27,4% (204/745)	0,20	0,14 – 0,28	<0,001	0,34	0,21 – 0,54	<0,001
65 ans et plus	16,1% (61/380)	0,10	0,07 – 0,15	<0,001	0,25	0,15 – 0,43	<0,001
Niveau d'études							
Primaire	10,7% (50/466)	—	—		—	—	
Secondaire	30,2% (117/387)	3,61	2,52 – 5,23	<0,001	2,59	1,77 – 3,83	<0,001
Technique / Professionnel	36,4% (216/594)	4,75	3,42 – 6,72	<0,001	2,86	1,98 – 4,17	<0,001
Supérieur	57,8% (255/441)	11,4	8,11 – 16,3	<0,001	6,63	4,55 – 9,80	<0,001

Table 22.9: tableau synthétique de l'analyse

Caractéristique	Pratique d'un sport	OR	95% IC	p- valeur	OR	95% IC	p- valeur
Non documenté	75,9% (85/112)	26,2	15,7 – 44,9	<0,001	8,59	4,53 – 16,6	<0,001
Rapport à la religion							
Pratiquant regulier	31,6% (84/266)	—	—		—	—	
Pratiquant occasionnel	33,3% (147/442)	1,08	0,78 – 1,50	0,6	0,98	0,68 – 1,42	>0,9
Appartenance sans pratique	37,8% (287/760)	1,31	0,98 – 1,78	0,071	0,99	0,71 – 1,40	>0,9
Ni croyance ni appartenance	40,1% (160/399)	1,45	1,05 – 2,02	0,026	0,81	0,55 – 1,18	0,3
Rejet	35,5% (33/93)	1,19	0,72 – 1,95	0,5	0,68	0,39 – 1,19	0,2
NSP ou NVPR	30,0% (12/40)	0,93	0,44 – 1,88	0,8	0,92	0,40 – 2,02	0,8
Heures de télévision / jour	2,00 (1,00 – 3,00)	0,79	0,74 – 0,84	<0,001	0,89	0,83 – 0,95	<0,001
Manquant	3						

Pour visualiser chaque étape du code, vous pouvez consulter le diaporama suivant : <https://larmarange.github.io/guide-R/analyses/ressources/flipbook-regression-logistique.html>

22.11 webin-R

La régression logistique est présentée sur YouTube dans le [webin-R #06](#) (*régression logistique (partie 1)*) et le [webin-R #07](#) (*régression logistique (partie 2)*).

<https://youtu.be/-bdMv2aAqUY>

<https://youtu.be/BUo9i7XTLYQ>

23 Sélection pas à pas d'un modèle

Il est toujours tentant lorsque l'on recherche les facteurs associés à un phénomène d'inclure un nombre important de variables explicatives potentielles dans son modèle de régression. Cependant, un tel modèle n'est pas forcément le plus efficace et certaines variables n'auront probablement pas d'effet significatif sur la variable d'intérêt.

Un autre problème potentiel est celui du [sur-ajustement ou surapprentissage](#). Un modèle sur-ajusté est un modèle statistique qui contient plus de paramètres que ne peuvent le justifier les données. Dès lors, il va être trop ajusté aux données observées mais perdre en capacité de généralisation.

Pour une présentation didactique du cadre théorique de la sélection de modèle, vous pouvez consulter en ligne le [cours de L. Rouvière sur la sélection/validation de modèles](#).

Les techniques de sélection pas à pas sont des approches visant à améliorer un modèle explicatif. On part d'un modèle initial puis on regarde s'il est possible d'améliorer le modèle en ajoutant ou en supprimant une des variables du modèle pour obtenir un nouveau modèle. Le processus est répété jusqu'à obtenir un modèle final que l'on ne peut plus améliorer.

23.1 Données d'illustration

Pour illustrer ce chapitre, nous allons prendre un modèle logistique inspiré de celui utilisé dans le chapitre sur la régression logistique binaire (cf. Chapitre 22).

```
library(tidyverse)
library(labelled)
library(gtsummary)
theme_gtsummary_language(
  "fr",
  decimal.mark = ",",
  big.mark = " "
)

data(hdv2003, package = "questionr")

d <-
```

```

hdv2003 |>
mutate(
  sexe = sexe |> fct_relevel("Femme"),
  groupe_ages = age |>
    cut(
      c(18, 25, 45, 65, 99),
      right = FALSE,
      include.lowest = TRUE,
      labels = c("18-24 ans", "25-44 ans",
                 "45-64 ans", "65 ans et plus")
    ),
  etudes = nivetud |>
    fct_recode(
      "Primaire" = "N'a jamais fait d'etudes",
      "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
      "Primaire" = "Derniere annee d'etudes primaires",
      "Secondaire" = "1er cycle",
      "Secondaire" = "2eme cycle",
      "Technique / Professionnel" = "Enseignement technique ou professionnel court",
      "Technique / Professionnel" = "Enseignement technique ou professionnel long",
      "Supérieur" = "Enseignement superieur y compris technique superieur"
    ) |>
    fct_na_value_to_level("Non documenté")
) |>
set_variable_labels(
  sport = "Pratique un sport ?",
  sexe = "Sexe",
  groupe_ages = "Groupe d'âges",
  etudes = "Niveau d'études",
  relig = "Rapport à la religion",
  lecture.bd = "Lit des bandes dessinées ?"
)

mod <- glm(
  sport ~ sexe + groupe_ages + etudes + relig + lecture.bd,
  family = binomial,
  data = d
)

```

23.2 Présentation de l'AIC

Il faut définir un critère pour déterminer la qualité d'un modèle. L'un des plus utilisés est le *Akaike Information Criterion* ou AIC. Il s'agit d'un compromis entre le nombre de degrés de liberté (e.g. le nombre de coefficients dans le modèle) que l'on cherche à minimiser et la variance expliquée que l'on cherche à maximiser (la vraisemblance).

Plus précisément $AIC = 2k - 2\ln(L)$ où L est le maximum de la fonction de vraisemblance du modèle et k le nombre de paramètres (i.e. de coefficients) du modèle. Plus l'AIC sera faible, meilleur sera le modèle.

L'AIC d'un modèle s'obtient aisément avec `AIC()`.

```
AIC(mod)
```

```
[1] 2257.101
```

23.3 Sélection pas à pas descendante

La fonction `step()` permet de sélectionner le meilleur modèle par une procédure pas à pas descendante basée sur la minimisation de l'AIC. La fonction affiche à l'écran les différentes étapes de la sélection et renvoie le modèle final.

```
mod2 <- step(mod)
```

```
Start:  AIC=2257.1
```

```
sport ~ sexe + groupe_ages + etudes + relig + lecture.bd
```

	Df	Deviance	AIC
- relig	5	2231.9	2251.9
- lecture.bd	1	2227.9	2255.9
<none>		2227.1	2257.1
- sexe	1	2245.6	2273.6
- groupe_ages	3	2280.1	2304.1
- etudes	4	2375.5	2397.5

```
Step:  AIC=2251.95
```

```
sport ~ sexe + groupe_ages + etudes + lecture.bd
```

	Df	Deviance	AIC
- lecture.bd	1	2232.6	2250.6

<none>		2231.9	2251.9
- sexe	1	2248.8	2266.8
- groupe_ages	3	2282.1	2296.1
- etudes	4	2380.5	2392.5

Step: AIC=2250.56

sport ~ sexe + groupe_ages + etudes

	Df	Deviance	AIC
<none>		2232.6	2250.6
- sexe	1	2249.2	2265.2
- groupe_ages	3	2282.5	2294.5
- etudes	4	2385.2	2395.2

Le modèle initial a un AIC de 2257,1.

À la première étape, il apparaît que la suppression de la variable *relig* permettrait diminuer l'AIC à 2251,9 et la suppression de la variable *lecture.bd* de le diminuer à 2255,9. Le gain maximal est obtenu en supprimant *relig* et donc cette variable est supprimée à ce stade. On peut noter que la suppression de la variable entraîne *de facto* une augmentation des résidus (colonne *Deviance*) et donc une baisse de la vraisemblance du modèle, mais cela est compensé par la réduction du nombre de degrés de liberté.

Le processus est maintenant répété. À la seconde étape, supprimer *lecture.bd* permettrait de diminuer encore l'AIC à 2250,6 et cette variable est supprimée.

À la troisième étape, tout retrait d'une variable additionnelle reviendrait à augmenter l'AIC.

Lors de la seconde étape, toute suppression d'une autre variable ferait augmenter l'AIC. La procédure s'arrête donc.

L'objet `mod2` renvoyé par `step()` est le modèle final.

`mod2`

```
Call: glm(formula = sport ~ sexe + groupe_ages + etudes, family = binomial,
  data = d)
```

Coefficients:

(Intercept)	sexeHomme
-1.2815	0.4234
groupe_ages25-44 ans	groupe_ages45-64 ans
-0.3012	-0.9261

groupe_ages65 ans et plus	etudesSecondaire
-1.2696	0.9670
etudesTechnique / Professionnel	etudesSupérieur
1.0678	1.9955
etudesNon documenté	
2.3192	

Degrees of Freedom: 1999 Total (i.e. Null); 1991 Residual
 Null Deviance: 2617
 Residual Deviance: 2233 AIC: 2251

On peut effectuer une analyse de variance ou ANOVA pour comparer les deux modèles avec la fonction `anova()`.

```
anova(mod, mod2, test = "Chisq")
```

Analysis of Deviance Table

Model 1: `sport ~ sexe + groupe_ages + etudes + relig + lecture.bd`

Model 2: `sport ~ sexe + groupe_ages + etudes`

	Resid. Df	Resid. Dev	Df	Deviance	Pr(>Chi)
1	1985	2227.1			
2	1991	2232.6	-6	-5.4597	0.4863

Il n'y a pas de différences significatives entre nos deux modèles ($p=0,55$). Autrement dit, notre second modèle explique tout autant de variance que notre premier modèle, tout en étant plus parcimonieux.

Astuce

Une alternative à la fonction `step()` est la fonction `MASS::stepAIC()` du package `{MASS}` qui fonctionne de la même manière. Si cela ne change rien aux régressions logistiques classiques, il arrive que pour certains types de modèle la méthode `step()` ne soit pas disponible, mais que `MASS::stepAIC()` puisse être utilisée à la place.

```
library(MASS)
```

Attachement du package : 'MASS'

L'objet suivant est masqué depuis 'package:gtsummary':

```

select

L'objet suivant est masqué depuis 'package:dplyr':

select

mod2bis <- stepAIC(mod)

Start:  AIC=2257.1
sport ~ sexe + groupe_ages + etudes + relig + lecture.bd

      Df Deviance    AIC
- relig      5  2231.9 2251.9
- lecture.bd  1  2227.9 2255.9
<none>                2227.1 2257.1
- sexe       1  2245.6 2273.6
- groupe_ages 3  2280.1 2304.1
- etudes      4  2375.5 2397.5

Step:  AIC=2251.95
sport ~ sexe + groupe_ages + etudes + lecture.bd

      Df Deviance    AIC
- lecture.bd  1  2232.6 2250.6
<none>                2231.9 2251.9
- sexe       1  2248.8 2266.8
- groupe_ages 3  2282.1 2296.1
- etudes      4  2380.5 2392.5

Step:  AIC=2250.56
sport ~ sexe + groupe_ages + etudes

      Df Deviance    AIC
<none>                2232.6 2250.6
- sexe       1  2249.2 2265.2
- groupe_ages 3  2282.5 2294.5
- etudes      4  2385.2 2395.2

```

On peut facilement comparer visuellement deux modèles avec `ggstats::ggcoef_compare()` de `{ggstats}`.


```
library(ggstats)
ggcoef_compare(
  list("modèle complet" = mod, "modèle réduit" = mod2),
  exponentiate = TRUE
)
```

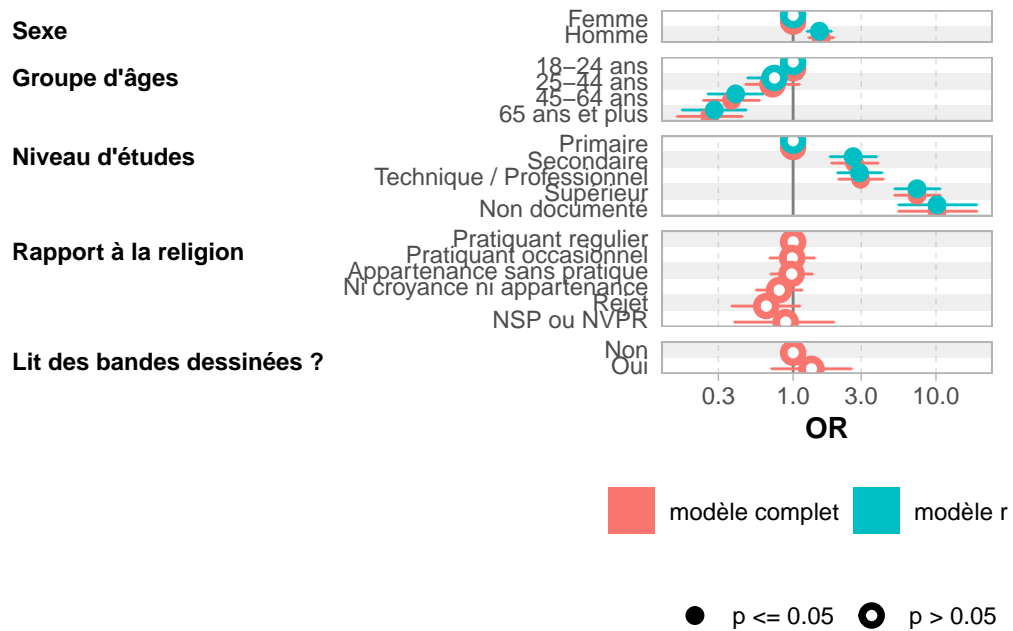


Figure 23.1: Comparaison visuelle des deux modèles (*dodge*)

```
ggcoef_compare(
  list("modèle complet" = mod, "modèle réduit" = mod2),
  type = "faceted",
  exponentiate = TRUE
)
```

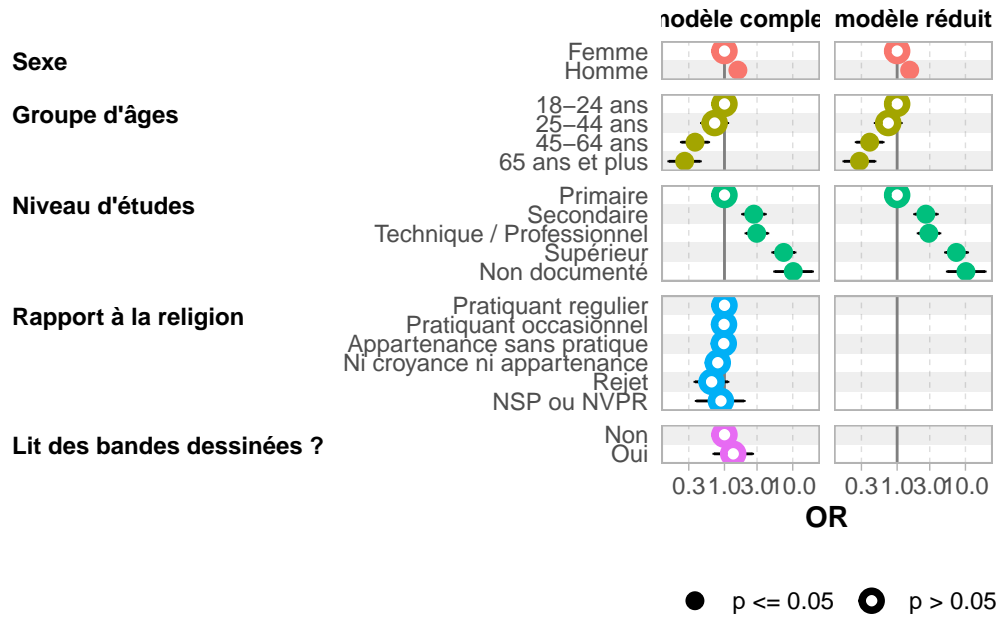


Figure 23.2: Comparaison visuelle des deux modèles (*faceted*)

23.4 Sélection pas à pas ascendante

Pour une approche ascendante, nous allons partir d'un modèle vide, c'est-à-dire d'un modèle sans variable explicative avec simplement un intercept.

```
mod_vide <- glm(
  sport ~ 1,
  family = binomial,
  data = d
)
```

Nous allons ensuite passer ce modèle vide à `step()` et préciser, via un élément nommé *upper* dans une liste passée à l'argument `scope`, la formule du modèle maximum à considérer. Nous précisons `direction = "forward"` pour indiquer que nous souhaitons une procédure ascendante.

```
mod3 <- step(
  mod_vide,
  direction = "forward",
```

```

scope = list(
  upper = ~ sexe + groupe_ages + etudes + relig + lecture.bd
)
)

```

Start: AIC=2619.11

sport ~ 1

	Df	Deviance	AIC
+ etudes	4	2294.9	2304.9
+ groupe_ages	3	2405.4	2413.4
+ sexe	1	2600.2	2604.2
+ lecture.bd	1	2612.7	2616.7
<none>		2617.1	2619.1
+ relig	5	2608.8	2620.8

Step: AIC=2304.92

sport ~ etudes

	Df	Deviance	AIC
+ groupe_ages	3	2249.2	2265.2
+ sexe	1	2282.5	2294.5
<none>		2294.9	2304.9
+ lecture.bd	1	2294.7	2306.7
+ relig	5	2293.0	2313.0

Step: AIC=2265.17

sport ~ etudes + groupe_ages

	Df	Deviance	AIC
+ sexe	1	2232.6	2250.6
<none>		2249.2	2265.2
+ lecture.bd	1	2248.8	2266.8
+ relig	5	2246.0	2272.0

Step: AIC=2250.56

sport ~ etudes + groupe_ages + sexe

	Df	Deviance	AIC
<none>		2232.6	2250.6
+ lecture.bd	1	2231.9	2251.9
+ relig	5	2227.9	2255.9

Cette fois-ci, à chaque étape, la fonction `step()` évalue le gain à ajouter chaque variable dans le modèle, ajoute la variable la plus pertinente, pour recommence le processus jusqu'à ce qu'il n'y ait plus de gain à ajouter une variable au modèle. Notons que nous aboutissons ici au même résultat.

Astuce

Nous aurions pu nous passer de préciser `direction = "forward"`. Dans cette situation, `step()` regarde simultanément les gains à ajouter une variable additionnelle au modèle et à supprimer une variable déjà incluse pour . Lorsque l'on part d'un modèle vide, cela ne change rien au résultat.

```
mod3 <- step(
  mod_vide,
  scope = list(
    upper = ~ sexe + groupe_ages + etudes + relig + lecture.bd
  )
)
```

```
Start:  AIC=2619.11
sport ~ 1
```

	Df	Deviance	AIC
+ etudes	4	2294.9	2304.9
+ groupe_ages	3	2405.4	2413.4
+ sexe	1	2600.2	2604.2
+ lecture.bd	1	2612.7	2616.7
<none>		2617.1	2619.1
+ relig	5	2608.8	2620.8

```
Step:  AIC=2304.92
sport ~ etudes
```

	Df	Deviance	AIC
+ groupe_ages	3	2249.2	2265.2
+ sexe	1	2282.5	2294.5
<none>		2294.9	2304.9
+ lecture.bd	1	2294.7	2306.7
+ relig	5	2293.0	2313.0
- etudes	4	2617.1	2619.1

```
Step:  AIC=2265.17
```

```
sport ~ etudes + groupe_ages
```

	Df	Deviance	AIC
+ sexe	1	2232.6	2250.6
<none>		2249.2	2265.2
+ lecture.bd	1	2248.8	2266.8
+ relig	5	2246.0	2272.0
- groupe_ages	3	2294.9	2304.9
- etudes	4	2405.4	2413.4

Step: AIC=2250.56

```
sport ~ etudes + groupe_ages + sexe
```

	Df	Deviance	AIC
<none>		2232.6	2250.6
+ lecture.bd	1	2231.9	2251.9
+ relig	5	2227.9	2255.9
- sexe	1	2249.2	2265.2
- groupe_ages	3	2282.5	2294.5
- etudes	4	2385.2	2395.2

23.5 Forcer certaines variables dans le modèle réduit

Même si l'on a recourt à `step()`, on peut vouloir forcer la présence de certaines variables dans le modèle, même si leur suppression minimiserait l'AIC. Par exemple, si l'on a des hypothèses spécifiques pour ces variables et que l'on a intérêt à montrer qu'elles n'ont pas d'effet dans le modèle multivarié.

Supposons que nous avons une hypothèse sur le lien entre la pratique d'un sport et la lecture de bandes dessinées. Nous souhaitons donc forcer la présence de la variable *lecture.bd* dans le modèle final. Cette fois-ci, nous allons indiquer, via la liste passée à `scope`, un élément *lower* indiquant le modèle minimum souhaité. Toutes les variables de ce modèle minimum seront donc conserver dans le modèle final.

```
mod4 <- step(
  mod,
  scope = list(
    lower = ~ lecture.bd
  )
)
```

```
Start:  AIC=2257.1
sport ~ sexe + groupe_ages + etudes + relig + lecture.bd
```

	Df	Deviance	AIC
- relig	5	2231.9	2251.9
<none>		2227.1	2257.1
- sexe	1	2245.6	2273.6
- groupe_ages	3	2280.1	2304.1
- etudes	4	2375.5	2397.5

```
Step:  AIC=2251.95
sport ~ sexe + groupe_ages + etudes + lecture.bd
```

	Df	Deviance	AIC
<none>		2231.9	2251.9
- sexe	1	2248.8	2266.8
- groupe_ages	3	2282.1	2296.1
- etudes	4	2380.5	2392.5

Cette fois-ci, nous constatons que la fonction `step()` n'a pas considéré la suppression éventuelle de la variable *lecture.bd* qui est donc conservée.

```
mod4$formula
```

```
sport ~ sexe + groupe_ages + etudes + lecture.bd
```

23.6 Minimisation du BIC

Un critère similaire à l'AIC est le critère BIC (*Bayesian Information Criterion*) appelé aussi SBC (*Schwarz information criterion*).

Sa formule est proche de celle de l'AIC : $BIC = \ln(n)k - 2\ln(L)$ où n correspond au nombre d'observations dans l'échantillon. Par rapport à l'AIC, il pénalise donc plus le nombre de degrés de liberté du modèle.

Pour réaliser une sélection pas à pas par optimisation du BIC, on appellera `step()` en ajoutant l'argument `k = log(n)` où n est le nombre d'observations incluses dans le modèle. Par défaut, un modèle est calculé en retirant les observations pour lesquelles des données sont manquantes. Dès lors, pour obtenir le nombre exact d'observations incluses dans le modèle, on peut utiliser la syntaxe `mod |> model.matrix() |> nrow()`, `model.matrix()` renvoyant la matrice de données ayant servi au calcul du modèle et `nrow()` le nombre de lignes.

```
mod5 <- mod |>
  step(
    k = mod |> model.matrix() |> nrow() |> log()
  )
```

Start: AIC=2341.11

sport ~ sexe + groupe_ages + etudes + relig + lecture.bd

	Df	Deviance	AIC
- relig	5	2231.9	2308.0
- lecture.bd	1	2227.9	2334.3
<none>		2227.1	2341.1
- sexe	1	2245.6	2352.0
- groupe_ages	3	2280.1	2371.3
- etudes	4	2375.5	2459.1

Step: AIC=2307.96

sport ~ sexe + groupe_ages + etudes + lecture.bd

	Df	Deviance	AIC
- lecture.bd	1	2232.6	2301.0
<none>		2231.9	2308.0
- sexe	1	2248.8	2317.2
- groupe_ages	3	2282.1	2335.3
- etudes	4	2380.5	2426.1

Step: AIC=2300.97

sport ~ sexe + groupe_ages + etudes

	Df	Deviance	AIC
<none>		2232.6	2301.0
- sexe	1	2249.2	2310.0
- groupe_ages	3	2282.5	2328.1
- etudes	4	2385.2	2423.2

23.7 Afficher les indicateurs de performance

Il existe plusieurs indicateurs de performance ou qualité d'un modèle. Pour les calculer/afficher (dont l'AIC et le BIC), on pourra avoir recours à `broom::glance()` ou encore à `performance::model_performance()`.

```
mod |> broom::glance()
```

```
# A tibble: 1 x 8
  null.deviance df.null logLik   AIC   BIC deviance df.residual  nobs
      <dbl>    <int>  <dbl> <dbl> <dbl>   <dbl>      <int> <int>
1      2617.    1999 -1114. 2257. 2341.   2227.      1985  2000
```

```
mod |> performance::model_performance()
```

```
# Indices of model performance
```

AIC	AICc	BIC	Tjur's R2	RMSE	Sigma	Log_loss	Score_log	Score_spl
2257.101	2257.343	2341.115	0.183	0.434	1.000	0.557	-Inf	

Le fonction `performance::compare_performance()` permet de comparer rapidement plusieurs modèles.

```
performance::compare_performance(mod, mod2, mod4)
```

```
# Comparison of Model Performance Indices
```

Name	Model	AIC (weights)	AICc (weights)	BIC (weights)	Tjur's R2	RMSE	Sigma
mod	glm	2257.1 (0.025)	2257.3 (0.023)	2341.1 (<.001)	0.183	0.434	1.000
mod2	glm	2250.6 (0.651)	2250.7 (0.654)	2301.0 (0.971)	0.181	0.435	1.000
mod4	glm	2252.0 (0.325)	2252.1 (0.323)	2308.0 (0.029)	0.181	0.435	1.000

Si l'on souhaite afficher l'AIC (ainsi que d'autres statistiques globales du modèle) en note du tableau des coefficients, on pourra utiliser `gtsummary::add_glance_source_note()`.

```
mod2 |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels() |>
  add_glance_source_note()
```

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

Table 23.1: Modèle obtenu après réduction du nombre de variables

Caractéristique	OR	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	1,53	1,25 – 1,87	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	0,74	0,48 – 1,15	0,2
45-64 ans	0,40	0,25 – 0,62	<0,001
65 ans et plus	0,28	0,17 – 0,47	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	2,63	1,80 – 3,88	<0,001
Technique / Professionnel	2,91	2,03 – 4,22	<0,001
Supérieur	7,36	5,10 – 10,8	<0,001
Non documenté	10,2	5,43 – 19,4	<0,001

23.8 Sélection pas à pas et valeurs manquantes

Si certaines de nos variables explicatives contiennent des valeurs manquantes (NA), cela peut entraîner des erreurs au moment d'avoir recours à `step()`, car le nombre d'observations dans le modèle va changer si on retire du modèle une variable explicative avec des valeurs manquantes.

Prenons un exemple, en ajoutant des valeurs manquantes à la variable *relig* (pour cela nous allons recoder les refus et les ne sait pas en NA).

```
d$relig_na <-
  d$relig |>
  fct_recode(
    NULL = "Rejet",
    NULL = "NSP ou NVPR"
  )

mod_na <- glm(
  sport ~ sexe + groupe_ages + etudes + relig_na + lecture.bd,
  family = binomial,
  data = d
)
```

Au moment d'exécuter `step()` nous obtenons l'erreur mentionnée précédemment.

```
step(mod_na)
```

```
Start:  AIC=2096.64
```

```
sport ~ sexe + groupe_ages + etudes + relig_na + lecture.bd
```

	Df	Deviance	AIC
- relig_na	3	2073.2	2093.2
- lecture.bd	1	2072.2	2096.2
<none>		2070.6	2096.6
- sexe	1	2088.6	2112.6
- groupe_ages	3	2118.0	2138.0
- etudes	4	2218.1	2236.1

```
Error in step(mod_na): le nombre de lignes utilisées a changé : supprimer les valeurs manquantes
```

Pas d'inquiétude ! Il y a moyen de s'en sortir en adoptant la stratégie suivante :

1. créer une copie du jeu de données avec uniquement des observations sans valeur manquante pour nos variables explicatives ;
2. calculer notre modèle complet à partir de ce jeu de données ;
3. appliquer `step()` ;
4. recalculer le modèle réduit en repartant du jeu de données complet.

Première étape, ne garder que les observations complètes à l'aide de `tidyr::drop_na()`, en lui indiquant la liste des variables dans lesquelles vérifier la présence ou non de NA.

```
d_complet <- d |>
  drop_na(sexe, groupe_ages, etudes, relig_na, lecture.bd)
```

Deuxième étape, calculons le modèle complet avec ce jeu données.

```
mod_na_alt <- glm(
  sport ~ sexe + groupe_ages + etudes + relig_na + lecture.bd,
  family = binomial,
  data = d_complet
)
```

Le modèle `mod_na_alt` est tout à fait identique au modèle `mod_na`, car `glm()` supprime de lui-même les valeurs manquantes quand elles existent. Nous pouvons maintenant utiliser `step()`.

```
mod_na_reduit <- step(mod_na_alt)
```

Start: AIC=2096.64

```
sport ~ sexe + groupe_ages + etudes + relig_na + lecture.bd
```

	Df	Deviance	AIC
- relig_na	3	2073.2	2093.2
- lecture.bd	1	2072.2	2096.2
<none>		2070.6	2096.6
- sexe	1	2088.6	2112.6
- groupe_ages	3	2118.0	2138.0
- etudes	4	2218.1	2236.1

Step: AIC=2093.19

```
sport ~ sexe + groupe_ages + etudes + lecture.bd
```

	Df	Deviance	AIC
- lecture.bd	1	2074.6	2092.6
<none>		2073.2	2093.2
- sexe	1	2090.2	2108.2
- groupe_ages	3	2118.5	2132.5
- etudes	4	2221.4	2233.4

Step: AIC=2092.59

```
sport ~ sexe + groupe_ages + etudes
```

	Df	Deviance	AIC
<none>		2074.6	2092.6
- sexe	1	2091.1	2107.1
- groupe_ages	3	2119.6	2131.6
- etudes	4	2227.2	2237.2

Cela s'exécute sans problème car tous les sous-modèles sont calculés à partir de `d_complet` et donc ont bien le même nombre d'observations. Cependant, dans notre modèle réduit, on a retiré 137 observations en raison d'une valeur manquante sur la variable `relig_na`, variable qui n'est plus présente dans notre modèle réduit. Il serait donc pertinent de réintégrer ces observations.

Nous allons donc recalculer le modèle réduit mais à partir de `d`. Inutile de recopier à la main la formule du modèle réduit, car nous pouvons l'obtenir directement avec `mod_na_reduit$formula`.

```
mod_na_reduit2 <- glm(
  mod_na_reduit$formula,
  family = binomial,
  data = d
)
```

Attention : `mod_na_reduit` et `mod_na_reduit2` ne sont pas identiques puisque le second a été calculé sur un plus grand nombre d'observations, ce qui change très légèrement les valeurs des coefficients.

💡 Astuce

Pour automatiser l'ensemble de ce processus, on peut copier/coller le code de la fonction générique suivante :

```
step_with_na <- function(model, ...) {
  # refit the model without NAs
  model_no_na <- update(model, data = model.frame(model))
  # apply step()
  model_simplified <- step(model_no_na, ...)
  # recompute simplified model using full data
  update(model, formula = terms(model_simplified))
}
```

Elle réalise l'ensemble des opérations décrites plus haut en profitant de la flexibilité offerte par la fonction `update()`. La fonction `model.frame()` permet de récupérer le jeu de données utilisé par un modèle (et dans lequel les lignes incomplètes ont été supprimées). La fonction `terms()` permet de récupérer l'équation du modèle.

Attention : il s'agit d'une fonction expérimentale et elle n'est peut-être pas compatible avec tous les types de modèles. Elle a été testée avec les modèles `lm()`, `glm()` et `nnet::multinom()`.

```
mod_na_reduit_direct <- step_with_na(mod_na, trace = 0)
```

Le résultat obtenu est strictement identique.

```
anova(mod_na_reduit2, mod_na_reduit_direct)
```

Analysis of Deviance Table

Model 1: sport ~ sexe + groupe_ages + etudes

Model 2: sport ~ sexe + groupe_ages + etudes

	Resid. Df	Resid. Dev	Df	Deviance
1	1991	2232.6		
2	1991	2232.6	0	0

24 Prédictions marginales, contrastes marginaux & effets marginaux

Avertissement

Ce chapitre nécessite une version récente de `{broom.helpers}` (version 1.12.0), de `{gtsummary}` (version 1.6.3), de `{ggstats}` (version 0.2.1) et de `{marginaleffects}` (version 0.10.0).

Les coefficients d'une régression multivariée ne sont pas toujours facile à interpréter car ils ne sont pas forcément exprimés dans la même dimension que la variable d'intérêt. C'est notamment le cas pour une régression logistique binaire (cf. Chapitre 22). Comment traduire la valeur d'un *odds ratio* en écart de probabilité ?

Dans certaines disciplines, notamment en économétrie, on préfère souvent présenter les effets marginaux qui tentent de traduire les résultats du modèle dans la dimension de la variable d'intérêt. Plusieurs approches existent et l'on trouve dans la littérature des expressions telles que effets marginaux, effets statistiques, moyennes marginales, pentes marginales, effets marginaux à la moyenne, et autres expressions similaires.

Différents auteurs peuvent utiliser la même expression pour désigner des indicateurs différents, ou bien des manières différentes de les calculer.

Note

Si vous n'êtes pas familier des estimations marginales et souhaitez aller à l'essentiel, vous pouvez, en première lecture, vous concentrer sur les prédictions marginales moyennes et les contrastes marginaux moyens, avant d'explorer les autres variantes.

24.1 Terminologie

Dans ce guide, nous avons décidé d'adopter une terminologie consistante avec celle du package `{broom.helpers}`, elle-même basée sur celle du package `{marginaleffects}`, dont la première version a été publiée en septembre 2021, et avec le [billet d'Andrew Heiss intitulé *Marginalia*](#) et publié en mai 2022.

Lorsque l'on utilise un modèle ajusté pour prédire l'*outcome* selon certaines combinaisons de valeurs des régresseurs / variables explicatives, par exemple leurs valeurs observées ou leur moyenne, on obtient des **prédictions ajustées**. Lorsque ces dernières sont moyennées selon un régresseur spécifique, nous parlerons alors de **prédictions marginales**.

Les **contrastes marginaux** correspondent au calcul d'une différence entre des prédictions marginales, que ce soit pour une variable catégorielle (e.g. différence entre deux modalités) ou pour une variable continue (différence observée au niveau de l'*outcome* pour un certain changement du prédicteur).

Les **pentés marginales** ou **effets marginaux** sont définis, pour des variables continues, comme la dérivée partielle (*slope*) de l'équation de régression pour certaines valeurs de la variable explicative. Dit autrement, un effet marginal correspond à la pente locale de la fonction de régression pour certaines valeurs choisies d'un prédicteur continue. De manière pratique, les effets marginaux sont similaires aux contrastes marginaux.

L'ensemble de ces indicateurs marginaux se calculent pour certaines valeurs typiques des variables explicatives, avec plusieurs approches possibles pour définir des valeurs typiques : moyenne / mode, valeurs observées, valeurs personnalisées...

Nous présenterons ces différents concepts plus en détail dans la suite de ce chapitre.

Plusieurs packages proposent des fonctions pour le calcul d'estimations marginales, `{marginaleffects}`, `{emmeans}`, `{margins}`, `{effects}`, ou encore `{ggeffects}`, chacun avec des approches et un vocabulaire légèrement différent.

Le package `{broom.helpers}` fournit plusieurs *tidiers* qui permettent d'appeler les fonctions de ces autres packages et de renvoyer un tableau de données compatible avec la fonction `broom.helpers::tidy_plus_plus()` et dès lors de pouvoir générer un tableau mis en forme avec `gtsummary::tbl_regression()` ou un graphique avec `ggstats::ggcoef_model()`.

24.2 Données d'illustration

Pour illustrer ce chapitre, nous allons prendre un modèle logistique issu du chapitre sur la régression logistique binaire (cf. Chapitre 22).

```
library(tidyverse)
library(labelled)
library(gtsummary)
theme_gtsummary_language(
  "fr",
  decimal.mark = ",",
  big.mark = " "
)
```

```

data(hdv2003, package = "questionr")

d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
      ) |>
      fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    heures.tv = "Heures de télévision / jour"
  )

mod <- glm(
  sport ~ sexe + groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)

```



```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels()
```

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

Table 24.1: Odds Ratios du modèle logistique

Caractéristique	OR	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	1,52	1,24 – 1,87	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	0,68	0,43 – 1,06	0,084
45-64 ans	0,36	0,23 – 0,57	<0,001
65 ans et plus	0,27	0,16 – 0,46	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	2,54	1,73 – 3,75	<0,001
Technique / Professionnel	2,81	1,95 – 4,10	<0,001
Supérieur	6,55	4,50 – 9,66	<0,001
Non documenté	8,54	4,51 – 16,5	<0,001
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

Il faut se rappeler que pour calculer le modèle, les observations ayant au moins une valeur manquante ont été exclues. Le modèle n'a donc pas été calculé sur 2000 observations (nombre de lignes de `hdv2003`) mais sur 1995. On peut obtenir le tableau de données du modèle (*model frame*), qui ne contient que les variables et les observations utilisées, avec `broom.helpers::model_get_model_frame()`.

```
mf <- mod %>%
  broom.helpers::model_get_model_frame()
nrow(mf)
```

```
[1] 1995
```

```
colnames(mf)
```

```
[1] "sport"          "sexe"           "groupe_ages"    "etudes"         "heures.tv"
```

24.3 Prédictions marginales

24.3.1 Prédictions marginales moyennes

Pour illustrer et mieux comprendre ce que représente la différence entre les femmes et les hommes, nous allons effectuer des prédictions avec notre modèle en ne faisant varier que la variable *sexe*.

Une première approche consiste à dupliquer nos données observées et à supposer que tous les individus sont des femmes, puis à supposer que tous les individus sont des hommes.

```
mf_femmes <- mf |> mutate(sexe = "Femme")  
mf_hommes <- mf |> mutate(sexe = "Homme")
```

Nos deux jeux de données sont donc identiques pour toutes les autres variables et ne varient que pour le *sexe*. Nous pouvons maintenant prédire, à partir de notre modèle ajusté, la probabilité de faire du sport de chacun des individus de ces deux nouveaux jeux de données, puis à en calculer la moyenne.

```
mod |> predict(type = "response", newdata = mf_femmes) |> mean()
```

```
[1] 0.324814
```

```
mod |> predict(type = "response", newdata = mf_hommes) |> mean()
```

```
[1] 0.4036624
```

Nous obtenons ainsi des **prédictions marginales moyennes**, *average marginal predictions* en anglais, de respectivement 32% et 40% pour les femmes et pour les hommes.

Le même résultat, avec en plus un intervalle de confiance, peut s'obtenir avec `marginalEffects::predictions()`

```
library(marginalEffects)  
mod |>  
  predictions(variables = "sexe", by = "sexe", type = "response")
```

sexe	Estimate	Std. Error	z	Pr(> z)	S	2.5 %	97.5 %
Femme	0.325	0.0130	25.0	<0.001	456.7	0.299	0.350
Homme	0.404	0.0147	27.5	<0.001	549.0	0.375	0.432

Columns: sexe, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: response

Pour une variable continue, on peut procéder de la même manière en générant des prédictions marginales pour certaines valeurs de la variable. Par défaut, `marginalEffects::predictions()` réalise des prédictions selon les 5 nombres de Tukey (*Tukey's five numbers*, à savoir minimum, premier quartile, médiane, troisième quartile et maximum).

```
mod |>
  predictions(variables = "heures.tv", by = "heures.tv", type = "response")
```

heures.tv	Estimate	Std. Error	z	Pr(> z)	S	2.5 %	97.5 %
0	0.410	0.01711	23.96	<0.001	419.0	0.3764	0.443
1	0.386	0.01220	31.64	<0.001	727.2	0.3621	0.410
2	0.363	0.00991	36.58	<0.001	970.7	0.3432	0.382
3	0.340	0.01145	29.66	<0.001	639.9	0.3173	0.362
12	0.168	0.04220	3.99	<0.001	13.9	0.0855	0.251

Columns: heures.tv, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: response

Le package `{broom.helpers}` fournit la fonction `broom.helpers::tidy_marginal_predictions()` qui génèrent les prédictions marginales de chaque variable¹ avec `marginalEffects::predictions()` et renvoie les résultat dans un format directement utilisable avec `gtsummary::tbl_regression()`.

Note

Il est à noter que `broom.helpers::tidy_marginal_predictions()` renvoie des p-valeurs qui, par défaut, teste si les valeurs prédites sont différentes de 0 (sur l'échelle de la fonction de lien, donc différentes de 50% dans le cas d'une régression logistique). Ce type de tests n'est pas vraiment pertinent dans le cas présent. On peut facilement masquer la colonne des p-valeurs avec `modify_column_hide("p.value")`.

¹La fonction `broom.helpers::tidy_marginal_predictions()` peut également gérer des combinaisons de variables ou interactions, voir Chapitre 26).

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    type = "response",
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
  bold_labels() |>
  modify_column_hide("p.value")
```

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

Table 24.2: Prédictions marginales moyennes

Caractéristique	Prédictions Marginales Moyennes	95% IC
Sexe		
Femme	32.5%	29.9% – 35.0%
Homme	40.4%	37.5% – 43.2%
Groupe d'âges		
18-24 ans	51.2%	42.2% – 60.1%
25-44 ans	42.7%	39.3% – 46.2%
45-64 ans	29.9%	26.6% – 33.2%
65 ans et plus	24.9%	19.7% – 30.0%
Niveau d'études		
Primaire	16.1%	11.9% – 20.4%
Secondaire	31.8%	27.2% – 36.4%
Technique / Professionnel	34.0%	30.3% – 37.7%
Supérieur	53.2%	48.4% – 57.9%
Non documenté	59.2%	47.0% – 71.5%
Heures de télévision / jour		
0	41.0%	37.6% – 44.3%
1	38.6%	36.2% – 41.0%
2	36.3%	34.3% – 38.2%
3	34.0%	31.7% – 36.2%
12	16.8%	8.6% – 25.1%

La fonction `broom.helpers::plot_marginal_predictions()` permet de visualiser les prédictions marginales à la moyenne en réalisant une liste de graphiques, un par variable, que nous

pouvons combiner avec `patchwork::wrap_plots()`. L'opérateur `&` permet d'appliquer une fonction de `{ggplot2}` à chaque sous-graphique. Ici, nous allons uniformiser l'axe des *y*.

```
p <- mod |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
  patchwork::wrap_plots() &
  scale_y_continuous(
    limits = c(0, .8),
    labels = scales::label_percent()
  )
```

p

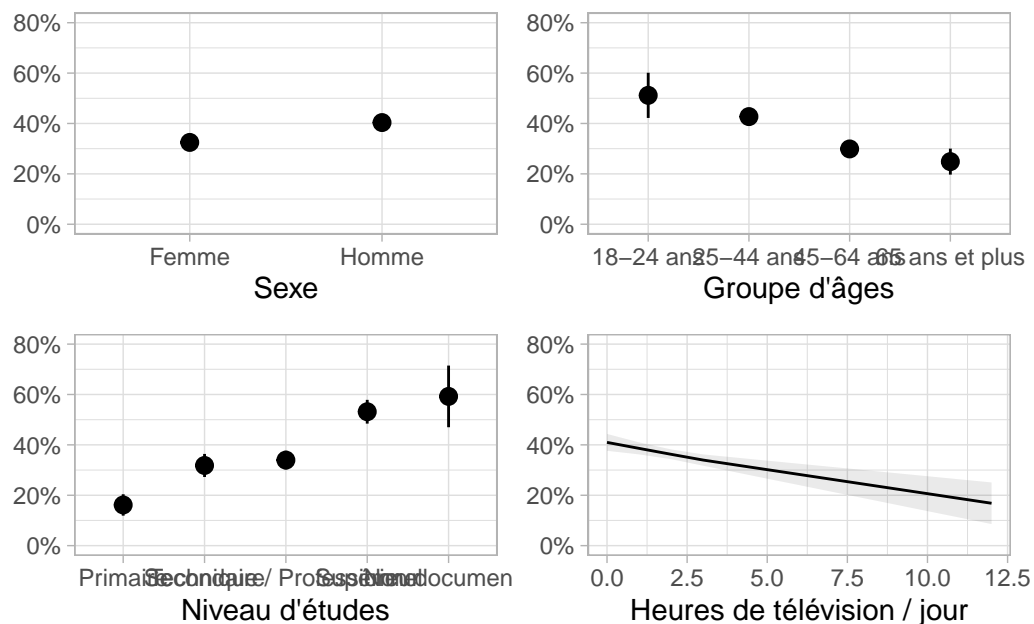


Figure 24.1: Prédictions marginales moyennes

Il est ici difficile de lire les étiquettes de la variable *etudes*. Nous pouvons éventuellement inverser l'axe des *x* et celui des *y* avec `ggplot2::coord_flip()`.

```
p & coord_flip()
```

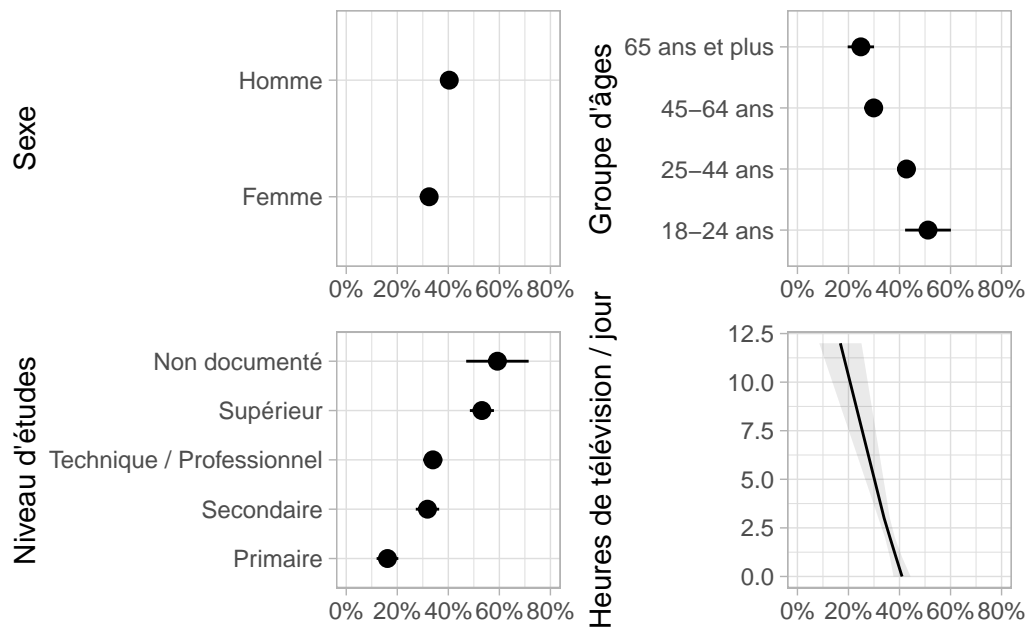


Figure 24.2: Prédictions marginales moyennes

Une alternative possible avec d'avoir recours à `ggstats::ggcoef_model()`.

```
mod |>
  ggstats::ggcoef_model(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    tidy_args = list(type = "response"),
    show_p_values = FALSE,
    signif_stars = FALSE,
    significance = NULL,
    vline = FALSE
  ) +
  scale_x_continuous(labels = scales::label_percent())
```

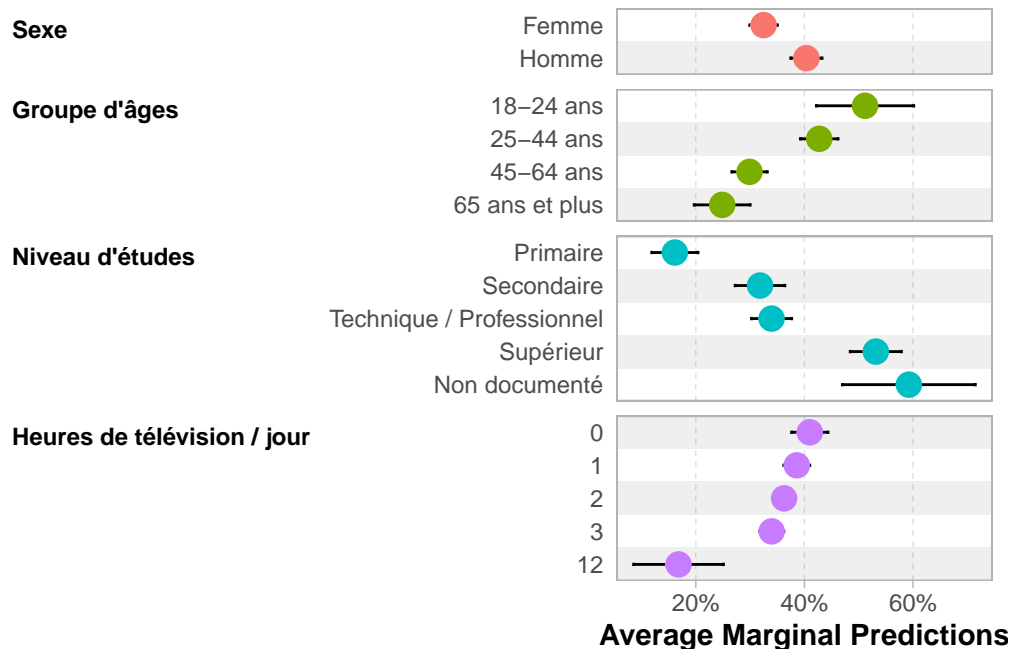


Figure 24.3: Prédictions marginales moyennes

! Importance de l'argument `type` pour les modèles `glm`

Lorsque l'on a recours à des modèles calculés avec `glm()`, il est possible de réaliser des prédictions selon deux échelles : l'échelle de notre *outcome* ou variable à expliquer (`type = "response"`), ici exprimée en probabilités ou proportions puisqu'il s'agit d'une régression logistique, ou bien selon l'échelle de la fonction de lien (`type = "link"`) du modèle, ici la fonction *logit* (voir Section 22.4).

Avec l'option `type = "reponse"`, on indique à `{marginaleffects}` de calculer pour chaque individu une prédiction selon l'échelle de l'outcome puis de procéder à la moyenne, ce que nous avons fait dans les exemples précédents.

Si nous avons indiqué `type = "link"`, les prédictions auraient été faites selon l'échelle de la fonction de lien avant d'être moyennée.

```
mod |> predict(type = "link", newdata = mf_femmes) |> mean()
```

```
[1] -0.910525
```

```
mod |> predict(type = "link", newdata = mf_hommes) |> mean()
```

```
[1] -0.4928844
```

```
mod |>
  predictions(variables = "sexe", by = "sexe", type = "link")
```

sexe	Estimate	Std. Error	z	Pr(> z)	S	2.5 %	97.5 %
Femme	-0.911	0.0751	-12.13	<0.001	110.0	-1.058	-0.763
Homme	-0.493	0.0779	-6.33	<0.001	31.9	-0.646	-0.340

Columns: sexe, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: link

Depuis la version 0.10.0 de `{marginaleffects}`, si l'on ne précise pas le paramètre `type` (i.e. si `type = NULL`), la fonction `marginaleffects::predictions()` réalise les prédictions selon l'échelle de la fonction de lien, calcule les moyennes puis re-transforme ce résultat selon l'échelle de la variable à expliquer.

```
logit_inverse <- binomial("logit") |> purrr::pluck("linkinv")
mod |> predict(type = "link", newdata = mf_femmes) |> mean() |> logit_inverse()
```

```
[1] 0.2868924
```

```
mod |> predict(type = "link", newdata = mf_hommes) |> mean() |> logit_inverse()
```

```
[1] 0.3792143
```

```
mod |>
  predictions(variables = "sexe", by = "sexe")
```

sexe	Estimate	Pr(> z)	S	2.5 %	97.5 %
Femme	0.287	<0.001	110.0	0.258	0.318
Homme	0.379	<0.001	31.9	0.344	0.416

Columns: sexe, estimate, p.value, s.value, conf.low, conf.high
Type: invlink(link)

Or, la plupart du temps, le logit inverse de la moyenne des prédictions est différent de la moyenne des logit inverse des prédictions !

Les résultats seront similaires et du même ordre de grandeur, mais pas identiques.

24.3.2 Prédictions marginales à la moyenne

Pour les prédictions marginales moyennes, nous avons réalisé des prédictions pour chaque observations du tableau d'origine, en faisant varier juste une variable à la fois, avant de calculer la moyenne des prédictions.

Une alternative consiste à générer une sorte d'individu moyen / typique puis à réaliser des prédictions pour cette unique individu, en faisant juste varier la variable explicative d'intérêt. On parle alors de **prédictions marginales à la moyenne**, *marginal predictions at the mean* en anglais.

24.3.2.1 avec {marginaleffects}

On peut réaliser cela avec {marginaleffects} en précisant `newdata = "mean"`. Prenons un exemple pour la variable *sexe* :

```
mod |> predictions(variables = "sexe", newdata = "mean")
```

groupe_ages	etudes	heures.tv	sexe	Estimate	Pr(> z)	S
45-64 ans Technique / Professionnel		2.25	Femme	0.239	<0.001	59.5
45-64 ans Technique / Professionnel		2.25	Homme	0.323	<0.001	29.5
2.5 %	97.5 %					
0.196	0.289					
0.273	0.378					

Columns: rowid, rowidcf, estimate, p.value, s.value, conf.low, conf.high, sport, groupe_ages
Type: invlink(link)

Dans ce cas de figure, {marginaleffects} considère pour chaque variable continue sa moyenne (ici 2.246 pour *heures.tv*) et pour chaque variable catégorielle son mode (la valeur observée la plus fréquente, ici "Technique / Professionnel" pour la variable *etudes*). On fait juste varier les modalités de *sexe* puis on calcule la probabilité de faire du sport de ces individus moyens.

On peut également passer le paramètre `newdata = "mean"` à `broom.helpers::tidy_marginal_predictions()` ou même à `gtsummary::tbl_regression()`².

²Les paramètres additionnels indiqués à `gtsummary::tbl_regression()` sont transmis en cascade à `broom.helpers::tidy_plus_plus()` puis à `broom.helpers::tidy_marginal_predictions()` et enfin à `marginaleffects::predictions()`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    newdata = "mean",
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
  bold_labels() |>
  modify_column_hide("p.value")
```

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

Table 24.3: Prédictions marginales à la moyenne

Caractéristique	Prédictions Marginales à la Moyenne	95% IC
Sexe		
Femme	23.9%	19.6% – 28.9%
Homme	32.3%	27.3% – 37.8%
Groupe d'âges		
18-24 ans	46.8%	36.1% – 57.8%
25-44 ans	37.3%	32.1% – 42.8%
45-64 ans	23.9%	19.6% – 28.9%
65 ans et plus	19.2%	14.0% – 25.6%
Niveau d'études		
Primaire	10.1%	7.3% – 13.7%
Secondaire	22.1%	17.7% – 27.2%
Technique / Professionnel	23.9%	19.6% – 28.9%
Supérieur	42.3%	36.0% – 48.9%
Non documenté	48.9%	34.7% – 63.2%
Heures de télévision / jour		
0	29.2%	23.6% – 35.5%
1	26.8%	21.9% – 32.3%
2	24.5%	20.1% – 29.5%
3	22.3%	18.1% – 27.2%
12	8.8%	4.6% – 16.4%

De même, on peut générer une représentation graphique :

```
p <- mod |>
  broom.helpers::plot_marginal_predictions(newdata = "mean") |>
  patchwork::wrap_plots() &
  scale_y_continuous(
    limits = c(0, .8),
    labels = scales::label_percent()
  ) &
  coord_flip()
```

p

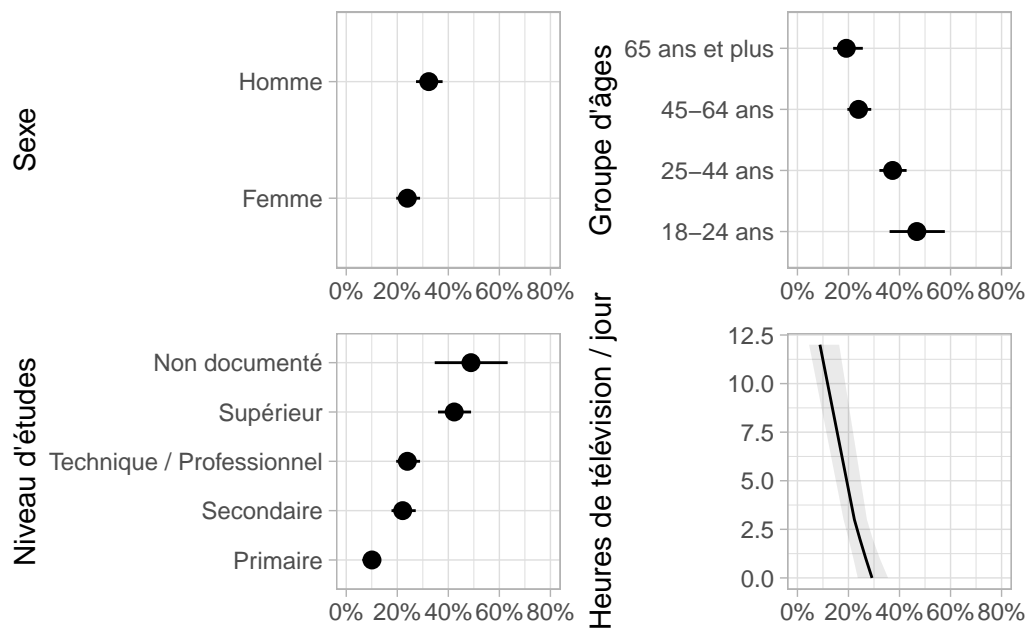


Figure 24.4: Prédictions marginales à la moyenne

Si l'on souhaite utiliser `ggstats::ggcoef_model()`, on peut directement indiquer `newdata = "mean"`. Il faudra passer cette option via `tidy_args` qui prend une liste d'arguments à transmettre à `tidy_fun`.

```
mod |>
  ggstats::ggcoef_model(
    tidy_fun = broom.helpers::tidy_marginal_predictions,
    tidy_args = list(newdata = "mean"),
```

```

show_p_values = FALSE,
signif_stars = FALSE,
significance = NULL,
vline = FALSE
) +
scale_x_continuous(labels = scales::label_percent())

```

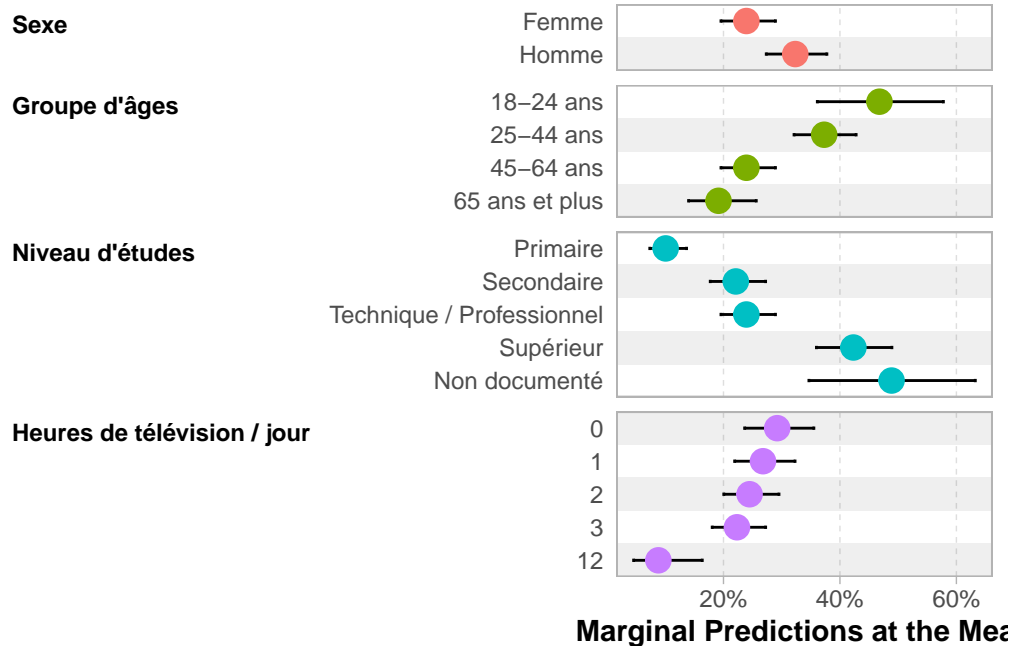


Figure 24.5: Prédictions marginales à la moyenne

24.3.2.2 avec {effects}

Le package `{effects}`³ adopte une approche un peu différente pour définir un individu moyen.

Calculons les prédictions marginales à la moyenne avec la fonction `effects::Effect()`.

```

e <- effects::Effect("sexe", mod)
e

```

³Malgré son nom, le package `{effects}` ne calcule pas des effets marginaux mais des prédictions marginales, selon la terminologie retenue au début de ce document.

```

sexe effect
sexe
  Femme      Homme
0.2868924 0.3792143

```

On le voit, les résultats sont là encore assez proches mais différents. Regardons de plus près les données utilisées pour les prédictions.

```
e$model.matrix
```

```

      (Intercept) sexeHomme groupe_ages25-44 ans groupe_ages45-64 ans
1             1             0             0.3533835             0.3719298
2             1             1             0.3533835             0.3719298
      groupe_ages65 ans et plus etudesSecondaire etudesTechnique / Professionnel
1                   0.1904762             0.193985                   0.2962406
2                   0.1904762             0.193985                   0.2962406
      etudesSupérieur etudesNon documenté heures.tv
1             0.2205514             0.05614035  2.246566
2             0.2205514             0.05614035  2.246566
attr(,"assign")
[1] 0 1 2 2 2 3 3 3 3 4
attr(,"contrasts")
attr(,"contrasts")$sexe
[1] "contr.treatment"

attr(,"contrasts")$groupe_ages
[1] "contr.treatment"

attr(,"contrasts")$etudes
[1] "contr.treatment"

```

Pour les variables continues, `{effects}` utilise la moyenne observée de la variable, comme précédemment avec `{marginaleffects}`. Par contre, pour les variables catégorielles, ce n'est pas le mode qui est utilisé, mais l'ensemble des modalités, pondérées selon leur proportion observée dans l'échantillon. Cette approche a l'avantage de moyenniser également les variables catégorielles, même si les individus pour lesquels une prédiction est réalisée sont complètement fictifs.

On peut utiliser `broom.helpers::tidy_all_effects()` pour générer un tableau de prédictions marginales avec `{effects}`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_all_effects,
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
  bold_labels()
```

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

Table 24.4: Prédictions marginales à la moyenne avec le package `effects`

Caractéristique	Prédictions Marginales à la Moyenne	95% IC
Sexe		
Femme	28.7%	25.8% – 31.8%
Homme	37.9%	34.4% – 41.6%
Groupe d'âges		
18-24 ans	51.2%	41.0% – 61.3%
25-44 ans	41.5%	37.4% – 45.7%
45-64 ans	27.3%	23.9% – 30.9%
65 ans et plus	22.0%	17.4% – 27.5%
Niveau d'études		
Primaire	14.9%	11.3% – 19.3%
Secondaire	30.7%	26.2% – 35.7%
Technique / Professionnel	32.9%	29.1% – 37.0%
Supérieur	53.4%	48.3% – 58.4%
Non documenté	59.9%	46.6% – 71.8%
Heures de télévision / jour		
0	38.9%	34.8% – 43.2%
3	30.7%	28.1% – 33.4%
6	23.6%	18.9% – 28.9%
9	17.7%	11.9% – 25.4%
10	16.0%	10.1% – 24.4%

Pour une représentation graphique, nous pouvons utiliser les fonctions internes d'`{effects}` en appliquant `plot()` aux résultats de `effects::allEffects()` qui calcule les prédictions marginales de chaque variable.

```
mod |>
  effects::allEffects() |>
  plot()
```

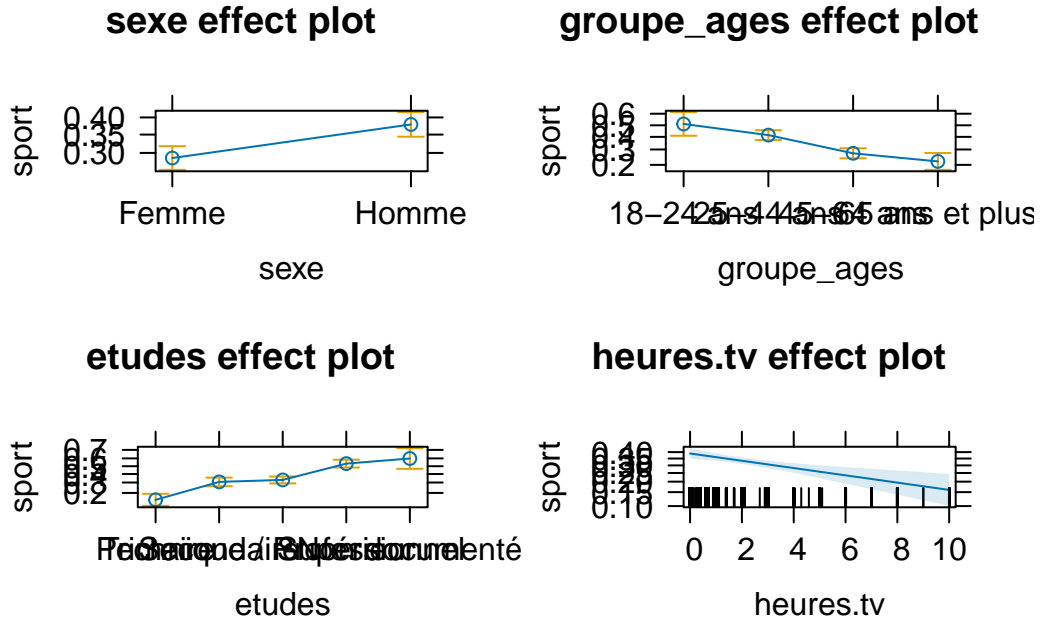


Figure 24.6: Prédictions marginales à la moyenne avec le package effects

On peut aussi utiliser `ggstats::ggcoef_model()`⁴.

```
mod |>
  ggstats::ggcoef_model(
    tidy_fun = broom.helpers::tidy_all_effects,
    vline = FALSE
  ) +
  scale_x_continuous(labels = scales::label_percent())
```

⁴De manière générale, `ggstats::ggcoef_model()` est compatible avec les mêmes `tidy_fun` que `gtsummary::tbl_regression()`, les deux fonctions utilisant en interne `broom.helpers::tidy_plus_plus()`.

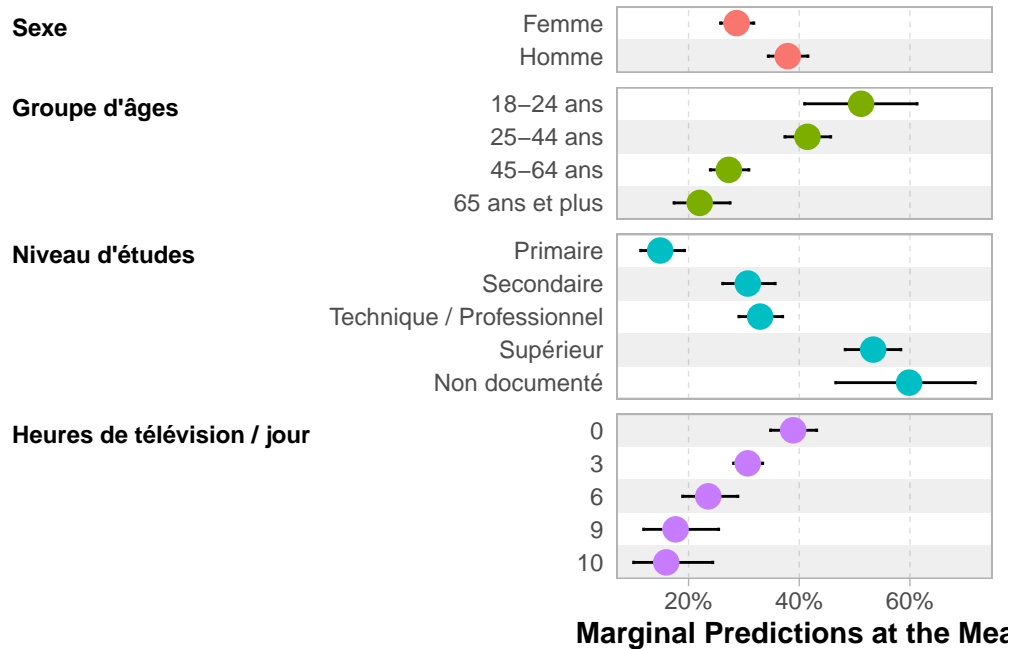


Figure 24.7: Prédictions marginales à la moyenne avec le package `effects`

24.3.3 Variantes

Le package `{ggeffects}` propose une fonction `ggeffects::ggpredict()` qui calcule des prédictions marginales à la moyenne des variables continues et à la première modalité (utilisée comme référence) des variables catégorielles. On ne peut donc plus, au sens strict, parler de prédictions à la moyenne. `{broom.helpers}` fournit une fonction `tidy_ggpredict()`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_ggpredict,
    estimate_fun = scales::label_percent(accuracy = 0.1)
  ) |>
  bold_labels()
```

Data were 'prettified'. Consider using ``terms="heures.tv [all]"`` to get smooth plots.

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

Table 24.5: Prédictions marginales avec ggpredict()

Caractéristique	Prédictions Marginales	95% IC
Sexe		
Femme	23.8%	15.3% – 35.1%
Homme	32.2%	21.4% – 45.4%
Groupe d'âges		
18-24 ans	23.8%	15.3% – 35.1%
25-44 ans	17.5%	12.7% – 23.5%
45-64 ans	10.1%	7.3% – 13.7%
65 ans et plus	7.8%	5.5% – 10.9%
Niveau d'études		
Primaire	23.8%	15.3% – 35.1%
Secondaire	44.2%	33.0% – 56.1%
Technique / Professionnel	46.8%	36.1% – 57.8%
Supérieur	67.2%	56.2% – 76.6%
Non documenté	72.8%	62.8% – 80.9%
Heures de télévision / jour		
0	29.1%	18.7% – 42.3%
1	26.7%	17.2% – 38.9%
2	24.4%	15.7% – 35.8%
3	22.2%	14.2% – 33.0%
4	20.2%	12.8% – 30.4%
5	18.3%	11.4% – 28.2%
6	16.6%	10.0% – 26.1%
7	15.0%	8.8% – 24.3%
8	13.5%	7.7% – 22.7%
9	12.1%	6.6% – 21.2%
10	10.9%	5.7% – 19.9%
11	9.8%	4.9% – 18.7%
12	8.8%	4.2% – 17.6%

Pour une représentation graphique, on peut utiliser les fonctionnalités natives incluses dans le package `{ggeffects}`.

```
mod |>
  ggeffects::ggpredict() |>
  lapply(plot) |>
  patchwork::wrap_plots()
```

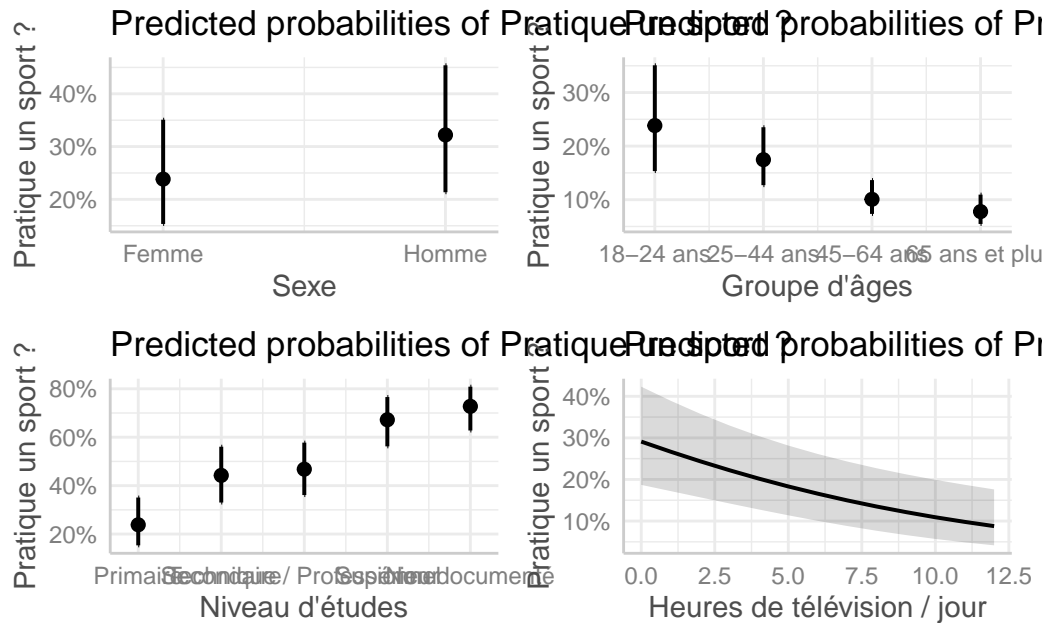


Figure 24.8: Prédictions marginales avec ggpredict()

24.4 Contrastes marginaux

Maintenant que nous savons estimer des prédictions marginales, nous pouvons facilement calculer des **contrastes marginaux**, à savoir des différences entre prédictions marginales.

24.4.1 Contrastes marginaux moyens

Considérons tout d'abord la variable catégorielle *sexe* et calculons les prédictions marginales moyennes avec `marginalEffects::predictions()`.

```
pred <- predictions(mod, variables = "sexe", by = "sexe", type = "response")
pred
```

sexe	Estimate	Std. Error	z	Pr(> z)	S	2.5 %	97.5 %
Femme	0.325	0.0130	25.0	<0.001	456.7	0.299	0.350
Homme	0.404	0.0147	27.5	<0.001	549.0	0.375	0.432

Columns: sexe, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: response

Le contraste entre les hommes et les femmes est tout simplement la différence et les deux prédictions marginales.

```
pred$estimate[2] - pred$estimate[1]
```

```
[1] 0.07884839
```

La fonction `marginalEffects::avg_comparisons()` permet de réaliser directement ce calcul.

```
avg_comparisons(mod, variables = "sexe")
```

Term	Contrast	Estimate	Std. Error	z	Pr(> z)	S	2.5 %	97.5 %
sexe Homme - Femme		0.0788	0.0197	4	<0.001	14.0	0.0402	0.117

Columns: term, contrast, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: response

Astuce

Dans les faits, `marginalEffects::avg_comparisons()` a calculé la différence entre les hommes et les femmes pour chaque observation d'origine puis a réalisé la moyenne des différences. Mathématiquement, la moyenne des différences est équivalente à la différence des moyennes.

Les contrastes calculés ici ont été moyennés sur l'ensemble des valeurs observées. On parle donc de **contrastes marginaux moyens** (*average marginal contrasts*).

Par défaut, chaque modalité est contrastée avec la première modalité prise comme référence (voir exemple ci-dessous avec la variable `groupe_ages`).

Regardons maintenant une variable continue.

```
avg_comparisons(mod, variables = "heures.tv")
```

Term	Contrast	Estimate	Std. Error	z	Pr(> z)	S	2.5 %	97.5 %
heures.tv	+1	-0.0224	0.00606	-3.7	<0.001	12.2	-0.0343	-0.0105

Columns: term, contrast, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: response

Par défaut, `marginalEffects::avg_comparisons()` calcule, pour chaque valeur observée de *heures.tv*, l'effet sur la probabilité de pratiquer un sport d'augmenter de 1 le nombre d'heures quotidiennes de télévision (plus précisément la différence des valeurs prédites pour la valeur observée plus 0,5 et la valeur observée moins 0,5).

On peut facilement obtenir la liste des contrastes marginaux pour l'ensemble des variables.

```
avg_comparisons(mod)
```

Term	Contrast	Estimate	Std. Error	z
etudes Non documenté - Primaire		0.4309	0.06915	6.23
etudes Secondaire - Primaire		0.1568	0.03143	4.99
etudes Supérieur - Primaire		0.3701	0.03370	10.98
etudes Technique / Professionnel - Primaire		0.1781	0.02948	6.04
groupe_ages 25-44 ans - 18-24 ans		-0.0844	0.04921	-1.71
groupe_ages 45-64 ans - 18-24 ans		-0.2127	0.05070	-4.20
groupe_ages 65 ans et plus - 18-24 ans		-0.2631	0.05558	-4.73
heures.tv +1		-0.0224	0.00606	-3.70
sexe Homme - Femme		0.0788	0.01970	4.00
Pr(> z)	S	2.5 %	97.5 %	
<0.001	31.0	0.2954	0.5665	
<0.001	20.6	0.0952	0.2184	
<0.001	90.8	0.3040	0.4361	
<0.001	29.3	0.1203	0.2359	
0.0865	3.5	-0.1808	0.0121	
<0.001	15.2	-0.3121	-0.1133	
<0.001	18.8	-0.3720	-0.1541	
<0.001	12.2	-0.0343	-0.0105	
<0.001	14.0	0.0402	0.1175	

Columns: term, contrast, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: response

Il est important de noter que le nom des colonnes n'est pas compatible avec les fonctions de `{broom.helpers}` et par extension avec `gtsummary::tbl_regression()` et

`ggstats::ggcoef_model()`. On utilisera donc `broom.helpers::tidy_marginal_contrasts()`⁵ qui remets en forme le tableau de résultats dans un format compatible. On pourra ainsi produire un tableau propre des résultats⁶. Au sens strict, les contrastes obtenus s'expriment en **points de pourcentage**. Pour éviter toute mauvaise interprétation des résultats, on privilégiera la notation *pp* (points de pourcentage) plutôt que le symbole %.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_marginal_contrasts,
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus",
      suffix = " pp"
    )
  ) |>
  bold_labels()
```

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

Table 24.6: Contrastes marginaux moyens

Caractéristique	Contrastes Marginaux		
	Moyens	95% IC	p-valeur
Sexe			
Homme - Femme	+7.9 pp	+4.0 pp – +11.7 pp	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-8.4 pp	-18.1 pp – +1.2 pp	0,086
45-64 ans - 18-24 ans	-21.3 pp	-31.2 pp – -11.3 pp	<0,001
65 ans et plus - 18-24 ans	-26.3 pp	-37.2 pp – -15.4 pp	<0,001

⁵Il existe également une fonction `broom.helpers::tidy_avg_comparisons()` mais on lui préférera `broom.helpers::tidy_marginal_contrasts()`. Pour un modèle sans interaction, les résultats sont identiques. Mais `broom.helpers::tidy_marginal_contrasts()` peut gérer des termes d'interactions, ce qui sera utile dans un prochain chapitre (cf. Chapitre 26).

⁶Notez l'utilisation de `style_positive = "plus"` dans l'appel de `scales::label_percent()` pour ajouter un signe + devant les valeurs positives, afin de bien indiquer que l'on représente le résultat d'une différence.

Table 24.6: Contrastes marginaux moyens

Caractéristique	Contrastes Marginaux		
	Moyens	95% IC	p-valeur
Niveau d'études			
Non documenté - Primaire	+43.1 pp	+29.5 pp – +56.6 pp	<0,001
Secondaire - Primaire	+15.7 pp	+9.5 pp – +21.8 pp	<0,001
Supérieur - Primaire	+37.0 pp	+30.4 pp – +43.6 pp	<0,001
Technique / Professionnel - Primaire	+17.8 pp	+12.0 pp – +23.6 pp	<0,001
Heures de télévision / jour			
+1	-2.2 pp	-3.4 pp – -1.1 pp	<0,001

De même, on peut représenter les contrastes marginaux moyens avec `ggstats::ggcoef_model()`.

```
ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_marginal_contrasts
) +
  ggplot2::scale_x_continuous(
    labels = scales::label_percent(
      style_positive = "plus",
      suffix = "pp"
    )
  )
```

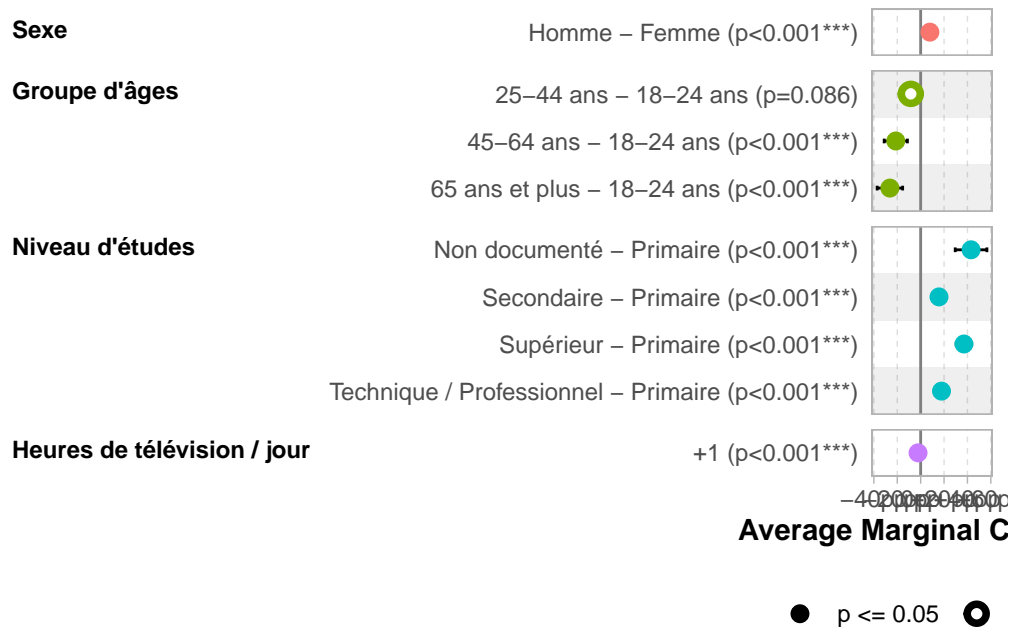


Figure 24.9: Contrastes marginaux moyens

💡 Astuce

Il est possible de personnaliser le type de contrastes calculés, variable par variable, avec l'option `variables_list` de `broom.helpers::tidy_marginal_contrasts()`. La syntaxe est un peu particulière : il faut transmettre une liste de listes.

```
mod |>
tbl_regression(
  tidy_fun = broom.helpers::tidy_marginal_contrasts,
  variables_list = list(
    list(heures.tv = 2),
    list(groupe_ages = "pairwise"),
    list(etudes = "sequential")
  ),
  estimate_fun = scales::label_percent(
    accuracy = 0.1,
    style_positive = "plus",
    suffix = " pp"
  )
) |>
bold_labels()
```

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

Caractéristique	Contrastes		
	Marginaux Moyens	95% IC	p-valeur
Heures de télévision / jour			
+2	-4.4 pp	-6.7 pp – -2.1 pp	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-8.4 pp	-18.1 pp – +1.2 pp	0,086
45-64 ans - 18-24 ans	-21.3 pp	-31.2 pp – -11.3 pp	<0,001
45-64 ans - 25-44 ans	-12.8 pp	-17.6 pp – -8.1 pp	<0,001
65 ans et plus - 18-24 ans	-26.3 pp	-37.2 pp – -15.4 pp	<0,001
65 ans et plus - 25-44 ans	-17.9 pp	-24.3 pp – -11.4 pp	<0,001
65 ans et plus - 45-64 ans	-5.0 pp	-11.0 pp – +0.9 pp	0,10
Niveau d'études			

Non documenté - Supérieur	+6.1 pp	-7.0 pp – +19.2 pp	0,4
Secondaire - Primaire	+15.7 pp	+9.5 pp – +21.8 pp	<0,001
Supérieur - Technique / Professionnel	+19.2 pp	+13.3 pp – +25.1 pp	<0,001
Technique / Professionnel - Secondaire	+2.1 pp	-3.8 pp – +8.0 pp	0,5

On peut obtenir le même résultat avec `broom.helpers::tidy_avg_comparison()` avec une syntaxe un peu plus simple (en passant une liste via `variables` au lieu d'une liste de listes via `variables_list`).

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_avg_comparisons,
    variables = list(
      heures.tv = 2,
      groupe_ages = "pairwise",
      etudes = "sequential"
    ),
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus",
      suffix = " pp"
    )
  ) |>
  bold_labels()
```

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

Caractéristique	Contrastes		
	Marginaux Moyens	95% IC	p-valeur
Niveau d'études			
Non documenté - Supérieur	+6.1 pp	-7.0 pp – +19.2 pp	0,4
Secondaire - Primaire	+15.7 pp	+9.5 pp – +21.8 pp	<0,001
Supérieur - Technique / Professionnel	+19.2 pp	+13.3 pp – +25.1 pp	<0,001
Technique / Professionnel - Secondaire	+2.1 pp	-3.8 pp – +8.0 pp	0,5
Groupe d'âges			
25-44 ans - 18-24 ans	-8.4 pp	-18.1 pp – +1.2 pp	0,086
45-64 ans - 18-24 ans	-21.3 pp	-31.2 pp – -11.3 pp	<0,001

45-64 ans - 25-44 ans	-12.8 pp	-17.6 pp – -8.1 pp	<0,001
65 ans et plus - 18-24 ans	-26.3 pp	-37.2 pp – -15.4 pp	<0,001
65 ans et plus - 25-44 ans	-17.9 pp	-24.3 pp – -11.4 pp	<0,001
65 ans et plus - 45-64 ans	-5.0 pp	-11.0 pp – +0.9 pp	0,10
Heures de télévision / jour			
+2	-4.4 pp	-6.7 pp – -2.1 pp	<0,001

24.4.2 Contrastes marginaux à la moyenne

Comme précédemment, plutôt que de calculer les contrastes marginaux pour chaque individu observé avant de faire la moyenne des résultats, une approche alternative consiste à considérer un individu moyen / typique et à calculer les contrastes marginaux pour cet individu. On parle alors de **contrastés marginaux à la moyenne** (*marginal contrasts at the mean*).

Avec `{marginaleffects}`, il suffit de spécifier `newdata = "mean"`. Les variables continues seront fixées à leur moyenne et les variables catégorielles à leur mode (modalité la plus fréquente dans l'échantillon).

```
mod |>
tbl_regression(
  tidy_fun = broom.helpers::tidy_marginal_contrasts,
  newdata = "mean",
  estimate_fun = scales::label_percent(
    accuracy = 0.1,
    style_positive = "plus",
    suffix = " pp"
  )
) |>
bold_labels()
```

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

Table 24.9: Contrastes marginaux à la moyenne

Caractéristique	Contrastes Marginaux à la Moyenne	95% IC	p-valeur
Sexe			
Homme - Femme	+8.4 pp	+4.3 pp – +12.5 pp	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-9.5 pp	-20.5 pp – +1.5 pp	0,090
45-64 ans - 18-24 ans	-22.9 pp	-33.8 pp – -11.9 pp	<0,001
65 ans et plus - 18-24 ans	-27.6 pp	-39.2 pp – -16.1 pp	<0,001
Niveau d'études			
Non documenté - Primaire	+38.8 pp	+24.1 pp – +53.5 pp	<0,001
Secondaire - Primaire	+12.0 pp	+7.0 pp – +17.1 pp	<0,001
Supérieur - Primaire	+32.2 pp	+25.7 pp – +38.7 pp	<0,001
Technique / Professionnel - Primaire	+13.9 pp	+9.0 pp – +18.7 pp	<0,001
Heures de télévision / jour			
+1	-2.1 pp	-3.3 pp – -1.0 pp	<0,001

Pour la fonction `ggstats::ggcoef_model()`, on utilisera l'argument `tidy_args` pour transmettre l'option `newdata = "mean"`.

```
ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_marginal_contrasts,
  tidy_args = list(newdata = "mean")
) +
ggplot2::scale_x_continuous(
  labels = scales::label_percent(
    style_positive = "plus",
    suffix = "pp"
  )
)
```

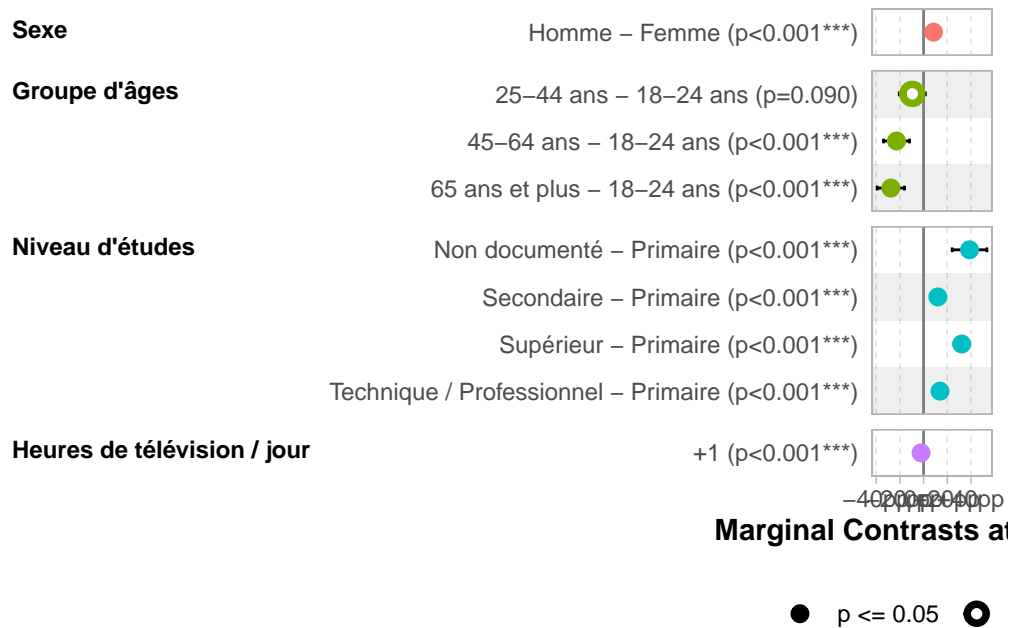


Figure 24.10: Contrastes marginaux à la moyenne

24.5 Pentés marginales / Effets marginaux

Les effets marginaux, ou plus précisément les pentes marginales, sont similaires aux contrastes marginaux, avec une différence subtile. Pour une variable continue, les contrastes marginaux sont une différence entre deux prédictions tandis que les **effets marginaux** (*marginal effects*) ou **pentés marginales** (*marginal slopes*). Dis autrement, l'effet marginal d'un régresseur continu x est la pente / dérivée $\partial y / \partial x$ la fonction de prédiction y , mesurée à des valeurs spécifiques de x .

Les effets marginaux sont le plus souvent calculés selon l'échelle de l'*outcome* et représentent le changement attendu de l'*outcome* pour une augmentation du régresseur d'une unité.

Par définition, les effets marginaux ne sont pas définis pour les variables catégorielles. La plupart des fonctions rapportent, à la place, les contrastes marginaux pour ces variables catégorielles.

Comme pour les prédictions marginales et les contrastes marginaux, plusieurs approches existent (voir par exemple la [vignette dédiée](#) du package `{marginaleffects}`).

24.5.1 Pententes marginales moyennes / Effets marginaux moyens

Les **effets marginaux moyens** (*average marginal effects*) sont calculés en deux temps : (i) un effet marginal est calculé pour chaque individu observé dans le modèle ; (ii) puis la moyenne de ces effets individuels est calculée.

On aura tout simplement recours à la fonction `marginalEffects::avg_slopes()`.

```
avg_slopes(mod)
```

	Term	Contrast	Estimate	Std. Error	z
etudes	Non documenté - Primaire		0.4309	0.0691	6.23
etudes	Secondaire - Primaire		0.1568	0.0314	4.99
etudes	Supérieur - Primaire		0.3701	0.0337	10.98
etudes	Technique / Professionnel - Primaire		0.1781	0.0295	6.04
groupe_ages	25-44 ans - 18-24 ans		-0.0844	0.0492	-1.71
groupe_ages	45-64 ans - 18-24 ans		-0.2127	0.0507	-4.20
groupe_ages	65 ans et plus - 18-24 ans		-0.2631	0.0556	-4.73
heures.tv	dY/dX		-0.0227	0.0062	-3.66
sexe	Homme - Femme		0.0788	0.0197	4.00

Pr(>|z|) S 2.5 % 97.5 %

<0.001	31.0	0.2954	0.5665
<0.001	20.6	0.0952	0.2184
<0.001	90.8	0.3040	0.4361
<0.001	29.3	0.1203	0.2359
0.0865	3.5	-0.1808	0.0121
<0.001	15.2	-0.3121	-0.1133
<0.001	18.8	-0.3720	-0.1541
<0.001	11.9	-0.0348	-0.0105
<0.001	14.0	0.0402	0.1175

Columns: term, contrast, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type: response

Pour un usage avec `broom.helpers::tidy_plus_plus()`, `gtsummary::tbl_regression()` ou `ggstats::ggcoef_model()`, on utilisera `broom.helpers::tidy_avg_slopes()`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_avg_slopes,
    estimate_fun = scales::label_percent(
```

```

    accuracy = 0.1,
    style_positive = "plus",
    suffix = " pp"
  )
) |>
bold_labels()

```

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

Table 24.10: Effets marginaux moyens

Caractéristique	Effets Marginaux Moyens	95% IC	p-valeur
Niveau d'études			
Non documenté - Primaire	+43.1 pp	+29.5 pp – +56.6 pp	<0,001
Secondaire - Primaire	+15.7 pp	+9.5 pp – +21.8 pp	<0,001
Supérieur - Primaire	+37.0 pp	+30.4 pp – +43.6 pp	<0,001
Technique / Professionnel - Primaire	+17.8 pp	+12.0 pp – +23.6 pp	<0,001
Groupe d'âges			
25-44 ans - 18-24 ans	-8.4 pp	-18.1 pp – +1.2 pp	0,086
45-64 ans - 18-24 ans	-21.3 pp	-31.2 pp – -11.3 pp	<0,001
65 ans et plus - 18-24 ans	-26.3 pp	-37.2 pp – -15.4 pp	<0,001
Heures de télévision / jour			
dY/dX	-2.3 pp	-3.5 pp – -1.1 pp	<0,001
Sexe			
Homme - Femme	+7.9 pp	+4.0 pp – +11.7 pp	<0,001

```

ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_avg_slopes
)

```

```
) +
  ggplot2::scale_x_continuous(
    labels = scales::label_percent(
      style_positive = "plus",
      suffix = "pp"
    )
  )
)
```

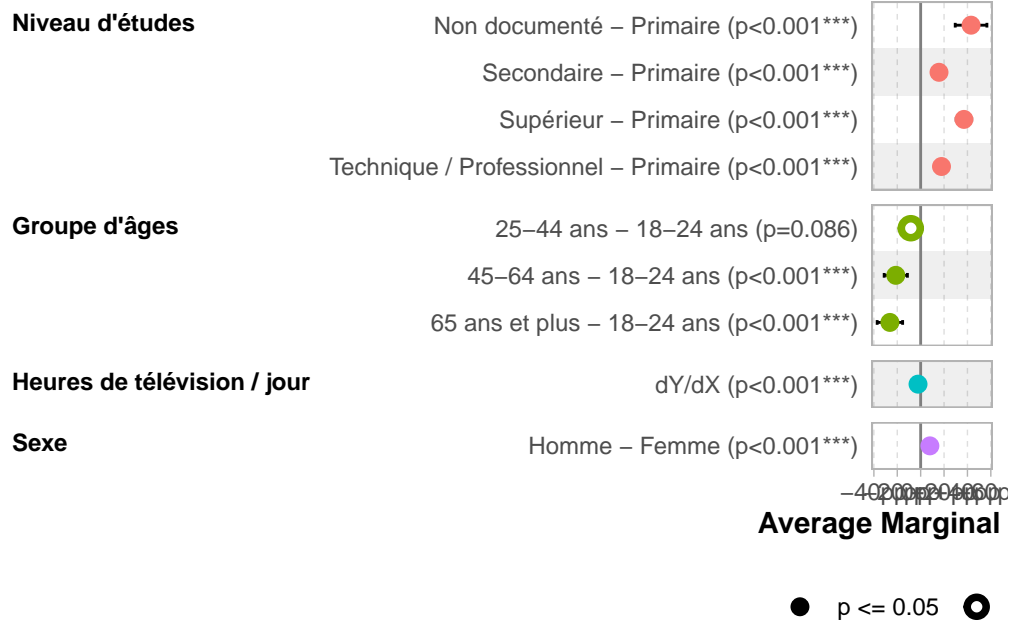


Figure 24.11: Effets marginaux moyens

Un résultat similaire peut être obtenu avec `margins::margins()`, le package `{margins}` s'inspirant de la commande **Stata** `margins`.

```
margins::margins(mod) %>% tidy()
```

```
# A tibble: 9 x 5
  term                estimate std.error statistic  p.value
  <chr>              <dbl>    <dbl>    <dbl>    <dbl>
1 etudesNon documenté 0.431    0.0691     6.23 4.60e-10
2 etudesSecondaire    0.157    0.0314     4.99 6.10e- 7
3 etudesSupérieur     0.370    0.0337    11.0 4.69e-28
```


4 etudesTechnique / Professionnel	0.178	0.0295	6.04	1.53e- 9
5 groupe_ages25-44 ans	-0.0844	0.0492	-1.71	8.65e- 2
6 groupe_ages45-64 ans	-0.213	0.0507	-4.20	2.73e- 5
7 groupe_ages65 ans et plus	-0.263	0.0556	-4.73	2.21e- 6
8 heures.tv	-0.0227	0.00620	-3.66	2.56e- 4
9 sexeHomme	0.0788	0.0197	4.00	6.26e- 5

For `{broom.helpers}`, `{gtsummary}` or `{ggstats}`, use `broom.helpers::tidy_margins()`.

```
mod |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_margins,
    estimate_fun = scales::label_percent(
      accuracy = 0.1,
      style_positive = "plus",
      suffix = " pp"
    )
  ) |>
  bold_labels()
```

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

Table 24.11: Effets marginaux moyens avec margins

Caractéristique	Effets Marginaux		p-valeur
	Moyens	95% IC	
Niveau d'études			
Primaire	—	—	
Non documenté	+43.1 pp	+29.5 pp – +56.6 pp	<0,001
Secondaire	+15.7 pp	+9.5 pp – +21.8 pp	<0,001
Supérieur	+37.0 pp	+30.4 pp – +43.6 pp	<0,001
Technique / Professionnel	+17.8 pp	+12.0 pp – +23.6 pp	<0,001
Groupe d'âges			
18-24 ans	—	—	

Table 24.11: Effets marginaux moyens avec margins

Caractéristique	Effets Marginaux Moyens	95% IC	p-valeur
25-44 ans	-8.4 pp	-18.1 pp – +1.2 pp	0,086
45-64 ans	-21.3 pp	-31.2 pp – -11.3 pp	<0,001
65 ans et plus	-26.3 pp	-37.2 pp – -15.4 pp	<0,001
Heures de télévision / jour	-2.3 pp	-3.5 pp – -1.1 pp	<0,001
Sexe			
Femme	—	—	
Homme	+7.9 pp	+4.0 pp – +11.7 pp	<0,001

```
ggstats::ggcoef_model(
  mod,
  tidy_fun = broom.helpers::tidy_margins
) +
ggplot2::scale_x_continuous(
  labels = scales::label_percent(
    style_positive = "plus",
    suffix = "pp"
  )
)
```

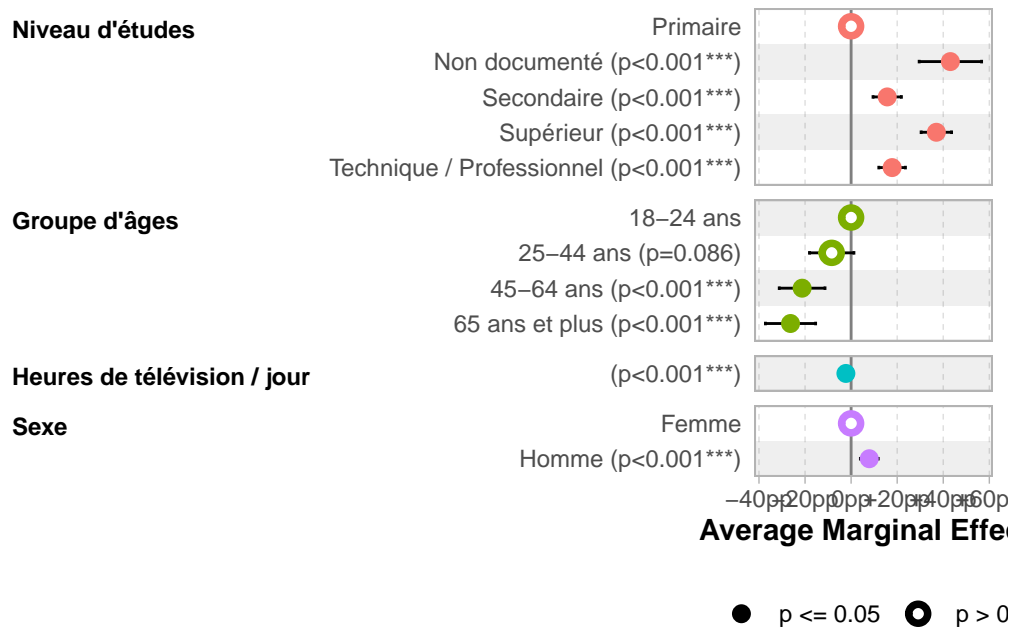


Figure 24.12: Effets marginaux moyens avec marges

24.5.2 Pententes marginales à la moyenne / Effets marginaux à la moyenne

Pour les **effets marginaux à la moyenne** (*marginal effects at the mean*), simplement indiquer `newdata = "mean"` à `broom.helpers::tidy_marginaleffects()`.

24.6 Lectures complémentaires (en anglais)

- [Documentation of the `marginalEffects` package](#) par Vincent Arel-Bundock
- [Marginalia: A guide to figuring out what the heck marginal effects, marginal slopes, average marginal effects, marginal effects at the mean, and all these other marginal things are](#) par Andrew Heiss
- [Introduction to Adjusted Predictions and Marginal Effects in R](#) par Daniel Lüdtke
- [An Introduction to `margins`](#)
- [Marginal effects / slopes, contrasts, means and predictions with `broom.helpers`](#)
- [The Marginal Effects Zoo](#) par Vincent Arel-Bundock

24.7 webin-R

La régression logistique est présentée sur YouTube dans le [webin-R #24](#) (*Prédictions, contrastes & effets marginaux*).

<https://youtu.be/rAAh8rNLAOA>

25 Contrastes (variables catégorielles)

Dans les modèles de régression (comme les modèles linéaires, cf. Chapitre 21, ou les modèles linéaires généralisés comme la régression logistique binaire, cf. Chapitre 22), une transformation des variables catégorielles est nécessaire pour qu'elles puissent être prises en compte dans le modèle. On va dès lors définir des **contrastes**.

De manière générale, une variable catégorielle à n modalités va être transformée en $n-1$ variables quantitatives. Il existe cependant plusieurs manières de faire (i.e. plusieurs types de contrastes). Et, selon les contrastes choisis, les coefficients du modèles ne s'interpréteront pas de la même manière.

25.1 Contrastes de type traitement

Par défaut, **R** applique des contrastes de type traitement pour un facteur non ordonné. Il s'agit notamment des contrastes utilisés par défaut dans les chapitres précédents.

25.1.1 Exemple 1 : un modèle linéaire avec une variable catégorielle

Commençons avec un premier exemple que nous allons calculer avec le jeu de données *trial* chargé en mémoire lorsque l'on appelle l'extension `{gtsummary}`. Ce jeu de données contient les observations de 200 patients. Nous nous intéressons à deux variables en particulier : *marker* une variable numérique correspondant à un marqueur biologique et *grade* un facteur à trois modalités correspondant à différent groupes de patients.

Regardons la moyenne de *marker* pour chaque valeur de *grade*.

```
library(tidyverse)
library(gtsummary)
trial |>
  select(marker, grade) |>
  tbl_summary(
    by = grade,
    statistic = marker ~ "{mean}",
    digits = marker ~ 4
```

```
) |>
  add_overall(last = TRUE)
```

Characteristic	I, N = 68	II, N = 68	III, N = 64	Overall, N = 200
Marker Level (ng/mL)	1.0669	0.6805	0.9958	0.9160
Unknown	2	5	3	10

Utilisons maintenant une régression linéaire pour modéliser la valeur de *marker* en fonction de *grade*.

```
mod1_trt <- lm(marker ~ grade, data = trial)
mod1_trt
```

Call:

```
lm(formula = marker ~ grade, data = trial)
```

Coefficients:

```
(Intercept)      gradeII      gradeIII
      1.0669      -0.3864      -0.0711
```

Le modèle obtenu contient trois coefficients ou termes : un intercept et deux termes associés à la variable *grade*.

Pour bien interpréter ces coefficients, il faut comprendre comment la variable *grade* a été transformée avant d'être incluse dans le modèle. Nous pouvons voir cela avec la fonction `contrasts()`.

```
contrasts(trial$grade)
```

```
      II III
I       0   0
II      1   0
III     0   1
```

Ce que nous montre cette matrice, c'est que la variable catégorielle *grade* à 3 modalités a été transformée en 2 variables binaires que l'on retrouve sous les noms de *gradeII* et *gradeIII* dans

le modèle : *gradeII* vaut 1 si *grade* est égal à *II* et 0 sinon; *gradeIII* vaut 1 si *grade* est égal à *III* et 0 sinon. Si *grade* est égal à *I*, alors *gradeII* et *gradeIII* valent 0.

Il s'agit ici d'un contraste dit de traitement ou la première modalité joue ici le rôle de **modalité de référence**.

Dans ce modèle linéaire, la valeur de l'intercept correspond à la moyenne de *marker* lorsque nous nous trouvons à la référence, donc quand *grade* est égal à *I* dans cet exemple. Et nous pouvons le constater dans notre tableau précédent des moyennes, 1.0669 correspond bien à la moyenne de *marker* pour la modalité *I*.

La valeur du coefficient associé à *markerII* correspond à l'écart par rapport à la référence lorsque *marker* est égal à *II*. Autrement dit, la moyenne de *marker* pour la modalité *II* correspond à la somme de l'intercept et du coefficient *markerII*. Et nous retrouvons bien la relation suivante : $0.6805 = 1.0669 + -0.3864$. De même, la moyenne de *marker* lorsque *grade* vaut *III* est égale à la somme de l'intercept et du terme *markerIII*.

Lorsqu'on utilise des contrastes de type traitement, chaque terme du modèle peut être associé à une et une seule modalité d'origine de la variable catégorielle. Dès lors, il est possible de rajouter la modalité de référence lorsque l'on présente les résultats et on peut même lui associer la valeurs 0, ce qui peut être fait avec `gtsummary::tbl_regression()` avec l'option `add_estimate_to_reference_rows = TRUE`.

```
mod1_trt |>
  tbl_regression(
    intercept = TRUE,
    add_estimate_to_reference_rows = TRUE
  )
```

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

Characteristic	Beta	95% CI	p-value
(Intercept)	1.1	0.86, 1.3	<0.001
Grade			
I	0.00	—	
II	-0.39	-0.68, -0.09	0.010
III	-0.07	-0.37, 0.23	0.6

25.1.2 Exemple 2 : une régression logistique avec deux variables catégorielles

Pour ce deuxième exemple, nous allons utiliser le jeu de données *hdv2003* fourni par l'extension `{questionr}` et recoder la variable *age* en groupes d'âges à 4 modalités.

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

library(tidyverse)
```

```
hdv2003 <- hdv2003 |>
  mutate(
    groupe_ages = cut(
      age,
      c(16, 25, 45, 65, 99),
      right = FALSE,
      include.lowest = TRUE
    ) |>
    fct_recode(
      "16-24" = "[16,25)",
      "25-44" = "[25,45)",
      "45-64" = "[45,65)",
      "65+" = "[65,99]"
    )
  ) |>
  labelled::set_variable_labels(
    groupe_ages = "Groupe d'âges",
    sexe = "Sexe"
  )
```

Nous allons faire une régression logistique binaire pour investiguer l'effet du *sexe* (variable à 2 modalités) et du *groupe d'âges* (variable à 4 modalités) sur la pratique du *sport*.

```
mod2_trt <- glm(
  sport ~ sexe + groupe_ages,
  family = binomial,
  data = hdv2003
)
mod2_trt
```



```
Call: glm(formula = sport ~ sexe + groupe_ages, family = binomial,
          data = hdv2003)
```

Coefficients:

(Intercept)	sexeFemme	groupe_ages25-44	groupe_ages45-64
0.9021	-0.4455	-0.6845	-1.6535
groupe_ages65+			
-2.3198			

Degrees of Freedom: 1999 Total (i.e. Null); 1995 Residual

Null Deviance: 2617

Residual Deviance: 2385 AIC: 2395

Le modèle contient 5 termes : 1 intercept, 1 coefficient pour la variable *sexe* et 3 coefficients pour la variable *groupe_ages*. Comme précédemment, nous pouvons constater que les variables à n modalités sont remplacées par défaut (contrastes de type traitement) par $n-1$ variables binaires, la première modalité jouant à chaque fois le rôle de modalité de référence.

```
contrasts(hdv2003$sexe)
```

	Femme
Homme	0
Femme	1

```
contrasts(hdv2003$groupe_ages)
```

	25-44	45-64	65+
16-24	0	0	0
25-44	1	0	0
45-64	0	1	0
65+	0	0	1

L'intercept correspond donc à la situation à la référence, c'est-à-dire à la prédiction du modèle pour les hommes (référence de *sexe*) âgés de 16 à 24 ans (référence de *groupe_ages*).

Il est possible d'exprimer cela en termes de probabilité en utilisant l'inverse de la fonction *logit* (puisque nous avons utilisé un modèle *logit*).

```
inv_logit <- binomial("logit")$linkinv
inv_logit(0.9021)
```

```
[1] 0.7113809
```

Selon le modèle, les hommes âgés de 16 à 24 ans ont donc 71% de chance de pratiquer du sport.

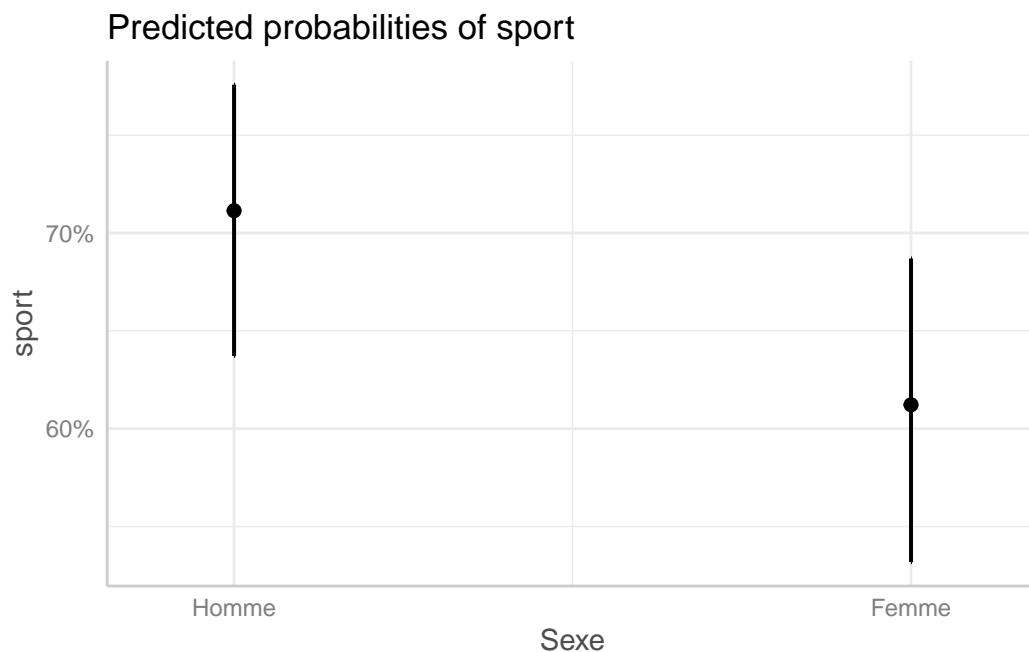
Regardons maintenant le coefficient associé à *sexeFemme* (-0.4455) : il représente (pour la modalité de référence des autres variables, soit pour les 16-24 ans ici) la correction à appliquer à l'intercept pour obtenir la probabilité de faire du sport. Il s'agit donc de la différence entre les femmes et les hommes pour le groupe des 16-24 ans.

```
inv_logit(0.9021 - 0.4455)
```

```
[1] 0.6122073
```

Autrement dit, selon le modèle, la probabilité de faire du sport pour une femme âgée de 16 à 24 ans est de 61%. On peut représenter cela avec la fonction `ggeffects::ggpredict()` de `{ggeffects}`, qui représente les prédictions d'une variable toutes les autres variables étant à la référence.

```
library(ggeffects)
ggpredict(mod2_trt, "sexe") |> plot()
```



Bien souvent, pour une régression logistique, on préfère représenter les exponentielles des coefficients qui correspondent à des *odds ratios*.

```
mod2_trt |>
  tbl_regression(
    exponentiate = TRUE,
    intercept = TRUE,
    add_estimate_to_reference_rows = TRUE
  ) |>
  bold_labels()
```

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

Characteristic	OR	95% CI	p-value
(Intercept)	2.46	1.76, 3.48	<0.001
Sexe			
Homme	1.00	—	
Femme	0.64	0.53, 0.78	<0.001
Groupe d'âges			
16-24	1.00	—	
25-44	0.50	0.35, 0.71	<0.001
45-64	0.19	0.13, 0.27	<0.001
65+	0.10	0.06, 0.15	<0.001

Or, 0,64 correspond bien à l'*odds ratio* entre 61% et 71% (que l'on peut calculer avec `questionr::odds.ratio()`).

```
questionr::odds.ratio(0.6122, 0.7114)
```

```
[1] 0.6404246
```

De la même manière, les différents coefficients associés à *groupe_ages* correspondent à la différence entre chaque groupe d'âges et sa modalité de référence (ici 16-24 ans), quand les autres variables (ici le *sexe*) sont à leur référence (ici les hommes).

Pour prédire la probabilité de faire du sport pour un profil particulier, il faut prendre en compte toutes les termes qui s'appliquent et qui s'ajoutent à l'intercept. Par exemple, pour une femme de 50 ans il faut considérer l'intercept (0.9021), le coefficient *sexeFemme* (-0.4455) et le coefficient *groupe_ages45-64* (-1.6535). Sa probabilité de faire du sport est donc de 23%.

```
inv_logit(0.9021 - 0.4455 - 1.6535)
```

```
[1] 0.2320271
```

25.1.3 Changer la modalité de référence

Il est possible de personnaliser les contrastes à utiliser et avoir un recours à un contraste de type traitement mais en utilisant une autre modalité que la première comme référence, avec la fonction `contr.treatment()`. Le premier argument de la fonction correspond au nombre de modalités de la variable et le paramètre `base` permet de spécifier la modalité de référence (1 par défaut).

```
contr.treatment(4, base = 2)
```

```
  1 3 4
1 1 0 0
2 0 0 0
3 0 1 0
4 0 0 1
```

`contr.SAS()` permet de spécifier un contraste de type traitement dont la modalité de référence est la dernière.

```
contr.SAS(4)
```

```
  1 2 3
1 1 0 0
2 0 1 0
3 0 0 1
4 0 0 0
```

Les contrastes peuvent être modifiés de deux manières : au moment de la construction du modèle (via l'option `contrasts`) ou comme attribut des variables (via la fonction `contrasts()`).

```
contrasts(hdv2003$sexe) <- contr.SAS(2)
mod2_trt_bis <- glm(
  sport ~ sexe + groupe_ages,
  family = binomial,
  data = hdv2003,
```

```

  contrasts = list(groupe_ages = contr.treatment(4, 3))
)
mod2_trt_bis |>
  tbl_regression(exponentiate = TRUE, intercept = TRUE) |>
  bold_labels()

```

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

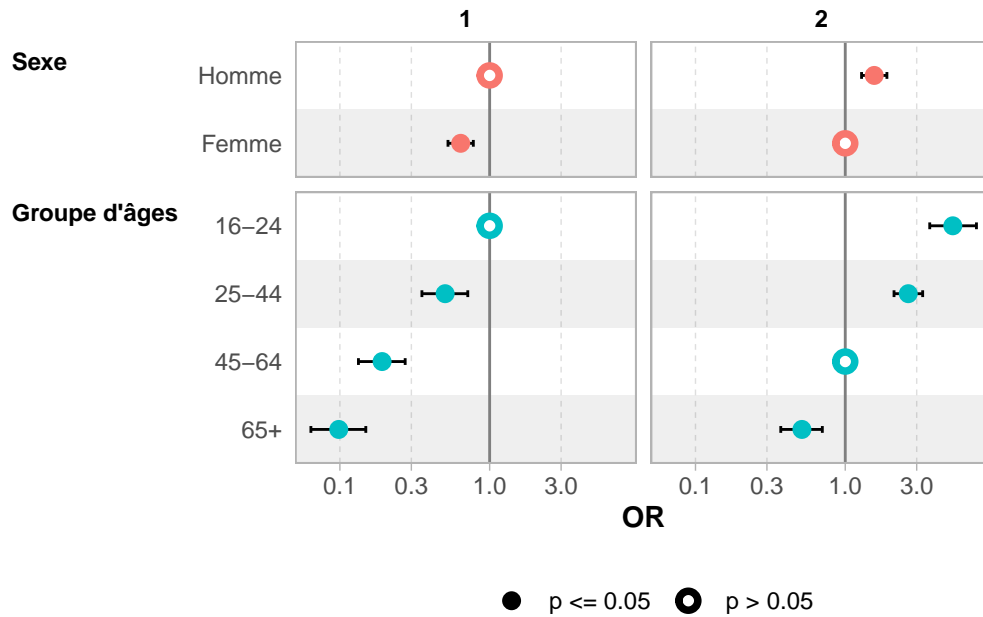
Characteristic	OR	95% CI	p-value
(Intercept)	0.30	0.25, 0.36	<0.001
Sexe			
Homme	1.56	1.29, 1.90	<0.001
Femme	—	—	
Groupe d'âges			
16-24	5.23	3.67, 7.52	<0.001
25-44	2.64	2.12, 3.29	<0.001
45-64	—	—	
65+	0.51	0.37, 0.70	<0.001

Comme les modalités de référence ont changé, l'intercept et les différents termes ont également changé (puisque l'on ne compare plus à la même référence).

```

ggstats::ggcoef_compare(
  list(mod2_trt, mod2_trt_bis),
  exponentiate = TRUE,
  type = "faceted"
)

```



Cependant, du point de vue explicatif et prédictif, les deux modèles sont rigoureusement identiques.

```
anova(mod2_trt, mod2_trt_bis, test = "Chisq")
```

Analysis of Deviance Table

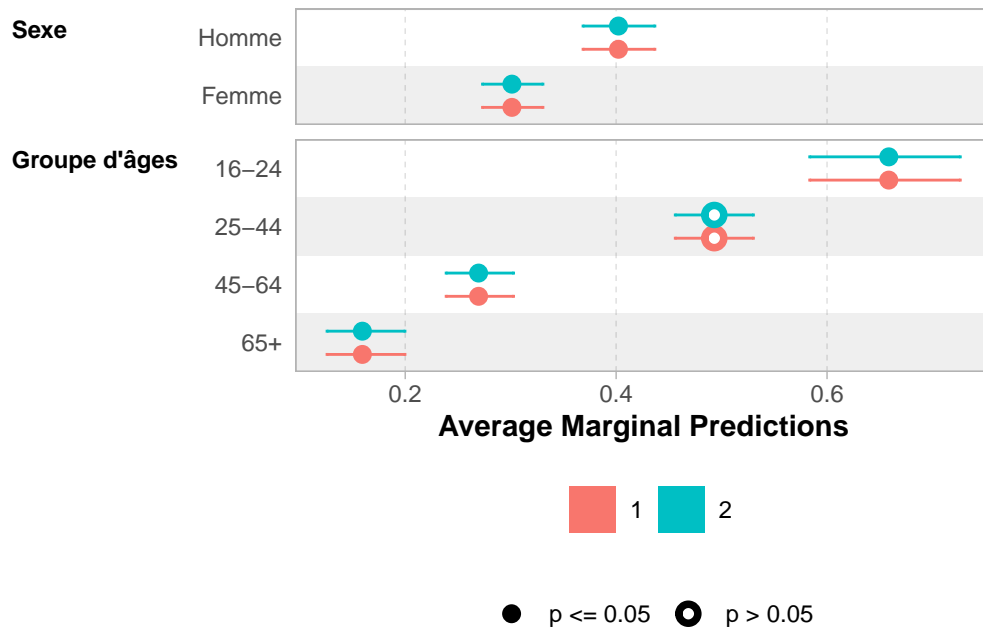
Model 1: sport ~ sexe + groupe_ages

Model 2: sport ~ sexe + groupe_ages

	Resid. Df	Resid. Dev	Df	Deviance	Pr(>Chi)
1	1995	2385.2			
2	1995	2385.2	0	0	

De même, leurs prédictions marginales (cf. Chapitre 24) sont identiques.

```
ggstats::ggcoef_compare(
  list(mod2_trt, mod2_trt_bis),
  tidy_fun = broom.helpers::tidy_marginal_predictions,
  type = "dodge",
  vline = FALSE
)
```



25.2 Contrastes de type somme

Nous l'avons vu, les contrastes de type traitement nécessitent de définir une modalité de référence et toutes les autres modalités seront comparées à cette modalité de référence. Une alternative consiste à comparer toutes les modalités à la grande moyenne, ce qui s'obtient avec un contraste de type somme que l'on obtient avec `contr.sum()`.

25.2.1 Exemple 1 : un modèle linéaire avec une variable catégorielle

Reprenons notre premier exemple de tout à l'heure et modifions seulement le contraste.

```
contrasts(trial$grade) <- contr.sum
mod1_sum <- lm(
  marker ~ grade,
  data = trial
)
mod1_sum
```

Call:

```
lm(formula = marker ~ grade, data = trial)
```

Coefficients:

(Intercept)	grade1	grade2
0.9144	0.1525	-0.2339

L'*intercept* correspond à ce qu'on appelle parfois la grande moyenne (ou *great average* en anglais). Il ne s'agit pas de la moyenne observée de *marker* mais de la moyenne des moyennes de chaque sous-groupe. Cela va constituer la situation de référence de notre modèle, en quelque sorte indépendante des effets de la variable *grade*.

```
mean(trial$marker, na.rm = TRUE)
```

```
[1] 0.9159895
```

```
moy_groupe <-  
  trial |>  
  dplyr::group_by(grade) |>  
  dplyr::summarise(moyenne_marker = mean(marker, na.rm = TRUE))  
moy_groupe
```

```
# A tibble: 3 x 2  
  grade moyenne_marker  
  <fct>          <dbl>  
1 I             1.07  
2 II            0.681  
3 III           0.996
```

```
mean(moy_groupe$moyenne_marker)
```

```
[1] 0.9144384
```

Le terme *grade1* correspond quant à lui au modificateur associé à la première modalité de la variable *grade* à savoir *I*. C'est l'écart, pour cette modalité, à la grande moyenne : $1.0669 - 0.9144 = 0.1525$.

De même, le terme *grade2* correspond à l'écart pour la modalité *II* par rapport à la grande moyenne : $0.6805 - 0.9144 = -0.2339$.

Qu'en est-il de l'écart à la grande moyenne pour la modalité *III* ? Pour cela, voyons tout d'abord comment la variable *grade* a été codée :


```
contrasts(trial$grade)
```

```
      [,1] [,2]  
I         1    0  
II        0    1  
III       -1   -1
```

Comme précédemment, cette variable à trois modalités a été codée avec deux termes. Les deux premiers termes correspondent aux écarts à la grande moyenne des deux premières modalités. La troisième modalité est, quant à elle, codée systématiquement -1 . C'est ce qui assure que la somme des contributions soit nulle et donc que l'*intercept* capture la grande moyenne.

L'écart à la grande moyenne pour la troisième modalité s'obtient donc en faisant la somme des autres termes et en l'inversant : $(0.1525 - 0.2339) * -1 = 0.0814 = 0.9958 - 0.9144$.

On peut calculer / afficher la valeur associée à la dernière modalité en précisant `add_estimate_to_reference_rows = TRUE` lorsque l'on appelle `gtsummary::tbl_regression()`.

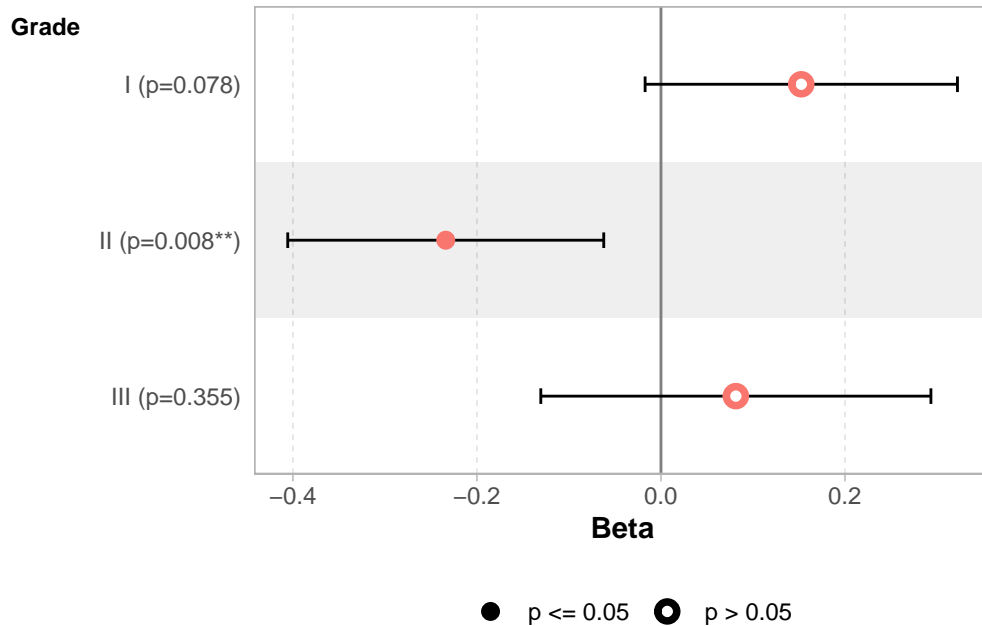
```
mod1_sum |>  
  tbl_regression(  
    intercept = TRUE,  
    add_estimate_to_reference_rows = TRUE  
  ) |>  
  bold_labels()
```

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

Characteristic	Beta	95% CI	p-value
(Intercept)	0.91	0.79, 1.0	<0.001
Grade			
I	0.15	-0.02, 0.32	0.078
II	-0.23	-0.41, -0.06	0.008
III	0.08	-0.13, 0.29	0.4

De même, cette valeur est correctement affichée par `ggstats::ggcoef_model()`.

```
ggstats::ggcoef_model(mod1_sum)
```



Le fait d'utiliser des contrastes de type traitement ou somme n'a aucun impact sur la valeur prédictive du modèle. La quantité de variance expliquée, la somme des résidus ou encore l'AIC sont identiques. En un sens, il s'agit du même modèle. C'est seulement la manière d'interpréter les coefficients du modèle qui change.

```
anova(mod1_trt, mod1_sum, test = "Chisq")
```

Analysis of Variance Table

Model 1: marker ~ grade

Model 2: marker ~ grade

	Res.Df	RSS	Df	Sum of Sq	Pr(>Chi)
1	187	134.17			
2	187	134.17	0	0	

25.2.2 Exemple 2 : une régression logistique avec deux variables catégorielles

Reprenons notre second exemple et codons les variables catégorielles avec un traitement de type somme.

```

mod2_sum <- glm(
  sport ~ sexe + groupe_ages,
  family = binomial,
  data = hdv2003,
  contrasts = list(sexe = contr.sum, groupe_ages = contr.sum)
)
mod2_sum |>
  tbl_regression(
    exponentiate = TRUE,
    intercept = TRUE,
    add_estimate_to_reference_rows = TRUE
  ) |>
  bold_labels()

```

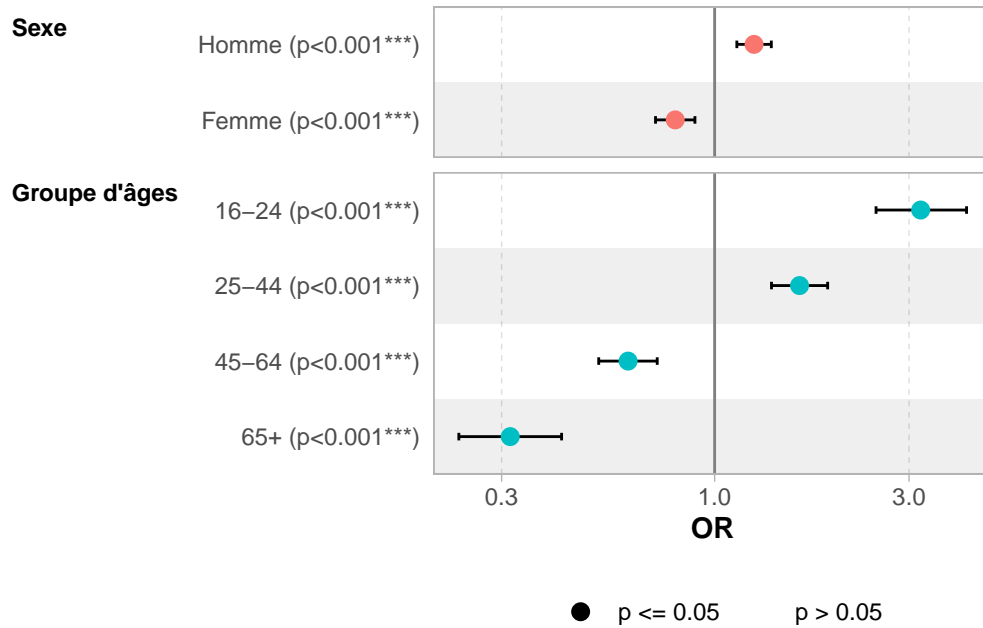
Table printed with `knitr::kable()`, not {gt}. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
 To suppress this message, include `message = FALSE` in code chunk header.

Characteristic	OR	95% CI	p-value
(Intercept)	0.62	0.55, 0.69	<0.001
Sexe			
Homme	1.25	1.13, 1.38	<0.001
Femme	0.80	0.72, 0.89	<0.001
Groupe d'âges			
16-24	3.20	2.49, 4.15	<0.001
25-44	1.62	1.38, 1.89	<0.001
45-64	0.61	0.52, 0.72	<0.001
65+	0.31	0.24, 0.42	<0.001

```

ggstats::ggcoef_model(mod2_sum, exponentiate = TRUE)

```



Cette fois-ci, l'*intercept* capture la situation à la grande moyenne à la fois du sexe et du groupe d'âges, et les coefficients s'interprètent donc comme modificateurs de chaque modalité par rapport à cette grande moyenne. En ce sens, les contrastes de type somme permettent donc de capturer l'effet de chaque modalité.

Du point de vue explicatif et prédictif, le fait d'avoir recours à des contrastes de type somme ou traitement n'a aucun impact : les deux modèles sont rigoureusement identiques. Il n'y a que la manière d'interpréter les coefficients qui change.

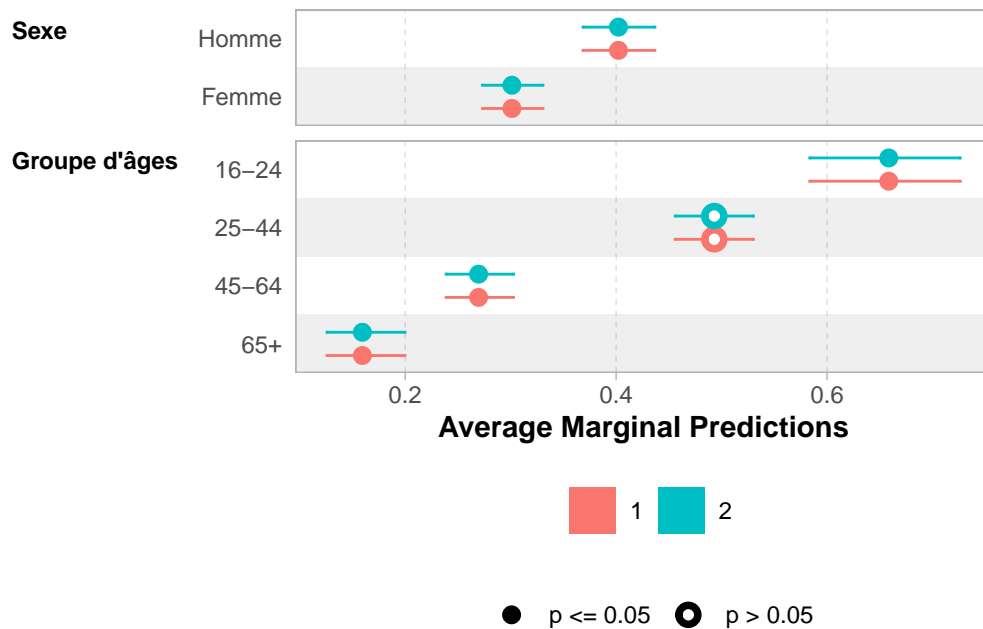
```
anova(mod2_trt, mod2_sum, test = "Chisq")
```

Analysis of Deviance Table

```
Model 1: sport ~ sexe + groupe_ages
Model 2: sport ~ sexe + groupe_ages
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1     1995     2385.2
2     1995     2385.2  0         0
```

Les prédictions marginales (cf. Chapitre 24) sont identiques.

```
ggstats::ggcoef_compare(
  list(mod2_trt, mod2_sum),
  tidy_fun = broom.helpers::tidy_marginal_predictions,
  type = "dodge",
  vline = FALSE
)
```



25.3 Contrastes par différences successives

Les contrastes par différences successives consistent à comparer la deuxième modalité à la première, puis la troisième modalité à la seconde, etc. Ils sont disponibles avec la fonction `MASS::contr.sdif()`.

Illustrons cela avec un exemple.

25.3.1 Exemple 1 : un modèle linéaire avec une variable catégorielle

```
mod1_sdif <- lm(
  marker ~ grade,
```

```

data = trial,
contrasts = list(grade = MASS::contr.sdif)
)
mod1_sdif |>
tbl_regression(intercept = TRUE) |>
bold_labels()

```

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

Characteristic	Beta	95% CI	p-value
(Intercept)	0.91	0.79, 1.0	<0.001
Grade			
II - I	-0.39	-0.68, -0.09	0.010
III - II	0.32	0.02, 0.62	0.040

En premier lieu, on notera que l'*intercept*, comme avec les contrastes de type somme, correspond ici à la grande moyenne.

```
mean(moy_groupe$moyenne_marker)
```

```
[1] 0.9144384
```

Cela est lié au fait que la somme des coefficients dans ce type de contrastes est égale à 0.

```
MASS::contr.sdif(3)
```

```

      2-1      3-2
1 -0.6666667 -0.3333333
2  0.3333333 -0.3333333
3  0.3333333  0.6666667

```

De plus, la matrice de contrastes est calculée de telle manière que l'écart entre les deux premières modalités vaut 1 pour le premier terme, et l'écart entre la seconde et la troisième modalité vaut également 1 pour le deuxième terme.

Ainsi, le terme `gradeII-I` correspond à la différence entre la moyenne du grade de niveau II et celle du niveau I¹.

```
moy_groupe$moyenne_marker[2] - moy_groupe$moyenne_marker[1]
```

```
[1] -0.3863997
```

Et le coefficient `gradeIII-II` à l'écart entre la moyenne du niveau III et celle du niveau II.

```
moy_groupe$moyenne_marker[3] - moy_groupe$moyenne_marker[2]
```

```
[1] 0.3152964
```

25.3.2 Exemple 2 : une régression logistique avec deux variables catégorielles

La même approche peut être appliquée à une régression logistique.

```
mod2_sdif <- glm(  
  sport ~ sexe + groupe_ages,  
  family = binomial,  
  data = hdv2003,  
  contrasts = list(  
    sexe = MASS::contr.sdif,  
    groupe_ages = MASS::contr.sdif  
  )  
)  
mod2_sdif |>  
  tbl_regression(  
    exponentiate = TRUE,  
    intercept = TRUE  
  ) |>  
  bold_labels()
```

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

¹On peut remarquer que la même valeur était obtenue avec un contraste de type traitement où toutes les modalités étaient comparées à la modalité de référence I.

Characteristic	OR	95% CI	p-value
(Intercept)	0.62	0.55, 0.69	<0.001
Sexe			
Femme / Homme	0.64	0.53, 0.78	<0.001
Groupe d'âges			
25-44 / 16-24	0.50	0.35, 0.71	<0.001
45-64 / 25-44	0.38	0.30, 0.47	<0.001
65+ / 45-64	0.51	0.37, 0.70	<0.001

On pourra noter que les *odds ratios* “femme/homme” et “25-44/16-24” obtenus ici sont équivalents à ceux que l’on avait obtenus précédemment avec des contrastes de types de traitement. Pour la modalité “45-64 ans” par contre, elle est ici comparée aux 25-44 ans, alors qu’avec un contraste de type traitement, toutes les comparaisons auraient eu lieu avec la même modalité de référence, à savoir les 16-24 ans.

Les contrastes par différences successives font donc plutôt sens lorsque les modalités sont ordonnées (d’où l’intérêt de comparer avec la modalité précédente), ce qui n’est pas forcément le cas lorsque les modalités ne sont pas ordonnées.

💡 Astuce

De manière générale, et quels que soient les contrastes utilisés pour le calcul du modèle, il est toujours possible de recalculer *a posteriori* les différences entre chaque combinaison de modalités deux à deux avec `emmeans::emmeans()`. Cela peut même se faire directement en passant l’argument `add_pairwise_contrasts = TRUE` à `tbl_regression()`.

```
mod2_trt |>
  tbl_regression(
    exponentiate = TRUE,
    add_pairwise_contrasts = TRUE
  ) |>
  bold_labels()
```

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

Characteristic	OR	95% CI	p-value
Sexe			
Femme / Homme	0.64	0.53, 0.78	<0.001

Groupe d'âges				
(25-44) / (16-24)	0.50	0.32, 0.80	<0.001	
(45-64) / (16-24)	0.19	0.12, 0.31	<0.001	
(45-64) / (25-44)	0.38	0.28, 0.51	<0.001	
(65+) / (16-24)	0.10	0.06, 0.17	<0.001	
(65+) / (25-44)	0.19	0.13, 0.29	<0.001	
(65+) / (45-64)	0.51	0.34, 0.78	<0.001	

25.4 Autres types de contrastes

25.4.1 Contrastes de type Helmert

Les contrastes de Helmert sont un peu plus complexes : ils visent à comparer la seconde modalité à la première, la troisième à la moyenne des deux premières, la quatrième à la moyenne des trois premières, etc.

Prenons un exemple avec une variable catégorielle à quatre modalités.

```
contrasts(trial$stage) <- contr.helmert
contrasts(trial$stage)
```

```
      [,1] [,2] [,3]
T1      -1   -1   -1
T2       1   -1   -1
T3       0    2   -1
T4       0    0    3
```

```
mod_helmert <- lm(
  marker ~ stage,
  data = trial
)
mod_helmert
```

Call:

```
lm(formula = marker ~ stage, data = trial)
```

Coefficients:

```
(Intercept)      stage1      stage2      stage3
      0.91661      0.19956      0.03294     -0.02085
```

Pour bien comprendre comment interpréter ces coefficients, calculons déjà la grande moyenne.

```
m <- trial |>
  dplyr::group_by(stage) |>
  dplyr::summarise(moy = mean(marker, na.rm = TRUE))
mean(m$moy)
```

```
[1] 0.9166073
```

On le voit, l'*intercept* (0.9166) capture ici cette grande moyenne, à savoir la moyenne des moyennes de chaque sous-groupe.

Maintenant, pour interpréter les coefficients, regardons comment évolue la moyenne à chaque fois que l'on ajoute une modalité. La fonction `dplyr::cummean()` nous permet de calculer la moyenne cumulée, c'est-à-dire la moyenne de la valeur actuelle et des valeurs des lignes précédentes. Avec `dplyr::lag()` nous pouvons obtenir la moyenne cumulée de la ligne précédente. Il nous est alors possible de calculer l'écart entre les deux, et donc de voir comment la moyenne a changé avec l'ajout d'une modalité.

```
m <- m |>
  dplyr::mutate(
    moy_cum = dplyr::cummean(moy),
    moy_cum_prec = dplyr::lag(moy_cum),
    ecart = moy_cum - moy_cum_prec
  )
m
```

```
# A tibble: 4 x 5
  stage   moy moy_cum moy_cum_prec   ecart
  <fct> <dbl>   <dbl>       <dbl>   <dbl>
1 T1    0.705   0.705         NA      NA
2 T2    1.10    0.905       0.705   0.200
3 T3    1.00    0.937       0.905   0.0329
4 T4    0.854    0.917       0.937  -0.0208
```

On le voit, les valeurs de la colonne *ecart* correspondent aux coefficients du modèle.

Le premier terme *stage1* compare la deuxième modalité (*T2*) à la première (*T1*) et indique l'écart entre la moyenne des moyennes de *T1* et *T2* et la moyenne de *T1*.

Le second terme *stage2* compare la troisième modalité (*T3*) aux deux premières (*T1* et *T2*) et indique l'écart entre la moyenne des moyennes de *T1*, *T2* et *T3* par rapport à la moyenne des moyennes de *T1* et *T2*.

Le troisième terme *stage3* compare la quatrième modalité (*T4*) aux trois premières (*T1*, *T2* et *T3*) et indique l'écart entre la moyenne des moyennes de *T1*, *T2*, *T3* et *T4* par rapport à la moyenne des moyennes de *T1*, *T2* et *T3*.

Les contrastes de Helmert sont ainsi un peu plus complexes à interpréter et à réserver à des cas particuliers où ils prennent tout leur sens.

25.4.2 Contrastes polynomiaux

Les contrastes polynomiaux, définis avec `contr.poly()`, sont utilisés par défaut pour les variables catégorielles ordonnées. Ils permettent de décomposer les effets selon une composante linéaire, une composante quadratique, une composante cubique, voire des composantes de degrés supérieurs.

```
contrasts(trial$stage) <- contr.poly
contrasts(trial$stage)
```

	.L	.Q	.C
T1	-0.6708204	0.5	-0.2236068
T2	-0.2236068	-0.5	0.6708204
T3	0.2236068	-0.5	-0.6708204
T4	0.6708204	0.5	0.2236068

```
mod_poly <- lm(
  marker ~ stage,
  data = trial
)
mod_poly
```

Call:

```
lm(formula = marker ~ stage, data = trial)
```

Coefficients:

(Intercept)	stage.L	stage.Q	stage.C
0.91661	0.07749	-0.27419	0.10092

Ici aussi, l'*intercept* correspond à la grande moyenne des moyennes. Il est par contre plus difficile de donner un sens interprétatif / sociologique aux différents coefficients.

25.5 Lectures additionnelles

- *A (sort of) Complete Guide to Contrasts in R* par Rose Maier
- *An introductory explanation of contrast coding in R linear models* par Athanassios Protopapas
- *Understanding Sum Contrasts for Regression Models: A Demonstration* par Mona Zhu

26 Interactions

Dans un modèle statistique classique, on fait l'hypothèse implicite que chaque variable explicative est indépendante des autres. Cependant, cela ne se vérifie pas toujours. Par exemple, l'effet de l'âge peut varier en fonction du sexe.

Nous pourrions dès lors ajouter à notre modèle des **interactions** entre variables.

26.1 Données d'illustration

Reprenons le modèle que nous avons utilisé dans le chapitre sur la régression logistique binaire (cf. Chapitre 22).

```
library(tidyverse)
library(labelled)

data(hdv2003, package = "questionr")

d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
```

```

    "Secondaire" = "2eme cycle",
    "Technique / Professionnel" = "Enseignement technique ou professionnel court",
    "Technique / Professionnel" = "Enseignement technique ou professionnel long",
    "Supérieur" = "Enseignement superieur y compris technique superieur"
  ) |>
  fct_na_value_to_level("Non documenté")
) |>
set_variable_labels(
  sport = "Pratique un sport ?",
  sexe = "Sexe",
  groupe_ages = "Groupe d'âges",
  etudes = "Niveau d'études",
  heures.tv = "Heures de télévision / jour"
)

```

26.2 Modèle sans interaction

Nous avons alors exploré les facteurs associés au fait de pratiquer du sport.

```

mod <- glm(
  sport ~ sexe + groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
library(gtsummary)
theme_gtsummary_language(
  "fr",
  decimal.mark = ",",
  big.mark = " "
)

```

```

mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels()

```

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

Table 26.1: Odds Ratios du modèle logistique simple

Caractéristique	OR	95% IC	p-valeur
Sexe			
Femme	—	—	
Homme	1,52	1,24 – 1,87	<0,001
Groupe d'âges			
18-24 ans	—	—	
25-44 ans	0,68	0,43 – 1,06	0,084
45-64 ans	0,36	0,23 – 0,57	<0,001
65 ans et plus	0,27	0,16 – 0,46	<0,001
Niveau d'études			
Primaire	—	—	
Secondaire	2,54	1,73 – 3,75	<0,001
Technique / Professionnel	2,81	1,95 – 4,10	<0,001
Supérieur	6,55	4,50 – 9,66	<0,001
Non documenté	8,54	4,51 – 16,5	<0,001
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001

Selon les résultats de notre modèle, les hommes pratiquent plus un sport que les femmes et la pratique du sport diminue avec l'âge.

Dans le chapitre sur les estimations marginales, cf. Chapitre 24, nous avons présenté la fonction `broom.helpers::plot_marginal_predictions()` qui permet de représenter les prédictions marginales moyennes du modèle.

```
mod |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
  patchwork::wrap_plots() &
  scale_y_continuous(
    limits = c(0, .8),
    labels = scales::label_percent()
  )
```

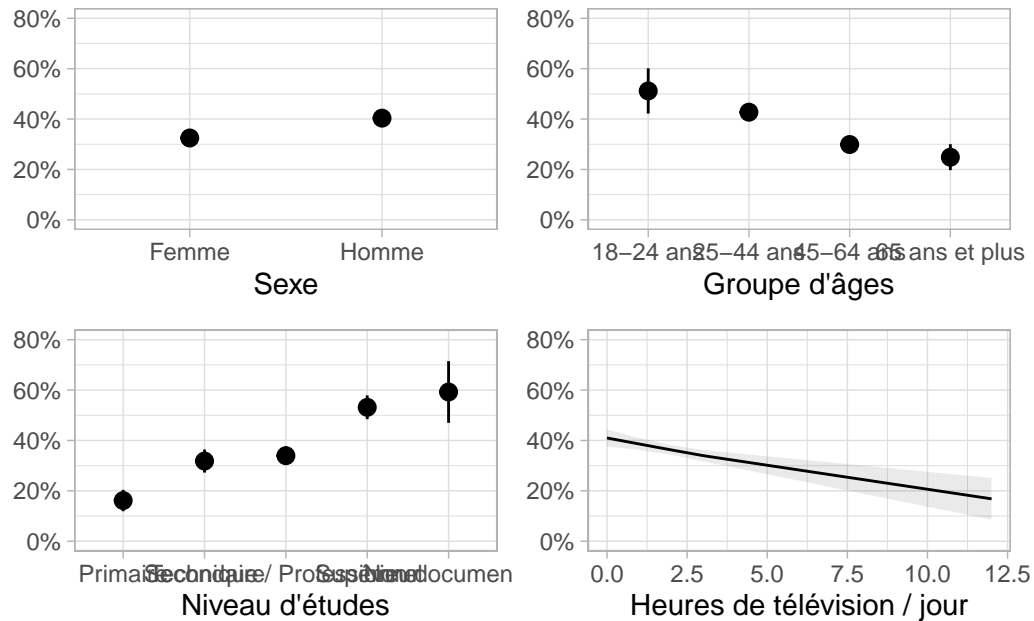


Figure 26.1: Prédictions marginales moyennes du modèle simple

26.3 Définition d'une interaction

Cependant, l'effet de l'âge est-il le même selon le sexe ? Nous allons donc introduire une interaction entre l'âge et le sexe dans notre modèle, ce qui sera représenté par `sexe * groupe_ages` dans l'équation du modèle.

```
mod2 <- glm(
  sport ~ sexe * groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
```

Commençons par regarder les prédictions marginales du modèle avec interaction.

```
mod2 |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
  patchwork::wrap_plots(ncol = 1) &
  scale_y_continuous(
    labels = scales::label_percent()
  )
```

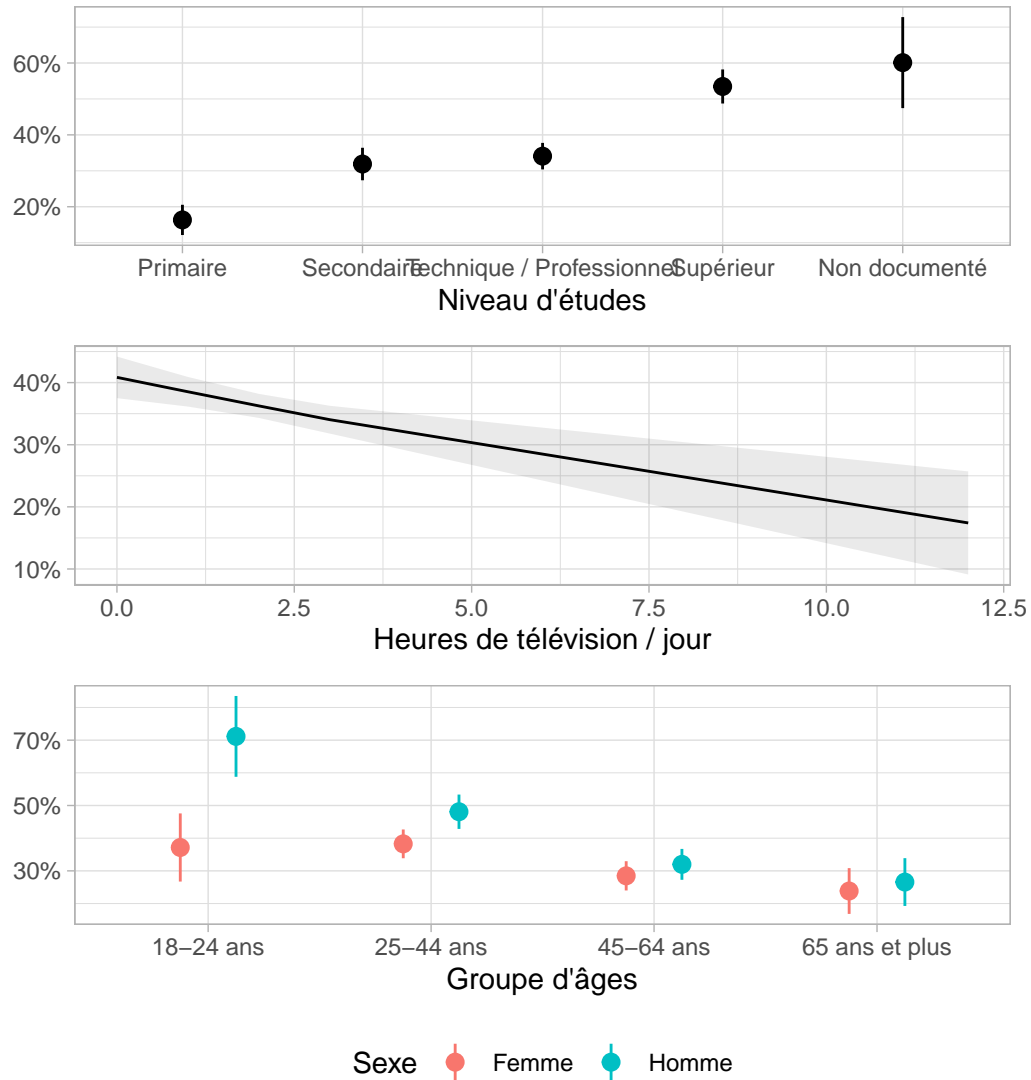



Figure 26.2: Prédictions marginales moyennes du modèle avec interaction

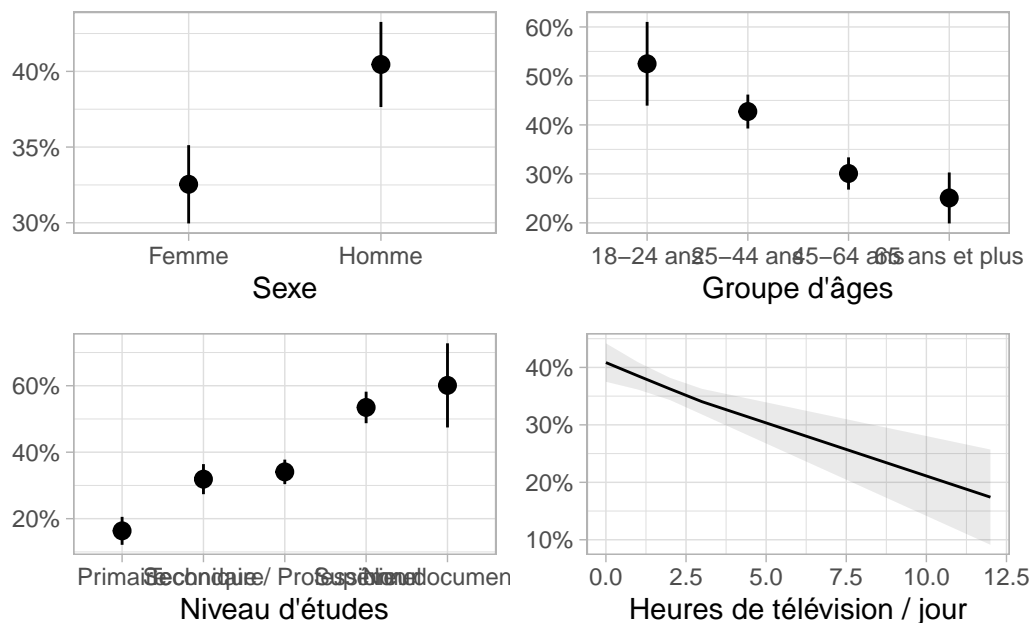
Sur ce graphique, on voit que la pratique d'un sport diminue fortement avec l'âge chez les hommes, tandis que cette diminution est bien plus modérée chez les femmes.

💡 Astuce

Par défaut, `broom.helpers::plot_marginal_predictions()` détecte la présence d'interactions dans le modèle et calcule les prédictions marginales pour chaque combi-

raison de variables incluent dans une interaction. Il reste possible de calculer des prédictions marginales individuellement pour chaque variable du modèle. Pour cela, il suffit d'indiquer `variables_list = "no_interaction"`.

```
mod2 |>
  broom.helpers::plot_marginal_predictions(
    variables_list = "no_interaction",
    type = "response"
  ) |>
  patchwork::wrap_plots() &
  scale_y_continuous(
    labels = scales::label_percent()
  )
```



26.4 Significativité de l'interaction

L'ajout d'une interaction au modèle augmente la capacité prédictive du modèle mais, dans le même temps, augmente le nombre de coefficients (et donc de degrés de liberté). La question se pose donc de savoir si l'ajout d'un terme d'interaction améliore notre modèle.

En premier lieu, nous pouvons comparer les AIC des modèles avec et sans interaction.

```
AIC(mod)
```

```
[1] 2230.404
```

```
AIC(mod2)
```

```
[1] 2223.382
```

L'AIC du modèle avec interaction est plus faible que celui sans interaction, nous indiquant un gain : notre modèle avec interaction est donc meilleur.

On peut tester avec `car::Anova()` si l'interaction est statistiquement significative¹.

```
car::Anova(mod2, type = "III")
```

Analysis of Deviance Table (Type III tests)

Response: sport

	LR	Chisq	Df	Pr(>Chisq)
sexe	19.349	1	1.089e-05	***
groupe_ages	15.125	3	0.0017131	**
etudes	125.575	4	< 2.2e-16	***
heures.tv	12.847	1	0.0003381	***
sexe:groupe_ages	13.023	3	0.0045881	**

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

La p-valeur associée au terme d'interaction (`sexe:groupe_ages`) est inférieure à 1% : l'interaction a donc bien un effet significatif.

Nous pouvons également utiliser `gtsummary::add_global_p()`.

```
mod2 |>
  tbl_regression(exponentiate = TRUE) |>
  add_global_p() |>
  bold_labels()
```

¹Lorsqu'il y a une interaction, il est préférable d'utiliser le type III, cf. Section 22.8. En toute rigueur, il serait préférable de coder nos variables catégorielles avec un contraste de type somme (cf. Chapitre 25). En pratique, nous pouvons nous en passer ici.

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

Table 26.2: Odds Ratios du modèle logistique avec interaction

Caractéristique	OR	95% IC	p-valeur
Sexe			<0,001
Femme	—	—	
Homme	5,10	2,41 – 11,4	
Groupe d'âges			0,002
18-24 ans	—	—	
25-44 ans	1,06	0,61 – 1,85	
45-64 ans	0,64	0,36 – 1,15	
65 ans et plus	0,49	0,25 – 0,97	
Niveau d'études			<0,001
Primaire	—	—	
Secondaire	2,55	1,74 – 3,78	
Technique / Professionnel	2,84	1,97 – 4,14	
Supérieur	6,69	4,60 – 9,89	
Non documenté	8,94	4,64 – 17,6	
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001
Sexe * Groupe d'âges			0,005
Homme * 25-44 ans	0,31	0,13 – 0,70	
Homme * 45-64 ans	0,24	0,10 – 0,54	
Homme * 65 ans et plus	0,23	0,09 – 0,60	

26.5 Interprétation des coefficients

Jetons maintenant un œil aux coefficients du modèle. Pour rendre les choses plus visuelles, nous aurons recours à `ggstats::ggcoef_model()`.

```
mod2 |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

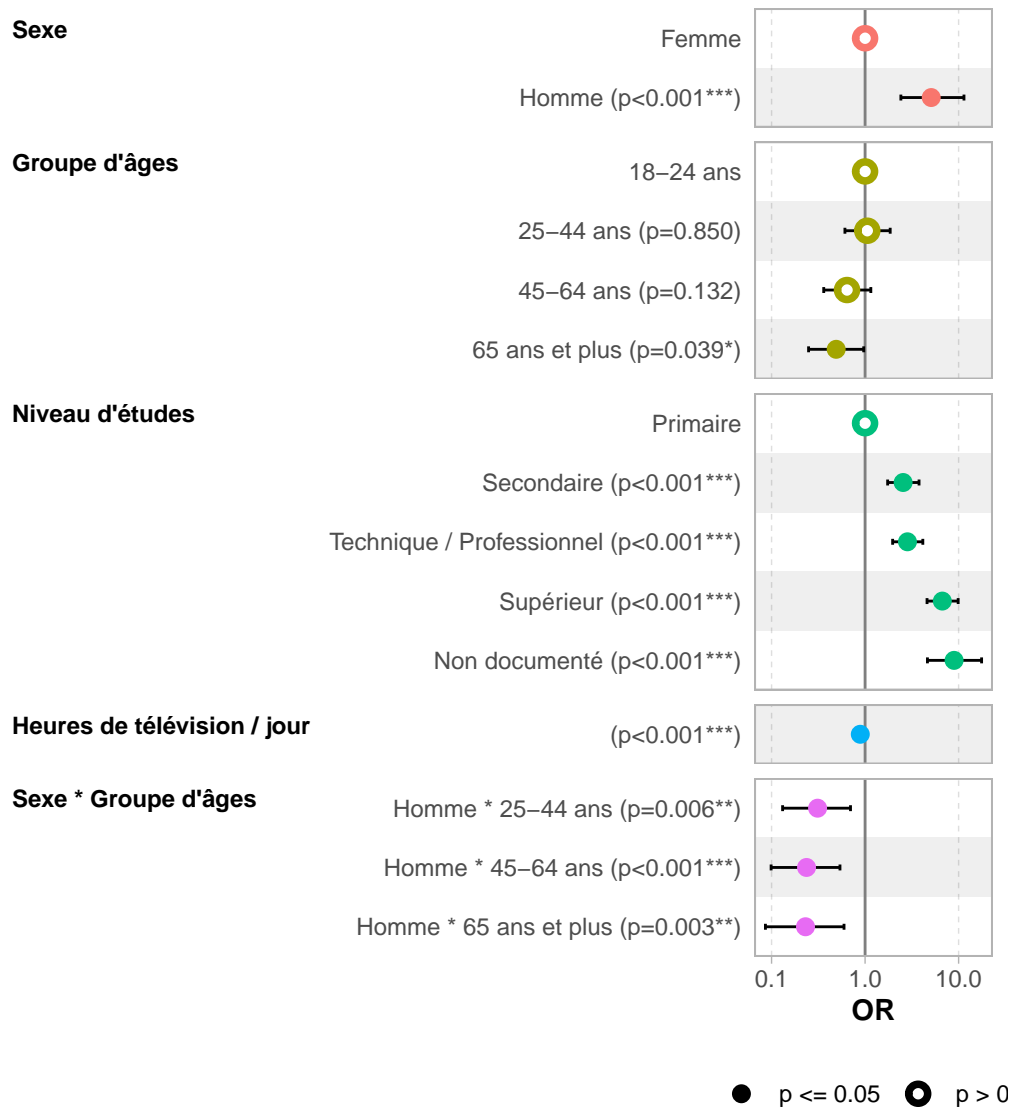


Figure 26.3: Coefficients (odds ratio) du modèle avec interaction

Concernant les variables *sexe* et *groupe_ages*, nous avons trois séries de coefficients : une série pour le sexe, une pour le groupe d'âges et enfin des coefficients pour l'interaction entre le sexe et le groupe d'âges.

Pour bien interpréter ces coefficients, il faut toujours avoir en tête les modalités choisies comme référence pour chaque variable.

Supposons une femme de 60 ans, dont toutes les autres variables correspondent aux modalités

de référence (i.e. de niveau primaire, qui ne regarde pas la télévision). Regardons ce que prédit le modèle quant à sa probabilité de faire du sport au travers d'une représentation graphique, grâce au package `{breakDown}`.

```
library(breakDown)
logit <- function(x) exp(x)/(1+exp(x))
nouvelle_observation <- d[1, ]
nouvelle_observation$sexe[1] = "Femme"
nouvelle_observation$groupe_ages[1] = "45-64 ans"
nouvelle_observation$etud[1] = "Primaire"
nouvelle_observation$heures.tv[1] = 0
plot(
  broken(mod2, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1) +
  ylab("Probabilité de faire du sport")
```

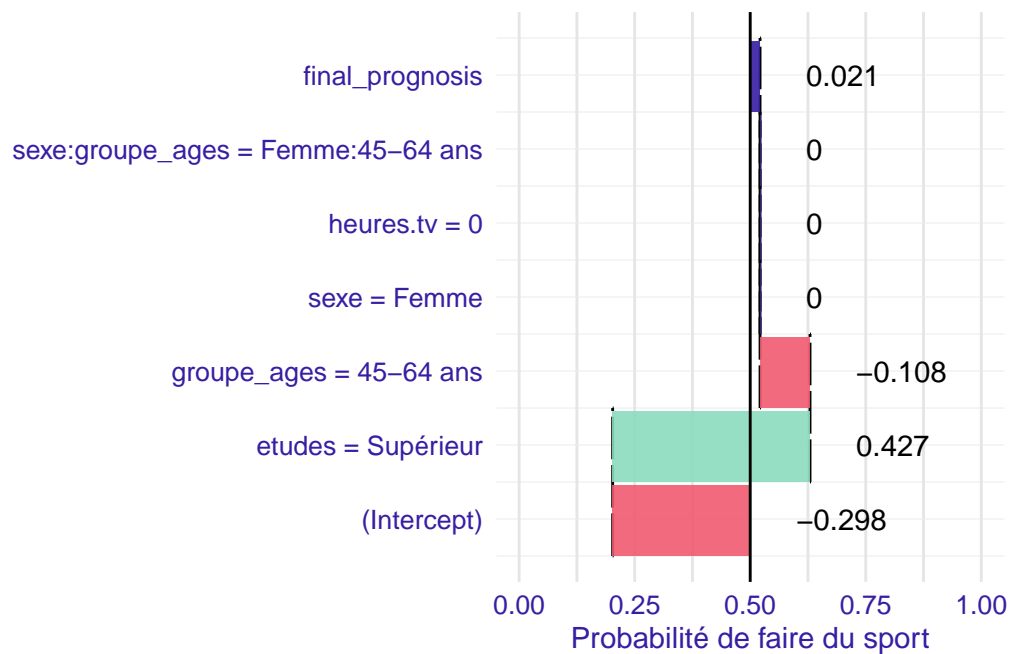


Figure 26.4: Représentation graphique de l'estimation de la probabilité de faire du sport pour une femme de 60 ans

En premier lieu, l'*intercept* s'applique et permet de déterminer la probabilité de base de faire du sport à la référence. Femme étant la modalité de référence pour la variable *sexe*, cela ne

modifie pas le calcul de la probabilité de faire du sport. Par contre, il y a une modification induite par la modalité 45-64 ans de la variable *groupe_ages*.

Regardons maintenant la situation d'un homme de 20 ans.

```
nouvelle_observation$sexe[1] = "Homme"
nouvelle_observation$groupe_ages[1] = "18-24 ans"
plot(
  broken(mod2, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1.2) +
  ylab("Probabilité de faire du sport")
```

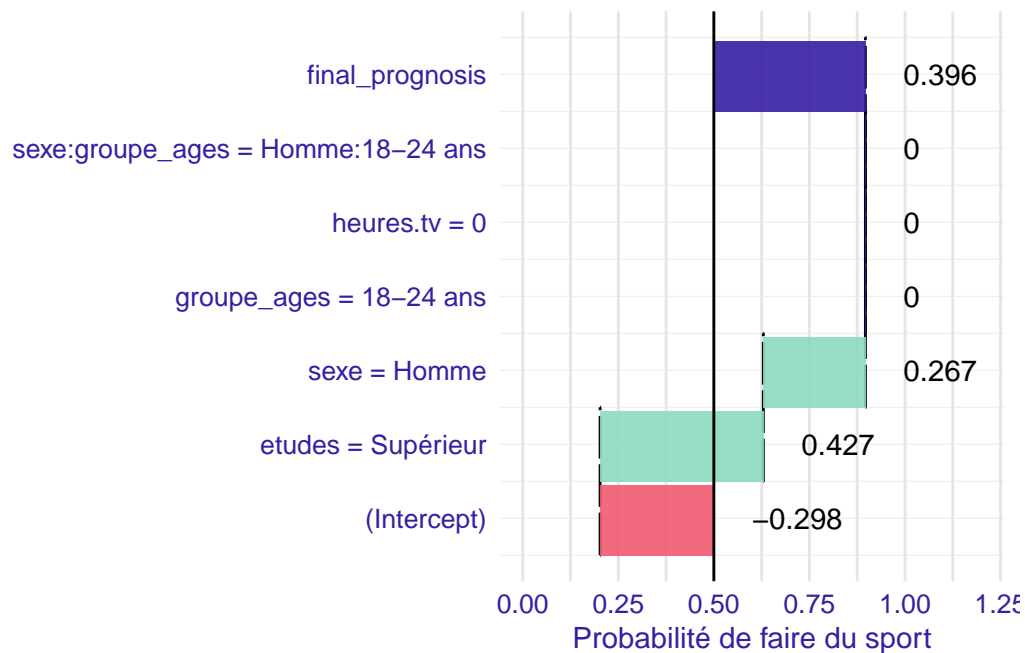


Figure 26.5: Représentation graphique de l'estimation de la probabilité de faire du sport pour un homme de 20 ans

Nous sommes à la modalité de référence pour l'âge par contre il y a un effet important du sexe. Le coefficient associé globalement à la variable sexe correspond donc à l'effet du sexe à la modalité de référence du groupe d'âges.

Regardons enfin la situation d'un homme de 60 ans.

```

nouvelle_observation$groupe_ages[1] = "45-64 ans"
plot(
  broken(mod2, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1.2) +
  ylab("Probabilité de faire du sport")

```

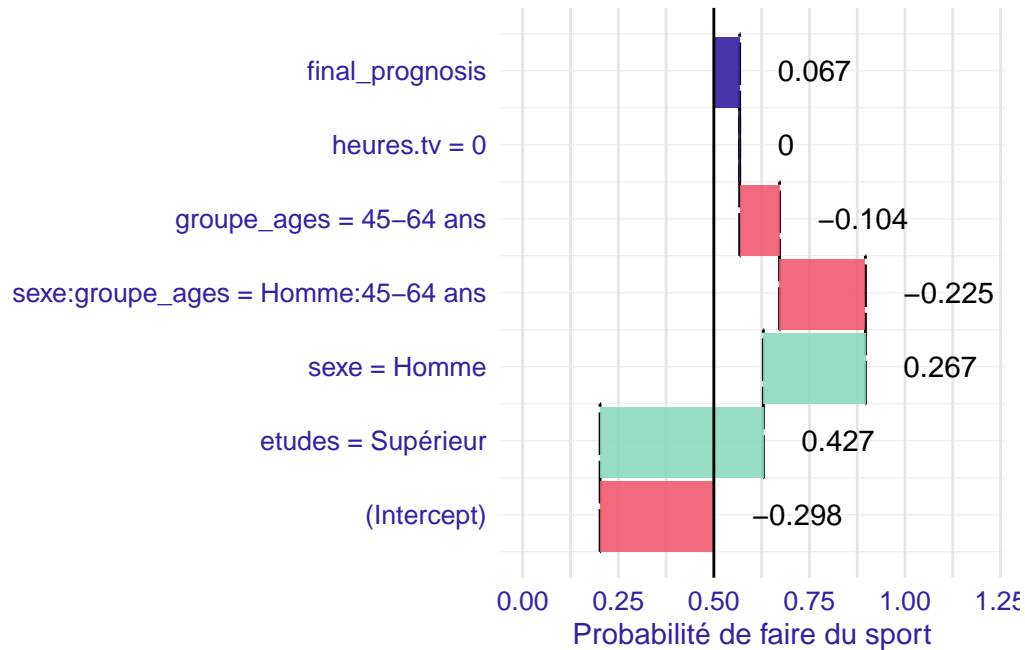


Figure 26.6: Représentation graphique de l'estimation de la probabilité de faire du sport pour un homme de 60 ans

Cette fois, plusieurs coefficients s'appliquent : à la fois le coefficient `sexe = Homme` (effet du sexe pour les 18-24 ans), le coefficient `groupe_ages = 45-64 ans` qui est l'effet de l'âge pour les femmes de 45-64 ans par rapport aux 18-24 ans et le coefficient `sexe:groupe_ages = Homme:45-64 ans` qui indique l'effet spécifique qui s'applique aux hommes de 45-64 ans, d'une part par rapport aux femmes du même âge et d'autre part par rapport aux hommes de 18-24 ans. L'effet des coefficients d'interaction doivent donc être interprétés par rapport aux autres coefficients du modèle qui s'appliquent, en tenant compte des modalités de référence.

26.6 Définition alternative de l'interaction

Il est cependant possible d'écrire le même modèle différemment. En effet, `sexe * groupe_ages` dans la formule du modèle est équivalent à l'écriture `sexe + groupe_ages + sexe:groupe_ages`, c'est-à-dire que l'on demande des coefficients pour la variable *sexe* à la référence de *groupe_ages*, des coefficients pour *groupe_ages* à la référence de *sexe* et enfin des coefficients pour tenir compte de l'interaction.

On peut se contenter d'une série de coefficients uniques pour l'interaction en indiquant seulement `sexe : groupe_ages`.

```
mod3 <- glm(  
  sport ~ sexe : groupe_ages + etudes + heures.tv,  
  family = binomial,  
  data = d  
)
```

Au sens strict, ce modèle explique tout autant le phénomène étudié que le modèle précédent. On peut le vérifier facilement avec `stats::anova()`.

```
anova(mod2, mod3, test = "Chisq")
```

Analysis of Deviance Table

Model 1: sport ~ sexe * groupe_ages + etudes + heures.tv

Model 2: sport ~ sexe:groupe_ages + etudes + heures.tv

	Resid. Df	Resid. Dev	Df	Deviance	Pr(>Chi)
1	1982	2197.4			
2	1982	2197.4	0		0

De même, les prédictions marginales sont les mêmes, comme nous pouvons le constater avec `ggstats::ggcoef_compare()`.

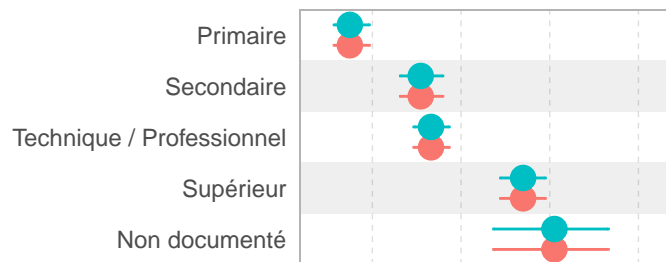
```
ggstats::ggcoef_compare(  
  list("sexe * groupe_ages" = mod2, "sexe : groupe_ages" = mod3),  
  tidy_fun = broom.helpers::tidy_marginal_predictions,  
  significance = NULL,  
  vline = FALSE  
) +  
  scale_x_continuous(labels = scales::label_percent())
```

Warning: Model matrix is rank deficient. Some variance-covariance parameters are missing.

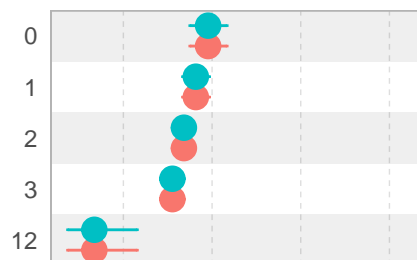
Warning: Model matrix is rank deficient. Some variance-covariance parameters are missing.

Warning: Model matrix is rank deficient. Some variance-covariance parameters are missing.

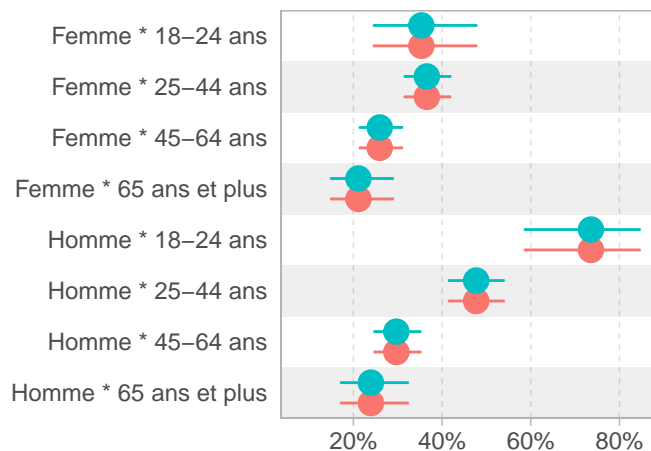
Niveau d'études



Heures de télévision / jour



Sexe * Groupe d'âges



Average Marginal Predictions

sexe * groupe_ages sexe : group

Figure 26.7: Comparaison des prédictions marginales moyennes des deux modèles avec interaction

Par contre, regardons d'un peu plus près les coefficients de ce nouveau modèle. Nous allons voir que leur interprétation est légèrement différente.

```
mod3 |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

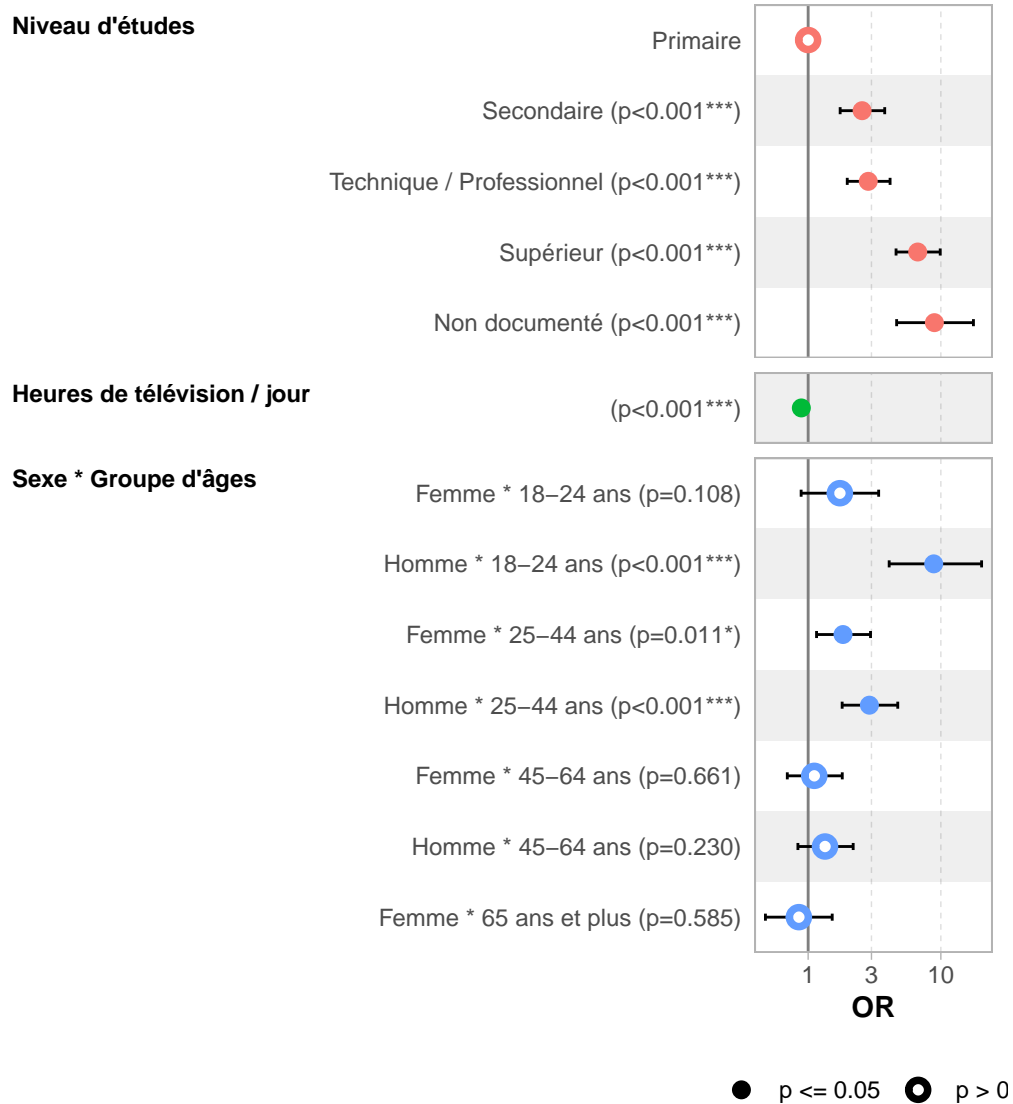


Figure 26.8: Coefficients (odds ratio) du modèle avec interaction simple entre le sexe et le groupe d'âges

Cette fois-ci, il n'y a plus de coefficients globaux pour la variable *sexe* ni pour *groupe_ages*

mais des coefficients pour chaque combinaison de ces deux variables. Reprenons l'exemple de notre homme de 60 ans.

```
plot(
  broken(mod3, nouvelle_observation, predict.function = betas),
  trans = logit
) +
  ylim(0, 1.2) +
  ylab("Probabilité de faire du sport")
```

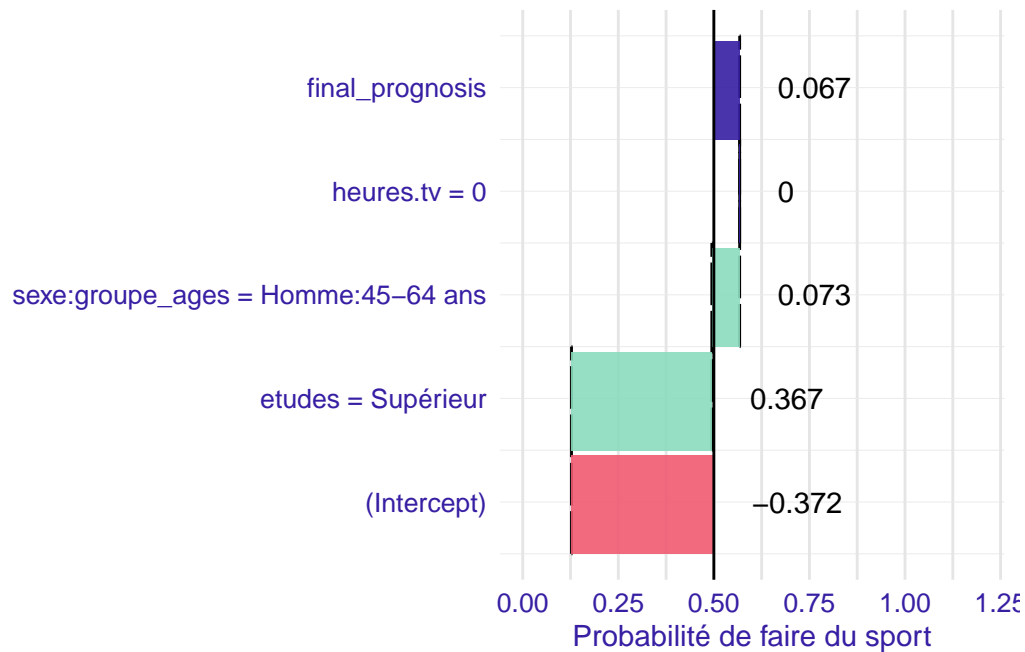


Figure 26.9: Représentation graphique de l'estimation de la probabilité de faire du sport pour un homme de 60 ans (interaction simple)

Cette fois-ci, le coefficient d'interaction fournit indique l'effet combiné du sexe et du groupe d'âges par rapport à la situation de référence (femme de 18-24 ans).

Que l'on définisse une interaction simple (`sexe:groupe_ages`) ou complète (`sexe*groupe_ages`), les deux modèles calculés sont donc identiques en termes prédictifs et explicatifs, mais l'interprétation de leurs coefficients diffèrent.

26.7 Identifier les interactions pertinentes

Il peut y avoir de multiples interactions dans un modèle, d'ordre 2 (entre deux variables) ou plus (entre trois variables ou plus). Il est toujours bon, selon notre connaissance du sujet et de la littérature, d'explorer manuellement les interactions attendues / prévisibles.

Mais, il est tentant de vouloir tester les multiples interactions possibles de manière itératives afin d'identifier celles à retenir.

Une possibilité² est d'avoir recours à une sélection de modèle pas à pas ascendante (voir Chapitre 23). Nous allons partir de notre modèle sans interaction, indiquer à `step()` l'ensemble des interactions possibles et voir si nous pouvons minimiser l'AIC.

```
mod4 <- mod |>
  step(scope = list(upper = ~ sexe * groupe_ages * etudes * heures.tv))
```

Start: AIC=2230.4

sport ~ sexe + groupe_ages + etudes + heures.tv

	Df	Deviance	AIC
+ sexe:groupe_ages	3	2197.4	2223.4
+ sexe:etudes	4	2199.6	2227.6
+ sexe:heures.tv	1	2207.6	2229.6
<none>		2210.4	2230.4
+ groupe_ages:heures.tv	3	2207.0	2233.0
+ etudes:heures.tv	4	2207.4	2235.4
+ groupe_ages:etudes	11	2194.6	2236.6
- heures.tv	1	2224.0	2242.0
- sexe	1	2226.4	2244.4
- groupe_ages	3	2260.6	2274.6
- etudes	4	2334.3	2346.3

Step: AIC=2223.38

sport ~ sexe + groupe_ages + etudes + heures.tv + sexe:groupe_ages

	Df	Deviance	AIC
+ sexe:heures.tv	1	2194.7	2222.7
<none>		2197.4	2223.4
+ groupe_ages:heures.tv	3	2193.5	2225.5
+ sexe:etudes	4	2192.1	2226.1
+ etudes:heures.tv	4	2194.6	2228.6

²On pourra également regarder du côté de `glmulti::glmulti()` pour des approches alternatives.

```

- sexe:groupe_ages      3    2210.4 2230.4
+ groupe_ages:etudes    11    2183.1 2231.1
- heures.tv            1    2210.2 2234.2
- etudes                4    2323.0 2341.0

```

Step: AIC=2222.67

```

sport ~ sexe + groupe_ages + etudes + heures.tv + sexe:groupe_ages +
      sexe:heures.tv

```

	Df	Deviance	AIC
<none>		2194.7	2222.7
- sexe:heures.tv	1	2197.4	2223.4
+ groupe_ages:heures.tv	3	2190.4	2224.4
+ sexe:etudes	4	2189.0	2225.0
+ etudes:heures.tv	4	2191.6	2227.6
- sexe:groupe_ages	3	2207.6	2229.6
+ groupe_ages:etudes	11	2180.7	2230.7
- etudes	4	2319.9	2339.9

```
mod4$formula
```

```

sport ~ sexe + groupe_ages + etudes + heures.tv + sexe:groupe_ages +
      sexe:heures.tv

```

Le modèle final suggéré comprends une interaction entre le sexe et le groupe d'âges et une interaction entre le sexe et le nombre quotidien d'heures de télévision. Nous pouvons utiliser `broom.helpers::plot_marginal_predictions()` pour visualiser l'effet de ces deux interactions.

```

mod4 |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
  patchwork::wrap_plots(ncol = 1) &
  scale_y_continuous(
    labels = scales::label_percent()
  )

```

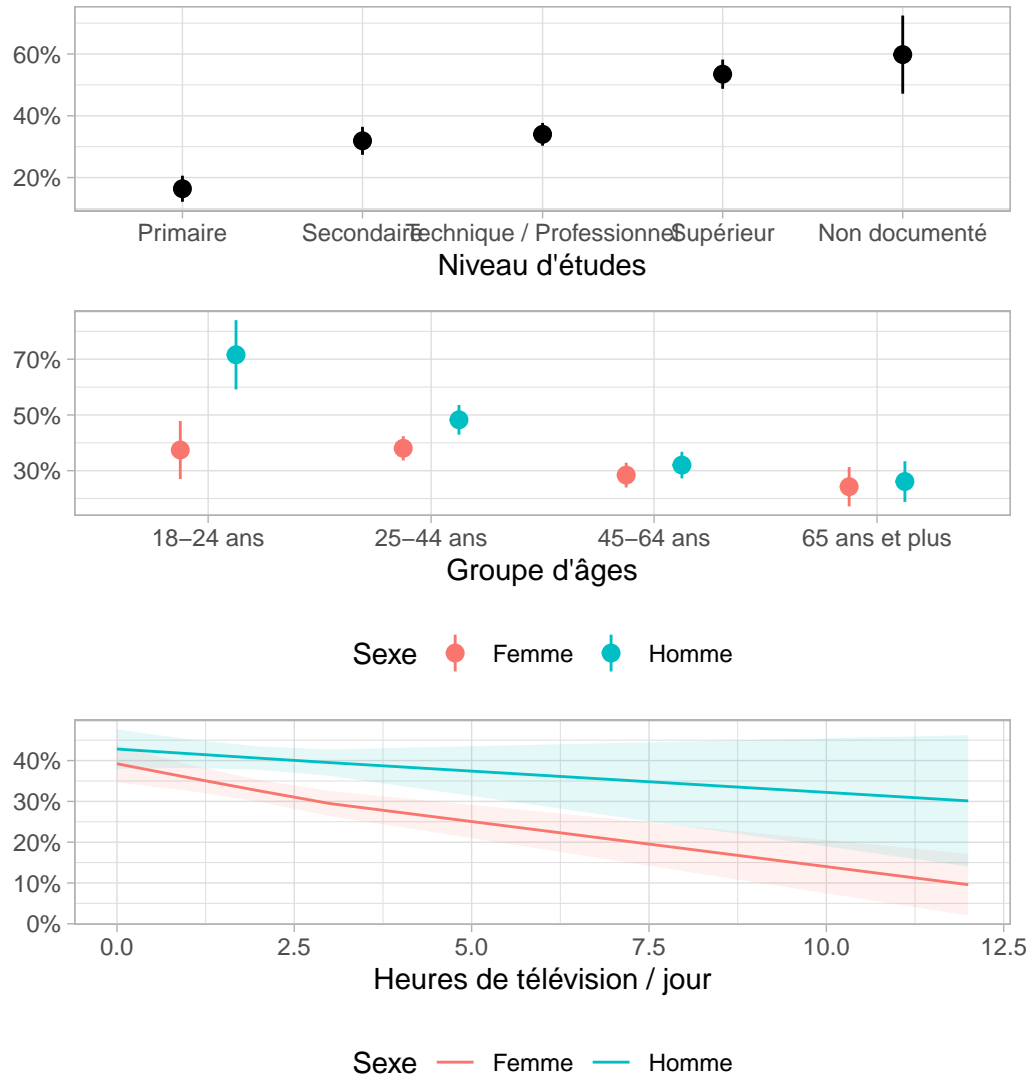


Figure 26.10: Prédictions marginales moyennes du modèle avec deux interactions

26.8 Pour aller plus loin

Il y a d'autres extensions dédiées à l'analyse des interactions d'un modèle, de même que de nombreux supports de cours en ligne dédiés à cette question.

- [Les effets d'interaction](#) par Jean-François Bickel
- [Analysing interactions of fitted models](#) par Helios De Rosario Martínez

26.9 webin-R

Les interactions sont abordées dans le webin-R #07 (*régression logistique partie 2*) sur [YouTube](#).

<https://youtu.be/BUo9i7XTLYQ>

27 Multicolinéarité

Dans une régression, la multicolinéarité est un problème qui survient lorsque certaines variables de prévision du modèle mesurent le même phénomène. Une multicolinéarité prononcée s'avère problématique, car elle peut augmenter la variance des coefficients de régression et les rendre instables et difficiles à interpréter. Les conséquences de coefficients instables peuvent être les suivantes :

- les coefficients peuvent sembler non significatifs, même lorsqu'une relation significative existe entre le prédicteur et la réponse ;
- les coefficients de prédicteurs fortement corrélés varieront considérablement d'un échantillon à un autre ;
- lorsque des termes d'un modèle sont fortement corrélés, la suppression de l'un de ces termes aura une incidence considérable sur les coefficients estimés des autres. Les coefficients des termes fortement corrélés peuvent même présenter le mauvais signe.

La multicolinéarité n'a aucune incidence sur l'adéquation de l'ajustement, ni sur la qualité de la prévision. Cependant, les coefficients individuels associés à chaque variable explicative ne peuvent pas être interprétés de façon fiable.

27.1 Définition

Au sens strict, on parle de multicolinéarité parfaite lorsqu'une des variables explicatives d'un modèle est une combinaison linéaire d'une ou plusieurs autres variables explicatives introduites dans le même modèle. L'absence de multicolinéarité parfaite est une des conditions requises pour pouvoir estimer un modèle linéaire et, par extension, un modèle linéaire généralisé (dont les modèles de régression logistique).

Dans les faits, une multicolinéarité parfaite n'est quasiment jamais observée. Mais une forte multicolinéarité entre plusieurs variables peut poser problème dans l'estimation et l'interprétation d'un modèle.

Une erreur fréquente est de confondre multicolinéarité et corrélation. Si des variables colinéaires sont *de facto* fortement corrélées entre elles, deux variables corrélées ne sont pas forcément colinéaires. En termes non statistiques, il y a colinéarité lorsque deux ou plusieurs variables mesurent la même chose.

Prenons un exemple. Nous étudions les complications après l'accouchement dans différentes maternités d'un pays en développement. On souhaite mettre dans le modèle, à la fois le milieu de résidence (urbain ou rural) et le fait qu'il y ait ou non un médecin dans la clinique. Or, dans la zone d'enquête, les maternités rurales sont dirigées seulement par des sage-femmes tandis que l'on trouve un médecin dans toutes les maternités urbaines sauf une. Dès lors, dans ce contexte précis, le milieu de résidence prédit presque totalement la présence d'un médecin et on se retrouve face à une multicollinéarité (qui serait même parfaite s'il n'y avait pas une clinique urbaine sans médecin). On ne peut donc distinguer l'effet de la présence d'un médecin de celui du milieu de résidence et il ne faut mettre qu'une seule de ces deux variables dans le modèle, sachant que du point de vue de l'interprétation elle capturera à la fois l'effet de la présence d'un médecin et celui du milieu de résidence.

Par contre, si dans notre région d'étude, seule la moitié des maternités urbaines disposait d'un médecin, alors le milieu de résidence n'aurait pas été suffisant pour prédire la présence d'un médecin. Certes, les deux variables seraient corrélées mais pas colinéaires. Un autre exemple de corrélation sans colinéarité, c'est la relation entre milieu de résidence et niveau d'instruction. Il y a une corrélation entre ces deux variables, les personnes résidant en ville étant généralement plus instruites. Cependant, il existe également des personnes non instruites en ville et des personnes instruites en milieu rural. Le milieu de résidence n'est donc pas suffisant pour prédire le niveau d'instruction.

27.2 Mesure de la colinéarité

Il existe différentes mesures de la multicollinéarité. L'extension `{mctest}` en fournit plusieurs, mais elle n'est utilisable que si l'ensemble des variables explicatives sont de type numérique.

L'approche la plus classique consiste à examiner les facteurs d'inflation de la variance (FIV) ou variance inflation factor (VIF) en anglais. Les FIV estiment de combien la variance d'un coefficient est augmentée en raison d'une relation linéaire avec d'autres prédicteurs. Ainsi, un FIV de 1,8 nous dit que la variance de ce coefficient particulier est supérieure de 80 % à la variance que l'on aurait dû observer si ce facteur n'est absolument pas corrélé aux autres prédicteurs.

Si tous les FIV sont égaux à 1, il n'existe pas de multicollinéarité, mais si certains FIV sont supérieurs à 1, les prédicteurs sont corrélés. Il n'y a pas de consensus sur la valeur au-delà de laquelle on doit considérer qu'il y a multicollinéarité. Certains auteurs, comme Paul Allison¹, disent de regarder plus en détail les variables avec un FIV supérieur à 2,5. D'autres ne s'inquiètent qu'à partir de 5. Il n'existe pas de test statistique qui permettrait de dire s'il y a colinéarité ou non².

¹[When Can You Safely Ignore Multicollinearity?](#)

²Pour plus de détails, voir ce post de Davig Giles, [Can You Actually TEST for Multicollinearity?](#), qui explique pourquoi ce n'est pas possible.

L'extension `{car}` fournit une fonction `car::vif()` permettant de calculer les FIV à partir d'un modèle. Elle implémente même une version généralisée permettant de considérer des facteurs catégoriels et des modèles linéaires généralisés comme la régression logistique.

Reprenons, pour exemple, un modèle logistique que nous avons déjà abordé dans d'autres chapitres.

```
library(tidyverse)
library(labelled)

data(hdv2003, package = "questionr")

d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
        "Technique / Professionnel" = "Enseignement technique ou professionnel long",
        "Supérieur" = "Enseignement superieur y compris technique superieur"
      ) |>
      fct_na_value_to_level("Non documenté")
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    etudes = "Niveau d'études",
    heures.tv = "Heures de télévision / jour"
```

```
)

mod <- glm(
  sport ~ sexe + groupe_ages + etudes + heures.tv,
  family = binomial,
  data = d
)
```

Le calcul des FIV se fait simplement en passant le modèle à la fonction `car::vif()`.

```
mod |> car::vif()
```

	GVIF	Df	$GVIF^{1/(2*Df)}$
sexe	1.024640	1	1.012245
groupe_ages	1.745492	3	1.097285
etudes	1.811370	4	1.077087
heures.tv	1.057819	1	1.028503

Dans notre exemple, tous les FIV sont proches de 1. Il n'y a donc pas de problème potentiel de colinéarité à explorer.

Pour un tableau propre, nous pouvons aussi utiliser `gtsummary::add_vif()`.

```
library(gtsummary)
theme_gtsummary_language(
  "fr",
  decimal.mark = ",",
  big.mark = " "
)
```

```
mod |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels() |>
  add_vif()
```

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

Table 27.1: Résumé du modèle logistique simple avec affichage des VIF généralisés

Caractéristique	OR	95% IC	p-valeur	GVIF	Adjusted GVIF
Sexe				1,0	1,0
Femme	—	—			
Homme	1,52	1,24 – 1,87	<0,001		
Groupe d'âges				1,7	1,1
18-24 ans	—	—			
25-44 ans	0,68	0,43 – 1,06	0,084		
45-64 ans	0,36	0,23 – 0,57	<0,001		
65 ans et plus	0,27	0,16 – 0,46	<0,001		
Niveau d'études				1,8	1,1
Primaire	—	—			
Secondaire	2,54	1,73 – 3,75	<0,001		
Technique / Professionnel	2,81	1,95 – 4,10	<0,001		
Supérieur	6,55	4,50 – 9,66	<0,001		
Non documenté	8,54	4,51 – 16,5	<0,001		
Heures de télévision / jour	0,89	0,83 – 0,95	<0,001	1,1	1,0

Le package `{performance}` propose quant à lui une fonction `performance::check_collinearity()` pour le calcul des FIV et de leur intervalle de confiance.

```
mod |> performance::check_collinearity()
```

```
# Check for Multicollinearity
```

```
Low Correlation
```

Term	VIF	VIF 95% CI	Increased SE	Tolerance	Tolerance 95% CI
sexe	1.02	[1.00, 1.16]	1.01	0.98	[0.86, 1.00]
groupe_ages	1.75	[1.64, 1.86]	1.32	0.57	[0.54, 0.61]
etudes	1.81	[1.70, 1.93]	1.35	0.55	[0.52, 0.59]
heures.tv	1.06	[1.02, 1.13]	1.03	0.95	[0.88, 0.98]

Les variables avec un FIV entre 5 et 10 sont présentées comme ayant une corrélation moyenne et celles avec un FIV de 10 ou plus une corrélation forte. Prenons un autre exemple.

```
mod2 <- lm(mpg ~ wt + am + gear + vs * cyl, data = mtcars)
mc <- mod2 |> performance::check_collinearity()
```

Model has interaction terms. VIFs might be inflated.
 You may check multicollinearity among predictors of a model without interaction terms.

```
mc
```

```
# Check for Multicollinearity
```

Low Correlation

Term	VIF	VIF 95% CI	Increased SE	Tolerance	Tolerance 95% CI
wt	3.64	[2.46, 5.77]	1.91	0.27	[0.17, 0.41]
am	4.32	[2.88, 6.89]	2.08	0.23	[0.15, 0.35]
gear	3.06	[2.11, 4.82]	1.75	0.33	[0.21, 0.47]

Moderate Correlation

Term	VIF	VIF 95% CI	Increased SE	Tolerance	Tolerance 95% CI
cyl	7.87	[5.03, 12.70]	2.80	0.13	[0.08, 0.20]

High Correlation

Term	VIF	VIF 95% CI	Increased SE	Tolerance	Tolerance 95% CI
vs	38.55	[23.69, 63.15]	6.21	0.03	[0.02, 0.04]
vs:cyl	25.79	[15.93, 42.16]	5.08	0.04	[0.02, 0.06]

Une représentation graphique des FIV peut être obtenue avec `plot()` appliquée au résultat de `performance::check_collinearity()`.

```
plot(mc)
```

Variable `Component` is not in your data frame :/

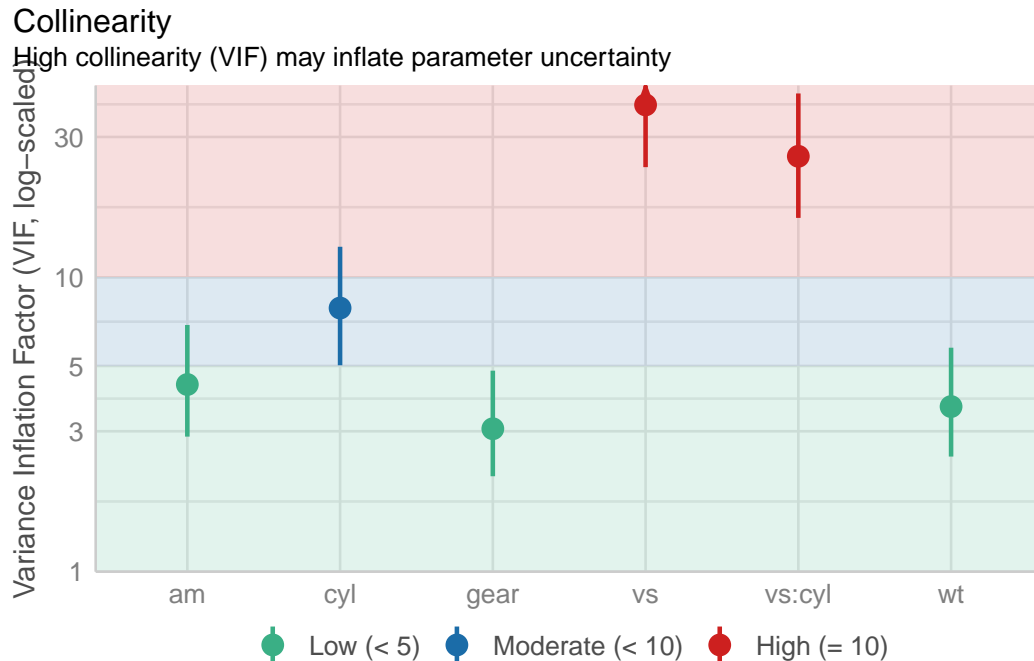


Figure 27.1: Représentation graphique des FIV d'un modèle

La fonction `performance::print_md()` peut être utilisée quant à elle pour une sortie des résultats dans un rapport markdown.

```
mc |> performance::print_md()
```

Table 27.2: Table des FIV d'un modèle

Term	VIF	VIF_CI_low	VIF_CI_high	SE_factor	Tolerance	Tolerance_CI_low	Tolerance_CI_high
wt	3.64	2.46	5.77	1.91	0.27	0.17	0.41
am	4.32	2.88	6.89	2.08	0.23	0.15	0.35
gear	3.06	2.11	4.82	1.75	0.33	0.21	0.47
vs	38.55	23.69	63.15	6.21	0.03	0.02	0.04
cyl	7.87	5.03	12.70	2.80	0.13	0.08	0.20
vs:cyl	25.79	15.93	42.16	5.08	0.04	0.02	0.06

27.3 La multicollinéarité est-elle toujours un problème ?

Là encore, il n'y a pas de consensus sur cette question. Certains analystes considèrent que tout modèle où certains prédicteurs seraient colinéaires n'est pas valable. Dans le billet [When](#)

[Can You Safely Ignore Multicollinearity?](#), Paul Allison évoque quant à lui des situations où la multicolinéarité peut être ignorée en toute sécurité. Le texte ci-dessous est une traduction de ce billet.

1. Les variables avec des FIV élevés sont des variables de contrôle, et les variables d'intérêt n'ont pas de FIV élevés.

Voici le problème de la multicolinéarité : ce n'est un problème que pour les variables qui sont colinéaires. Il augmente les erreurs-types de leurs coefficients et peut rendre ces coefficients instables de plusieurs façons. Mais tant que les variables colinéaires ne sont utilisées que comme variables de contrôle, et qu'elles ne sont pas colinéaires avec vos variables d'intérêt, il n'y a pas de problème. Les coefficients des variables d'intérêt ne sont pas affectés et la performance des variables de contrôle n'est pas altérée.

Voici un exemple tiré de ces propres travaux : l'échantillon est constitué de collèves américains, la variable dépendante est le taux d'obtention de diplôme et la variable d'intérêt est un indicateur (factice) pour les secteurs public et privé. Deux variables de contrôle sont les scores moyens au SAT et les scores moyens à l'ACT pour l'entrée en première année. Ces deux variables ont une corrélation supérieure à ,9, ce qui correspond à des FIV d'au moins 5,26 pour chacune d'entre elles. Mais le FIV pour l'indicateur public/privé n'est que de 1,04. Il n'y a donc pas de problème à se préoccuper et il n'est pas nécessaire de supprimer l'un ou l'autre des deux contrôles, à condition que l'on ne cherche pas à interpréter ou comparer l'un par rapport à l'autre les coefficients de ces deux variables de contrôle.

2. Les FIV élevés sont causés par l'inclusion de puissances ou de produits d'autres variables.

Si vous spécifiez un modèle de régression avec x et x^2 , il y a de bonnes chances que ces deux variables soient fortement corrélées. De même, si votre modèle a x , z et xz , x et z sont susceptibles d'être fortement corrélés avec leur produit. Il n'y a pas de quoi s'inquiéter, car la valeur p de xz n'est pas affectée par la multicolinéarité. Ceci est facile à démontrer : vous pouvez réduire considérablement les corrélations en centrant les variables (c'est-à-dire en soustrayant leurs moyennes) avant de créer les puissances ou les produits. Mais la valeur p pour x^2 ou pour xz sera exactement la même, que l'on centre ou non. Et tous les résultats pour les autres variables (y compris le R^2 mais sans les termes d'ordre inférieur) seront les mêmes dans les deux cas. La multicolinéarité n'a donc pas de conséquences négatives.

3. Les variables avec des FIV élevés sont des variables indicatrices (factices) qui représentent une variable catégorielle avec trois catégories ou plus.

Si la proportion de cas dans la catégorie de référence est faible, les variables indicatrices auront nécessairement des FIV élevés, même si la variable catégorielle n'est pas associée à d'autres variables dans le modèle de régression.

Supposons, par exemple, qu'une variable de l'état matrimonial comporte trois catégories : actuellement marié, jamais marié et anciennement marié. Vous choisissez anciennement marié comme catégorie de référence, avec des variables d'indicateur pour les deux autres. Ce qui

se passe, c'est que la corrélation entre ces deux indicateurs devient plus négative à mesure que la fraction de personnes dans la catégorie de référence diminue. Par exemple, si 45 % des personnes ne sont jamais mariées, 45 % sont mariées et 10 % sont anciennement mariées, les valeurs du FIV pour les personnes mariées et les personnes jamais mariées seront d'au moins 3,0.

Est-ce un problème ? Eh bien, cela signifie que les valeurs p des variables indicatrices peuvent être élevées. Mais le test global selon lequel tous les indicateurs ont des coefficients de zéro n'est pas affecté par des FIV élevés. Et rien d'autre dans la régression n'est affecté. Si vous voulez vraiment éviter des FIV élevés, il suffit de choisir une catégorie de référence avec une plus grande fraction des cas. Cela peut être souhaitable pour éviter les situations où aucun des indicateurs individuels n'est statistiquement significatif, même si l'ensemble des indicateurs est significatif.

27.4 webin-R

La multicolinéarité est abordée dans le webin-R #07 (*régression logistique partie 2*) sur [YouTube](#).

<https://youtu.be/BUo9i7XTLYQ>

partie IV

Données pondérées avec survey

28 Définir un plan d'échantillonnage

Lorsque l'on travaille avec des données d'enquêtes, il est fréquent que les données soient **pondérées**. Cette pondération est nécessaire pour assurer la représentativité des données lorsque les participants ont été sélectionnés avec des probabilités variables. C'est par exemple le cas lorsqu'on réalise une enquête en grappes et/ou stratifiée.

De nombreuses fonctions acceptent une variable de pondération. Cependant, la simple prise en compte de la pondération est souvent insuffisante si l'on ne tient pas compte du plan d'échantillonnage de l'enquête. Le plan d'échantillonnage ne joue pas seulement sur la pondération des données, mais influence le calcul des variances et par ricochet tous les tests statistiques. Deux échantillons identiques avec la même variable de pondération mais des designs différents produiront les mêmes moyennes et proportions mais des intervalles de confiance différents.

Diverses fonctions de **R** peuvent prendre en compte une variable de pondération. Mais, en règle générale, elles sont incapables de tenir compte du plan d'échantillonnage. Il est donc préférable de privilégier le package `{survey}` qui est spécialement dédié au traitement d'enquêtes ayant des techniques d'échantillonnage et de pondération potentiellement très complexes. `{survey}` peut également être utilisée pour des pondérations simples.

28.1 Différents types d'échantillonnage

L'**échantillonnage aléatoire simple** ou **échantillonnage équiprobable** est une méthode pour laquelle tous les échantillons possibles (de même taille) ont la même probabilité d'être choisis et tous les éléments de la population ont une chance égale de faire partie de l'échantillon. C'est l'échantillonnage le plus simple : chaque individu a la même probabilité d'être sélectionné.

L'**échantillonnage stratifié** est une méthode qui consiste d'abord à subdiviser la population en groupes homogènes (strates) pour ensuite extraire un échantillon aléatoire de chaque strate. Cette méthode suppose une connaissance de la structure de la population. Pour estimer les paramètres, les résultats doivent être pondérés par l'importance relative de chaque strate dans la population.

L'échantillonnage par grappes est une méthode qui consiste à choisir un échantillon aléatoire d'unités qui sont elles-mêmes des sous-ensembles de la population (grappes ou clusters en

anglais). Cette méthode suppose que les unités de chaque grappe sont représentatives. Elle possède l'avantage d'être souvent plus économique.

Il est possible de combiner plusieurs de ces approches. Par exemple, les *Enquêtes Démographiques et de Santé*¹ (EDS) sont des enquêtes stratifiées en grappes à deux degrés. Dans un premier temps, la population est divisée en strates par région et milieu de résidence. Dans chaque strate, des zones d'enquêtes, correspondant à des unités de recensement, sont tirées au sort avec une probabilité proportionnelle au nombre de ménages de chaque zone au dernier recensement de population. Enfin, au sein de chaque zone d'enquête sélectionnée, un recensement de l'ensemble des ménages est effectué puis un nombre identique de ménages par zone d'enquête est tiré au sort de manière aléatoire simple.

28.2 Avec `survey::svydesign()`

La fonction `survey::svydesign()` accepte plusieurs arguments décrits en détail sur sa page d'aide (obtenue avec la commande `?svydesign`).

L'argument `data` permet de spécifier le tableau de données contenant les observations.

L'argument `ids` est obligatoire et spécifie sous la forme d'une formule les identifiants des différents niveaux d'un tirage en grappe. S'il s'agit d'un échantillon aléatoire simple, on entrera `ids = ~ 1`. Autre situation : supposons une étude portant sur la population française. Dans un premier temps, on a tiré au sort un certain nombre de départements français. Dans un second temps, on tire au sort dans chaque département des communes. Dans chaque commune sélectionnée, on tire au sort des quartiers. Enfin, on interroge de manière exhaustive toutes les personnes habitant les quartiers enquêtés. Notre fichier de données devra donc comporter pour chaque observation les variables `id_departement`, `id_commune` et `id_quartier`. On écrira alors pour l'argument `ids` la valeur suivante :

```
ids = ~ id_departement + id_commune + id_quartier.
```

Si l'échantillon est stratifié, on spécifiera les strates à l'aide de l'argument `strata` en spécifiant la variable contenant l'identifiant des strates. Par exemple : `strata = ~ id_strate`.

Il faut encore spécifier les probabilités de tirage de chaque cluster /grappe ou bien la pondération des individus. Si l'on dispose de la probabilité de chaque observation d'être sélectionnée, on utilisera l'argument `probs`. Si, par contre, on connaît la pondération de chaque observation (qui doit être proportionnelle à l'inverse de cette probabilité), on utilisera l'argument `weights`.

Si l'échantillon est stratifié, qu'au sein de chaque strate les individus ont été tirés au sort de manière aléatoire et que l'on connaît la taille de chaque strate, il est possible de ne pas avoir à spécifier la probabilité de tirage ou la pondération de chaque observation. Il est préférable

¹Vaste programme d'enquêtes réalisées à intervalles réguliers dans les pays à faible et moyen revenu, disponibles sur <https://dhsprogram.com/>.

de fournir une variable contenant la taille de chaque strate à l'argument `fpc`. De plus, dans ce cas-là, une petite correction sera appliquée au modèle pour prendre en compte la taille finie de chaque strate.

On peut tout à fait définir un **échantillonnage aléatoire simple** (on considère donc que toutes les observations ont le même poids, égal à 1). Pour rappel, en l'absence de clusters/grappes, il faut préciser `ids = ~ 1`, ce paramètre n'ayant pas de valeur par défaut.

```
p_iris <- survey::svydesign(  
  ids = ~ 1,  
  data = iris  
)
```

```
Warning in svydesign.default(ids = ~1, data = iris): No weights or  
probabilities supplied, assuming equal probability
```

```
p_iris
```

```
Independent Sampling design (with replacement)  
survey::svydesign(ids = ~1, data = iris)
```

Pour un jeu de données **simplement pondéré** (chaque ligne représente plusieurs observations) :

```
titanic <- dplyr::as_tibble(Titanic)  
titanic |> labelled::look_for()
```

pos	variable	label	col_type	missing	values
1	Class	-	chr	0	
2	Sex	-	chr	0	
3	Age	-	chr	0	
4	Survived	-	chr	0	
5	n	-	dbl	0	

```
p_titanic <- survey::svydesign(  
  ids = ~ 1,  
  data = titanic,  
  weights = ~ n  
)  
p_titanic
```

```
Independent Sampling design (with replacement)
survey::svydesign(ids = ~1, data = titanic, weights = ~n)
```

Pour un **échantillon stratifié** pour lequel les strates sont indiquées dans la variable *stype* et les poids indiquées dans la variable *pw*.

```
data("api", package = "survey")
p_strates <- survey::svydesign(
  id = ~ 1,
  strata = ~ stype,
  weights = ~ pw,
  data = apistrat
)
p_strates
```

```
Stratified Independent Sampling design (with replacement)
survey::svydesign(id = ~1, strata = ~stype, weights = ~pw, data = apistrat)
```

Pour une **enquête en grappes à 1 degré**, pour laquelle l'identifiant des grappes (*clusters*) est indiqué par la variable *dnum*.

```
data("api", package = "survey")
p_grappes <- survey::svydesign(
  id = ~ dnum,
  weights = ~ pw,
  data = apiclus1
)
p_grappes
```

```
1 - level Cluster Sampling design (with replacement)
With (15) clusters.
survey::svydesign(id = ~dnum, weights = ~pw, data = apiclus1)
```

Voici un exemple un peu plus complexe d'une **enquête en grappes à deux degrés** (les deux niveaux étant donnés par les variables *dnum* et *snum*). Les poids ne sont pas fournis mais la taille des grappes est connue et renseignée dans les variables *fpc1* et *fpc2* que nous pourrions donc transmettre via l'argument *fpc*.

```
data("api", package = "survey")
p_grappes2 <- survey::svydesign(
  id = ~ dnum + snum,
  fpc = ~ fpc1 + fpc2,
  data = apiclus2
)
p_grappes2
```

2 - level Cluster Sampling design

With (40, 126) clusters.

```
survey::svydesign(id = ~dnum + snum, fpc = ~fpc1 + fpc2, data = apiclus2)
```

Dans le cas présent, `{survey}` a calculé les poids s'appliquant à chaque individu. On peut les obtenir avec la fonction `weights()`, en l'occurrence avec `p_grappes2 |> weights()`.

Enfin, prenons l'exemple d'une *Enquête Démographique et de Santé*. Le nom des différentes variables est standardisé et commun quelle que soit l'enquête. Nous supposons que vous avez importé le fichier *individus* dans un tableau de données nommés `eds`. Le poids statistique de chaque individu est fourni par la variable *V005* qui doit au préalable être divisée par un million. Les grappes d'échantillonnage au premier degré sont fournies par la variable *V021* (*primary sample unit*). Si elle n'est pas renseignée, on pourra utiliser le numéro de grappe *V001*. Enfin, le milieu de résidence (urbain / rural) est fourni par *V025* et la région par *V024*. Pour rappel, l'échantillon a été stratifié à la fois par région et par milieu de résidence. Certaines enquêtes fournissent directement un numéro de strate via *V022*. Si tel est le cas, on pourra préciser le plan d'échantillonnage ainsi :

```
eds$poids <- eds$V005/1000000
p_eds <- survey::svydesign(
  ids = ~ V021,
  data = eds,
  strata = ~ V022,
  weights = ~ poids
)
```

28.3 Avec `srvyr::as_survey_design()`

Dans le prochain chapitre (cf. Chapitre 29), nous aborderons le package `{srvyr}` qui est aux objets `{survey}` ce que `{dplyr}` est aux tableaux de données : plus précisément, ce package étend les verbes de `{dplyr}` aux plans d'échantillonnage complexe.

La fonction `srvyr::as_survey_design()` est équivalente à `survey::svydesign()` mais avec quelques différences :

- le paramètre `ids` dispose d'une valeur par défaut et on peut l'ignorer en l'absence de grappes ;
- les variables ne sont pas spécifiées avec une formule mais avec les mêmes sélecteurs que `dplyr::select()` ;
- l'objet renvoyé est à la fois du type `"survey.design"` et du type `"tbl_svy"`, une sorte de *tibble* pour les objets `{survey}`.

Reprenons nos exemples précédents en commençant par un **échantillonnage aléatoire simple**.

```
t_iris <- iris |>
  srvyr::as_survey_design()
t_iris
```

Independent Sampling design (with replacement)

Called via `srvyr`

Sampling variables:

- `ids: `1``

Data variables: `Sepal.Length (dbl)`, `Sepal.Width (dbl)`, `Petal.Length (dbl)`,
`Petal.Width (dbl)`, `Species (fct)`

```
class(t_iris)
```

```
[1] "tbl_svy"          "survey.design2" "survey.design"
```

Pour un jeu de données **simplement pondéré** (chaque ligne représente plusieurs observations) :

```
titanic <- dplyr::as_tibble(Titanic)
t_titanic <- titanic |>
  srvyr::as_survey_design(weights = n)
t_titanic
```

Independent Sampling design (with replacement)

Called via `srvyr`

Sampling variables:

- `ids: `1``

- `weights: n`

Data variables: `Class (chr)`, `Sex (chr)`, `Age (chr)`, `Survived (chr)`, `n (dbl)`

Pour un **échantillon stratifié** pour lequel les strates sont indiquées dans la variable *stype* et les poids indiquées dans la variable *pw*.

```
data("api", package = "survey")
t_strates <- apistrat |>
  srvyr::as_survey_design(strata = stype, weights = pw)
t_strates
```

Stratified Independent Sampling design (with replacement)

Called via srvyr

Sampling variables:

- ids: `1`
- strata: stype
- weights: pw

Data variables: cds (chr), stype (fct), name (chr), sname (chr), snum (dbl),
dname (chr), dnum (int), cname (chr), cnum (int), flag (int), pcttest (int),
api00 (int), api99 (int), target (int), growth (int), sch.wide (fct),
comp.imp (fct), both (fct), awards (fct), meals (int), ell (int), yr.rnd
(fct), mobility (int), acs.k3 (int), acs.46 (int), acs.core (int), pct.resp
(int), not.hsg (int), hsg (int), some.col (int), col.grad (int), grad.sch
(int), avg.ed (dbl), full (int), emer (int), enroll (int), api.stu (int), pw
(dbl), fpc (dbl)

Pour une **enquête en grappes à 1 degré**, pour laquelle l'identifiant des grappes (*clusters*) est indiqué par la variable *dnum*.

```
data("api", package = "survey")
t_grappes <- apiclus1 |>
  srvyr::as_survey_design(id = dnum, weights = pw)
t_grappes
```

1 - level Cluster Sampling design (with replacement)

With (15) clusters.

Called via srvyr

Sampling variables:

- ids: dnum
- weights: pw

Data variables: cds (chr), stype (fct), name (chr), sname (chr), snum (dbl),
dname (chr), dnum (int), cname (chr), cnum (int), flag (int), pcttest (int),
api00 (int), api99 (int), target (int), growth (int), sch.wide (fct),
comp.imp (fct), both (fct), awards (fct), meals (int), ell (int), yr.rnd

```
(fct), mobility (int), acs.k3 (int), acs.46 (int), acs.core (int), pct.resp
(int), not.hsg (int), hsg (int), some.col (int), col.grad (int), grad.sch
(int), avg.ed (dbl), full (int), emer (int), enroll (int), api.stu (int), fpc
(dbl), pw (dbl)
```

Voici un exemple un peu plus complexe d'une **enquête en grappes à deux degrés** (les deux niveaux étant donnés par les variables *dnum* et *snum*). Les poids ne sont pas fournis mais la taille des grappes est connue et renseignée dans les variables *fpc1* et *fpc2* que nous pourrions donc transmettre via l'argument *fpc*.

```
data("api", package = "survey")
data("api", package = "survey")
t_grappes2 <- apiclus2 |>
  srvyr::as_survey_design(id = c(dnum, snum), fpc = c(fpc1, fpc2))
t_grappes2
```

2 - level Cluster Sampling design

With (40, 126) clusters.

Called via *srvyr*

Sampling variables:

```
- ids: `dnum + snum`
- fpc: `fpc1 + fpc2`
```

Data variables: *cds* (chr), *stype* (fct), *name* (chr), *sname* (chr), *snum* (dbl), *dname* (chr), *dnum* (int), *cname* (chr), *cnum* (int), *flag* (int), *pcttest* (int), *api00* (int), *api99* (int), *target* (int), *growth* (int), *sch.wide* (fct), *comp.imp* (fct), *both* (fct), *awards* (fct), *meals* (int), *ell* (int), *yr.rnd* (fct), *mobility* (int), *acs.k3* (int), *acs.46* (int), *acs.core* (int), *pct.resp* (int), *not.hsg* (int), *hsg* (int), *some.col* (int), *col.grad* (int), *grad.sch* (int), *avg.ed* (dbl), *full* (int), *emer* (int), *enroll* (int), *api.stu* (int), *pw* (dbl), *fpc1* (dbl), *fpc2* (int[1d])

Enfin, prenons l'exemple d'une *Enquête Démographique et de Santé*. Le nom des différentes variables est standardisé et commun quelle que soit l'enquête. Nous supposons que vous avez importé le fichier *individus* dans un tableau de données nommés *eds*. Le poids statistique de chaque individu est fourni par la variable *V005* qui doit au préalable être divisée par un million. Les grappes d'échantillonnage au premier degré sont fournies par la variable *V021* (*primary sample unit*). Si elle n'est pas renseignée, on pourra utiliser le numéro de grappe *V001*. Enfin, le milieu de résidence (urbain / rural) est fourni par *V025* et la région par *V024*. Pour rappel, l'échantillon a été stratifié à la fois par région et par milieu de résidence. Certaines enquêtes fournissent directement un numéro de strate via *V022*. Si tel est le cas, on pourra préciser le plan d'échantillonnage ainsi :

```
eds$poids <- eds$V005/1000000
t_eds <- eds |>
  srvyr::as_survey_design(
    ids = V021,
    strata = V022,
    weights = poids
  )
```

28.4 webin-R

La statistique univariée est présentée dans le webin-R #10 (*données pondérées, plan d'échantillonnage complexe & survey*) sur [YouTube](#).

<https://youtu.be/aXCn9SyhcTE>

29 Manipulation de données pondérées

L'objet créé avec `survey::svydesign()` ou `srvyr::as_survey_design()` n'est plus un tableau de données, mais plutôt un tableau de données auquel est attaché un plan d'échantillonnage. Les colonnes du tableau d'origine ne sont plus directement accessibles avec l'opérateur `$`. En fait, elles sont stockées dans un sous-objet `$variables`.

```
titanic <- dplyr::as_tibble(Titanic)
t_titanic <- titanic |>
  srvyr::as_survey_design(weights = n)
t_titanic$variables |> dplyr::glimpse()
```

Rows: 32

Columns: 5

```
$ Class      <chr> "1st", "2nd", "3rd", "Crew", "1st", "2nd", "3rd", "Crew", "1s~
$ Sex        <chr> "Male", "Male", "Male", "Male", "Female", "Female", "Female",~
$ Age        <chr> "Child", "Child", "Child", "Child", "Child", "Child", "Child"~
$ Survived   <chr> "No", "No", "No", "No", "No", "No", "No", "No", "No", "No", "~
$ n          <dbl> 0, 0, 35, 0, 0, 0, 17, 0, 118, 154, 387, 670, 4, 13, 89, 3, 5~
```

Il n'est pas aisé de modifier des variables dans un objet de ce type. Il est donc préférable de procéder à l'ensemble des nettoyages, recodages de variables (et au besoin transformation des vecteurs labellisés en facteur), avant de définir le plan d'échantillonnage et de procéder aux analyses.

Si l'on souhaite manipuler les données, le plus simple est d'avoir recours au package `{srvyr}` qui étend les verbes de `{dplyr}` (cf. Chapitre 8) aux objets `{survey}`.

29.1 Utilisation de `{srvyr}`

`{srvyr}` fournit les verbes `srvyr::select()` et `srvyr::filter()` pour sélectionner respectivement des colonnes et des lignes.

```
library(srvyr)
```

Attachement du package : 'srvyr'

L'objet suivant est masqué depuis 'package:stats':

```
filter
```

```
t_titanic |> select(Sex, Age)
```

Independent Sampling design (with replacement)

Called via srvyr

Sampling variables:

- ids: `1`
- weights: n

Data variables: Sex (chr), Age (chr)

```
t_titanic |> filter(Sex == "Female")
```

Independent Sampling design (with replacement)

Called via srvyr

Sampling variables:

- ids: `1`
- weights: n

Data variables: Class (chr), Sex (chr), Age (chr), Survived (chr), n (dbl)

On peut aussi utiliser `srvyr::pull()` pour extraire le contenu d'une colonne ou `srvyr::drop_na()` pour supprimer les observations contenant des valeurs manquantes.

Avertissement

Par contre, le verbe `arrange()` (tri du tableau) ou encore les fonctions de jointures (telles que `left_join()`) ne sont pas implémentées car ce type d'opération entraînerait des modifications du plan d'échantillonnage. Il est donc préférable de réaliser ce type d'opérations avant la déclaration du plan d'échantillonnage (quand les données sont donc encore stockées dans un tableau de données classiques).

`srvyr` fournit également le verbe `srvyr::summarize()` permettant de calculer des statistiques sur l'ensemble du fichier ou par sous-groupe (en combinant `summarize()`

avec `group_by()`). Afin de prendre en compte correctement la pondération et le plan d'échantillonnage, `srvyr` fournit des fonctions adaptées pour un usage au sein de `summarize()` : `srvyr::survey_mean()`, `srvyr::survey_total()`, `srvyr::survey_prop()`, `srvyr::survey_ratio()`, `srvyr::survey_quantile()` ou encore `srvyr::survey_median()`.

```
t_titanic |>
  group_by(Sex, Class, Survived) |>
  summarise(taux_survie = survey_prop()) |>
  filter(Survived == "Yes")
```

When ``proportion`` is unspecified, ``survey_prop()`` now defaults to ``proportion = TRUE``.
i This should improve confidence interval coverage.
This message is displayed once per session.

Warning: There were 24 warnings in ``dplyr::summarise()``.

The first warning was:

i In argument: ``taux_survie = survey_prop()``.

i In group 1: ``Sex = "Female"`, `Class = "1st"`, `Survived = "No"`.`

Caused by warning in ``summary.glm()``:

! les observations de poids nul n'ont pas été utilisées pour le calcul de la dispersion

i Run ``dplyr::last_dplyr_warnings()`` to see the 23 remaining warnings.

A tibble: 8 x 5

Groups: Sex, Class [8]

	Sex	Class	Survived	taux_survie	taux_survie_se
	<chr>	<chr>	<chr>	<dbl>	<dbl>
1	Female	1st	Yes	0.972	0.0384
2	Female	2nd	Yes	0.877	0.145
3	Female	3rd	Yes	0.459	0.306
4	Female	Crew	Yes	0.870	0.163
5	Male	1st	Yes	0.344	0.312
6	Male	2nd	Yes	0.140	0.150
7	Male	3rd	Yes	0.173	0.183
8	Male	Crew	Yes	0.223	0.249

29.2 Lister / Rechercher des variables

La fonction `labelled::look_for()`, que nous avons déjà abordée (cf. Section 4.3), est compatible avec les objets `{survey}` et peut donc être utilisée pour lister ou rechercher des variables.

```
t_titanic <- titanic |>
  labelled::set_variable_labels(
    Class = "Class du passager",
    Sex = "Sexe du passager",
    Age = "Enfant ou adulte ?",
    Survived = "A survécu au naufrage ?",
    n = "Nombre d'observations"
  ) |>
  srvyr::as_survey_design(weights = n)
t_titanic |> labelled::look_for()
```

pos	variable	label	col_type	missing values
1	Class	Class du passager	chr	0
2	Sex	Sexe du passager	chr	0
3	Age	Enfant ou adulte ?	chr	0
4	Survived	A survécu au naufrage ?	chr	0
5	n	Nombre d'observations	dbl	0

```
t_titanic |> labelled::look_for("nau")
```

pos	variable	label	col_type	missing values
4	Survived	A survécu au naufrage ?	chr	0

29.3 Extraire un sous-échantillon

Si l'on souhaite travailler sur un sous-échantillon de l'enquête, il importe de définir le plan d'échantillonnage sur l'ensemble du jeu de données **avant** de procéder à la sélection des observations.

La fonction classique pour sélectionner des lignes est `subset()`. Cependant, elle a un inconvénient lorsque nos données comportent des étiquettes de variables (cf. Chapitre 11) ou de valeurs (Chapitre 12), car les étiquettes ne sont pas conservées après l'opération.

On préférera donc avoir recours à `srvyr::filter()` qui conservent les attributs associés aux colonnes du tableau de données.

```
t_subset <- t_titanic |> subset(Sex == "Female")
t_subset |> labelled::look_for()
```

pos	variable	label	col_type	missing	values
1	Class	-	chr	0	
2	Sex	-	chr	0	
3	Age	-	chr	0	
4	Survived	-	chr	0	
5	n	-	dbl	0	

```
t_filter <- t_titanic |> filter(Sex == "Female")
t_filter |> labelled::look_for()
```

pos	variable	label	col_type	missing	values
1	Class	Class du passager	chr	0	
2	Sex	Sexe du passager	chr	0	
3	Age	Enfant ou adulte ?	chr	0	
4	Survived	A survécu au naufrage ?	chr	0	
5	n	Nombre d'observations	dbl	0	

30 Analyses uni- et bivariées pondérées

30.1 La fonction `tbl_svysummary()`

Dans les chapitres sur la statistique univariée (cf. Chapitre 18) et la statistique bivariée (cf. Chapitre 19), nous avons abordé la fonction `gtsummary::tbl_summary()` qui permet de générer des tris à plats et des tableaux croisés prêts à être publiés.

Son équivalent pour les objets `{survey}` existe : il s'agit de la fonction `gtsummary::tbl_svysummary()` qui fonctionne de manière similaire.

Pour illustrer son fonctionnement, nous allons utiliser le jeu de données *fecondite* fourni dans le package `{questionr}`. Ce jeu de données fournit un tableau de données `femmes` comportant une variable *poids* de pondération que nous allons utiliser. Les données catégorielles étant stockées sous forme de vecteurs numériques avec étiquettes de valeurs (cf. Chapitre 12), nous allons les convertir en facteurs avec `labelled::unlabelled()`. De même, certaines valeurs manquantes sont indiquées sous formes de *user NAs* (cf. Section 13.2) : nous allons les convertir en valeurs manquantes classiques (*regular NAs*) avec `labelled::user_na_to_na()`.

```
data("fecondite", package = "questionr")
library(srvyr)
```

Attachement du package : 'srvyr'

L'objet suivant est masqué depuis 'package:stats':

`filter`

```
dp <- femmes |>
  labelled::user_na_to_na() |>
  labelled::unlabelled() |>
  as_survey_design(weights = poids)
dp
```

Independent Sampling design (with replacement)

Called via srvyr

Sampling variables:

- ids: `1`
- weights: poids

Data variables: id_femme (dbl), id_menage (dbl), poids (dbl), date_entretien (date), date_naissance (date), age (dbl), milieu (fct), region (fct), educ (fct), travail (fct), matri (fct), religion (fct), journal (fct), radio (fct), tv (fct), nb_enf_ideal (dbl), test (fct)

Chargeons {gtsummary} et définissons le français comme langue de rendu des tableaux.

```
library(gtsummary)
theme_gtsummary_language(
  language = "fr",
  decimal.mark = ",",
  big.mark = ""
)
```

Setting theme `language: fr`

Pour réaliser un tableau croisé, il nous suffit d'appeler `gtsummary::tbl_svysummary()` de la même manière que l'on aurait procédé avec `gtsummary::tbl_summary()`. En arrière plan, `gtsummary::tbl_svysummary()` appellera les différentes fonctions statistiques de {survey} : la pondération ainsi que les spécificités du plan d'échantillonnage seront donc correctement prises en compte.

```
dp |>
  tbl_svysummary(
    by = milieu,
    include = c(age, educ, travail)
  ) |>
  add_overall(last = TRUE) |>
  bold_labels()
```

Warning: There were 5 warnings in `mutate()`.

The first warning was:

i In argument: `df_stats = pmap(...)`.

Caused by warning in `svymean.survey.design2()`:

! Sample size greater than population size: are weights correctly scaled?

i Run `dplyr::last_dplyr_warnings()` to see the 4 remaining warnings.

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

Table 30.1: Tableau croisé sur des données pondérées

Caractéristique	urbain, N = 1026	rural, N = 1002	Total, N = 2027
Âge révolu (en années) à la date de passation du questionnaire	26 (20 – 33)	28 (22 – 36)	27 (21 – 35)
Niveau d'éducation			
aucun	414 (40%)	681 (68%)	1095 (54%)
primaire	251 (24%)	257 (26%)	507 (25%)
secondaire	303 (30%)	61 (6,1%)	364 (18%)
supérieur	58 (5,7%)	3 (0,3%)	61 (3,0%)
A un emploi ?			
non	401 (39%)	269 (27%)	670 (33%)
oui	621 (61%)	731 (73%)	1351 (67%)
Manquant	5	1	6

! Important

Par défaut, les effectifs (ainsi que les pourcentages et autres statistiques) affichés sont pondérés. Il est important de bien comprendre ce que représentent ces effectifs pondérés pour les interpréter correctement. Pour cela, il faut savoir comment les poids de l'enquête ont été calculés.

Dans certains cas, lorsque la population totale est connue, la somme des poids est égale à cette population totale dans laquelle l'échantillon a été tiré au sort. Les effectifs pondérés représentent donc une estimation des effectifs dans la population totale et ne représentent en rien le nombre d'observations dans l'enquête.

Dans d'autres enquêtes, les poids sont générés de telle manière que la somme des poids correspondent au nombre total de personnes enquêtées. Dans ce genre de situation, on a souvent tendance, à tort, à interpréter les effectifs pondérés comme un nombre d'observations. Or, il peut y avoir un écart important entre le nombre d'observations dans l'enquête et les effectifs pondérés.

On pourra éventuellement présenter séparément le nombre d'observations (i.e. les effectifs non pondérés) et les proportions pondérées. `gtsummary::tbl_svsummary()` fournit justement à la fois ces données pondérées et non pondérées. Il est vrai que cela nécessite quand même quelques manipulations. Pour les cellules, on précisera le type d'effectifs à afficher avec l'argument `statistic`. Pour persona-

liser l’affiche du nombre de valeurs manquantes, cela doit se faire à un niveau plus global via `gtsummary::set_gtsummary_theme()`. Enfin, on passera par `gtsummary::modify_header()` pour personnaliser les en-têtes de colonne.

```
set_gtsummary_theme(
  list("tbl_summary-str:missing_stat" = "{N_miss_unweighted} obs.")
)
dp |>
  tbl_svysummary(
    by = milieu,
    include = c(educ, travail),
    statistic = all_categorical() ~ "{p}% ({n_unweighted} obs.)",
    digits = all_categorical() ~ c(1, 0)
  ) |>
  modify_header(
    all_stat_cols() ~ "**{level}** ({n_unweighted} obs.)"
  ) |>
  bold_labels()
```

Warning: There were 4 warnings in `mutate()`.

The first warning was:

i In argument: `df_stats = pmap(...)`.

Caused by warning in `svymean.survey.design2()`:

! Sample size greater than population size: are weights correctly scaled?

i Run `dplyr::last_dplyr_warnings()` to see the 3 remaining warnings.

Table printed with `knitr::kable()`, not {gt}. Learn why at

<https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>

To suppress this message, include `message = FALSE` in code chunk header.

Caractéristique	urbain (912 obs.)	rural (1088 obs.)
Niveau d’éducation		
aucun	40,3% (375 obs.)	68,0% (763 obs.)
primaire	24,4% (213 obs.)	25,6% (247 obs.)
secondaire	29,5% (275 obs.)	6,1% (73 obs.)
supérieur	5,7% (49 obs.)	0,3% (5 obs.)
A un emploi ?		
non	39,2% (370 obs.)	26,9% (296 obs.)
oui	60,8% (537 obs.)	73,1% (790 obs.)
Manquant	5 obs.	2 obs.

Il faut noter qu'une modification du thème impactera tous les tableaux suivants, jusqu'à ce que le thème soit à nouveau modifié ou bien que l'on fasse appel à `gtsummary::reset_gtsummary_theme()`.

30.2 Calcul manuel avec {survey}

Lorsque l'on travail avec un plan d'échantillonnage, on ne peut utiliser les fonctions statistiques classiques de **R**. On aura recours à leurs équivalents fournis par `{survey}` :

- `survey::svymean()`, `survey::svyvar()`, `survey::svytotal()`, `survey::svyquantile()` : moyenne, variance, total, quantiles
- `survey::svytable()` : tri à plat et tableau croisé
- `survey::svychisq()` : test du χ^2
- `survey::svyby()` : statistiques selon un facteur
- `survey::svyttest()` : test t de Student de comparaison de moyennes
- `survey::svyciprop()` : intervalle de confiance d'une proportion
- `survey::svyratio()` : ratio de deux variables continues

Ces fonctions prennent leurs arguments sous forme de formules pour spécifier les variables d'intérêt.

```
survey::svymean(~ age, dp)
```

	mean	SE
age	28.468	0.2697

```
survey::svymean(~ age, dp) |> confint()
```

	2.5 %	97.5 %
age	27.93931	28.99653

```
survey::svyquantile(~age, dp, quantile = c(0.25, 0.5, 0.75), ci = TRUE)
```

```
$age
      quantile ci.2.5 ci.97.5      se
0.25         21      21      22 0.2549523
0.5          27      27      28 0.2549523
0.75         35      35      37 0.5099045
```

```
attr("hasci")
[1] TRUE
attr("class")
[1] "newsvyquantile"
```

Les tris à plat se déclarent en passant comme argument le nom de la variable précédé d'un tilde (~), tandis que les tableaux croisés utilisent les noms des deux variables séparés par un signe plus (+) et précédés par un tilde (~)¹.

```
survey::svytable(~region, dp)
```

```
region
      Nord      Est      Sud      Ouest
611.0924 175.7404 329.2220 911.2197
```

```
survey::svytable(~milieu + educ, dp)
```

```
educ
milieu      aucun      primaire secondaire      supérieur
urbain 413.608780 250.665214 303.058978 58.412688
rural  681.131096 256.694363 61.023980 2.679392
```

La fonction `questionr::freq()` peut être utilisée si on lui passe en argument non pas la variable elle-même, mais son tri à plat obtenu avec `survey::svytable()` :

```
survey::svytable(~region, dp) |>
  questionr::freq(total = TRUE)
```

```
      n      %  val%
Nord  611.1 30.1 30.1
Est   175.7  8.7  8.7
Sud   329.2 16.2 16.2
Ouest 911.2 44.9 44.9
Total 2027.3 100.0 100.0
```

¹Cette syntaxe est similaire à celle de `xtabs()`.

Les fonctions `questionr::rprop()` et `questionr::cprop()` peuvent être utilisées pour calculer les pourcentages en ligne ou en colonne.

```
survey::svytable(~milieu + educ, dp) |>
  questionr::cprop()
```

	educ				
milieu	aucun	primaire	secondaire	supérieur	Ensemble
urbain	37.8	49.4	83.2	95.6	50.6
rural	62.2	50.6	16.8	4.4	49.4
Total	100.0	100.0	100.0	100.0	100.0

Le principe de la fonction `survey::svyby()` est similaire à celui de `tapply()` (cf. Section 19.2.3). Elle permet de calculer des statistiques selon plusieurs sous-groupes définis par un facteur.

```
survey::svyby(~age, ~region, dp, survey::svymean)
```

	region	age	se
Nord	Nord	29.03299	0.4753268
Est	Est	27.54455	0.5261669
Sud	Sud	28.96830	0.6148223
Ouest	Ouest	28.08626	0.4458201

30.3 Intervalles de confiance et tests statistiques

La fonction `gtsummary::add_ci()` peut être appliquée à des tableaux produits avec `gtsummary::tbl_svysummary()`². Les méthodes utilisées sont adaptées à la prise en compte d'un plan d'échantillonnage. On se référera à la document de la fonction pour plus de détails sur les méthodes statistiques utilisées. **Rappel :** pour les variables continues, on sera vigilant à ce que la statistique affichée (médiane par défaut) corresponde au type d'intervalle de confiance calculé (moyenne par défaut).

```
dp |>
  tbl_svysummary(
    include = c(age, region),
    statistic = all_continuous() ~ "{mean} ({sd})"
  ) |>
```

²Cela requiert une version récente (1.7.0) de `gtsummary`.

```
add_ci() |>
bold_labels()
```

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

Table 30.3: Intervalles de confiance avec prise en compte du plan d'échantillonnage

Caractéristique	N = 2027	95% CI
Âge révolu (en années) à la date de passation du questionnaire	28 (9)	28, 29
Région de résidence		
Nord	611 (30%)	28%, 33%
Est	176 (8,7%)	7,7%, 9,8%
Sud	329 (16%)	14%, 18%
Ouest	911 (45%)	42%, 48%

De même, on peut aisément effectuer des tests de comparaison avec `gtsummary::add_p()`. Là aussi, les tests utilisés sont des adaptations des tests classiques avec différentes corrections pour tenir compte à la fois de la pondération et du plan d'échantillonnage.

```
dp |>
tbl_svsummary(
  include = c(age, region),
  by = milieu
) |>
add_p() |>
bold_labels()
```

Warning: There were 3 warnings in ``mutate()``.
The first warning was:
i In argument: ``df_stats = pmap(...)``.
Caused by warning in ``svymean.survey.design2()``:
! Sample size greater than population size: are weights correctly scaled?
i Run ``dplyr::last_dplyr_warnings()`` to see the 2 remaining warnings.

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

Table 30.4: Tests de comparaison avec prise en compte du plan d'échantillonnage

Caractéristique	urbain, N = 1026	rural, N = 1002	p-valeur
Âge révolu (en années) à la date de passation du questionnaire	26 (20 – 33)	28 (22 – 36)	<0,001
Région de résidence			<0,001
Nord	265 (26%)	346 (35%)	
Est	48 (4,7%)	128 (13%)	
Sud	79 (7,7%)	250 (25%)	
Ouest	633 (62%)	278 (28%)	

30.4 Calcul manuel avec {srvyr}

On peut avoir besoin de calculer des moyennes et/ou des proportions par sous-groupe, avec leurs intervalles de confiance, de manière manuelle, par exemple en amont d'un graphique à représenter avec {ggplot2}. Dans ce cas de figure, les fonctions natives de {survey} ne sont pas toujours très facile d'emploi et l'on pourra avantageusement recourir à {srvyr} pour bénéficier d'une syntaxe à la {dplyr}.

Par exemple, pour calculer l'âge moyen des femmes par région, on combinera `srvyr::survey_mean()` avec `srvyr::group_by()` et `srvyr::summarise()` :

```
dp |>
  group_by(region) |>
  summarise(moy = survey_mean())
```

```
# A tibble: 4 x 3
  region    moy moy_se
  <fct>   <dbl> <dbl>
1 Nord   0.301  0.0127
2 Est    0.0867 0.00544
3 Sud    0.162  0.0102
4 Ouest  0.449  0.0147
```

Par défaut, cela renvoie les moyennes et les erreurs standards. Pour les intervalles de confiance, on précisera simplement `vartype = "ci"`.

```
dp |>
  group_by(region) |>
  summarise(moy = survey_mean(vartype = "ci"))
```

```
# A tibble: 4 x 4
  region    moy moy_low moy_upp
  <fct>    <dbl>   <dbl>   <dbl>
1 Nord    0.301    0.277    0.326
2 Est     0.0867   0.0760   0.0974
3 Sud     0.162    0.142    0.182
4 Ouest   0.449    0.421    0.478
```

Pour des proportions, on aura recours à `srvyr::survey_prop()`. Par exemple, pour un tri à plat du niveau d'éducation :

```
dp |>
  group_by(educ) |>
  summarise(prop = survey_prop())
```

When ``proportion`` is unspecified, ``survey_prop()`` now defaults to ``proportion = TRUE``.
 i This should improve confidence interval coverage.
 This message is displayed once per session.

```
# A tibble: 4 x 3
  educ      prop prop_se
  <fct>    <dbl>   <dbl>
1 aucun    0.540  0.0144
2 primaire 0.250  0.0127
3 secondaire 0.180  0.0110
4 supérieur 0.0301 0.00487
```

Là encore, on peut passer l'option `vartpe = "ci"` pour obtenir les intervalles de confiance³.

Si l'on passe plusieurs variables dans le `group_by()`, les proportions sont calculées pour la dernière variable pour chaque combinaison des autres variables. Par exemple, pour la distribution du niveau d'éducation par milieu de résidence (i.e. la somme des proportions est de 100% pour le milieu urbain et de 100% pour celles du milieu rural, soit 200% au total) :

```
dp |>
  group_by(milieu, educ) |>
  summarise(prop = survey_prop(vartype = "ci", proportion = TRUE))
```

³Par défaut, les intervalles de confiance sont calculés avec `survey::svymean()` et peuvent générer des valeurs inférieures à 0 ou supérieures à 1. Pour un calcul plus précis reposant sur `survey::svyciprop()`, on précisera `proportion = TRUE`. Plusieurs méthodes existent pour ce calcul, voir l'aide de `survey::svyciprop()`.

```
# A tibble: 8 x 5
# Groups:   milieu [2]
  milieu educ      prop prop_low prop_upp
  <fct> <fct>    <dbl>   <dbl>   <dbl>
1 urbain aucun    0.403    0.364    0.444
2 urbain primaire 0.244    0.210    0.282
3 urbain secondaire 0.295    0.260    0.334
4 urbain supérieur 0.0569   0.0410   0.0785
5 rural  aucun    0.680    0.643    0.715
6 rural  primaire 0.256    0.223    0.292
7 rural  secondaire 0.0609   0.0454   0.0813
8 rural  supérieur 0.00268  0.00108  0.00660
```

Si l'on souhaite les pourcentages que représentent chaque combinaison au sein de l'ensemble de l'échantillon (i.e. que la somme de toutes les proportions soit de 100%), on aura recours à `srvyr::interact()`.

```
dp |>
  group_by(interact(milieu, educ)) |>
  summarise(prop = survey_prop(vartype = "ci", proportion = TRUE))
```

```
# A tibble: 8 x 5
  milieu educ      prop prop_low prop_upp
  <fct> <fct>    <dbl>   <dbl>   <dbl>
1 urbain aucun    0.204    0.182    0.228
2 urbain primaire 0.124    0.105    0.145
3 urbain secondaire 0.149    0.130    0.171
4 urbain supérieur 0.0288   0.0207   0.0400
5 rural  aucun    0.336    0.310    0.363
6 rural  primaire 0.127    0.109    0.146
7 rural  secondaire 0.0301   0.0224   0.0404
8 rural  supérieur 0.00132  0.000535 0.00326
```

30.5 Impact du plan d'échantillonnage

Lorsque l'on calcul des proportions, moyennes ou médianes pondérées, seuls les poids entrent en ligne de compte. Le plan d'échantillonnage (strates et/ou grappes) n'a de son côté pas d'effet. Par contre, le plan d'échantillonnage a un impact important sur le calcul des variances et, par extension, sur le calcul des intervalles de confiance et des tests de comparaison.

Pour illustrer cela, nous allons considérer un même jeu de données, avec la même variable de poids, mais en faisant varier la présence de strates et de grappes.

Commençons par regarder le jeu de données *apistrat* fourni par `{survey}`.

```
data("api", package = "survey")
nrow(apistrat)
```

```
[1] 200
```

```
summary(apistrat$pw)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
15.10	19.05	32.28	30.97	44.21	44.21

```
sum(apistrat$pw)
```

```
[1] 6194
```

Nous avons ici un tableau de données de 200 lignes, avec des poids variant entre 15 et 44. Nous pouvons définir une pondération simple et croiser deux variables.

```
d_ponderation_simple <- apistrat |>
  as_survey_design(weights = pw)
tbl <- survey::svytable(~ awards + yr.rnd, design = d_ponderation_simple)
tbl
```

	yr.rnd	
awards	No	Yes
No	2068.34	168.09
Yes	3274.06	683.51

Réalisons un test du Chi² entre ces deux variables. Si nous appliquons la fonction classique `chisq.test()` sur ce tableau, cette fonction considérerait que nous avons 6194 observations (somme des poids) et dès lors nous obtiendrions une p-valeur très faible.

```
tbl |> chisq.test()
```

Pearson's Chi-squared test with Yates' continuity correction

```
data: tbl
X-squared = 113.84, df = 1, p-value < 2.2e-16
```

Le calcul précédent ne tient pas compte que nous n'avons que 200 observations dans notre échantillon. Refaisons le calcul `survey::svychisq()` qui est adaptée aux plans d'échantillonnage.

```
survey::svychisq(~ awards + yr.rnd, design = d_ponderation_simple)
```

Pearson's X^2 : Rao & Scott adjustment

```
data: NextMethod()
F = 2.9162, ndf = 1, ddf = 199, p-value = 0.08926
```

Le résultat est ici tout autre et notre test n'est plus significatif au seuil de 5% ! Ici, les corrections de Rao & Scott permettent justement de tenir compte que nous avons un échantillon de seulement 200 observations.

Regardons maintenant si, à poids égal, il y a une différence entre une enquête stratifiée et une enquête en grappes.

```
# Pondération simple
survey::svytable(~ awards + yr.rnd, design = d_ponderation_simple)
```

	yr.rnd	
awards	No	Yes
No	2068.34	168.09
Yes	3274.06	683.51

```
survey::svychisq(~ awards + yr.rnd, design = d_ponderation_simple)
```

Pearson's X^2 : Rao & Scott adjustment

```
data: NextMethod()
F = 2.9162, ndf = 1, ddf = 199, p-value = 0.08926
```

```
# Enquête stratifiée
d_strates <- apistrat |>
  as_survey_design(weights = pw, strata = stype)
survey::svytable(~ awards + yr.rnd, design = d_strates)
```

	yr.rnd	
awards	No	Yes
No	2068.34	168.09
Yes	3274.06	683.51

```
survey::svychisq(~ awards + yr.rnd, design = d_strates)
```

Pearson's X^2 : Rao & Scott adjustment

```
data: NextMethod()
F = 2.9007, ndf = 1, ddf = 197, p-value = 0.09012
```

```
# Enquête en grappes
d_grappes <- apistrat |>
  as_survey_design(weights = pw, ids = dnum)
survey::svytable(~ awards + yr.rnd, design = d_grappes)
```

	yr.rnd	
awards	No	Yes
No	2068.34	168.09
Yes	3274.06	683.51

```
survey::svychisq(~ awards + yr.rnd, design = d_grappes)
```

Pearson's X^2 : Rao & Scott adjustment

```
data: NextMethod()
F = 3.1393, ndf = 1, ddf = 134, p-value = 0.0787
```

On le constate : dans les trois cas les tableaux croisés sont identiques, mais pour autant les trois p-valeurs diffèrent.

Dès lors qu'un calcul de variance est impliqué, la simple prise en compte des poids est insuffisante : il faut appliquer des corrections en fonction du plan d'échantillonnage !

Pas d'inquiétude, `{survey}` s'en occupe pour vous, dès lors que le plan d'échantillonnage a correctement été défini.

31 Graphiques pondérés

Le package `{ggplot2}` n'est compatible directement avec les objets `{survey}`. Cependant, il accepte une esthétique *weight* qui permet de définir une variable de pondération.

Avertissement

ATTENTION : les graphiques obtenus ne sont corrects qu'à la condition que seuls les poids soient nécessaires pour les construire, ce qui est le cas d'un nuage de points ou d'un diagramme en barres.

Par contre, si le calcul du graphique implique le calcul de variances, la représentation sera incorrecte. Par exemple, avec `ggplot2::geom_smooth()`, les intervalles de confiance affichés ne prendront pas correctement en compte le plan d'échantillonnage.

Reprenons le jeu de données *fecondite* que nous avons abordé dans le chapitre sur les analyses bivariées pondérées, cf. Chapitre 30. Les poids d'enquête y sont indiqués dans la colonne *poids*. Pour rappel, les données catégorielles étant stockées sous forme de vecteurs numériques avec étiquettes de valeurs (cf. Chapitre 12), nous allons les convertir en facteurs avec `labelled::unlabelled()`.

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

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr      1.1.4      v readr      2.1.5
v forcats    1.0.0      v stringr    1.5.1
v ggplot2    3.5.1      v tibble     3.2.1
v lubridate  1.9.3      v tidyr      1.3.1
v purrr      1.0.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
d <- labelled::unlabelled(femmes)
```


Pour réaliser un graphique, nous pouvons reprendre ce que nous avons vu dans notre chapitre introductif sur `{ggplot2}`, cf. Chapitre 17, en spécifiant simplement l'esthétique *weight*.

```
ggplot(d) +  
  aes(x = region, fill = test, weight = poids) +  
  geom_bar(position = "fill")
```

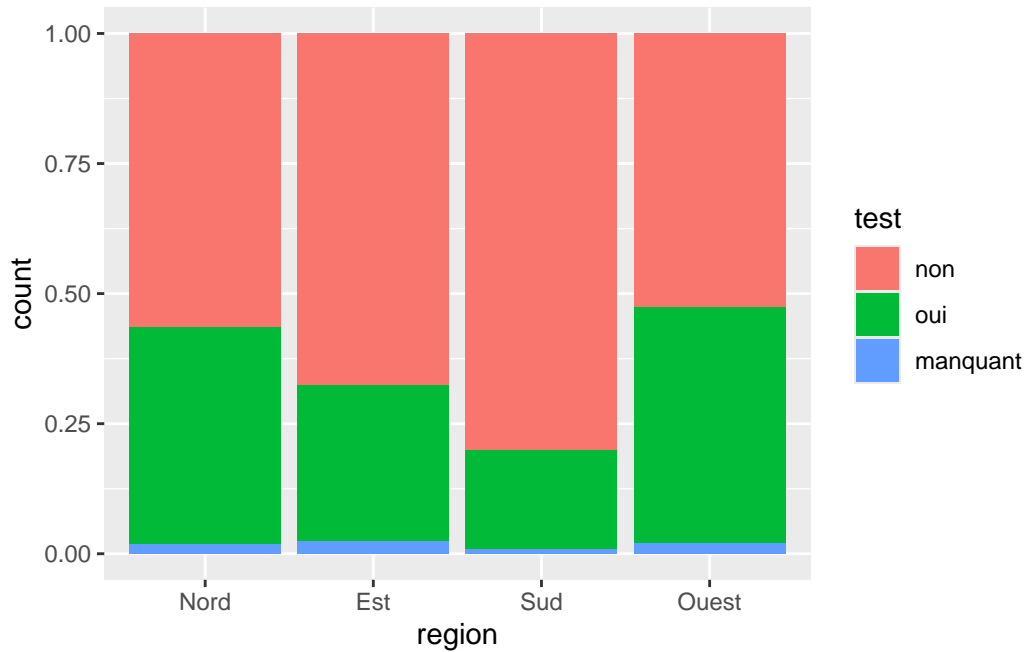


Figure 31.1: Un graphique en barres pondéré

Si l'on a déjà créé un objet `{survey}`, on peut obtenir les poids des observations avec la fonction `weights()`. Les données sont quant à elle accessibles via le sous-élément nommé *variables*.

```
library(srvyr)
```

Attachement du package : 'srvyr'

L'objet suivant est masqué depuis 'package:stats':

```
filter
```

```
dp <- femmes |>
  labelled::unlabelled() |>
  as_survey_design(weights = poids)
```

```
ggplot(dp$variables) +
  aes(x = region, fill = test, weight = weights(dp)) +
  geom_bar(position = "fill")
```

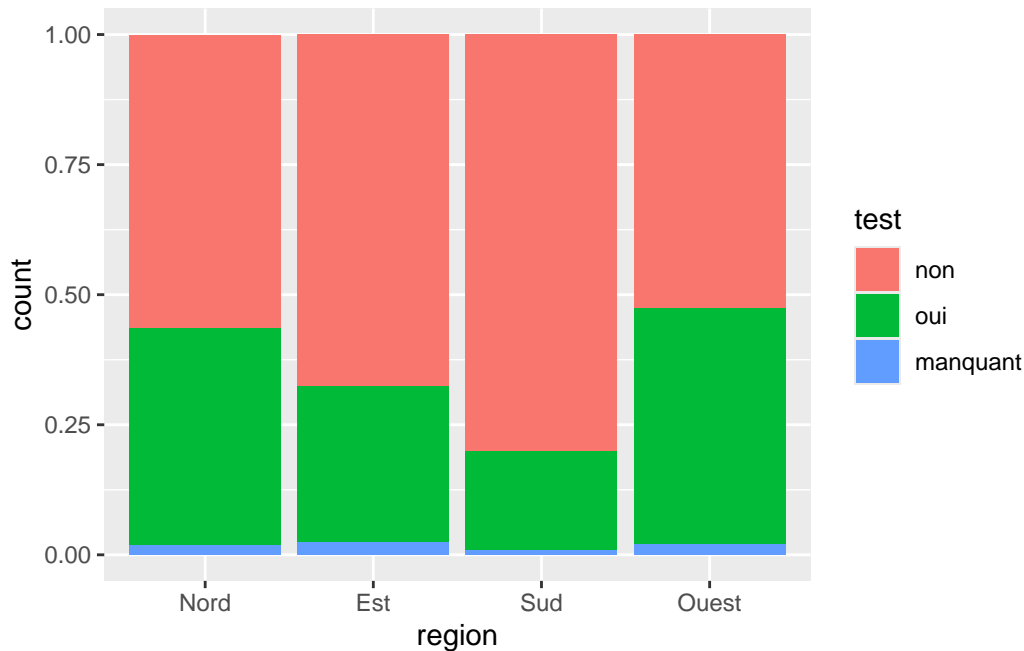


Figure 31.2: Un graphique en barres pondéré

Pour se faciliter les choses, on peut avoir directement recours à la fonction `ggstats::ggsurvey()`, que l'on utilisera à la place de `ggplot2::ggplot()`, et qui fait exactement la même chose que dans notre exemple précédent : on lui passe un objet de type `{survey}` et la fonction en extrait le sous-élément `variables` pour le passer à `ggplot2::ggplot()` et les poids qui sont automatiquement associés à l'esthétique `weight`.

Ainsi, le code de notre graphique précédent s'écrit tout simplement¹ :

¹Notez que les poids ont déjà été associés à la bonne esthétique et qu'il n'est donc pas nécessaire de le refaire dans l'appel à `aes()`.

```
ggstats::ggsurvey(dp) +
  aes(x = region, fill = test) +
  geom_bar(position = "fill")
```

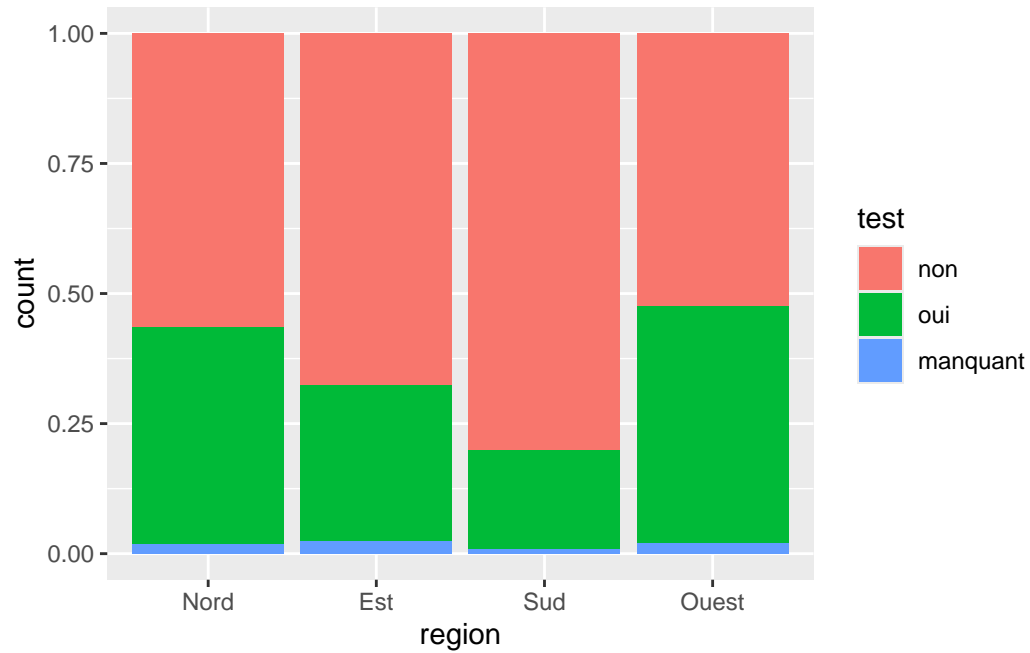


Figure 31.3: Un graphique en barres pondéré

32 Régression logistique binaire pondérée

Nous avons abordé la régression logistique binaire non pondérée dans un chapitre dédié, cf. Chapitre 22. Elle se réalise classiquement avec la fonction `glm()` en spécifiant `family = binomial`.

Lorsque l'on utilise des données d'enquêtes, l'approche est similaire sauf que l'on aura recours à la fonction `survey::svyglm()` qui sait gérer des objets `{survey}` : non seulement la pondération sera prise en compte, mais le calcul des intervalles de confiance et des p-valeurs sera ajusté en fonction du plan d'échantillonnage.

32.1 Données des exemples

Nous allons reprendre les mêmes données issues de l'enquête *Histoire de vie 2003*, mais en tenant compte cette fois-ci des poids de pondération fournis dans la variable *poids*.

```
library(tidyverse)
library(labelled)
data(hdv2003, package = "questionr")
d <-
  hdv2003 |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
```

```

    "Secondaire" = "1er cycle",
    "Secondaire" = "2eme cycle",
    "Technique / Professionnel" = "Enseignement technique ou professionnel court",
    "Technique / Professionnel" = "Enseignement technique ou professionnel long",
    "Supérieur" = "Enseignement superieur y compris technique superieur"
  ) |>
  fct_na_value_to_level("Non documenté")
) |>
set_variable_labels(
  sport = "Pratique un sport ?",
  sexe = "Sexe",
  groupe_ages = "Groupe d'âges",
  etudes = "Niveau d'études",
  relig = "Rapport à la religion",
  heures.tv = "Heures de télévision / jour",
  poids = "Pondération de l'enquête"
)

```

Il ne nous reste qu'à définir notre objet `{survey}` en spécifiant la pondération fournie avec l'enquête. La documentation ne mentionne ni strates ni grappes.

```

library(srvyr)
library(survey)
dp <- d |>
  as_survey_design(weights = poids)

```

32.2 Calcul de la régression logistique binaire

La syntaxe de `survey::svyglm()` est similaire à celle de `glm()` sauf qu'elle a un argument `design` au lieu de `data`.

La plupart du temps, les poids de pondération ne sont pas des nombres entiers, mais des nombres décimaux. Or, la famille de modèles binomiaux repose sur des nombres entiers de succès et d'échecs. Avec une version récente¹ de **R**, cela n'est pas problématique. Nous aurons simplement un avertissement.

¹Si vous utilisez une version ancienne de **R**, cela n'était tout simplement pas possible. Vous obteniez un message d'erreur et le modèle n'était pas calculé. Si c'est votre cas, optez pour un modèle quasi-binomial ou bien mettez à jour **R**.

```
mod_binomial <- svyglm(
  sport ~ sexe + groupe_ages + etudes + relig + heures.tv,
  family = binomial,
  design = dp
)
```

Warning in eval(family\$initialize): nombre de succès non entier dans un glm binomial !

Une alternative consiste à avoir recours à la famille quasi-binomiale, que l'on spécifie avec `family = quasibinomial` et qui constitue une extension de la famille binomiale pouvant gérer des poids non entiers. La distribution quasi-binomiale, bien que similaire à la distribution binomiale, possède un paramètre supplémentaire qui tente de décrire une variance supplémentaire dans les données qui ne peut être expliquée par une distribution binomiale seule (on parle alors de *surdispersion*). Les coefficients obtenus sont les mêmes, mais les intervalles de confiance peuvent être un peu plus large.

```
mod_quasi <- svyglm(
  sport ~ sexe + groupe_ages + etudes + relig + heures.tv,
  family = quasibinomial,
  design = dp
)
```

Simple, non ?

32.3 Sélection de modèle

Comme précédemment (cf. Chapitre 23), il est possible de procéder à une sélection de modèle pas à pas, par minimisation de l'AIC, avec `step()`.

```
mod_quasi2 <- step(mod_quasi)
```

Start: AIC=2309.89

```
sport ~ sexe + groupe_ages + etudes + relig + heures.tv
```

	Df	Deviance	AIC
- relig	5	2266.3	2302.2
<none>		2263.9	2309.9
- heures.tv	1	2276.2	2320.2

```
- sexe          1    2276.4 2320.4
- groupe_ages  3    2313.9 2353.8
- etudes       4    2383.5 2421.2
```

Step: AIC=2296.28

```
sport ~ sexe + groupe_ages + etudes + heures.tv
```

	Df	Deviance	AIC
<none>		2266.3	2296.3
- heures.tv	1	2278.4	2306.4
- sexe	1	2279.0	2307.0
- groupe_ages	3	2318.3	2342.1
- etudes	4	2387.2	2408.8

⚠ Sélection pas à pas et valeurs manquantes

Nous avons abordé dans le chapitre sur la sélection de modèle pas à pas la problématique des valeurs manquantes lors d'une sélection pas à pas descendante par minimisation de l'AIC (cf. Section 23.8). La même approche peut être appliquée avec des données pondérées. Cependant, la fonction `step_with_na()` que nous avons présenté n'est pas compatible avec les modèles `survey::svyglm()` puisqu'ils prennent en entrée un argument `design` et non `data`.

On pourra essayer la variante `step_with_na_survey()` ci-dessous qui nécessite qu'on lui passe également l'objet `{survey}` ayant servi au calcul du modèle.

```
step_with_na_survey <- function(model, design, ...) {
  # list of variables
  variables <- broom.helpers::model_list_variables(
    model,
    only_variable = TRUE
  )
  # design with no na
  design_no_na <- design |>
    srvyr::drop_na(dplyr::any_of(variables))
  # refit the model without NAs
  model_no_na <- update(model, data = design_no_na)
  # apply step()
  model_simplified <- step(model_no_na, ...)
  # recompute simplified model using full data
  update(model, formula = terms(model_simplified))
}
```

```
mod2_binomial <- step_with_na_survey(mod_binomial, dp)
```

32.4 Affichage des résultats

Nous pouvons tout à fait utiliser `gtsummary::tbl_regression()` avec ce type de modèles. De même, on peut utiliser `gtsummary::add_global_p()` pour calculer les p-valeurs globales des variables ou encore `gtsummary::add_vif()` pour vérifier la multicollinéarité (cf. Chapitre 27).

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",", big.mark = " ")
```

```
mod_quasi2 |>
  tbl_regression(exponentiate = TRUE) |>
  add_global_p(keep = TRUE) |>
  add_vif() |>
  bold_labels()
```

```
Warning in printHypothesis(L, rhs, names(b)): one or more coefficients in the hypothesis include
  arithmetic operators in their names;
  the printed representation of the hypothesis will be omitted
Warning in printHypothesis(L, rhs, names(b)): one or more coefficients in the hypothesis include
  arithmetic operators in their names;
  the printed representation of the hypothesis will be omitted
```

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

Table 32.1: Facteurs associés à la pratique d'un sport (régression logistique pondérée)

Caractéristique	OR	95% IC	p-valeur	GVIF	Adjusted GVIF
Sexe			0,005	1,0	1,0
Femme	—	—			
Homme	1,44	1,12 – 1,87	0,005		
Groupe d'âges			<0,001	2,1	1,1

Table 32.1: Facteurs associés à la pratique d'un sport (régression logistique pondérée)

Caractéristique	OR	95% IC	p-valeur	GVIF	Adjusted GVIF
18-24 ans	—	—			
25-44 ans	0,85	0,48 – 1,51	0,6		
45-64 ans	0,40	0,22 – 0,73	0,003		
65 ans et plus	0,37	0,19 – 0,72	0,004		
Niveau d'études			<0,001	2,2	1,1
Primaire	—	—			
Secondaire	2,66	1,62 – 4,38	<0,001		
Technique / Professionnel	3,09	1,90 – 5,00	<0,001		
Supérieur	6,54	3,99 – 10,7	<0,001		
Non documenté	10,3	4,60 – 23,0	<0,001		
Heures de télévision / jour	0,89	0,82 – 0,97	0,006	1,1	1,0

Pour un graphique des coefficients, nous pouvons utiliser `ggstats::ggcoef_model()`.

```
mod_quasi2 |>
  ggstats::ggcoef_model(exponentiate = TRUE)
```

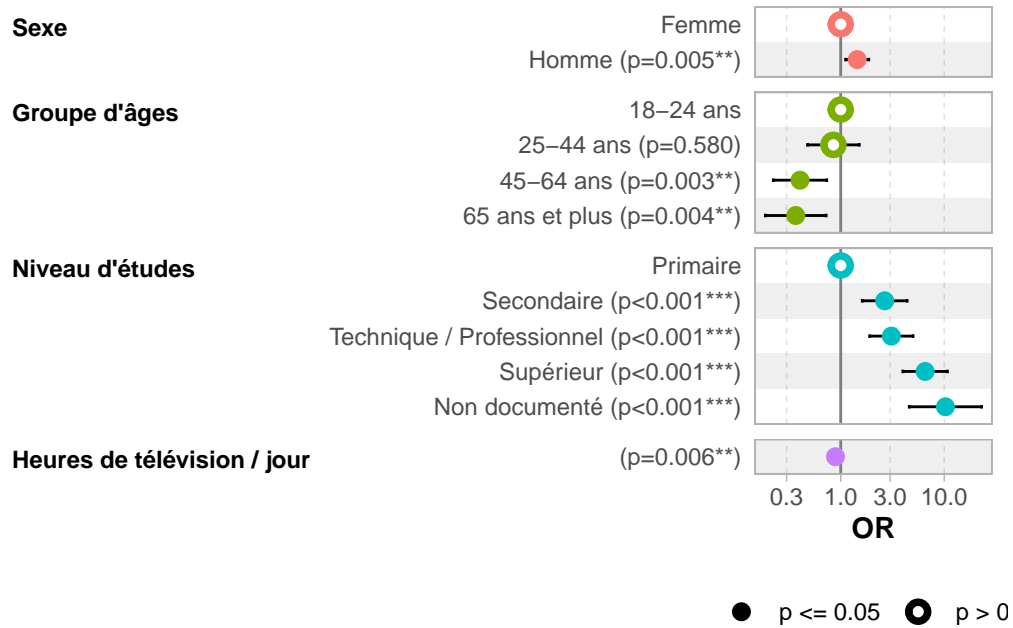


Figure 32.1: Facteurs associés à la pratique d'un sport (régression logistique pondérée)

32.5 Prédictions marginales

Pour visualiser les prédictions marginales moyennes du modèle (cf. Section 24.3), nous pouvons utiliser `broom.helpers::plot_marginal_predictions()`.

```
mod_quasi2 |>
  broom.helpers::plot_marginal_predictions(type = "response") |>
  patchwork::wrap_plots() &
  scale_y_continuous(
    limits = c(0, .8),
    labels = scales::label_percent()
  )
```

Warning: With models of this class, it is normally good practice to specify weights using the `wts` argument. Otherwise, weights will be ignored in the computation of quantities of interest.

Warning: With models of this class, it is normally good practice to specify weights using the `wts` argument. Otherwise, weights will be ignored in the computation of quantities of interest.

Warning: With models of this class, it is normally good practice to specify weights using the `wts` argument. Otherwise, weights will be ignored in the computation of quantities of interest.

Warning: With models of this class, it is normally good practice to specify weights using the `wts` argument. Otherwise, weights will be ignored in the computation of quantities of interest.

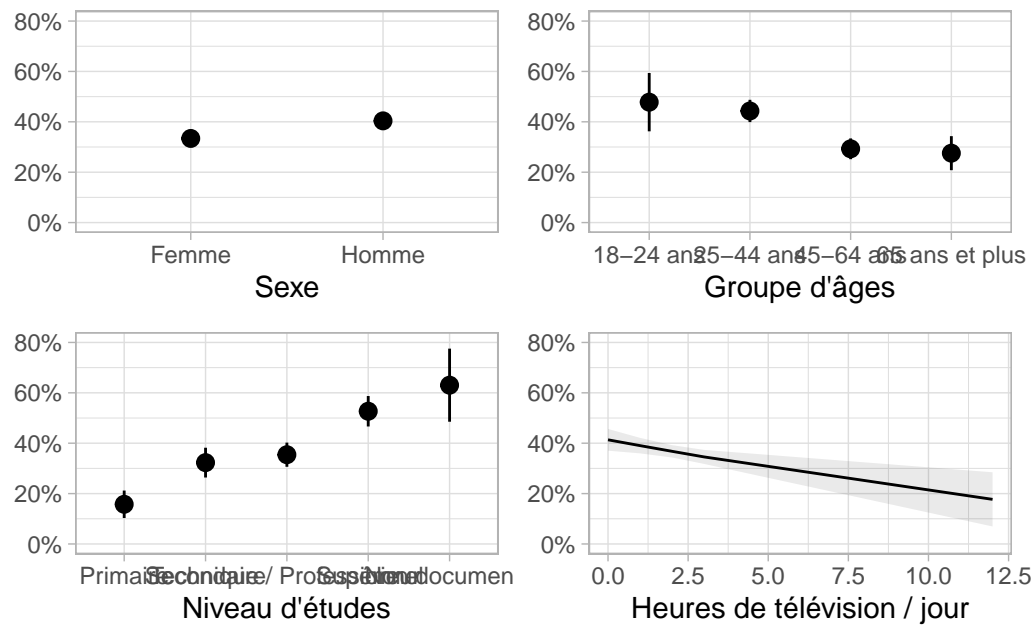


Figure 32.2: Prédictions marginales moyennes du modèle pondéré

partie V

Manipulation avancée

33 Fusion de tables

Il est fréquent d’avoir à gérer des données réparties dans plusieurs tables de données, notamment lorsque l’on a une enquête réalisée à différents niveaux (par exemple, un questionnaire ménage et un questionnaire individu) ou des données longitudinales.

On peut distinguer deux types d’actions :

- l’ajout de variables (jointure entre tables)
- l’ajout d’observations (concaténation de tables)

33.1 Jointures avec dplyr

Le jeu de données `{nycflights13}` est un exemple de données réparties en plusieurs tables. Ici on en a trois : les informations sur les vols, celles sur les aéroports et celles sur les compagnies aériennes sont dans trois tables distinctes.

`{dplyr}` propose différentes fonctions permettant de travailler avec des données structurées de cette manière.

```
library(tidyverse)
library(nycflights13)
data(flights)
data(airports)
data(airlines)
```

33.1.1 Clés implicites

Lorsque les données sont réparties dans plusieurs tables différentes, il est essentiel de repérer les identifiants permettant de naviguer d’une table à l’autre. Dans notre exemple, on peut voir que la table `flights` contient le code de la compagnie aérienne du vol dans la variable *carrier* :

```
flights |> labelled::look_for()
```

pos	variable	label	col_type	missing	values
1	year	-	int	0	
2	month	-	int	0	
3	day	-	int	0	
4	dep_time	-	int	8255	
5	sched_dep_time	-	int	0	
6	dep_delay	-	dbl	8255	
7	arr_time	-	int	8713	
8	sched_arr_time	-	int	0	
9	arr_delay	-	dbl	9430	
10	carrier	-	chr	0	
11	flight	-	int	0	
12	tailnum	-	chr	2512	
13	origin	-	chr	0	
14	dest	-	chr	0	
15	air_time	-	dbl	9430	
16	distance	-	dbl	0	
17	hour	-	dbl	0	
18	minute	-	dbl	0	
19	time_hour	-	dtm	0	

Et que par ailleurs la table `airlines` contient une information supplémentaire relative à ces compagnies, à savoir le nom complet.

```
airlines |> labelled::look_for()
```

pos	variable	label	col_type	missing	values
1	carrier	-	chr	0	
2	name	-	chr	0	

Il est donc naturel de vouloir associer les deux, en l'occurrence pour ajouter les noms complets des compagnies à la table `flights`. Dans ce cas on va faire une *jointure* : les lignes d'une table seront associées à une autre en se basant non pas sur leur position, mais sur les valeurs d'une ou plusieurs colonnes. Ces colonnes sont appelées des *clés*.

Pour faire une jointure de ce type, on va utiliser la fonction `dplyr::left_join()` :

```
fusion <- flights |> left_join(airlines)
```

Joining with ``by = join_by(carrier)``

Pour faciliter la lecture, on va afficher seulement certaines colonnes du résultat et les premières lignes de la table :

```
fusion |>
  select(month, day, carrier, name) |>
  head(10)
```

```
# A tibble: 10 x 4
  month   day carrier name
  <int> <int> <chr>   <chr>
1     1     1    UA   United Air Lines Inc.
2     1     1    UA   United Air Lines Inc.
3     1     1    AA   American Airlines Inc.
4     1     1    B6   JetBlue Airways
5     1     1    DL   Delta Air Lines Inc.
6     1     1    UA   United Air Lines Inc.
7     1     1    B6   JetBlue Airways
8     1     1    EV   ExpressJet Airlines Inc.
9     1     1    B6   JetBlue Airways
10    1     1    AA   American Airlines Inc.
```

On voit que la table obtenue est bien la fusion des deux tables d'origine selon les valeurs des deux colonnes clés *carrier*. On est parti de la table `flights`, et pour chaque ligne on a ajouté les colonnes de `airlines` pour lesquelles la valeur de *carrier* est la même. On a donc bien une nouvelle colonne `name` dans notre table résultat, avec le nom complet de la compagnie aérienne.

Note

Nous sommes ici dans le cas le plus simple concernant les clés de jointure : les deux clés sont uniques et portent le même nom dans les deux tables. Par défaut, si on ne lui spécifie pas explicitement les clés, `{dplyr}` fusionne en utilisant l'ensemble des colonnes communes aux deux tables. On peut d'ailleurs voir dans cet exemple qu'un message a été affiché précisant que la jointure s'est faite sur la variable *carrier*.

33.1.2 Clés explicites

La table `airports`, elle, contient des informations supplémentaires sur les aéroports : nom complet, altitude, position géographique, etc. Chaque aéroport est identifié par un code contenu dans la colonne *faa*.

Si on regarde la table `flights`, on voit que le code d'identification des aéroports apparaît à deux endroits différents : pour l'aéroport de départ dans la colonne *origin*, et pour celui d'arrivée dans la colonne *dest*. On a donc deux clés de jointures possibles, et qui portent un nom différent de la clé de `airports`.

On va commencer par fusionner les données concernant l'aéroport de départ. Pour simplifier l'affichage des résultats, on va se contenter d'un sous-ensemble des deux tables :

```
flights_ex <- flights |> select(month, day, origin, dest)
airports_ex <- airports |> select(faa, alt, name)
```

Si on se contente d'un `dplyr::left_join()` comme à l'étape précédente, on obtient un message d'erreur car aucune colonne commune ne peut être identifiée comme clé de jointure :

```
flights_ex |> left_join(airports_ex)
```

```
Error in `left_join()` :
! `by` must be supplied when `x` and `y` have no common variables.
i Use `cross_join()` to perform a cross-join.
```

On doit donc spécifier explicitement les clés avec l'argument `by` de `dplyr::left_join()`. Ici la clé est nommée *origin* dans la première table, et *faa* dans la seconde. La syntaxe est donc la suivante :

```
flights_ex |>
  left_join(airports_ex, by = c("origin" = "faa")) |>
  head(10)
```

```
# A tibble: 10 x 6
  month   day origin dest    alt name
  <int> <int> <chr>  <chr> <dbl> <chr>
1     1     1 EWR    IAH     18 Newark Liberty Intl
2     1     1 LGA    IAH     22 La Guardia
3     1     1 JFK    MIA     13 John F Kennedy Intl
4     1     1 JFK    BQN     13 John F Kennedy Intl
5     1     1 LGA    ATL     22 La Guardia
6     1     1 EWR    ORD     18 Newark Liberty Intl
7     1     1 EWR    FLL     18 Newark Liberty Intl
8     1     1 LGA    IAD     22 La Guardia
9     1     1 JFK    MCO     13 John F Kennedy Intl
10    1     1 LGA    ORD     22 La Guardia
```


On constate que les deux nouvelles colonnes *name* et *alt* contiennent bien les données correspondant à l'aéroport de départ.

On va stocker le résultat de cette jointure dans `flights_ex` :

```
flights_ex <- flights_ex |>
  left_join(airports_ex, by = c("origin" = "faa"))
```

Supposons qu'on souhaite maintenant fusionner à nouveau les informations de la table `airports`, mais cette fois pour les aéroports d'arrivée de notre nouvelle table `flights_ex`. Les deux clés sont donc désormais *dest* dans la première table, et *faa* dans la deuxième. La syntaxe est donc la suivante :

```
flights_ex |>
  left_join(airports_ex, by=c("dest" = "faa")) |>
  head(10)
```

```
# A tibble: 10 x 8
  month   day origin dest alt.x name.x alt.y name.y
  <int> <int> <chr>  <chr> <dbl> <chr>    <dbl> <chr>
1     1     1   EWR   IAH     18 Newark Liberty Intl    97 George Bush Interco~
2     1     1   LGA   IAH     22 La Guardia             97 George Bush Interco~
3     1     1   JFK   MIA     13 John F Kennedy Intl     8 Miami Intl
4     1     1   JFK   BQN     13 John F Kennedy Intl    NA <NA>
5     1     1   LGA   ATL     22 La Guardia          1026 Hartsfield Jackson ~
6     1     1   EWR   ORD     18 Newark Liberty Intl   668 Chicago Ohare Intl
7     1     1   EWR   FLL     18 Newark Liberty Intl     9 Fort Lauderdale Hol~
8     1     1   LGA   IAD     22 La Guardia          313 Washington Dulles I~
9     1     1   JFK   MCO     13 John F Kennedy Intl    96 Orlando Intl
10    1     1   LGA   ORD     22 La Guardia          668 Chicago Ohare Intl
```

Cela fonctionne, les informations de l'aéroport d'arrivée ont bien été ajoutées, mais on constate que les colonnes ont été renommées. En effet, ici les deux tables fusionnées contenaient toutes les deux des colonnes *name* et *alt*. Comme on ne peut pas avoir deux colonnes avec le même nom dans un tableau, `{dplyr}` a renommé les colonnes de la première table en *name.x* et *alt.x*, et celles de la deuxième en *name.y* et *alt.y*.

C'est pratique, mais pas forcément très parlant. On pourrait renommer manuellement les colonnes pour avoir des intitulés plus explicites avec `dplyr::rename()`, mais on peut aussi utiliser l'argument `suffix` de `dplyr::left_join()`, qui permet d'indiquer les suffixes à ajouter aux colonnes. Ainsi, on peut faire :

```
flights_ex |>
  left_join(
    airports_ex,
    by = c("dest" = "faa"),
    suffix = c("_depart", "_arrivee")
  ) |>
  head(10)
```

```
# A tibble: 10 x 8
  month   day origin dest alt_depart name_depart alt_arrivee name_arrivee
  <int> <int> <chr>  <chr>    <dbl> <chr>          <dbl> <chr>
1     1     1   EWR   IAH        18 Newark Liberty ~    97 George Bush~
2     1     1   LGA   IAH        22 La Guardia      97 George Bush~
3     1     1   JFK   MIA        13 John F Kennedy ~    8 Miami Intl
4     1     1   JFK   BQN        13 John F Kennedy ~    NA <NA>
5     1     1   LGA   ATL        22 La Guardia     1026 Hartsfield ~
6     1     1   EWR   ORD        18 Newark Liberty ~   668 Chicago Oha~
7     1     1   EWR   FLL        18 Newark Liberty ~    9 Fort Lauder~
8     1     1   LGA   IAD        22 La Guardia     313 Washington ~
9     1     1   JFK   MCO        13 John F Kennedy ~    96 Orlando Intl
10    1     1   LGA   ORD        22 La Guardia     668 Chicago Oha~
```

On obtient ainsi directement des noms de colonnes nettement plus clairs.

33.1.3 Types de jointures

Jusqu'à présent nous avons utilisé la fonction `dplyr::left_join()`, mais il existe plusieurs types de jointures.

Partons de deux tables d'exemple, `personnes` et `voitures` :

```
personnes <- tibble(
  nom = c("Sylvie", "Sylvie", "Monique", "Gunter", "Rayan", "Rayan"),
  voiture = c("Twingo", "Ferrari", "Scenic", "Lada", "Twingo", "Clio")
)
personnes
```

```
# A tibble: 6 x 2
  nom      voiture
  <chr>    <chr>
1 Sylvie   Twingo
2 Sylvie   Ferrari
3 Monique  Scenic
4 Gunter   Lada
5 Rayan    Twingo
6 Rayan    Clio
```

```

1 Sylvie Twingo
2 Sylvie Ferrari
3 Monique Scenic
4 Gunter Lada
5 Rayan Twingo
6 Rayan Clio

```

```

voitures <- tibble(
  voiture = c("Twingo", "Ferrari", "Clio", "Lada", "208"),
  vitesse = c("140", "280", "160", "85", "160")
)
voitures

```

```

# A tibble: 5 x 2
  voiture vitesse
  <chr>   <chr>
1 Twingo 140
2 Ferrari 280
3 Clio   160
4 Lada   85
5 208    160

```

33.1.3.1 left_join()

Si on fait un `dplyr::left_join()` de `voitures` sur `personnes` :

```

personnes |> left_join(voitures, by = "voiture")

```

```

# A tibble: 6 x 3
  nom      voiture vitesse
  <chr>   <chr>   <chr>
1 Sylvie Twingo 140
2 Sylvie Ferrari 280
3 Monique Scenic <NA>
4 Gunter Lada 85
5 Rayan Twingo 140
6 Rayan Clio 160

```

On voit que chaque ligne de `personnes` est bien présente, et qu'on lui a ajouté une ligne de `voitures` correspondante si elle existe. Dans le cas du *Scenic*, il n'y avait pas de ligne dans

`voitures`, donc *vitesse* a été peuplée avec la valeur manquante `NA`. Dans le cas de la *208*, présente dans `voitures` mais pas dans `personnes`, la ligne n'apparaît pas.

La clé de fusion étant unique dans la table de droite, le nombre de lignes de la table de gauche est donc bien préservée.

```
personnes |> nrow()
```

```
[1] 6
```

```
personnes |> left_join(voitures, by = "voiture") |> nrow()
```

```
[1] 6
```

Si on fait un `dplyr::left_join()` cette fois de `personnes` sur `voitures`, c'est l'inverse :

```
voitures |> left_join(personnes, by = "voiture")
```

```
# A tibble: 6 x 3
  voiture vitesse nom
  <chr>    <chr>  <chr>
1 Twingo   140     Sylvie
2 Twingo   140     Rayan
3 Ferrari  280     Sylvie
4 Clio     160     Rayan
5 Lada     85     Gunter
6 208      160    <NA>
```

La ligne *208* est bien là avec la variable *nom* remplie avec une valeur manquante `NA`. Par contre *Monique* est absente.

! Important

On remarquera que la ligne *Twingo*, présente deux fois dans `personnes`, a été dupliquée pour être associée aux deux lignes de données de *Sylvie* et *Rayan*. Autrement dit, si la clé de fusion n'est pas unique dans la table de droite, certaines de lignes de la table de gauche seront dupliquées.

En résumé, quand on fait un `left_join(x, y)`, toutes les lignes de `x` sont présentes, et dupliquées si nécessaire quand elles apparaissent plusieurs fois dans `y`. Les lignes de `y` non présentes dans `x` disparaissent. Les lignes de `x`

non présentes dans y se voient attribuer des valeurs manquantes NA pour les nouvelles colonnes.

Intuitivement, on pourrait considérer que `left_join(x, y)` signifie ramener l'information de la table y sur la table x.

En général, `dplyr::left_join()` sera le type de jointures le plus fréquemment utilisé.

33.1.3.2 right_join()

La jointure `dplyr::right_join()` est l'exacte symétrique de `dplyr::left_join()`, c'est-à-dire que `x |> right_join(y)` est équivalent¹ à `y |> left_join(x)` :

```
personnes |> right_join(voitures, by = "voiture")
```

```
# A tibble: 6 x 3
  nom      voiture vitesse
<chr>   <chr>   <chr>
1 Sylvie Twingo  140
2 Sylvie Ferrari 280
3 Gunter Lada    85
4 Rayan  Twingo  140
5 Rayan  Clio    160
6 <NA>    208    160
```

```
voitures |> left_join(personnes, by = "voiture")
```

```
# A tibble: 6 x 3
  voiture vitesse nom
<chr>   <chr>   <chr>
1 Twingo  140    Sylvie
2 Twingo  140    Rayan
3 Ferrari 280    Sylvie
4 Clio    160    Rayan
5 Lada    85     Gunter
6 208     160    <NA>
```

¹À l'exception de l'ordre des variables dans le tableau final.

33.1.3.3 inner_join()

Dans le cas de `dplyr::inner_join()`, seules les lignes présentes à la fois dans `x` et `y` sont présentes (et si nécessaire dupliquées) dans la table résultat :

```
personnes |> inner_join(voitures, by = "voiture")
```

```
# A tibble: 5 x 3
  nom      voiture vitesse
  <chr>   <chr>   <chr>
1 Sylvie Twingo   140
2 Sylvie Ferrari 280
3 Gunter Lada     85
4 Rayan  Twingo   140
5 Rayan  Clio     160
```

Ici la ligne *208* est absente, ainsi que la ligne *Monique*, qui dans le cas d'un `dplyr::left_join()` avait été conservée et s'était vue attribuer NA à *vitesse*.

33.1.3.4 full_join()

Dans le cas de `dplyr::full_join()`, toutes les lignes de `x` et toutes les lignes de `y` sont conservées (avec des NA ajoutés si nécessaire) même si elles sont absentes de l'autre table :

```
personnes |> full_join(voitures, by = "voiture")
```

```
# A tibble: 7 x 3
  nom      voiture vitesse
  <chr>   <chr>   <chr>
1 Sylvie Twingo   140
2 Sylvie Ferrari 280
3 Monique Scenic  <NA>
4 Gunter Lada     85
5 Rayan  Twingo   140
6 Rayan  Clio     160
7 <NA>   208     160
```

33.1.3.5 semi_join() et anti_join()

`dplyr::semi_join()` et `dplyr::anti_join()` sont des jointures *filtrantes*, c'est-à-dire qu'elles sélectionnent les lignes de `x` sans ajouter les colonnes de `y`.

Ainsi, `dplyr::semi_join()` ne conservera que les lignes de `x` pour lesquelles une ligne de `y` existe également, et supprimera les autres. Dans notre exemple, la ligne *Monique* est donc supprimée :

```
personnes |> semi_join(voitures, by = "voiture")
```

```
# A tibble: 5 x 2
  nom      voiture
  <chr>   <chr>
1 Sylvie Twingo
2 Sylvie Ferrari
3 Gunter Lada
4 Rayan   Twingo
5 Rayan   Clio
```

Un `dplyr::anti_join()` fait l'inverse, il ne conserve que les lignes de `x` absentes de `y`. Dans notre exemple, on ne garde donc que la ligne *Monique* :

```
personnes |> anti_join(voitures, by = "voiture")
```

```
# A tibble: 1 x 2
  nom      voiture
  <chr>   <chr>
1 Monique Scenic
```

33.2 Jointures avec merge()

La fonction `merge()` est la fonction de **R base** pour fusionner des tables entre elles.

Par défaut, elle réalise un *inner join*, c'est-à-dire qu'elle ne garde que les observations dont la clé est retrouvée dans les deux tableaux fusionnés

```
merge(personnes, voitures, by = "voiture")
```

	voiture	nom	vitesse
1	Clio	Rayan	160
2	Ferrari	Sylvie	280
3	Lada	Gunter	85
4	Twingo	Sylvie	140
5	Twingo	Rayan	140

Les paramètres `all.x` et `all.y` permettent de réaliser fusions à gauche, à droite ou complète. L'équivalent de `dplyr::left_join()` sera obtenu avec `all.x = TRUE`, celui de `dplyr::right_join()` avec `all.y = TRUE` et celui de `dplyr::full_join()` avec `all.x = TRUE`, `all.y = TRUE`.

```
merge(personnes, voitures, by = "voiture", all.x = TRUE)
```

	voiture	nom	vitesse
1	Clio	Rayan	160
2	Ferrari	Sylvie	280
3	Lada	Gunter	85
4	Scenic	Monique	<NA>
5	Twingo	Sylvie	140
6	Twingo	Rayan	140

```
personnes |> left_join(voitures)
```

Joining with ``by = join_by(voiture)``

```
# A tibble: 6 x 3
  nom      voiture vitesse
<chr>    <chr>    <chr>
1 Sylvie  Twingo    140
2 Sylvie  Ferrari   280
3 Monique Scenic   <NA>
4 Gunter  Lada       85
5 Rayan   Twingo    140
6 Rayan   Clio      160
```


33.3 Ajouter des observations avec `bind_rows()`

La fonction `base::rbind()`, fournie nativement avec **R** pour ajouter des observations à un tableau, doit être évitée car elle générera des résultats non pertinents si les tableaux que l'on concatène n'ont pas exactement les mêmes colonnes dans le même ordre.

La fonction `dplyr::bind_rows()` de `{dplyr}` permet d'ajouter des lignes à une table à partir d'une ou plusieurs autres tables.

L'exemple suivant (certes très artificiel) montre l'utilisation de `dplyr::bind_rows()`. On commence par créer trois tableaux `t1`, `t2` et `t3` :

```
t1 <- airports |>
  select(faa, name, lat, lon) |>
  slice(1:2)
t1
```

```
# A tibble: 2 x 4
  faa   name                lat   lon
  <chr> <chr>                <dbl> <dbl>
1 04G   Lansdowne Airport      41.1 -80.6
2 06A   Moton Field Municipal Airport 32.5 -85.7
```

```
t2 <- airports |>
  select(name, faa, lon, lat) |>
  slice(5:6)
t2
```

```
# A tibble: 2 x 4
  name                faa   lon   lat
  <chr>                <chr> <dbl> <dbl>
1 Jekyll Island Airport 09J  -81.4  31.1
2 Elizabethton Municipal Airport 0A9  -82.2  36.4
```

```
t3 <- airports |>
  select(faa, name) |>
  slice(100:101)
t3
```

```
# A tibble: 2 x 2
  faa   name
  <chr> <chr>
1 ADW   Andrews Afb
2 AET   Allakaket Airport
```

On concatène ensuite les trois tables avec `dplyr::bind_rows()` :

```
bind_rows(t1, t2, t3)
```

```
# A tibble: 6 x 4
  faa   name                lat   lon
  <chr> <chr>                <dbl> <dbl>
1 04G   Lansdowne Airport      41.1 -80.6
2 06A   Moton Field Municipal Airport 32.5 -85.7
3 09J   Jekyll Island Airport   31.1 -81.4
4 0A9   Elizabethton Municipal Airport 36.4 -82.2
5 ADW   Andrews Afb            NA    NA
6 AET   Allakaket Airport      NA    NA
```

On remarquera que si des colonnes sont manquantes pour certaines tables, comme les colonnes *lat* et *lon* de *t3*, des valeurs manquantes NA sont automatiquement insérées.

De plus, peu importe l'ordre des variables entre les différentes tables, `dplyr::bind_rows()` les ré-associera en considérant que deux colonnes ayant le même nom dans deux tableaux correspondent à la même variable.

Il peut être utile, quand on concatène des lignes, de garder une trace du tableau d'origine de chacune des lignes dans le tableau final. C'est possible grâce à l'argument `.id` de `dplyr::bind_rows()`. On passe à cet argument le nom d'une colonne qui contiendra l'indicateur d'origine des lignes :

```
bind_rows(t1, t2, t3, .id = "source")
```

```
# A tibble: 6 x 5
  source faa   name                lat   lon
  <chr> <chr> <chr>                <dbl> <dbl>
1 1      04G   Lansdowne Airport      41.1 -80.6
2 1      06A   Moton Field Municipal Airport 32.5 -85.7
3 2      09J   Jekyll Island Airport   31.1 -81.4
4 2      0A9   Elizabethton Municipal Airport 36.4 -82.2
5 3      ADW   Andrews Afb            NA    NA
6 3      AET   Allakaket Airport      NA    NA
```

Par défaut la colonne `.id` ne contient qu'un nombre, différent pour chaque tableau. On peut lui spécifier des valeurs plus explicites en "nommant" les tables dans `dplyr::bind_rows()` de la manière suivante :

```
bind_rows(table1 = t1, table2 = t2, table3 = t3, .id = "source")
```

```
# A tibble: 6 x 5
  source faa   name                lat   lon
  <chr>  <chr> <chr>                <dbl> <dbl>
1 table1 04G   Lansdowne Airport    41.1 -80.6
2 table1 06A   Moton Field Municipal Airport 32.5 -85.7
3 table2 09J   Jekyll Island Airport 31.1 -81.4
4 table2 0A9   Elizabethton Municipal Airport 36.4 -82.2
5 table3 ADW   Andrews Afb         NA    NA
6 table3 AET   Allakaket Airport    NA    NA
```

Une alternative à `dplyr::bind_rows()` est la fonction `plyr::rbind.fill()` de l'extension `{plyr}` qui fonctionne de manière similaire.

34 Dates avec lubridate

Dans cette section, nous aborderons la gestion des dates des heures dans **R**. Si ce type de données peut paraître simple à première vue, la situation se complique dès lors que l'on souhaite effectuer des calculs entre dates. En effet, le nombre de jours varie d'un mois à un autre, voir d'une année à l'autre si l'on tient compte des années bissextiles. Quand on manipule des heures, il faut aussi pouvoir prendre en compte les différents fuseaux horaires ainsi que les changements liés à l'heure d'été.

Heureusement, le package `{lubridate}` permet de résoudre la plupart des problèmes posés par la manipulation de date. Il est chargé par défaut avec la commande `library(tidyverse)`. Nous allons également charger en mémoire le package `{nycflights13}` qui nous fournira les données utilisées dans nos exemples.

```
library(tidyverse)
library(nycflights13)
```

34.1 Création de dates / de dates-heures

Il existe trois types de variables pour représenter des dates et des heures :

- une **date**, de la classe `Date` et représentée dans un tibble avec `<date>`
- une **heure**, de la classe `hms` et représentée dans un tibble avec `<time>`
- une **date-heure**, de la classe `POSIXct` et représentée dans un tibble avec `<dtm>`

Les classes `Date` et `POSIXct` sont gérées nativement par **R** tandis que la classe `hms` est fournie par le package homonyme `{hms}`. Cette dernière classe est d'un usage plus spécifique. Dans cette section, nous allons nous concentrer sur les dates et les dates-heures.

Il est toujours préférable d'utiliser la classe la plus simple. Si vous gérez uniquement des dates, privilégiez la classe `Date`. La classe `POSIXct`, plus complexe, permet d'ajouter une heure associée à un fuseau horaire.

Pour obtenir la date ou la date-heure courante, vous pouvez appeler `today()` ou `now()` :

```
today()
```

```
[1] "2024-04-23"
```

```
now()
```

```
[1] "2024-04-23 20:13:31 CEST"
```

34.1.1 lors de l'import d'un fichier CSV

Si le fichier CSV contient des dates ou des dates-heures au format ISO8601, `readr::read_csv()` saura les reconnaître automatiquement :

```
csv <- "  
  date,datetime  
  2022-01-02,2022-01-02 05:12  
"  
read_csv(csv)
```

```
# A tibble: 1 x 2  
  date      datetime  
  <date>    <dtm>  
1 2022-01-02 2022-01-02 05:12:00
```

Astuce

Le format **ISO8601** est un standard international pour l'écriture de dates¹ sous la forme AAAA-MM-JJ afin d'éviter la confusion entre les habitudes de différents pays, par exemple JJ/MM/AAAA en France ou MM/JJ/AA dans les pays anglo-saxons.

Pour les autres formats, non standards, il sera nécessaire d'utiliser `col_types` avec `col_date()` pour spécifier comment lire et interpréter les chaînes de caractères. `{readr}` comprend la spécification **POSIX** qui permet de décrire un format de date². Il s'agit de codes commençant par le symbole % et indiquant un composant d'une date. Par exemple, `%Y-%m-%d` correspond au format ISO8601, par exemple 2023-10-03 pour le 3 octobre 2023. Le tableau Table 34.1 liste les principales options.

¹<https://xkcd.com/1179/>

²La spécification complète est décrite dans l'aide de la fonction `strptime()`.

Table 34.1: Les formats de dates compris par readr

Type	Code	Signification	Exemple
Année	%Y	année sur 4 chiffres	2021
	%y	année sur 2 chiffres	21
Mois	%m	numéro du mois	2
	%b	nom abrégé	Feb
	%B	nom complet	February
Jour	%d	jour sur 2 chiffres	02
	%e	jour sur 1 ou 2 chiffres	2
Heure	%H	heure sur 24 heures	13
	%I	heure sur 12 heures	1
	%p	AM ou PM	pm
Minute	%M	minutes	35
Seconde	%S	secondes	45
	%OS	secondes avec une composante décimale	45.35
Fuseau horaire	%Z	nom du fuseau	America/Chicago
	%z	décalage du fuseau par rapport au temps universel UTC	+0800
Autre	%.	sauter un caractère (autre qu'un chiffre)	:
	%*	sauter un nombre quelconque de caractères (autres qu'un chiffre)	

Et voici un exemple de code induisant une lecture différente d'une date ambiguë.

```
csv <- "
  date
  01/02/15
"

read_csv(csv, col_types = cols(date = col_date("%m/%d/%y")))
```

```
# A tibble: 1 x 1
  date
  <date>
1 2015-01-02
```

```
read_csv(csv, col_types = cols(date = col_date("%d/%m/%y")))
```

```
# A tibble: 1 x 1
  date
<date>
1 2015-02-01
```

```
read_csv(csv, col_types = cols(date = col_date("%y/%m/%d")))
```

```
# A tibble: 1 x 1
  date
<date>
1 2001-02-15
```

Quel que soit le format original, les dates importées seront toujours affichées par **R** au format ISO.

Astuce

Si vous utilisez %b ou %B, il est essentiel de spécifier la langue utilisée avec le paramètre `local` de `col_date()`. Pour voir l'ensemble des langues couvertes, vous pouvez appelez `readr::date_names_langs()` et pour voir les chaînes de langues correspondantes `readr::date_names_lang()`. Si vos données n'utilise pas des noms standards, vous pouvez créer votre propre jeu de correspondance avec `readr::date_names()`.

```
date_names_langs()
```

```
[1] "af" "agq" "ak" "am" "ar" "as" "asa" "az" "bas" "be" "bem" "bez"
[13] "bg" "bm" "bn" "bo" "br" "brx" "bs" "ca" "cgg" "chr" "cs" "cy"
[25] "da" "dav" "de" "dje" "dsb" "dua" "dyo" "dz" "ebu" "ee" "el" "en"
[37] "eo" "es" "et" "eu" "ewo" "fa" "ff" "fi" "fil" "fo" "fr" "fur"
[49] "fy" "ga" "gd" "gl" "gsw" "gu" "guz" "gv" "ha" "haw" "he" "hi"
[61] "hr" "hsb" "hu" "hy" "id" "ig" "ii" "is" "it" "ja" "jgo" "jmc"
[73] "ka" "kab" "kam" "kde" "kea" "khq" "ki" "kk" "kkj" "kl" "kln" "km"
[85] "kn" "ko" "kok" "ks" "ksb" "ksf" "ksh" "kw" "ky" "lag" "lb" "lg"
[97] "lkt" "ln" "lo" "lt" "lu" "luo" "luy" "lv" "mas" "mer" "mfe" "mg"
[109] "mgh" "mgo" "mk" "ml" "mn" "mr" "ms" "mt" "mua" "my" "naq" "nb"
[121] "nd" "ne" "nl" "nmg" "nn" "nnh" "nus" "nyn" "om" "or" "os" "pa"
[133] "pl" "ps" "pt" "qu" "rm" "rn" "ro" "rof" "ru" "rw" "rwk" "sah"
[145] "saq" "sbp" "se" "seh" "ses" "sg" "shi" "si" "sk" "sl" "smn" "sn"
[157] "so" "sq" "sr" "sv" "sw" "ta" "te" "teo" "th" "ti" "to" "tr"
[169] "twq" "tzm" "ug" "uk" "ur" "uz" "vai" "vi" "vun" "wae" "xog" "yav"
[181] "yi" "yo" "zgh" "zh" "zu"
```

```
date_names_lang("fr")
```

```
<date_names>
```

```
Days:  dimanche (dim.), lundi (lun.), mardi (mar.), mercredi (mer.), jeudi  
       (jeu.), vendredi (ven.), samedi (sam.)  
Months: janvier (janv.), février (févr.), mars (mars), avril (avr.), mai (mai),  
        juin (juin), juillet (juil.), août (août), septembre (sept.),  
        octobre (oct.), novembre (nov.), décembre (déc.)  
AM/PM:  AM/PM
```

```
date_names_lang("en")
```

```
<date_names>
```

```
Days:  Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed), Thursday  
       (Thu), Friday (Fri), Saturday (Sat)  
Months: January (Jan), February (Feb), March (Mar), April (Apr), May (May),  
        June (Jun), July (Jul), August (Aug), September (Sep), October  
        (Oct), November (Nov), December (Dec)  
AM/PM:  AM/PM
```

```
csv <- "date  
3 de febrero de 2001"
```

```
read_csv(  
  csv,  
  col_types = cols(date = col_date("%d de %B de %Y")),  
  locale = locale("es")  
)
```

```
# A tibble: 1 x 1  
  date  
  <date>  
1 2001-02-03
```

34.1.2 à partir d'une chaîne de caractères

Le langage de spécification de la date et du temps est puissant, mais il nécessite une analyse minutieuse du format de la date. Une autre approche consiste à utiliser les fonctions de `{lubridate}` qui tentent de déterminer automatiquement le format une fois que vous avez spécifié l'ordre des composants. Pour les utiliser, identifiez l'ordre dans lequel l'année, le mois

et le jour apparaissent dans vos dates, puis placez “y”, “m” et “d” dans le même ordre. Cela vous donne le nom de la fonction {lubridate} qui analysera votre date. Par exemple :

```
ymd("2017-01-31")
```

```
[1] "2017-01-31"
```

```
mdy("January 31st, 2017")
```

```
[1] "2017-01-31"
```

```
dmy("31-Jan-2017")
```

```
[1] "2017-01-31"
```

ymd() et ses sœurs créent des dates. Pour des dates-heures, ajoutez un tiret bas et les lettres “h”, “m” et/ou “s” :

```
ymd_hms("2017-01-31 20:11:59")
```

```
[1] "2017-01-31 20:11:59 UTC"
```

```
mdy_hm("01/31/2017 08:01")
```

```
[1] "2017-01-31 08:01:00 UTC"
```

34.1.3 à partir des composants

Parfois, les différentes composantes d’une date (jour, mois, année...) sont stockées dans des colonnes séparées. C’est le cas par exemple dans la table `flights` issue du package {nycflights13}.

```
flights |>  
  select(year, month, day, hour, minute) |>  
  head()
```

```
# A tibble: 6 x 5
  year month   day hour minute
  <int> <int> <int> <dbl> <dbl>
1  2013     1     1     5     15
2  2013     1     1     5     29
3  2013     1     1     5     40
4  2013     1     1     5     45
5  2013     1     1     6      0
6  2013     1     1     5     58
```

Pour créer une date ou une date-heure à partir de colonnes séparées, il suffit d'utiliser `lubridate::make_date()` pour les dates et `lubridate::make_datetime()` pour les dates-heures :

```
flights |>
  select(year, month, day, hour, minute) |>
  mutate(
    departure = make_datetime(year, month, day, hour, minute),
    departure_date = make_date(year, month, day)
  ) |>
  head()
```

```
# A tibble: 6 x 7
  year month   day hour minute departure departure_date
  <int> <int> <int> <dbl> <dbl> <dtm>         <date>
1  2013     1     1     5     15 2013-01-01 05:15:00 2013-01-01
2  2013     1     1     5     29 2013-01-01 05:29:00 2013-01-01
3  2013     1     1     5     40 2013-01-01 05:40:00 2013-01-01
4  2013     1     1     5     45 2013-01-01 05:45:00 2013-01-01
5  2013     1     1     6      0 2013-01-01 06:00:00 2013-01-01
6  2013     1     1     5     58 2013-01-01 05:58:00 2013-01-01
```

34.1.4 conversion

Pour convertir une date en date-heure, ou l'inverse, utilisez `lubridate::as_datetime()` ou `lubridate::as_date()` :

```
as_datetime(today())
```

```
[1] "2024-04-23 UTC"
```

```
as_date(now())
```

```
[1] "2024-04-23"
```

34.2 Manipuler les composants d'une date/date-heure

34.2.1 Extraire un composant

Pour extraire un composant d'une date ou d'une date-heure, il suffit d'utiliser l'une des fonctions suivantes : `year()` (année), `month()` (mois), `mday()` (jour du mois), `yday()` (jours de l'année), `wday()` (jour de la semaine), `hour()` (heure), `minute()` (minute), ou `second()` (seconde).

```
datetime <- ymd_hms("2026-07-08 12:34:56")
```

```
year(datetime)
```

```
[1] 2026
```

```
month(datetime)
```

```
[1] 7
```

```
mday(datetime)
```

```
[1] 8
```

```
yday(datetime)
```

```
[1] 189
```

```
wday(datetime)
```

```
[1] 4
```

Pour `month()` et `wday()`, vous pouvez indiquer `label = TRUE` pour récupérer le nom abrégé du mois ou du jours de la semaine. Ajoutez `abbr = FALSE` pour le nom complet.

```
month(datetime, label = TRUE)
```

```
[1] juil  
12 Levels: janv < févr < mars < avr < mai < juin < juil < août < ... < déc
```

```
wday(datetime, label = TRUE, abbr = FALSE)
```

```
[1] mercredi  
7 Levels: dimanche < lundi < mardi < mercredi < jeudi < ... < samedi
```

Les noms sont affichés dans la langue de votre ordinateur. On peut utiliser le paramètre `locale` pour changer la langue. Attention : le code peut varier selon votre système d'exploitation. Vous pouvez essayer déjà de simplement indiquer le code à 2 lettres de la langue visée, par exemple `"de"` pour l'allemand. Si cela ne fonctionne pas, essayez `"de_DE"` (allemand utilisé en Allemagne), `"de_DE.UTF-8"` (format utilisé par MacOS et plusieurs distributions Linux), la variante `"de_DE.utf8"` (utilisée par certaines distributions Linux) ou bien encore `"German.UTF-8"` (utilisé par Windows).

```
month(datetime, label = TRUE, abbr = FALSE, locale = "en")
```

```
[1] July  
12 Levels: January < February < March < April < May < June < ... < December
```

```
month(datetime, label = TRUE, abbr = FALSE, locale = "es_ES.utf8")
```

```
[1] julio  
12 Levels: enero < febrero < marzo < abril < mayo < junio < ... < diciembre
```

```
month(datetime, label = TRUE, abbr = FALSE, locale = "German.UTF-8")
```

```
[1] Juli  
12 Levels: Januar < Februar < März < April < Mai < Juni < Juli < ... < Dezember
```

34.2.2 Arrondis

Les fonctions `lubridate::round_date()`, `lubridate::floor_date()` et `lubridate::ceiling_date()` permettent d'arrondir une date à l'unité la plus proche, inférieure ou supérieure. On devra préciser avec `unit` l'unité utilisée pour arrondir. Les valeurs acceptées sont "second", "minute", "hour", "day", "week", "month", "bimonth" (bimestre, i.e. période de 2 mois), "quarter" (trimestre), `season` (saison), `halfyear` (semestre) et `year`, ou un multiple de ces valeurs.

```
d <- ymd("2022-05-14")
floor_date(d, unit = "week")
```

```
[1] "2022-05-08"
```

```
floor_date(d, unit = "month")
```

```
[1] "2022-05-01"
```

```
floor_date(d, unit = "3 months")
```

```
[1] "2022-04-01"
```

```
floor_date(d, unit = "year")
```

```
[1] "2022-01-01"
```

34.2.3 Modifier un composant

Les mêmes fonctions peuvent être utilisées pour modifier un composant particulier d'une date-heure.

```
datetime <- ymd_hms("2026-07-08 12:34:56")
year(datetime) <- 2030
datetime
```

```
[1] "2030-07-08 12:34:56 UTC"
```

```
month(datetime) <- 01
datetime
```

```
[1] "2030-01-08 12:34:56 UTC"
```

```
hour(datetime) <- hour(datetime) + 1
datetime
```

```
[1] "2030-01-08 13:34:56 UTC"
```

Une alternative, plutôt que de modifier une date-heure, consiste à créer une copie modifiée avec `lubridate::update()`. Cela permet également de modifier plusieurs éléments à la fois :

```
update(datetime, year = 2030, month = 2, mday = 2, hour = 2)
```

```
[1] "2030-02-02 02:34:56 UTC"
```

Si les valeurs sont trop importantes (trop de jours par exemple), la fonction ajoutera les unités en trop pour générer une date valide :

```
update(ymd("2023-02-01"), mday = 30)
```

```
[1] "2023-03-02"
```

```
update(ymd("2023-02-01"), hour = 400)
```

```
[1] "2023-02-17 16:00:00 UTC"
```

34.3 Durées, périodes, intervalles & Arithmétique

Il existe plusieurs manières de représenter les intervalles de temps entre deux dates :

- les **durées** (*Duration*), qui représentent un nombre exact de secondes ;
- les **périodes** (*Periods*), qui représentent une durée sous la forme d'unités de temps telles que des semaines ou des mois ;
- les **intervalles** (*Intervals*), qui sont définis par une date-heure de début et une date-heure de fin.

34.3.1 Durées (Duration)

Avec **R**, lorsque l'on soustrait deux dates, on obtient un objet de la classe `difftime`.

```
diff <- ymd("2021-06-30") - ymd("1979-10-14")
diff
```

```
Time difference of 15235 days
```

Un objet `difftime` enregistre une durée sous la forme d'un nombre de secondes, de minutes, d'heures, de jours ou de semaines. Du fait de variations de l'unité d'un objet à l'autre, ils ne sont pas toujours faciles à manipuler. Pour lever toute ambiguïté, on préférera les objets de la classe `Duration` qui stockent les durées sous la forme d'un nombre de secondes. La conversion peut se faire avec `lubridate::as.duration()`.

```
as.duration(diff)
```

```
[1] "1316304000s (~41.71 years)"
```

Il est possible de créer facilement des durées avec une série de fonctions dédiées dont le nom commence par "d"

```
dseconds(15)
```

```
[1] "15s"
```

```
dminutes(10)
```

```
[1] "600s (~10 minutes)"
```

```
dhours(c(12, 24))
```

```
[1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
ddays(0:5)
```

```
[1] "0s" "86400s (~1 days)" "172800s (~2 days)"
[4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
```

```
dweeks(3)
```

```
[1] "1814400s (~3 weeks)"
```

```
dyears(1)
```

```
[1] "31557600s (~1 years)"
```

Les durées sont toujours exprimées en secondes. Des unités plus grandes sont créées en convertissant les minutes, les heures, les jours, les semaines et les années en secondes : 60 secondes dans une minute, 60 minutes dans une heure, 24 heures dans un jour et 7 jours dans une semaine. Les unités de temps plus grandes posent davantage de problèmes. Une année utilise le nombre « moyen » de jours dans une année, c'est-à-dire 365,25. Il n'existe aucun moyen de convertir un mois en durée, car les variations sont trop importantes.

Il est possible d'additionner et de multiplier les durées :

```
2 * dyears(1)
```

```
[1] "63115200s (~2 years)"
```

```
dyears(1) + dweeks(12) + dhours(15)
```

```
[1] "38869200s (~1.23 years)"
```

On peut ajouter ou soustraire des durées à une date.

```
demain <- today() + ddays(1)
il_y_a_un_an <- today() - dyears(1)
```

Cependant, comme les durées représentent un nombre exact de secondes, vous pouvez parfois obtenir un résultat inattendu :

```
one_am <- ymd_hms("2026-03-08 01:00:00", tz = "America/New_York")
one_am
```

```
[1] "2026-03-08 01:00:00 EST"
```



```
one_am + ddays(1)
```

```
[1] "2026-03-09 02:00:00 EDT"
```

Pourquoi lorsqu'on ajoute un jour, on passe de 1 heure du matin à 2 heures du matin ? Si vous regardez attentivement la date, vous remarquerez que le fuseau a changé. Le 8 mars 2026 n'aura que 23 heures aux États-Unis en raison du passage à l'heure d'été. En ajoutant une durée de 1 jour, nous avons ajouté exactement 24 heures. Le même type de phénomène peut s'observer en ajoutant une durée d'une année, car on considère que cela représente en moyenne 365.25 jours.

34.3.2 Périodes (Period)

Pour résoudre ce problème, `{lubridate}` a introduit les périodes (de classe `Period`) qui représentent une durée en nombre de secondes, minutes, heures, jours, mois et années, sans préciser la durée exacte de chaque mois ou année. Cela permet de faire des calculs plus intuitifs :

```
one_am
```

```
[1] "2026-03-08 01:00:00 EST"
```

```
one_am + days(1)
```

```
[1] "2026-03-09 01:00:00 EDT"
```

Comme pour les durées, on peut créer facilement des périodes avec des fonctions dédiées (notez ici le pluriel des noms de fonction, alors que celles permettant d'extraire un composant d'une date étaient au singulier) :

```
hours(c(12, 24))
```

```
[1] "12H 0M 0S" "24H 0M 0S"
```

```
days(7)
```

```
[1] "7d 0H 0M 0S"
```

```
months(1:6)
```

```
[1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"  
[5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
```

On peut ajouter, soustraire et multiplier les périodes entre elles.

```
10 * (months(6) + days(1))
```

```
[1] "60m 10d 0H 0M 0S"
```

```
days(50) + hours(25) + minutes(2)
```

```
[1] "50d 25H 2M 0S"
```

Bien sûr, on peut ajouter ou soustraire une période à une date :

```
# Exemple avec une année bissextile  
ymd("2024-01-01") + dyears(1)
```

```
[1] "2024-12-31 06:00:00 UTC"
```

```
ymd("2024-01-01") + years(1)
```

```
[1] "2025-01-01"
```

```
# Exemple avec un passage à l'heure d'été  
one_am + ddays(1)
```

```
[1] "2026-03-09 02:00:00 EDT"
```

```
one_am + days(1)
```

```
[1] "2026-03-09 01:00:00 EDT"
```

Restent malgré tout quelques cas problématiques. Essayons d'ajouter 1 mois à la date du 31 janvier 2021.

```
ymd("2021-01-31") + months(1)
```

```
[1] NA
```

Ce calcul a ajouté 1 au mois, sans toucher à l'année ni au jour, produisant la date du 31 février 2021 qui n'existe pas, produisant ainsi `NA`. Pour du calcul impliquant des dates et des périodes, il est préférable d'utiliser les opérateurs dédiés `%m+%` pour l'addition et `%m-%` pour la soustraction.

```
ymd("2021-01-31") %m+% months(1)
```

```
[1] "2021-02-28"
```

Lorsque le résultat produit une date inexistante, cela renvoie la dernière date correcte, ici le 28 février 2021, ce qui correspond bien à la fin du mois considéré.

34.3.3 Intervalles (Interval)

Quelle est la durée réelle d'une année ? En 2015, il s'agissait de 365 jours alors qu'en 2016 on en comptait 366. Quand on s'intéresse aux mois, la situation est encore plus compliquée car il y a une grande variation du nombre de jours d'un mois à l'autre.

Pour des calculs précis entre deux dates, les durées et les intervalles sont souvent insuffisants. On pourra alors avoir recours aux intervalles (de la classe `Interval`) qui sont définis avec une date de début et une date de fin.

On peut créer un intervalle avec la fonction `lubridate::interval()` :

```
interval(ymd("2022-05-13"), ymd("2022-08-15"))
```

```
[1] 2022-05-13 UTC--2022-08-15 UTC
```

On peut également utiliser l'opérateur `%--%` :

```
y2023 <- ymd("2023-01-01") %--% ymd("2024-01-01")
y2024 <- ymd("2024-01-01") %--% ymd("2025-01-01")

y2023
```

```
[1] 2023-01-01 UTC--2024-01-01 UTC
```

```
y2024
```

```
[1] 2024-01-01 UTC--2025-01-01 UTC
```

On peut tester si une date est située dans un intervalle donné avec l'opérateur `%within%`.

```
int <- interval(ymd("2001-01-01"), ymd("2002-01-01"))  
ymd("2001-05-03") %within% int
```

```
[1] TRUE
```

On peut même tester si un intervalle est situé à l'intérieur d'un intervalle :

```
int2 <- interval(ymd("2001-06-01"), ymd("2001-11-11"))  
int2 %within% int
```

```
[1] TRUE
```

Cela n'est valable que si l'ensemble du premier intervalle est situé à l'intérieur du second intervalle.

```
int3 <- interval(ymd("2001-06-01"), ymd("2002-06-01"))  
int3 %within% int
```

```
[1] FALSE
```

Pour tester si deux intervalles ont une partie en commun, on pourra utiliser `lubridate::int_overlaps()`. La fonction `intersect()` renvoie la partie partagée par les deux intervalles.

```
int_overlaps(int3, int)
```

```
[1] TRUE
```

```
intersect(int3, int)
```

```
[1] 2001-06-01 UTC--2002-01-01 UTC
```

{lubridate} fournit plusieurs fonctions, de la forme `int_*()`, pour manipuler les intervalles.

```
int
```

```
[1] 2001-01-01 UTC--2002-01-01 UTC
```

```
int_start(int)
```

```
[1] "2001-01-01 UTC"
```

```
int_end(int)
```

```
[1] "2002-01-01 UTC"
```

```
int_flip(int)
```

```
[1] 2002-01-01 UTC--2001-01-01 UTC
```

On peut calculer facilement la durée d'un intervalle avec la fonction `lubridate::time_length()` :

```
time_length(int) # en seconde par défaut
```

```
[1] 31536000
```

```
time_length(int, unit = "weeks")
```

```
[1] 52.14286
```

```
time_length(int, unit = "days")
```

```
[1] 365
```

La fonction `time_length()` permet notamment de calculer correctement un âge.

34.4 Calcul d'un âge

En tant que démographe, je suis toujours attentif au calcul des âges. Les démographes distinguent l'**âge exact**, exprimé en années avec une partie décimale, et qui correspond à la durée entre la date considérée et la date de naissance ; l'**âge révolu**, qui correspond à l'âge au dernier anniversaire et exprimé avec un nombre entier d'années (c'est l'âge que nous utilisons dans notre vie quotidienne) ; et l'**âge atteint** ou **âge par différence de millésimes**, qui correspond à la différence entre l'année en cours et l'année de naissance (c'est l'âge que l'on aura cette année le jour de son anniversaire).

Pour calculer un âge exact en années, nous ne pouvons pendre la durée en jours entre les deux dates et diviser par 365 puisqu'il y a des années bissextiles. Une approche correcte est déjà de considérer l'âge au dernière anniversaire pour la partie entière, puis de calculer la partie décimale comme étant le ratio entre la durée depuis le dernière anniversaire et la durée entre le dernier et le prochain anniversaire. C'est exactement ce que fait `lubridate::time_length()`.

```
naiss <- ymd("1979-11-28")
evt <- ymd("2022-07-14")
age_exact <- time_length(naiss %--% evt, unit = "years")
age_exact
```

```
[1] 42.62466
```

Pour un âge révolu, il suffit de ne garder que la partie entière de l'âge exact avec `trunc()`.

```
age_revolu <- trunc(age_exact)
age_revolu
```

```
[1] 42
```

Enfin, pour un âge atteint ou un âge par différence de millésimes, nous extrairons les deux années avant d'en faire la soustraction.

```
age_atteint <- year(evt) - year(naiss)
age_atteint
```

```
[1] 43
```

Astuce

Le calcul d'un âge moyen s'effectue normalement à partir d'âges exacts. Il arrive fréquemment que l'on ne dispose dans les données d'enquêtes que de l'âge révolu. Auquel cas, il faut bien penser à rajouter 0,5 au résultat obtenu. En effet, un âge révolu peut être vu comme une classe d'âges exacts : les individus ayant 20 ans révolus ont entre 20 et 21 ans exacts, soit en moyenne 20,5 ans !

34.5 Fuseaux horaires

Les **fuseaux horaires** sont un sujet extrêmement complexe en raison de leur interaction avec les entités géopolitiques. Heureusement, nous n'avons pas besoin d'entrer dans tous les détails, car ils ne sont pas tous importants pour l'analyse des données, mais il y a quelques défis que nous devons relever.

Le premier défi est que les noms courants des fuseaux horaires ont tendance à être ambigus. Par exemple, si vous êtes américain, vous connaissez probablement l'EST (*Eastern Standard Time*). Cependant, l'Australie et le Canada ont également une heure normale de l'Est ! Pour éviter toute confusion, **R** utilise les fuseaux horaires standard internationaux de l'IANA. Ceux-ci utilisent un schéma de dénomination cohérent `{zone}/{lieu}`, généralement sous la forme `{continent}/{ville}` ou `{océan}/{ville}`. Parmi les exemples, citons "America/New_York", "Europe/Paris" et "Pacific/Auckland".

On peut se demander pourquoi le fuseau horaire utilise une ville, alors que l'on pense généralement que les fuseaux horaires sont associés à un pays ou à une région à l'intérieur d'un pays. La raison en est que la base de données de l'IANA doit enregistrer des dizaines d'années de règles relatives aux fuseaux horaires. Au fil des décennies, les pays changent de nom (ou se séparent) assez fréquemment, mais les noms de villes ont tendance à rester inchangés. Un autre problème réside dans le fait que le nom doit refléter non seulement le comportement actuel, mais aussi l'ensemble de l'histoire. Par exemple, il existe des fuseaux horaires pour "America/New_York" et "America/Detroit". Cela vaut la peine de lire la base de données brute des fuseaux horaires (disponible à l'adresse <https://www.iana.org/time-zones>) rien que pour lire certaines de ces histoires !

Vous pouvez découvrir ce que **R** pense être votre fuseau horaire actuel avec `Sys.timezone()` :

```
Sys.timezone()
```

```
[1] "Europe/Paris"
```

La liste complète des fuseaux horaires est disponible avec `OlsonNames()` :

```
length(OlsonNames())
```

```
[1] 596
```

```
head(OlsonNames())
```

```
[1] "Africa/Abidjan"      "Africa/Accra"        "Africa/Addis_Ababa"  
[4] "Africa/Algiers"      "Africa/Asmara"        "Africa/Asmera"
```

Dans **R**, le fuseau horaire est un attribut de la date-heure qui ne contrôle que l’affichage. Par exemple, ces trois objets représentent le même instant dans le temps :

```
x1 <- ymd_hms("2024-06-01 12:00:00", tz = "America/New_York")  
x1
```

```
[1] "2024-06-01 12:00:00 EDT"
```

```
x2 <- ymd_hms("2024-06-01 18:00:00", tz = "Europe/Copenhagen")  
x2
```

```
[1] "2024-06-01 18:00:00 CEST"
```

```
x3 <- ymd_hms("2024-06-02 04:00:00", tz = "Pacific/Auckland")  
x3
```

```
[1] "2024-06-02 04:00:00 NZST"
```

Sauf indication contraire, `{lubridate}` utilise toujours l’heure UTC. UTC ([Temps universel coordonné](#), compromis entre l’anglais CUT *Coordinated universal time* et le français TUC *Temps universel coordonné*) est le fuseau horaire standard utilisé par la communauté scientifique et est à peu près équivalent à GMT (*Greenwich Mean Time*). Il n’y a pas d’heure d’été, ce qui en fait une représentation pratique pour les calculs. Les opérations qui combinent des dates-heure, comme `c()`, ne tiennent souvent pas compte du fuseau horaire. Dans ce cas, les dates-heure s’afficheront dans le fuseau horaire du premier élément :

```
x4 <- c(x1, x2, x3)  
x4
```



```
[1] "2024-06-01 12:00:00 EDT" "2024-06-01 12:00:00 EDT"
[3] "2024-06-01 12:00:00 EDT"
```

Vous pouvez modifier le fuseau horaire de deux manières :

- Conserver le même instant dans le temps, mais modifier la façon dont il est affiché. Utilisez cette option lorsque l'instant est correct, mais que vous souhaitez un affichage plus naturel.

```
x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
x4a
```

```
[1] "2024-06-02 02:30:00 +1030" "2024-06-02 02:30:00 +1030"
[3] "2024-06-02 02:30:00 +1030"
```

```
x4a - x4
```

```
Time differences in secs
[1] 0 0 0
```

- Modifier l'instant sous-jacent dans le temps. Utilisez cette option lorsqu'un instant a été étiqueté avec un fuseau horaire incorrect et que vous devez le corriger.

```
x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b
```

```
[1] "2024-06-01 12:00:00 +1030" "2024-06-01 12:00:00 +1030"
[3] "2024-06-01 12:00:00 +1030"
```

```
x4b - x4
```

```
Time differences in hours
[1] -14.5 -14.5 -14.5
```

34.6 Pour aller plus loin

- le chapitre [Dates and times](#) de l'ouvrage *R for Data Science* (2e édition)
- la documentation du package `{lubridate}` : <https://lubridate.tidyverse.org/>

35 Chaînes de texte avec stringr

Les fonctions de `{forcats}` vues précédemment permettent de modifier des modalités d'une variable 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)
```

Note

`{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 avenue de la Libération"
  ),
  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

nom	adresse	ville
M. Martial Truc	12 rue du 17 octobre 1961	Vénissieux
Mme Huguette Chose	221 avenue de la Libération	Marseille

35.1 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"
[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"
```

Note

À noter que `paste()` et `paste0()` sont des fonctions R de base. L'équivalent pour `{stringr}` se nomme `stringr::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 :

```
d$ville |> paste(collapse = ", ")
```

```
[1] "Nouméa, Marseille, Vénissieux, Marseille"
```

35.2 Convertir en majuscules / minuscules

Les fonctions `stringr::str_to_lower()`, `stringr::str_to_upper()` et `stringr::str_to_title()` permettent respectivement de mettre en minuscules, mettre en majuscules, ou de capitaliser les éléments d'un vecteur de chaînes de caractères :

```
d$nom |> str_to_lower()
```

```
[1] "mr félicien machin" "mme raymonde bidule" "m. martial truc"  
[4] "mme huguette chose"
```

```
d$nom |> str_to_upper()
```

```
[1] "MR FÉLICIEIN MACHIN" "MME RAYMONDE BIDULE" "M. MARTIAL TRUC"  
[4] "MME HUGUETTE CHOSE"
```

```
d$nom |> str_to_title()
```

```
[1] "Mr Félicien Machin" "Mme Raymonde Bidule" "M. Martial Truc"  
[4] "Mme Huguette Chose"
```

35.3 Découper des chaînes

La fonction `stringr::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 :

```
"un-deux-trois" |>  
  str_split("-")
```

```
[[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`.

```
d$nom |>  
  str_split(" ", 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 `tidyr::separate()` de l'extension `{tidyr}` (cf. Section 36.5).

Voici juste un exemple de son utilisation :

```

d |>
  tidyr::separate(
    col = nom,
    into = c("genre", "prenom", "nom")
  )

```

Warning: Expected 3 pieces. Additional pieces discarded in 1 rows [1].

```

# A tibble: 4 x 5
  genre prenom  nom      adresse          ville
  <chr> <chr>    <chr>  <chr>          <chr>
1 Mr    F        licien 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

```

35.4 Extraire des sous-chaînes par position

La fonction `stringr::str_sub()` permet d'extraire des sous-chaînes par position, en indiquant simplement les positions des premier et dernier caractères :

```

d$ville |> str_sub(1, 3)

```

```

[1] "Nou" "Mar" "Vén" "Mar"

```

35.5 Détecter des motifs

`stringr::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 :

```
d$adresse |> str_detect("Libération")
```

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

`stringr::str_detect()` renvoi un vecteur de valeurs logiques et peut donc être utilisée, par exemple, avec le verbe `dplyr::filter()` pour extraire des sous-populations.

```
d |>
  filter(adresse |> str_detect("Libération"))
```

```
# A tibble: 2 x 3
  nom          adresse          ville
  <chr>        <chr>          <chr>
1 Mme Raymonde Bidule 47 ave de la Libération Marseille
2 Mme Huguette Chose  221 avenue de la Libération Marseille
```

Une variante, `stringr::str_count()`, compte le nombre d'occurrences d'une chaîne pour chaque élément d'un vecteur :

```
d$ville
```

```
[1] "Nouméa"      "Marseille"  "Vénissieux" "Marseille"
```

```
d$ville |> str_count("s")
```

```
[1] 0 1 2 1
```

! Important

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 `stringr::str_subset()` pour ne garder d'un vecteur que les éléments correspondant au motif :

```
d$adresse |> str_subset("Libération")
```

```
[1] "47 ave de la Libération"      "221 avenue de la Libération"
```

35.6 Expressions régulières

Les fonctions présentées ici 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,}
```

Les exemples donnés dans ce chapitre ont utilisés autant que possible de simples chaînes de texte, sans expression régulière. Mais si vous pensez manipuler des données textuelles, il peut être très utile de s'intéresser à cette syntaxe.

35.7 Extraire des motifs

`stringr::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 :

```
d$adresse |>  
  str_extract("Libération")
```

```
[1] NA          "Libération" NA          "Libération"
```

¹Il 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...

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.

```
d$adresse |> str_extract("^\\d+")
```

```
[1] "3"   "47"  "12"  "221"
```

`stringr::str_extract()` ne récupère que la première occurrence du motif. Si on veut toutes les extraire on peut utiliser `stringr::str_extract_all()`. Ainsi, si on veut extraire l'ensemble des nombres présents dans les adresses :

```
d$adresse |> str_extract_all("\\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 plutôt utiliser la fonction `tidyr::extract()` de l'extension `{tidyr}` (cf. Section [36.8](#)).

Par exemple :

```
d |>
  tidyr::extract(
    col = adresse,
    into = "type_rue",
    regex = "^\\d+ (.*) ",
    remove = FALSE
  )
```

```
# A tibble: 4 x 4
  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
```

35.8 Remplacer des motifs

La fonction `stringr::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 :

```
d$nom |>
  str_replace("Mr", "M.")
```

```
[1] "M. Félicien Machin" "Mme Raymonde Bidule" "M. Martial Truc"
[4] "Mme Huguette Chose"
```

La variante `stringr::str_replace_all()` permet de spécifier plusieurs remplacements d’un coup :

```
d$adresse |>
  str_replace_all(
    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"
```

35.9 Modificateurs de motifs

Par défaut, les motifs passés aux fonctions comme `stringr::str_detect()`, `stringr::str_extract()` ou `stringr::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 `stringr::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 :

```
d$nom |> str_count(".")
```

```
[1] 18 19 15 18
```

Il faut donc spécifier que notre point est bien un point avec `stringr::fixed()` :

```
d$nom |> str_count(fixed("."))
```

```
[1] 0 0 1 0
```

On peut aussi modifier le comportement des expressions régulières à l'aide de la fonction `stringr::regex()`. On peut ainsi rendre les motifs insensibles à la casse avec `ignore_case` :

```
d$nom |> str_detect("mme")
```

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

```
d$nom |>  
  str_detect(regex("mme", ignore_case = TRUE))
```

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

On peut également permettre aux expressions régulières d'être multilignes avec l'option `multiline = TRUE`, etc.

35.10 Insérer une variable dans une chaîne de caractères

La fonction `stringr::str_glue()` repose sur l'extension `{glue}`. Elle permet, à l'aide d'une syntaxe un peu spécifique, de pouvoir insérer facilement les valeurs d'une ou plusieurs variables dans une chaîne de caractères. Prenons un exemple :

```
prenom <- "Fred"
age <- 28
anniversaire <- as.Date("1991-10-12")
str_glue(
  "Je m'appelle {prenom}. ",
  "L'année prochaine j'aurai {age + 1} ans, ",
  "car je suis né le {format(anniversaire, '%A %d %B %Y')})."
)
```

Je m'appelle Fred. L'année prochaine j'aurai 29 ans, car je suis né le samedi 12 octobre 1991.

Sa variante `stringr::str_glue_data()` est adaptée lorsque l'on travaille sur un tableau de données.

```
d |> str_glue_data("{nom} habite à {ville}.")
```

Mr Félicien Machin habite à Nouméa.
Mme Raymonde Bidule habite à Marseille.
M. Martial Truc habite à Vénissieux.
Mme Huguette Chose habite à Marseille.

35.11 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 le package [stringi](#) sur lequel est basé `{stringr}`.

36 Réorganisation avec tidyr

36.1 Tidy data

Comme indiqué dans le chapitre sur les tibbles (cf. Chapitre 5), 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 :

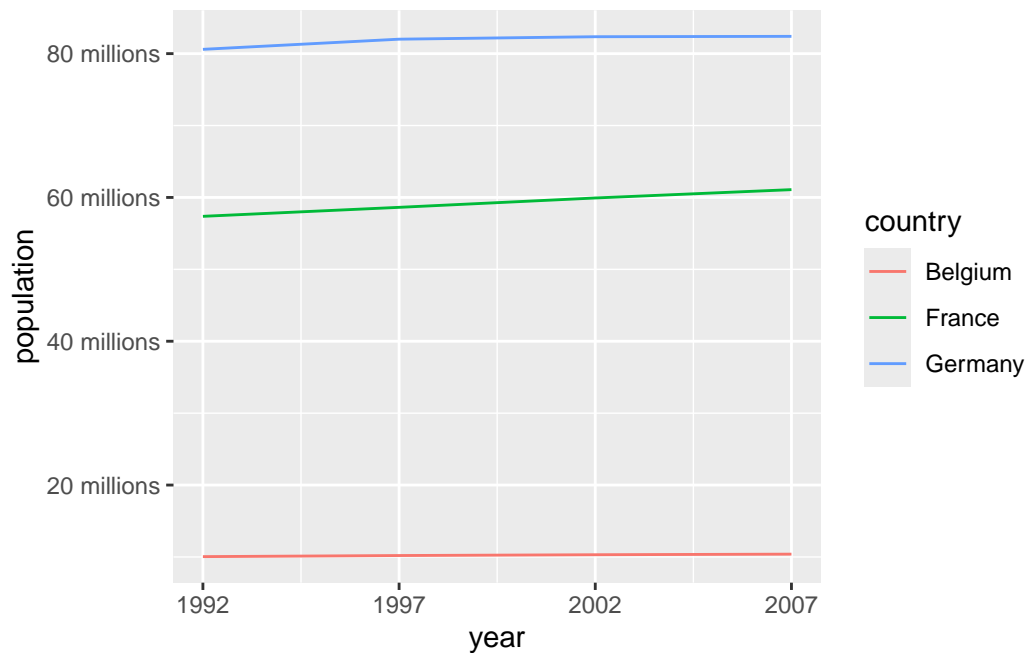
country	1992	1997	2002	2007
Belgium	10045622	10199787	10311970	10392226
France	57374179	58623428	59925035	61083916
Germany	80597764	82011073	82350671	82400996

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	year	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 :

```
library(tidyverse)
ggplot(d) +
  aes(x = year, y = population, color = country) +
  geom_line() +
  scale_x_continuous(breaks = unique(d$year)) +
  scale_y_continuous(
    labels = scales::label_number(
      scale = 10^-6,
      suffix = " millions"
    )
  )
)
```



C'est la même chose pour `{dplyr}`, par exemple si on voulait calculer la population minimale pour chaque pays avec `dplyr::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
```

36.2 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, si 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 l'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
France	1992	57374179
Germany	1992	80597764
Belgium	1997	10199787
France	1997	58623428
Germany	1997	82011073
Belgium	2002	10311970
France	2002	59925035
Germany	2002	82350671
Belgium	2007	10392226
France	2007	61083916

country	annee	population
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	flights_per_year
2013	1	1	517	UA	United Air Lines Inc.	58665
2013	1	1	533	UA	United Air Lines Inc.	58665
2013	1	1	542	AA	American Airlines Inc.	32729
2013	1	1	554	UA	United Air Lines Inc.	58665
2013	1	1	558	AA	American Airlines Inc.	32729
2013	1	1	558	UA	United Air Lines Inc.	58665
2013	1	1	558	UA	United Air Lines Inc.	58665
2013	1	1	559	AA	American Airlines Inc.	32729

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	flights_per_year
UA	United Air Lines Inc.	58665
AA	American Airlines Inc.	32729

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.

L'objectif de `{tidyr}` 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* (`|>`, cf. Chapitre 7), les *pipelines*, permettant d'enchaîner les opérations.

36.3 `pivot_longer()` : rassembler des colonnes

Prenons le tableau d suivant, qui liste la population de 4 pays en 2002 et 2007 :

country	2002	2007
Belgium	10311970	10392226
France	59925035	61083916
Germany	82350671	82400996
Spain	40152517	40448191

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 `tidyr::pivot_longer()` :

```
d |>
  pivot_longer(
    cols = c(`2002`, `2007`),
    names_to = "annee",
    values_to = "population"
  )
```

```
# A tibble: 8 x 3
  country annee population
<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 Spain    2002    40152517
8 Spain    2007    40448191

```

La fonction `tidyr::pivot_longer()` prend comme arguments la liste des colonnes à rassembler (on peut également y utiliser les différentes fonctions de sélection de variables utilisables avec `dplyr::select()`), ainsi que deux arguments `names_to` et `values_to` :

- `names_to` est le nom de la colonne qui va contenir les “noms” des colonnes originelles, c’est-à-dire les identifiants des différentes observations
- `values_to` est le nom de la colonne qui va contenir la valeur des observations

Parfois il est plus rapide d’indiquer à `tidyr::pivot_longer()` les colonnes qu’on ne souhaite pas rassembler. On peut le faire avec la syntaxe suivante :

```

d |>
  pivot_longer(
    -country,
    names_to = "annee",
    values_to = "population"
  )

```

```

# A tibble: 8 x 3
  country annee population
<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 Spain   2002      40152517
8 Spain   2007      40448191

```

36.4 pivot_wider() : disperser des lignes

La fonction `tidyr::pivot_wider()` est l’inverse de `tidyr::pivot_longer()`.

Soit le tableau d suivant :

country	continent	year	variable	value
Belgium	Europe	2002	lifeExp	78.320
Belgium	Europe	2007	lifeExp	79.441
France	Europe	2002	lifeExp	79.590
France	Europe	2007	lifeExp	80.657
Germany	Europe	2002	lifeExp	78.670
Germany	Europe	2007	lifeExp	79.406
Belgium	Europe	2002	pop	10311970.000
Belgium	Europe	2007	pop	10392226.000
France	Europe	2002	pop	59925035.000
France	Europe	2007	pop	61083916.000
Germany	Europe	2002	pop	82350671.000
Germany	Europe	2007	pop	82400996.000

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 `tidyr::pivot_wider()` pour disperser 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
```

`tidyr::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 manquantes sont indiquées avec `NA`).

Exemple avec le tableau `d` suivant :

country	continent	year	variable	value
Belgium	Europe	2002	lifeExp	78.320
Belgium	Europe	2007	lifeExp	79.441
France	Europe	2002	lifeExp	79.590
France	Europe	2007	lifeExp	80.657
Germany	Europe	2002	lifeExp	78.670
Germany	Europe	2007	lifeExp	79.406
Belgium	Europe	2002	pop	10311970.000
Belgium	Europe	2007	pop	10392226.000
France	Europe	2002	pop	59925035.000
France	Europe	2007	pop	61083916.000
Germany	Europe	2002	pop	82350671.000
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

```

36.5 separate() : séparer une colonne en plusieurs colonnes

Parfois on a plusieurs informations réunies en une seule colonne et on souhaite les séparer. Soit le tableau d'exemple caricatural suivant, nommé `df` :

```

df <- tibble(
  eleve = c("Alex Petit", "Bertrand Dupont", "Corinne Durand"),
  note = c("5/20", "6/10", "87/100")
)
df

```

```

# A tibble: 3 x 2
  eleve      note
  <chr>    <chr>
1 Alex Petit 5/20
2 Bertrand Dupont 6/10
3 Corinne Durand 87/100

```

`tidyr::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 Alex Petit      5      20
2 Bertrand Dupont 6      10
3 Corinne Durand 87     100
```

`tidyr::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 `tidyr::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 |>
  tidyr::separate(
    eleve,
    c("prenom", "nom"),
    sep = " "
  )
```

```
# A tibble: 3 x 3
  prenom    nom    note
  <chr>    <chr> <chr>
1 Alex    Petit 5/20
2 Bertrand Dupont 6/10
3 Corinne Durand 87/100
```

36.6 `separate_rows()` : séparer une colonne en plusieurs lignes

La fonction `tidyr::separate_rows()` est utile lorsque plusieurs valeurs sont contenues dans la même variable. Mais, alors que `tidyr::separate()` permet de répartir ces différentes valeurs dans plusieurs colonnes, `tidyr::separate_rows()` va créer une ligne pour chaque valeur. Prenons cet exemple trivial où les différentes notes de chaque élève sont contenues dans la colonne `notes`.

```
df <- tibble(
  eleve = c("Alex Petit", "Bertrand Dupont", "Corinne Durand"),
  notes = c("10,15,16", "18,12,14", "16,17")
)
df
```

```
# A tibble: 3 x 2
  eleve      notes
  <chr>      <chr>
1 Alex Petit 10,15,16
2 Bertrand Dupont 18,12,14
3 Corinne Durand 16,17
```

Appliquons `tidyr::separate_rows()`.

```
df |>
  separate_rows(notes) |>
  rename(note = notes)
```

```
# A tibble: 8 x 2
  eleve      note
  <chr>      <chr>
1 Alex Petit 10
2 Alex Petit 15
3 Alex Petit 16
4 Bertrand Dupont 18
5 Bertrand Dupont 12
6 Bertrand Dupont 14
7 Corinne Durand 16
8 Corinne Durand 17
```

Par défaut `tidyr::separate_rows()` sépare les valeurs dès qu'elle trouve un caractère qui ne soit ni un chiffre ni une lettre, mais on peut spécifier le séparateur à l'aide de l'argument `sep` (qui accepte une chaîne de caractère ou même une expression régulière) :

```
df |>
  separate_rows(notes, sep = ",") |>
  rename(note = notes)
```

```
# A tibble: 5 x 2
  eleve      note
  <chr>      <chr>
1 Alex Petit 10
2 Alex Petit 15
3 Alex Petit 16
4 Bertrand Dupont 18
5 Bertrand Dupont 12
```

```
6 Bertrand Dupont 14
7 Corinne Durand 16
8 Corinne Durand 17
```

36.7 unite() : regrouper plusieurs colonnes en une seule

`tidyr::unite()` est l'opération inverse de `tidyr::separate()`. Elle permet de regrouper plusieurs colonnes en une seule. Imaginons qu'on obtient le tableau `d` suivant :

code_département	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 `tidyr::unite()` pour cela on indique d'abord le nom de la nouvelle variable puis la liste des variables à concaténer :

```
d |>
  unite(code_insee, code_département, 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 `tidyr::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
```

36.8 extract() : créer de nouvelles colonnes à partir d'une colonne de texte

`tidyr::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
Alex Petit	5/20
Bertrand Dupont	6/10
Corinne Durand	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 Alex    Petit  5/20
2 Bertrand Dupont 6/10
3 Corinne Durand 87/100
```

On passe donc à `tidyr::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 |>
  tidyr::extract(
    eleve,
    c("initiale_prenom", "initiale_nom"),
    "^(.).* (.)*$",
    remove = FALSE
  )
```

```
# A tibble: 3 x 4
  eleve          initiale_prenom initiale_nom note
  <chr>          <chr>           <chr>      <chr>
1 Alex Petit    A                P        5/20
2 Bertrand Dupont B                D        6/10
3 Corinne Durand C                D       87/100
```

36.9 complete() : compléter des combinaisons de variables manquantes

Imaginons qu'on ait le tableau de résultats suivants :

eleve	matiere	note
Alain	Maths	16

eleve	matiere	note
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 `tidyr::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 `tidyr::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 `tidyr::complete()` comme précédemment, le résultat n'est pas bon car il génère des valeurs manquantes pour `id`.

```
df |>
  complete(eleve, matiere)
```

```
# A tibble: 6 x 4
  eleve   matiere      id  note
  <chr>   <chr>   <dbl> <dbl>
1 Alain Français 1001001     9
2 Alain Maths   1001001    16
3 Barnabé Français    NA    NA
4 Barnabé Maths   1001002    17
5 Chantal Français 1001003    11
6 Chantal Maths    NA    NA
```

Et si nous ajoutons `id` dans l'appel de la fonction, nous obtenons 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 `tidyr::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 `tidyr::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

36.10 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* (doi: [10.18637/jss.v059.i10](https://doi.org/10.18637/jss.v059.i10)), présente de manière détaillée le concept éponyme (mais il utilise des extensions désormais obsolètes qui ont depuis été remplacées par `{dplyr}` et `{tidyr}`).

Le site de l’extension est accessible à l’adresse : <http://tidyr.tidyverse.org/> et contient une liste des fonctions et les pages d’aide associées.

En particulier, on pourra se référer à la [vignette dédiée](#) à `tidyr::pivot_wider()` et `tidyr::pivot_longer()` pour des exemples avancés de réorganisation des données.

Pour des usages avancés, il est possible avec `{tidyr}` de gérer des données nichées (*nested data*), c’est-à-dire des tableaux de données dans des tableaux de données. Ces fonctionnalités, réservées aux utilisateurs avancés, sont décrites dans une [vignette spécifique](#).

36.11 Fichiers volumineux

Si l’on a des tableaux de données particulièrement volumineux (plusieurs Go), les fonctions de `{tidyr}` ne sont pas les plus performantes.

On aura alors intérêt à regarder du côté des fonctions `data.table::melt()` et `data.table::dcast()` de l’extension `{data.table}` développées pour optimiser la performance sur les grands tableaux de données.

Pour plus de détails, voir la vignette dédiée : <https://rdatatable.gitlab.io/data.table/articles/datatable-reshape.html>

36.12 webin-R

Le package `{tidyr}` est évoqué sur YouTube dans le [webin-R #13](#) (*exemples de graphiques avancés*) et le [webin-R #17](#) (*trajectoires de soins : un exemple de données longitudinales*).

<https://youtu.be/5sD4Z8bTIMM>

<https://youtu.be/JV1Srrg09oI>

37 Conditions logiques

Dans ce chapitre, nous allons aborder les conditions et vecteurs logiques qui sont composés de trois valeurs possibles : **TRUE** (vrai), **FALSE** (faux) et **NA** (manquant). Les vecteurs logiques sont notamment utilisés pour sélectionner des observations, par exemple avec `dplyr::filter()`.

Nous avons également déjà abordé les conditions pour combiner ensemble plusieurs variables (cf. Chapitre 10), notamment avec `dplyr::if_else()` ou `dplyr::case_when()`.

37.1 Opérateurs de comparaison

Une manière commune de créer un vecteur logique consiste à utiliser l'un des opérateurs de comparaison suivants : `<` (strictement inférieur), `<=` (inférieur ou égal), `>` (strictement supérieur), `>=` (supérieur ou égal), `==` (est égal à), `!=` (est différent de).

On peut comparer un vecteur de plusieurs valeurs avec une valeur unique.

```
x <- c(1, 5, 2, 8)
x < 3
```

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

Si l'on prend deux vecteurs de même longueur, la comparaison se fera ligne à ligne.

```
y <- c(3, 5, 1, 7)
y >= x
```

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

```
y == x
```

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



```
y != x
```

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

On peut ainsi facilement sélectionner des lignes d'un tableau de données à partir d'une condition sur certaines variables.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr      1.1.4      v readr      2.1.5
v forcats    1.0.0      v stringr    1.5.1
v ggplot2     3.5.1      v tibble     3.2.1
v lubridate   1.9.3      v tidyr      1.3.1
v purrr       1.0.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
data("hdv2003", package = "questionr")
hdv2003 |> nrow()
```

```
[1] 2000
```

```
hdv2003 |>
  filter(sexe == "Femme") |>
  nrow()
```

```
[1] 1101
```

```
hdv2003 |>
  filter(age < 25) |>
  nrow()
```

```
[1] 169
```

💡 Tester l'égalité d'une valeur décimale

Lorsque l'on effectue un test d'égalité avec des valeurs décimales, le test échouera si les deux valeurs ne sont pas parfaitement identiques.

Prenons un exemple :

```
x <- 1 / 49 * 49  
x
```

```
[1] 1
```

```
x == 1
```

```
[1] FALSE
```

Pourquoi ce test échoue-t-il ? Le nombre de décimales stockées par l'ordinateur est limité et, de ce fait, il peut y avoir quelques écarts d'arrondis. Ainsi, `x` n'est pas tout à fait égal à 1, ce qui devient visible si on l'affiche avec un nombre élevé de décimales.

```
print(x, digits = 16)
```

```
[1] 0.9999999999999999
```

Dans ce cas là, on pourra avoir recours à `dplyr::near()` qui prendra en compte la précision de l'ordinateur dans la comparaison.

```
near(x, 1)
```

```
[1] TRUE
```

On peut aussi utiliser cette fonction en personnalisant le niveau de tolérance pour la comparaison.

```
near(c(2.1, 3.4), 2, tol = 1)
```

```
[1] TRUE FALSE
```

37.2 Comparaison et valeurs manquantes

Les valeurs manquantes (NA) peuvent être parfois problématiques lors d'une comparaison car elles renvoient systématiquement une valeur manquante.

```
2 < NA
```

```
[1] NA
```

```
NA == 6
```

```
[1] NA
```

Lorsque l'on sélectionne des observations avec la syntaxe des crochets (`[]`, voir Chapitre 4), cela va générer des lignes vides / manquantes.

```
d <- tibble(  
  a = c(1, NA, 3, 4),  
  b = c("x", "y", "x", "y")  
)  
d[d$a > 2, ]
```

```
# A tibble: 3 x 2  
      a b  
  <dbl> <chr>  
1    NA <NA>  
2     3 x  
3     4 y
```

Le recours à `dplyr::filter()` est plus sûr car les lignes pour lesquelles la condition renvoie NA ne sont pas sélectionnées.

```
d |> filter(a > 2)
```

```
# A tibble: 2 x 2  
      a b  
  <dbl> <chr>  
1     3 x  
2     4 y
```

L'opérateur `==` ne peut pas être utilisé pour tester si une valeur est manquante. On utilisera à la place la fonction `is.na()`.

```
d$a == NA
```

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

```
is.na(d$a)
```

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

Astuce

Voici deux petites fonctions permettant de tester si deux valeurs sont identiques ou différentes, en tenant compte des NA comme l'un des valeurs possibles (deux NA seront alors considérés comme égaux).

```
is_different <- function(x, y) {  
  (x != y & !is.na(x) & !is.na(y)) | xor(is.na(x), is.na(y))  
}
```

```
is_equal <- function(x, y) {  
  (x == y & !is.na(x) & !is.na(y)) | (is.na(x) & is.na(y))  
}
```

```
v <- c(1, NA, NA, 2)
```

```
w <- c(1, 2, NA, 3)
```

```
v == w
```

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

```
is_equal(v, w)
```

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

```
v != w
```

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

```
is_different(v, w)
```

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

37.3 Opérateurs logiques (algèbre booléenne)

Les opérateurs logiques permettent de combiner ensemble plusieurs vecteurs logiques :

- `&` : opérateur et (`x & y` est vrai si à la fois `x` et `y` sont vrais) ;
- `|` : opérateur ou (`x | y` est vrai si `x` ou `y` ou les deux sont vrais) ;
- `xor()` : opérateur ou exclusif (`xor(x, y)` est vrai si seulement `x` ou seulement `y` est vrai, mais pas les deux) ;
- `!` : opérateur non (`!x` est vrai si `x` est faux).

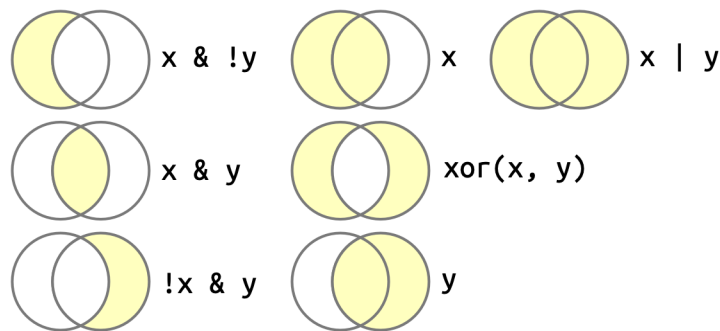


Figure 37.1: Représentation graphique de l'ensemble des opérations logiques. Le cercle de gauche représente `x` et celui de droite `y`. La région colorée représente le résultat de l'opération.

Ils permettent de combiner plusieurs conditions entre elles.

```
hdv2003 |>
  filter(sexe == "Femme" & age < 25) |>
  nrow()
```

```
[1] 93
```

```
hdv2003 |>
  filter(sexe == "Femme" | age < 25) |>
  nrow()
```

```
[1] 1177
```

Pour des conditions complexes, on utilisera des parenthèses pour indiquer dans quel ordre effectuer les opérations.

```
# sélectionne les jeunes femmes et les hommes âgés
hdv2003 |>
  filter(
    (sexe == "Femme" & age < 25) |
    (sexe == "Homme" & age > 60)) |>
  nrow()
```

```
[1] 315
```

37.3.1 Opérations logiques et Valeurs manquantes

On sera vigilant avec les valeurs manquantes. Cela peut paraître un peu obscur au premier abord, mais est en fait parfaitement logique.

```
df <- tibble(x = c(TRUE, FALSE, NA))

df |>
  mutate(
    et_na = x & NA,
    ou_na = x | NA
  )
```

```
# A tibble: 3 x 3
  x      et_na ou_na
<lgl> <lgl> <lgl>
1 TRUE  NA     TRUE
2 FALSE FALSE  NA
3 NA    NA     NA
```

TRUE | NA vaut TRUE car la condition reste vraie quelle que soit la valeur du deuxième paramètre, tandis que FALSE | NA renvoie NA car le résultat est indéterminé (il dépend du deuxième paramètre).

37.3.2 L'opérateur %in%

Il est fréquent de vouloir tester simultanément plusieurs égalités. Par exemple :

```
x <- c("a", "b", "c", "d")
x == "a" | x == "b"
```

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

On aura alors avantageusement recours à l'opérateur `%in%` que l'on peut traduire par appartient à et qui teste si les éléments appartiennent à un certain ensemble.

```
x %in% c("a", "b")
```

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

37.4 Aggrégation

Pour résumer un ensemble de valeurs logiques en une seule, on utilisera les fonction `all()` et `any()` qui teste si toutes les valeurs / au moins une valeur est vrai. Ces deux fonctions acceptent un argument `na.rm` permettant de ne pas tenir compte des valeurs manquantes.

```
x <- c(TRUE, NA, FALSE, FALSE)
any(x)
```

```
[1] TRUE
```

```
all(x)
```

```
[1] FALSE
```

Un vecteur logique peut-être vu comme un vecteur de valeur binaire (0 si `FALSE`, 1 si `TRUE`). On peut dès lors effectuer des opérations comme la somme ou la moyenne.

```
sum(x, na.rm = TRUE)
```

```
[1] 1
```

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

```
[1] 0.3333333
```

37.5 Programmation

Lorsque l'on programme avec R, notamment avec des structures conditionnelles telles que `if ... else ...`, on a besoin d'écrire des conditions qui ne renvoient qu'une et une seule valeur logique.

Les opérateurs `&` et `|` s'appliquent sur des vecteurs et donc renvoient potentiellement plusieurs valeurs. On privilégiera alors les variantes `&&` et `||` qui ne renvoient qu'une seule valeur et produise une erreur sinon.

De même, pour vérifier qu'un objet est bien égal à `TRUE` ou à `FALSE`, n'est pas nul, n'est pas manquant et est de longueur 1, on utilisera `isTRUE()` et `isFALSE()`.

```
isTRUE(TRUE)
```

```
[1] TRUE
```

```
isTRUE(NA)
```

```
[1] FALSE
```

```
isTRUE(NULL)
```

```
[1] FALSE
```

```
isTRUE(c(TRUE, TRUE))
```

```
[1] FALSE
```


38 Transformations multiples

38.1 Transformations multiples sur les colonnes

Il est souvent utile d'effectuer la même opération sur plusieurs colonnes, mais le copier-coller est à la fois fastidieux et source d'erreurs :

```
df %>%
  group_by(g1, g2) %>%
  summarise(
    a = mean(a),
    b = mean(b),
    c = mean(c),
    d = mean(d)
  )
```

Dans cette section, nous allons introduire `dplyr::across()` qui permet de réécrire la même commande de manière plus succincte.

```
df %>%
  group_by(g1, g2) %>%
  summarise(across(a:d, mean))
```

38.1.1 Usage de base

`dplyr::across()` a deux arguments principaux :

- le premier, `.cols`, permet de sélectionner les colonnes sur lesquelles on souhaite agir et accepte la même syntaxe de `dplyr::select()` ;
- le second, `.fns`, est une fonction (ou une liste de fonctions) à appliquer à chaque colonne sélectionnée.

Voici quelques exemples avec `dplyr::summarise()`.

Dans ce premier exemple, nous utilisons `tidyselect::where()` qui permet de sélectionner les colonnes en fonction de leur type (ici les colonnes textuelles car `where()` est utilisé en

conjonction avec la fonction `is.character()`). Notez que l'on passe `is.character()` sans ajouter de parenthèse. En effet, `is.character` **renvoie** la fonction du même nom, tandis que `is.character()` **appelle** la fonction pour l'exécuter. La fonction `dplyr::n_distinct()`, quant à elle, compte le nombre de valeurs uniques. Le tableau ci-dessous renvoie donc, pour chaque variable textuelle, le nombre de valeurs uniques observées dans les données.

```
library(tidyverse)
starwars |>
  summarise(across(where(is.character), n_distinct))
```

```
# A tibble: 1 x 8
  name hair_color skin_color eye_color sex gender homeworld species
<int>   <int>      <int>    <int> <int> <int>      <int> <int>
1    87        12        31        15    5    3        49    38
```

Dans ce second exemple, nous indiquons simplement la liste de nos variables d'intérêt.

```
starwars |>
  group_by(species) |>
  filter(n() > 1) |>
  summarise(across(c(sex, gender, homeworld), n_distinct))
```

```
# A tibble: 9 x 4
  species    sex gender homeworld
<chr>    <int> <int>    <int>
1 Droid      1     2      3
2 Gungan     1     1      1
3 Human      2     2     15
4 Kaminoan   2     2      1
5 Mirialan   1     1      1
6 Twi'lek    2     2      1
7 Wookiee    1     1      1
8 Zabrak     1     1      2
9 <NA>       1     1      3
```

Dans ce troisième exemple, nous allons calculer la moyenne pour chaque variable numérique.

```
starwars %>%
  group_by(homeworld) |>
  filter(n() > 1) |>
  summarise(across(where(is.numeric), mean))
```

```
# A tibble: 10 x 4
  homeworld height  mass birth_year
  <chr>      <dbl> <dbl>      <dbl>
1 Alderaan   176.   NA         NA
2 Corellia   175  78.5       25
3 Coruscant  174.   NA         NA
4 Kamino     208.   NA         NA
5 Kashyyyk   231  124        NA
6 Mirial     168  53.1       49
7 Naboo      177.   NA         NA
8 Ryloth     179   NA         NA
9 Tatooine   170.   NA         NA
10 <NA>       NA    NA         NA
```

Il y a beaucoup de valeurs manquantes. Nous devons donc passer `na.rm = TRUE` à `mean()`. Différentes approches sont possibles :

- écrire notre propre fonction `ma_fonction()` ;
- utiliser `purrr::partial()` qui permet de renvoyer une fonction avec des valeurs par défaut différentes ;
- la syntaxe native de **R** pour déclarer des fonctions anonymes avec le raccourci `\(arg) expr` ;
- une formule définissant une fonction dans le style du package **purrr**, c'est-à-dire une formule commençant par `~` et dont le premier argument sera noté `.x`¹.

```
ma_fonction <- function(x) {mean(x, na.rm = TRUE)}
starwars %>%
  group_by(homeworld) |>
  filter(n() > 1) |>
  summarise(across(where(is.numeric), ma_fonction))
```

```
# A tibble: 10 x 4
  homeworld height  mass birth_year
  <chr>      <dbl> <dbl>      <dbl>
1 Alderaan   176.   64         43
2 Corellia   175  78.5       25
3 Coruscant  174.   50         91
4 Kamino     208.  83.1      31.5
5 Kashyyyk   231  124        200
6 Mirial     168  53.1       49
```

¹Cette syntaxe particulière n'est compatible que dans certaines fonctions du `{tidyverse}`. Ce n'est pas une syntaxe standard de **R**.

7 Naboo	177.	64.2	55
8 Ryloth	179	55	48
9 Tatooine	170.	85.4	54.6
10 <NA>	139.	82	334.

```
starwars %>%
  group_by(homeworld) |>
  filter(n() > 1) |>
  summarise(across(where(is.numeric), purrr::partial(mean, na.rm = TRUE)))
```

```
# A tibble: 10 x 4
  homeworld height  mass birth_year
  <chr>      <dbl> <dbl>      <dbl>
1 Alderaan  176.   64         43
2 Corellia  175   78.5        25
3 Coruscant 174.   50         91
4 Kamino    208.  83.1       31.5
5 Kashyyyk  231  124        200
6 Mirial    168   53.1        49
7 Naboo     177.  64.2        55
8 Ryloth    179   55         48
9 Tatooine  170.  85.4       54.6
10 <NA>     139.  82         334.
```

```
starwars %>%
  group_by(homeworld) |>
  filter(n() > 1) |>
  summarise(across(where(is.numeric), \(x) {mean(x, na.rm = TRUE)}))
```

```
# A tibble: 10 x 4
  homeworld height  mass birth_year
  <chr>      <dbl> <dbl>      <dbl>
1 Alderaan  176.   64         43
2 Corellia  175   78.5        25
3 Coruscant 174.   50         91
4 Kamino    208.  83.1       31.5
5 Kashyyyk  231  124        200
6 Mirial    168   53.1        49
7 Naboo     177.  64.2        55
8 Ryloth    179   55         48
9 Tatooine  170.  85.4       54.6
10 <NA>     139.  82         334.
```

```
starwars %>%
  group_by(homeworld) |>
  filter(n() > 1) |>
  summarise(across(where(is.numeric), ~ mean(.x, na.rm = TRUE)))
```

```
# A tibble: 10 x 4
  homeworld height  mass birth_year
  <chr>      <dbl> <dbl>      <dbl>
1 Alderaan   176.   64         43
2 Corellia   175  78.5        25
3 Coruscant  174.   50         91
4 Kamino     208.  83.1       31.5
5 Kashyyyk   231  124        200
6 Mirial     168  53.1        49
7 Naboo      177.  64.2        55
8 Ryloth     179   55         48
9 Tatooine   170.  85.4       54.6
10 <NA>      139.  82         334.
```

Comme `dplyr::across()` est souvent utilisée au sein de `dplyr::mutate()` ou de `dplyr::summarise()`, les variables de groupement ne sont jamais sélectionnée par `dplyr::across()` pour éviter tout accident.

```
df <- data.frame(
  g = c(1, 1, 2),
  x = c(-1, 1, 3),
  y = c(-1, -4, -9)
)
df %>%
  group_by(g) %>%
  summarise(across(where(is.numeric), sum))
```

```
# A tibble: 2 x 3
  g     x     y
  <dbl> <dbl> <dbl>
1     1     0    -5
2     2     3    -9
```

38.1.2 Fonctions multiples

Vous pouvez transformer chaque variable avec plus d'une fonction en fournissant une liste nommée de fonctions dans le deuxième argument :

```
min_max <- list(
  min = \(x) min(x, na.rm = TRUE),
  max = \(x) max(x, na.rm = TRUE)
)
starwars |>
  summarise(across(where(is.numeric), min_max))
```

```
# A tibble: 1 x 6
  height_min height_max mass_min mass_max birth_year_min birth_year_max
    <int>      <int>    <dbl>    <dbl>         <dbl>         <dbl>
1         66        264      15    1358             8          896
```

On peut contrôler le nom des variables produites avec l'option `.names` qui prend une chaîne de caractère au format du package [glue](#).

```
starwars |>
  summarise(
    across(
      where(is.numeric),
      min_max,
      .names = "{.fn}.{.col}"
    )
  )
```

```
# A tibble: 1 x 6
  min.height max.height min.mass max.mass min.birth_year max.birth_year
    <int>      <int>    <dbl>    <dbl>         <dbl>         <dbl>
1         66        264      15    1358             8          896
```

38.1.3 Accéder à la colonne courante

Si vous en avez besoin, vous pouvez accéder au nom de la colonne courante à l'intérieur d'une fonction en appelant `dplyr::cur_column()`. Cela peut être utile si vous voulez effectuer une sorte de transformation dépendante du contexte qui est déjà encodée dans un vecteur :

```
df <- tibble(x = 1:3, y = 3:5, z = 5:7)
mult <- list(x = 1, y = 10, z = 100)

df |>
  mutate(
    across(
      all_of(names(mult)),
      ~ .x * mult[[cur_column()]]
    )
  )
```

```
# A tibble: 3 x 3
      x     y     z
<dbl> <dbl> <dbl>
1     1    30   500
2     2    40   600
3     3    50   700
```

Jusqu'à présent, nous nous sommes concentrés sur l'utilisation de `across()` avec `summarise()`, mais cela fonctionne avec n'importe quel autre verbe `{dplyr}` qui utilise le masquage de données.

Par exemple, nous pouvons rééchelonner toutes les variables numériques pour se situer entre 0 et 1.

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df <- tibble(x = 1:4, y = rnorm(4))
df |>
  mutate(across(where(is.numeric), rescale01))
```

```
# A tibble: 4 x 2
      x     y
<dbl> <dbl>
1  0     0
2 0.333 0.402
3 0.667 1
4 1     0.791
```

38.1.4 pick()

Pour certains verbes, comme `dplyr::group_by()`, `dplyr::count()` et `dplyr::distinct()`, il n'est pas nécessaire de fournir une fonction de résumé, mais il peut être utile de pouvoir sélectionner dynamiquement un ensemble de colonnes.

Dans ce cas, nous recommandons d'utiliser le complément de `dplyr::across()`, `dplyr::pick()`, qui fonctionne comme `across()` mais n'applique aucune fonction et renvoie à la place un cadre de données contenant les colonnes sélectionnées.

```
starwars |>
  distinct(pick(contains("color")))
```

```
# A tibble: 67 x 3
  hair_color    skin_color eye_color
  <chr>         <chr>      <chr>
1 blond        fair        blue
2 <NA>          gold        yellow
3 <NA>          white, blue red
4 none         white        yellow
5 brown        light        brown
6 brown, grey  light        blue
7 brown        light        blue
8 <NA>          white, red  red
9 black        light        brown
10 auburn, white fair        blue-gray
# i 57 more rows
```

```
starwars |>
  count(pick(contains("color")), sort = TRUE)
```

```
# A tibble: 67 x 4
  hair_color skin_color eye_color    n
  <chr>      <chr>      <chr>  <int>
1 brown     light     brown     6
2 brown     fair      blue      4
3 none      grey     black      4
4 black     dark     brown      3
5 blond     fair      blue      3
6 black     fair     brown      2
7 black     tan      brown      2
8 black     yellow   blue      2
```



```

 9 brown      fair      brown      2
10 none      white     yellow     2
# i 57 more rows

```

`dplyr::across()` ne fonctionne pas avec `dplyr::select()` ou `dplyr::rename()` parce qu'ils utilisent déjà une syntaxe de sélection dynamique. Si vous voulez transformer les noms de colonnes avec une fonction, vous pouvez utiliser `dplyr::rename_with()`.

38.2 Sélection de lignes à partir d'une sélection de colonnes

Nous ne pouvons pas utiliser directement `across()` dans `dplyr::filter()` car nous avons besoin d'une étape supplémentaire pour combiner les résultats. À cette fin, `filter()` dispose de deux fonctions complémentaires spéciales :

`dplyr::if_any()` conserve les lignes pour lesquelles le prédicat est vrai pour *au moins une* colonne sélectionnée :

```

starwars |>
  filter(if_any(everything(), ~ !is.na(.x)))

```

```

# A tibble: 87 x 14
   name      height  mass hair_color skin_color eye_color birth_year sex  gender
   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
1 Luke Sk~    172    77 blond      fair        blue        19   male masculin
2 C-3P0      167    75 <NA>      gold        yellow     112   none masculin
3 R2-D2       96    32 <NA>      white, bl~ red         33   none masculin
4 Darth V~   202   136 none      white       yellow     41.9 male masculin
5 Leia Or~   150    49 brown      light       brown       19   fema~ féminin
6 Owen La~   178   120 brown, gr~ light       blue        52   male masculin
7 Beru Wh~   165    75 brown      light       blue        47   fema~ féminin
8 R5-D4       97    32 <NA>      white, red red         NA   none masculin
9 Biggs D~   183    84 black      light       brown       24   male masculin
10 Obi-Wan~   182    77 auburn, w~ fair        blue-gray   57   male masculin
# i 77 more rows
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>

```

`dplyr::if_all()` sélectionne les lignes pour lesquelles le prédicat est vrai pour *toutes* les colonnes sélectionnées :

```
starwars |>
  filter(if_all(everything(), ~ !is.na(.x)))
```

```
# A tibble: 29 x 14
  name      height  mass hair_color skin_color eye_color birth_year sex  gender
  <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
1 Luke Sk~    172    77 blond      fair        blue        19   male masculi~
2 Darth V~    202   136 none       white       yellow      41.9 male masculi~
3 Leia Or~    150    49 brown      light       brown       19   fema~ feminin~
4 Owen La~    178   120 brown, gr~ light       blue       52   male masculi~
5 Beru Wh~    165    75 brown      light       blue       47   fema~ feminin~
6 Biggs D~    183    84 black      light       brown       24   male masculi~
7 Obi-Wan~    182    77 auburn, w~ fair        blue-gray   57   male masculi~
8 Anakin ~    188    84 blond      fair        blue       41.9 male masculi~
9 Chewbac~    228   112 brown      unknown     blue       200   male masculi~
10 Han Solo    180    80 brown      fair        brown       29   male masculi~
# i 19 more rows
# i 5 more variables: homeworld <chr>, species <chr>, films <list>,
#   vehicles <list>, starships <list>
```

38.3 Transformations multiples sur les lignes

{dplyr}, et **R** de manière générale, sont particulièrement bien adaptés à l'exécution d'opérations sur les colonnes, alors que l'exécution d'opérations sur les lignes est beaucoup plus difficile. Ici, nous verrons comment réaliser des calculs ligne par ligne avec `dplyr::rowwise()`.

38.3.1 Création

Les opérations par ligne requièrent un type spécial de regroupement où chaque groupe est constitué d'une seule ligne. Vous créez ce type de groupe avec `dplyr::rowwise()` :

```
df <- tibble(x = 1:2, y = 3:4, z = 5:6)
df |> rowwise()
```

```
# A tibble: 2 x 3
# Rowwise:
      x     y     z
  <int> <int> <int>
```

1	1	3	5
2	2	4	6

Comme `group_by()`, `rowwise()` ne fait rien en soi ; elle modifie simplement le fonctionnement des autres verbes. Par exemple, comparez les résultats de `mutate()` dans le code suivant :

```
df |>
  mutate(m = mean(c(x, y, z)))
```

```
# A tibble: 2 x 4
      x     y     z     m
  <int> <int> <int> <dbl>
1     1     3     5  3.5
2     2     4     6  3.5
```

```
df |>
  rowwise() |>
  mutate(m = mean(c(x, y, z)))
```

```
# A tibble: 2 x 4
# Rowwise:
      x     y     z     m
  <int> <int> <int> <dbl>
1     1     3     5     3
2     2     4     6     4
```

Si vous utilisez `mutate()` avec un tableau de données classique, il calcule la moyenne de `x`, `y` et `z` sur toutes les lignes. Si vous l'appliquez à un tableau de données *row-wise*, il calcule la moyenne séparément pour chaque ligne.

Vous pouvez optionnellement fournir des variables identifiantes dans votre appel à `rowwise()`. Ces variables sont conservées lorsque vous appelez `summarise()`, de sorte qu'elles se comportent de manière similaire aux variables de regroupement passées à `group_by()`:

```
df <- tibble(
  name = c("Mara", "Hadley"),
  x = 1:2,
  y = 3:4,
  z = 5:6
)
```

```
df |>
  rowwise() |>
  summarise(m = mean(c(x, y, z)))
```

```
# A tibble: 2 x 1
      m
  <dbl>
1     3
2     4
```

```
df |>
  rowwise(name) |>
  summarise(m = mean(c(x, y, z)))
```

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

```
# A tibble: 2 x 2
# Groups:   name [2]
  name     m
  <chr> <dbl>
1 Mara     3
2 Hadley    4
```

`rowwise()` n'est qu'une forme spéciale de regroupement : donc si vous voulez enlever sa déclaration, appelez simplement `ungroup()`.

38.3.2 Statistiques ligne par ligne

`dplyr::summarise()` permet de résumer facilement les valeurs d'une ligne à l'autre à l'intérieur d'une colonne. Combinée à `rowwise()`, elle permet également de résumer les valeurs de plusieurs colonnes à l'intérieur d'une même ligne. Pour voir comment, commençons par créer un petit jeu de données :

```
df <- tibble(
  id = 1:6,
  w = 10:15,
  x = 20:25,
  y = 30:35,
```

```

    z = 40:45
  )
df

```

```

# A tibble: 6 x 5
   id     w     x     y     z
<int> <int> <int> <int> <int>
1     1    10    20    30    40
2     2    11    21    31    41
3     3    12    22    32    42
4     4    13    23    33    43
5     5    14    24    34    44
6     6    15    25    35    45

```

Supposons que nous voulions calculer la somme de **w**, **x**, **y** et **z** pour chaque ligne. Nous pouvons utiliser `mutate()` pour ajouter une nouvelle colonne ou `summarise()` pour renvoyer ce seul résumé :

```

df |>
  rowwise(id) |>
  mutate(total = sum(c(w, x, y, z)))

```

```

# A tibble: 6 x 6
# Rowwise:   id
   id     w     x     y     z total
<int> <int> <int> <int> <int> <int>
1     1    10    20    30    40   100
2     2    11    21    31    41   104
3     3    12    22    32    42   108
4     4    13    23    33    43   112
5     5    14    24    34    44   116
6     6    15    25    35    45   120

```

```

df |>
  rowwise(id) |>
  summarise(total = sum(c(w, x, y, z)))

```

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

```
# A tibble: 6 x 2
# Groups:   id [6]
      id total
  <int> <int>
1     1    100
2     2    104
3     3    108
4     4    112
5     5    116
6     6    120
```

Bien sûr, si vous avez beaucoup de variables, il sera fastidieux de taper chaque nom de variable. Au lieu de cela, vous pouvez utiliser `dplyr::c_across()` qui utilise une syntaxe *tidy selection* afin de sélectionner succinctement de nombreuses variables :

```
df |>
  rowwise(id) |>
  mutate(total = sum(c_across(w:z)))
```

```
# A tibble: 6 x 6
# Rowwise:   id
      id     w     x     y     z total
  <int> <int> <int> <int> <int> <int>
1     1    10    20    30    40    100
2     2    11    21    31    41    104
3     3    12    22    32    42    108
4     4    13    23    33    43    112
5     5    14    24    34    44    116
6     6    15    25    35    45    120
```

```
df |>
  rowwise(id) |>
  mutate(total = sum(c_across(where(is.numeric))))
```

```
# A tibble: 6 x 6
# Rowwise:   id
      id     w     x     y     z total
  <int> <int> <int> <int> <int> <int>
1     1    10    20    30    40    100
2     2    11    21    31    41    104
3     3    12    22    32    42    108
```

4	4	13	23	33	43	112
5	5	14	24	34	44	116
6	6	15	25	35	45	120

Vous pouvez combiner cela avec des opérations par colonne (voir la section précédente) pour calculer la proportion du total pour chaque colonne :

```
df |>
  rowwise(id) |>
  mutate(total = sum(c_across(w:z))) |>
  ungroup() |>
  mutate(across(w:z, ~ . / total))
```

```
# A tibble: 6 x 6
      id      w      x      y      z total
  <int> <dbl> <dbl> <dbl> <dbl> <int>
1     1 0.1   0.2   0.3   0.4   100
2     2 0.106 0.202 0.298 0.394  104
3     3 0.111 0.204 0.296 0.389  108
4     4 0.116 0.205 0.295 0.384  112
5     5 0.121 0.207 0.293 0.379  116
6     6 0.125 0.208 0.292 0.375  120
```

! Important

L'approche `rowwise()` fonctionne pour n'importe quelle fonction de résumé. Mais si vous avez besoin d'une plus grande rapidité, il est préférable de rechercher une variante intégrée de votre fonction de résumé. Celles-ci sont plus efficaces car elles opèrent sur l'ensemble du cadre de données ; elles ne le divisent pas en lignes, ne calculent pas le résumé et ne joignent pas à nouveau les résultats.

Par exemple, **R** fournit nativement les fonctions `rowSums()` et `rowMeans()` pour calculer des sommes et des moyennes par ligne. Elles sont de fait bien plus efficaces.

```
df |>
  mutate(total = rowSums(pick(where(is.numeric), -id)))
```

```
# A tibble: 6 x 6
      id      w      x      y      z total
  <int> <int> <int> <int> <int> <dbl>
1     1    10    20    30    40   100
2     2    11    21    31    41   104
3     3    12    22    32    42   108
```

4	4	13	23	33	43	112
5	5	14	24	34	44	116
6	6	15	25	35	45	120

```
df |>
  mutate(mean = rowMeans(pick(where(is.numeric), -id)))
```

```
# A tibble: 6 x 6
      id     w     x     y     z mean
  <int> <int> <int> <int> <int> <dbl>
1     1    10    20    30    40    25
2     2    11    21    31    41    26
3     3    12    22    32    42    27
4     4    13    23    33    43    28
5     5    14    24    34    44    29
6     6    15    25    35    45    30
```


partie VI

Analyses avancées

39 Analyse factorielle

Il existe plusieurs techniques d'analyse factorielle dont les plus courantes sont l'*analyse en composante principale* (ACP) portant sur des variables quantitatives, l'*analyse factorielle des correspondances* (AFC) portant sur deux variables qualitatives et l'*analyse des correspondances multiples* (ACM) portant sur plusieurs variables qualitatives (il s'agit d'une extension de l'AFC). Pour combiner des variables à la fois quantitatives et qualitatives, on pourra avoir recours à l'*analyse factorielle avec données mixtes*.

Bien que ces techniques soient disponibles dans les extensions standards de **R**, il est souvent préférable d'avoir recours à deux autres packages plus complets, `{ade4}` et `{FactoMineR}`, chacun ayant ses avantages et des possibilités différentes¹. Voici les fonctions à retenir :

Analyse	Variables	Fonction standard	Fonction <code>{ade4}</code>	Fonctions <code>{FactoMineR}</code>
ACP	plusieurs variables quantitatives	<code>stats::princomp()</code>	<code>ade4::dudi.pca()</code>	<code>FactoMineR::PCA()</code>
AFC	deux variables qualitatives	<code>MASS::corresp()</code>	<code>ade4::dudi.cora()</code>	<code>FactoMineR::CA()</code>
ACM	plusieurs variables qualitatives	<code>MASS::mca()</code>	<code>ade4::dudi.acm()</code>	<code>FactoMineR::MCA()</code>
Analyse mixte	plusieurs variables quantitatives et/ou qualitatives	—	<code>ade4::dudi.mfcca()</code>	<code>FactoMineR::FAMD()</code>

Notons que l'extension `{GDATools}` propose une variation de l'ACM permettant de neutraliser tout en conservant les valeurs manquantes : on parle alors d'*Analyse des correspondances multiples spécifique* qui se calcule avec la fonction `GDATools::speMCA()`.

Le package `{FactoMineR}` propose également deux variantes avancées : l'*analyse factorielle multiple* permettant d'organiser les variables en groupes (fonction `FactoMineR::MFA()`) et l'*analyse factorielle multiple hiérarchique* (fonction `FactoMineR::MFA()`).

¹Ces deux packages sont très complets et il est difficile de privilégier l'un par rapport à l'autre. Tous deux sont activement maintenus et bénéficient d'une documentation complète. À titre personnel, j'aurai peut être une petite préférence pour `{ade4}`. En effet, `{FactoMineR}` a une tendance à réaliser toutes les étapes de l'analyse en un seul appel de fonction, là où `{ade4}` nécessitera quelques étapes supplémentaires. Or, ce qui pourrait être vu comme un inconvénient permet à l'inverse d'avoir une meilleure conscience de ce que l'on fait. À l'inverse, les options graphiques offertes par `{factoextra}` sont plus nombreuses quand l'analyse factorielle a été réalisée avec `{FactoMineR}`. Enfin, nous le verrons plus loin, la gestion de variables additionnelles est bien plus facile avec `{FactoMineR}`.

Deux autres packages nous seront particulièrement utiles dans ce chapitre : `{explor}` pour une exploration visuelle interactive des résultats et `{factoextra}` pour diverses représentations graphiques.

39.1 Principe général

L'analyse des correspondances multiples est une technique descriptive visant à résumer l'information contenu dans un grand nombre de variables afin de faciliter l'interprétation des corrélations existantes entre ces différentes variables. On cherche à savoir quelles sont les modalités corrélées entre elles.

L'idée générale est la suivante. L'ensemble des individus peut être représenté dans un espace à plusieurs dimensions où chaque axe représente les différentes variables utilisées pour décrire chaque individu. Plus précisément, pour chaque variable qualitative, il y a autant d'axes que de modalités moins un. Ainsi il faut trois axes pour décrire une variable à quatre modalités. Un tel nuage de points est aussi difficile à interpréter que de lire directement le fichier de données. On ne voit pas les corrélations qu'il peut y avoir entre modalités, par exemple qu'aller au cinéma est plus fréquent chez les personnes habitant en milieu urbain. Afin de mieux représenter ce nuage de points, on va procéder à un changement de systèmes de coordonnées. Les individus seront dès lors projetés et représentés sur un nouveau système d'axe. Ce nouveau système d'axes est choisis de telle manière que la majorité des variations soit concentrées sur les premiers axes. Les deux-trois premiers axes permettront d'expliquer la majorité des différences observées dans l'échantillon, les autres axes n'apportant qu'une faible part additionnelle d'information. Dès lors, l'analyse pourra se concentrer sur ses premiers axes qui constitueront un bon résumé des variations observables dans l'échantillon.

! Important

Avant toute analyse factorielle, il est indispensable de réaliser une analyse préliminaire de chaque variable, afin de voir si toutes les classes sont aussi bien représentées ou s'il existe un déséquilibre. L'analyse factorielle est sensible aux petits effectifs. Aussi il peut être préférable de regrouper les classes peu représentées le cas échéant.

De même, il peut être tentant de mettre toutes les variables disponibles dans son jeu de données directement dans une analyse factorielle pour voir ce que ça donne. Il est préférable de réfléchir en amont aux questions que l'on veut poser et de choisir ensuite un jeu de variables en fonction.

39.2 Première illustration : ACM sur les loisirs

Pour ce premier exemple, nous allons considérer le jeu de données `hdv2003` fourni dans le package `{questionr}` et correspondant à un extrait de l'enquête *Histoire de Vie* réalisée par l'Insee en 2003.

Nous allons considérer 7 variables binaires (oui/non) portant sur la pratique de différents loisirs (écouter du hard rock, lire des bandes dessinées, pratiquer la pêche ou la chasse, cuisiner, bricoler ou pratiquer un sport). Pour le moment, nous n'allons pas intégrer à l'analyse de variable socio-démographique, car nous souhaitons explorer comment ces activités se corrèlent entre elles, indépendamment de toute autre considération.

Notons, avec `questionr::freq.na()` ou avec `labelled::look_for()`, qu'il n'y a pas de valeurs manquantes dans nos données.

```
library(tidyverse)
data("hdv2003", package = "questionr")
d <- hdv2003 |>
  select(hard.rock:sport)
d |> questionr::freq.na()
```

	missing %
hard.rock	0 0
lecture.bd	0 0
peche.chasse	0 0
cuisine	0 0
bricol	0 0
cinema	0 0
sport	0 0

```
d |> labelled::look_for()
```

pos	variable	label	col_type	missing	values
1	hard.rock	-	fct	0	Non Oui
2	lecture.bd	-	fct	0	Non Oui
3	peche.chasse	-	fct	0	Non Oui
4	cuisine	-	fct	0	Non Oui
5	bricol	-	fct	0	Non

				Oui
6	cinema	-	fct	0
				Non
				Oui
7	sport	-	fct	0
				Non
				Oui

Comme l'ensemble de nos variables sont catégorielles nous allons réaliser une analyse des correspondances multiples (ACM).

39.2.1 Calcul de l'ACM

Avec `{ade4}`, l'ACM s'obtient à l'aide la fonction `ade4::dudi.acm()`. Par défaut, si l'on exécute seulement `ade4::dudi.acm(d)`, la fonction va afficher un graphique indiquant la variance expliquée par chaque axe et une invite dans la console va demander le nombre d'axes à conserver pour l'analyse. Une invite de commande n'est pas vraiment adaptée dans le cadre d'un script que l'on souhaite pouvoir exécuter car cela implique une intervention manuelle. On pourra désactiver cette invitation avec `scannf = FALSE` et indiquer le nombre d'axes à conserver avec l'argument `nf` (`nf = Inf` permet de conserver l'ensemble des axes).

```
acm1_ad <- d |>
  ade4::dudi.acm(scannf = FALSE, nf = Inf)
```

Avec `{FactoMineR}`, l'ACM s'obtient avec `FactoMineR::MCA()`. Par défaut, seuls les 5 premiers axes sont conservés, ce qui est modifiable avec l'argument `nbp`. De plus, la fonction affiche par défaut un graphique des résultats avant de renvoyer les résultats de l'ACM. Ce graphique peut être désactivé avec `graph = FALSE`.

```
acm1_fm <- d |>
  FactoMineR::MCA(nbp = Inf, graph = FALSE)
```

Les deux ACM sont ici identiques. Par contre, les deux objets renvoyés ne sont pas structurés de la même manière.

39.2.2 Exploration graphique interactive

Le package `{explor}` permet d'explorer les résultats de manière interactive. Il fonctionne à la fois avec les analyses factorielles produites avec `{FactoMineR}` et celles réalisées avec `{ade4}`.

Pour lancer l'exploration interactive, il suffit de passer les résultats de l'ACM à la fonction `explor::explor()`.

```
acm1_ad |> explor::explor()
```

Les graphiques réalisés avec `explor::explor()` peuvent être exportés en fichier image SVG (via le bouton dédié en bas à gauche dans l'interface). De même, il est possible d'obtenir un code **R** que l'on pourra copier dans un script pour reproduire le graphique (**ATTENTION** : le graphique produit est interactif et donc utilisable uniquement dans document web).

```
res <- explor::prepare_results(acm1_ad)
p <- explor::MCA_var_plot(
  res,
  xax = 1,
  yax = 2,
  var_sup = FALSE,
  var_sup_choice = ,
  var_lab_min_contrib = 0,
  col_var = "Variable",
  symbol_var = "Variable",
  size_var = "Contrib",
  size_range = c(52.5, 700),
  labels_size = 12,
  point_size = 56,
  transitions = FALSE,
  labels_positions = "auto",
  labels_prepend_var = TRUE,
  xlim = c(-2.58, 1.69),
  ylim = c(-1.33, 2.94)
)
```

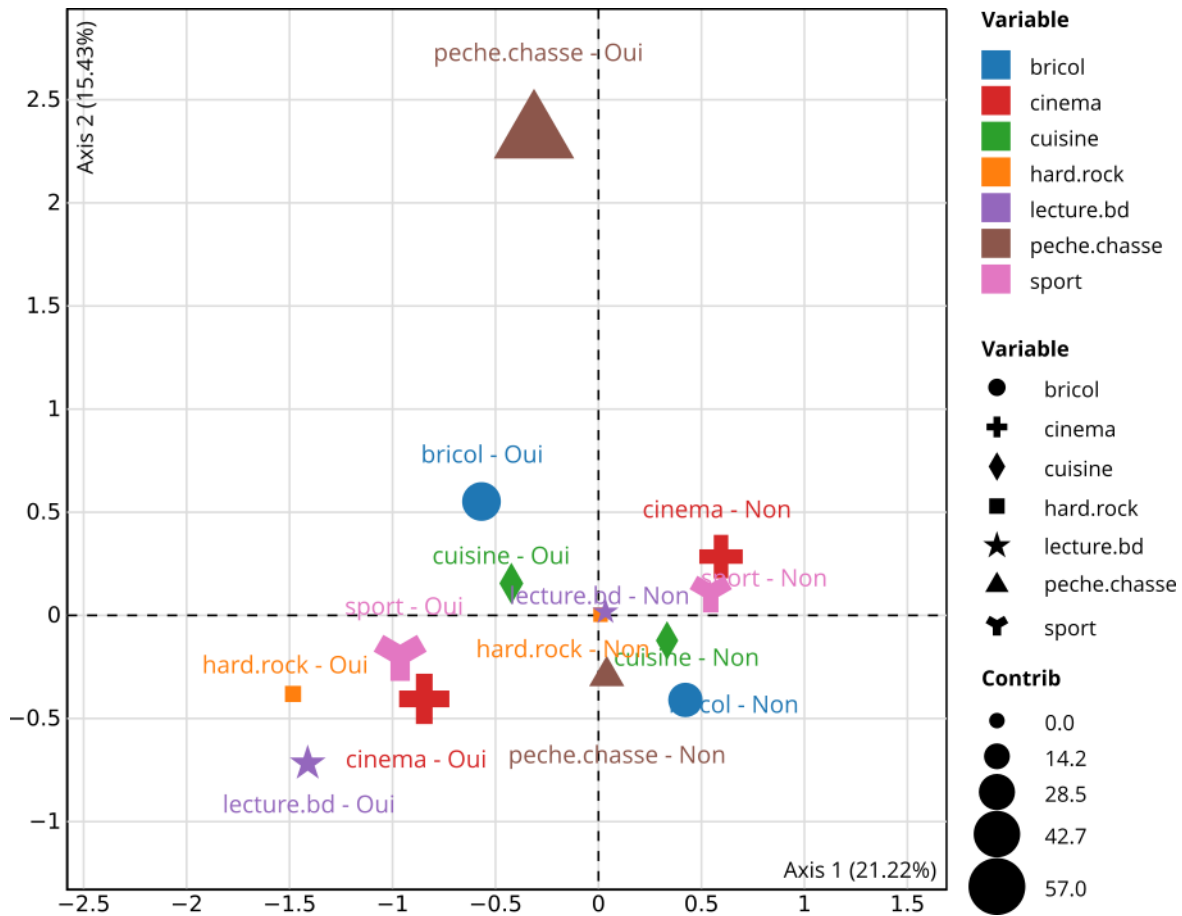


Figure 39.1: Exemple de figure exportée à partir de explor

39.2.3 Représentations graphiques

À la fois `{ade4}` et `{FactoMineR}` disposent de leurs propres fonctions graphiques dédiées. Elles sont cependant spécifiques à chaque package. Les fonctions graphiques de `{ade4}` ne peuvent pas être utilisées avec un objet `{FactoMineR}` et inversement.

Le package `{factoextra}` permet de palier à ce problème. Ces fonctions graphiques sont en effet compatibles avec les deux packages et reposent sur `{ggplot2}`, ce qui permet facilement de personnaliser les graphiques obtenus.

39.2.4 Variance expliquée et valeurs propres

Les valeurs propres (*eigen values* en anglais) correspondent à la quantité de variance capturée par chaque axe (on parle également d'inertie). On peut les obtenir aisément avec la fonction

```
factoextra::get_eigenvalue().
```

```
acm1_ad |>
  factoextra::get_eigenvalue()
```

	eigenvalue	variance.percent	cumulative.variance.percent
Dim.1	0.2121854	21.21854	21.21854
Dim.2	0.1542803	15.42803	36.64657
Dim.3	0.1468192	14.68192	51.32849
Dim.4	0.1387719	13.87719	65.20568
Dim.5	0.1349671	13.49671	78.70239
Dim.6	0.1192865	11.92865	90.63104
Dim.7	0.0936896	9.36896	100.00000

On notera que les axes sont ordonnées en fonction de la quantité de variation qu'ils capturent. Le premier axe est toujours celui qui capture le plus de variance, suivi du deuxième, etc.

La somme totale des valeurs propres indique la variation totale des données. Souvent, les valeurs propres sont exprimées en pourcentage du total. Dans notre exemple, l'axe 1 capture 21,2 % de la variance et l'axe 2 en capture 15,4 %. Ainsi, le plan factoriel composé des deux premiers axes permet de capturer à lui seul plus du tiers (36,6 %) de la variance totale.

Une représentation graphique des valeurs propres s'obtient avec `factoextra::fviz_screplot()`.

```
acm1_ad |>
  factoextra::fviz_screplot()
```

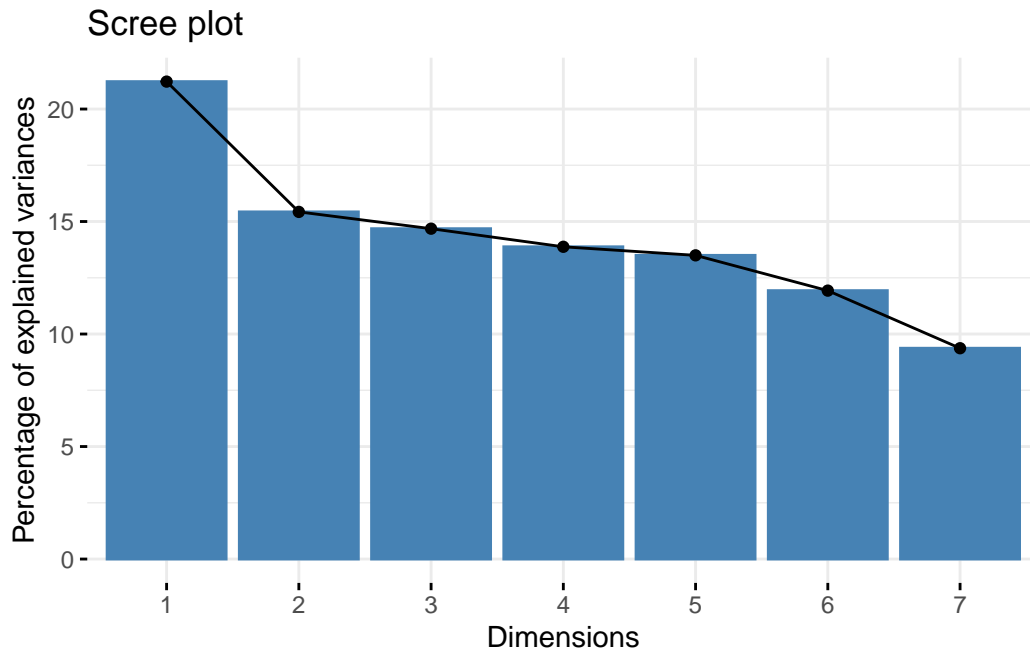



Figure 39.2: Représentation graphique de la variance expliquée par les différents axes de l'ACM

Il n'y a pas de règle absolue concernant le nombre d'axes à explorer pour l'analyse et l'interprétation des résultats. L'objectif d'une analyse factorielle étant justement de réduire le nombre de dimension considérée pour ce concentrer sur les principales associations entre modalités, il est fréquent de se limiter aux deux premiers ou aux trois premiers axes.

Une approche fréquente consiste à regarder s'il y a un coude, un saut plus marqué qu'un autre dans le graphique des valeurs propres. Dans notre exemple, qui ne comporte qu'un petit nombre de variable, on voit un saut marqué entre le premier axe et les autres, suggérant de se focaliser en particulier sur ce premier axe.

39.2.5 Contribution aux axes

La fonction `factoextra::fviz_contrib()` permet de visualiser la contribution des différentes modalités à un axe donnée. Regardons le premier axe.

```
acm1_ad |>
  factoextra::fviz_contrib(choice = "var", axes = 1)
```

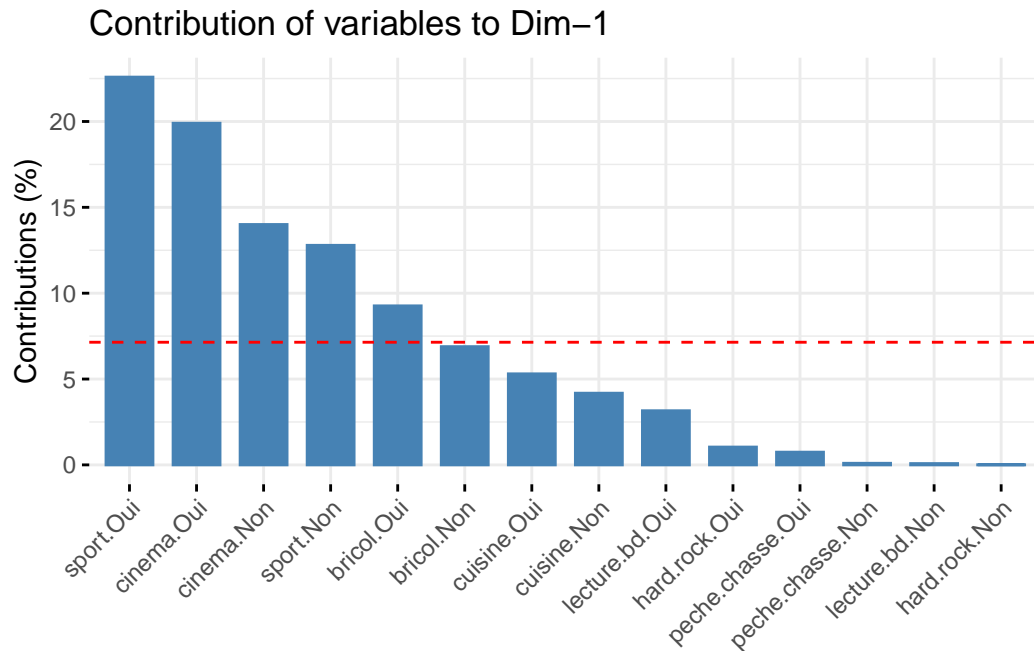


Figure 39.3: Contribution des modalités au premier axe

La ligne en pointillés rouges indique la contribution attendue de chaque modalité si la répartition était uniforme. Nous constatons ici que le premier axe est surtout déterminé par la pratique d'une activité sportive et le fait d'aller au cinéma.

```
acm1_ad |>
  factoextra::fviz_contrib(choice = "var", axes = 2)
```

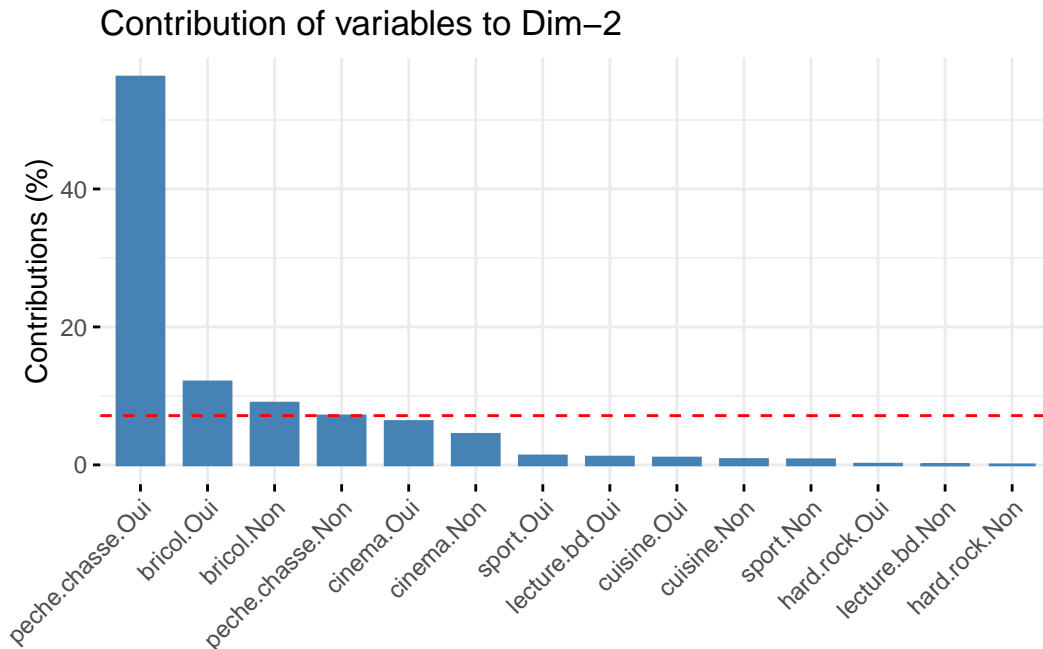


Figure 39.4: Contribution des modalités au deuxième axe

Le deuxième axe, quant à lui, est surtout marqué par la pratique de la pêche ou de la chasse et, dans une moindre mesure, par le fait de bricoler.

39.2.6 Représentation des modalités dans le plan factoriel

Pour représenter les modalités dans le plan factoriel, on aura recours à `factoextra::fviz_mca_var()`. En termes de visualisation, c'est moins ergonomique que ce que propose `{explor}`. On aura donc tout intérêt à profiter de ce dernier. Si l'on a réalisé un autre type d'analyse factorielle, il faudra choisir la fonction correspondante, par exemple `factoextra::fviz_famd_var()` pour une analyse sur données mixtes. La liste complète est visible sur le [site de documentation du package](#).

```
acm1_ad |>
  factoextra::fviz_mca_var()
```

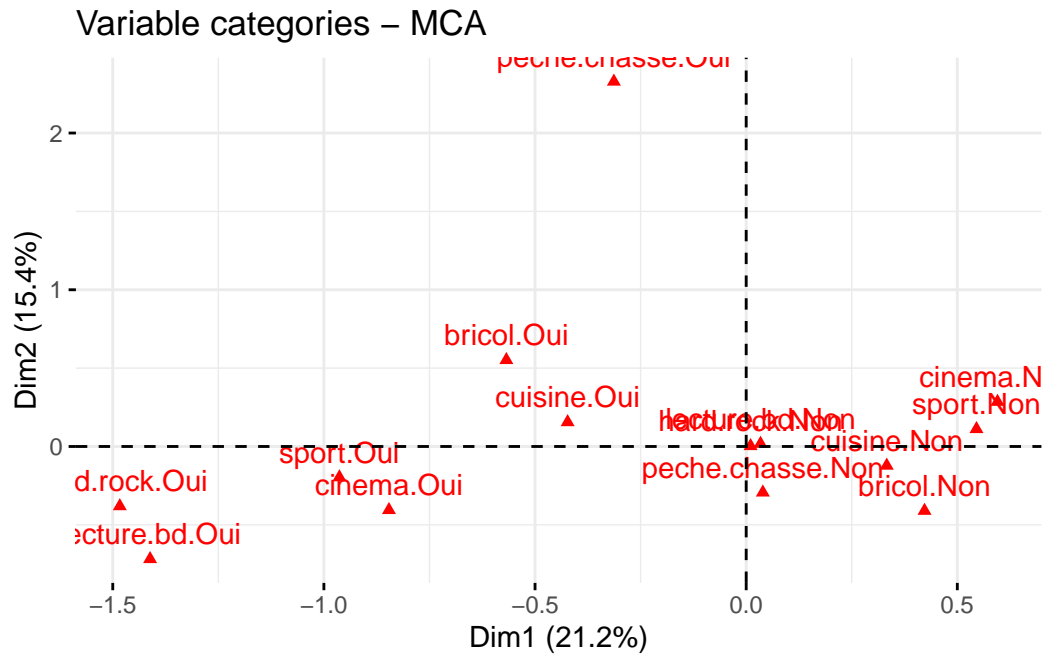
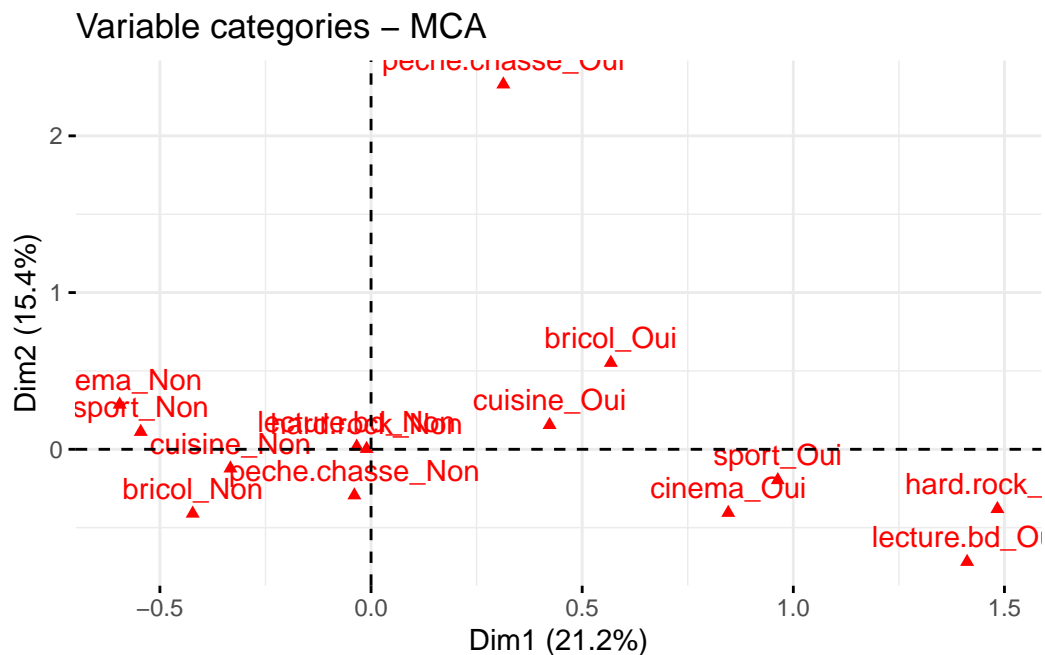


Figure 39.5: Projection des modalités dans le plan factoriel

i Note

Si l'on réalise le même graphique à partir de l'ACM réalisée avec `{FactoMineR}`, nous aurons à première vue un résultat différent.

```
acm1_fm |>
  factoextra::fviz_mca_var()
```



Si l'on regarde plus attentivement, les valeurs sont inversées sur l'axe 1 : les valeurs positives sont devenues négatives et inversement. Ceci est dû à de légères différences dans les algorithmes de calcul des deux packages. Pour autant, les résultats sont bien strictement équivalents, le sens des axes n'ayant pas de signification en soi.

39.2.7 Représentation des individus dans le plan factoriel

Pour projeter les individus (i.e. les observations) dans le plan factoriel, nous aurons recours à `factoextra::fviz_mca_ind()`.

```
acm1_ad |>
  factoextra::fviz_mca_ind()
```

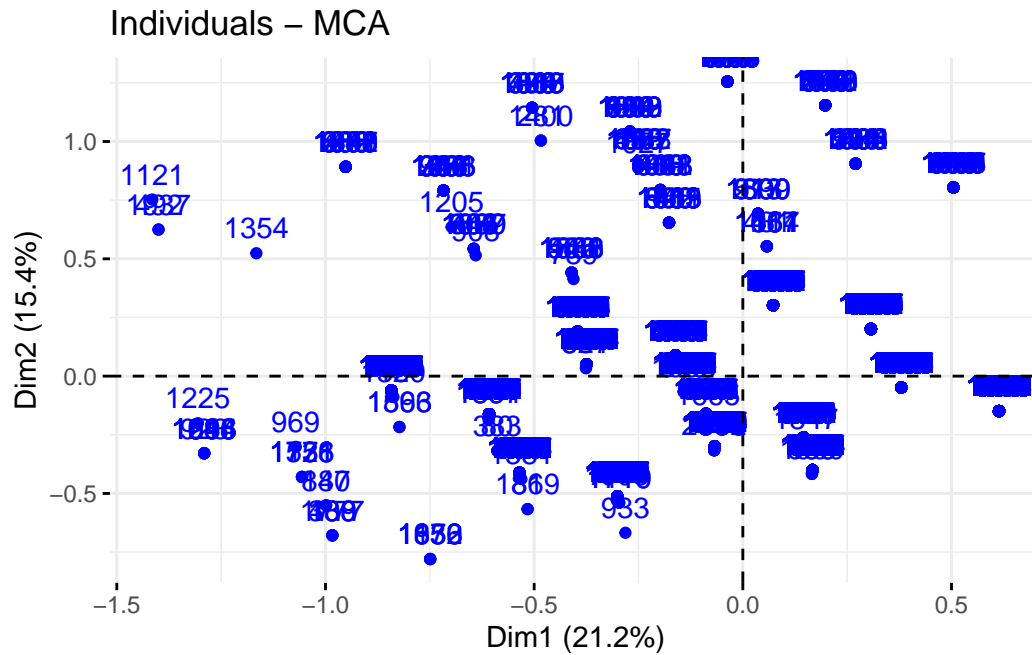


Figure 39.6: Projection des individus dans le plan factoriel

Le graphique sera un peu plus lisible en n'affichant que les points avec un effet de transparence (pour les points superposés).

```
acm1_ad |>
  factoextra::fviz_mca_ind(
    geom.ind = "point",
    alpha.ind = 0.1
  )
```

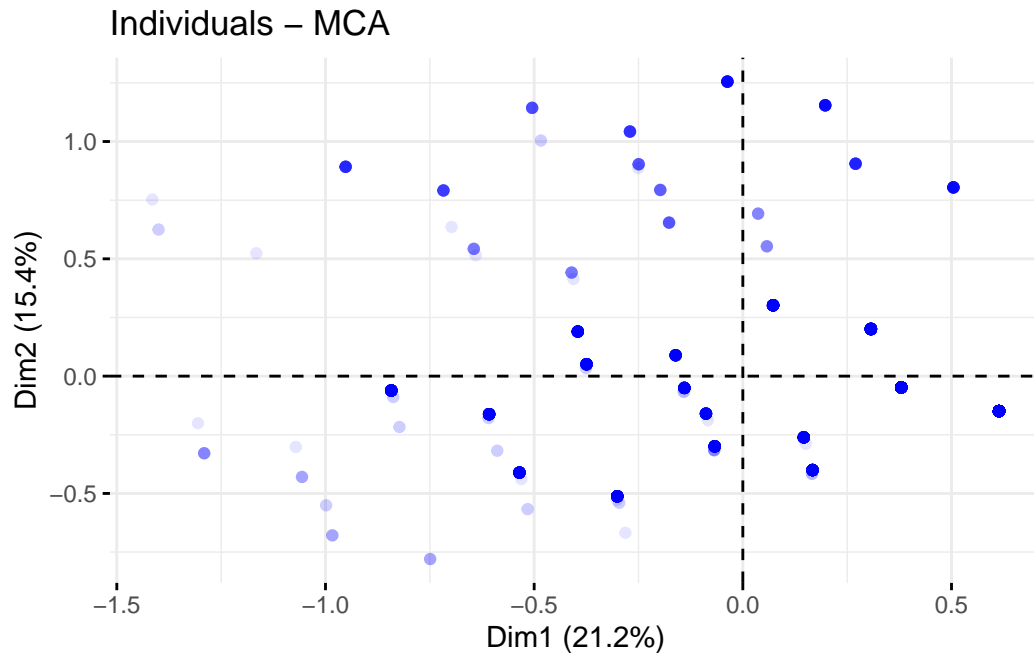


Figure 39.7: Projection des individus dans le plan factoriel

Il est souvent intéressant de colorier les individus selon une variable catégorielle tierce, que ce soit une des variables ayant contribué ou non à l'ACM. Par exemple, nous allons regarder la répartition des individus dans le plan factoriel selon leur pratique d'un sport, puisque cette variable contribuait fortement au premier axe.

Nous indiquerons la variable considérée à `factoextra::fviz_mca_ind()` via l'argument `habillage`. L'option `addEllipses = TRUE` permet de dessiner des ellipses autour des observations.

```
acm1_ad |>
  factoextra::fviz_mca_ind(
    habillage = d$sport,
    addEllipses = TRUE,
    geom.ind = "point",
    alpha.ind = 0.1
  )
```

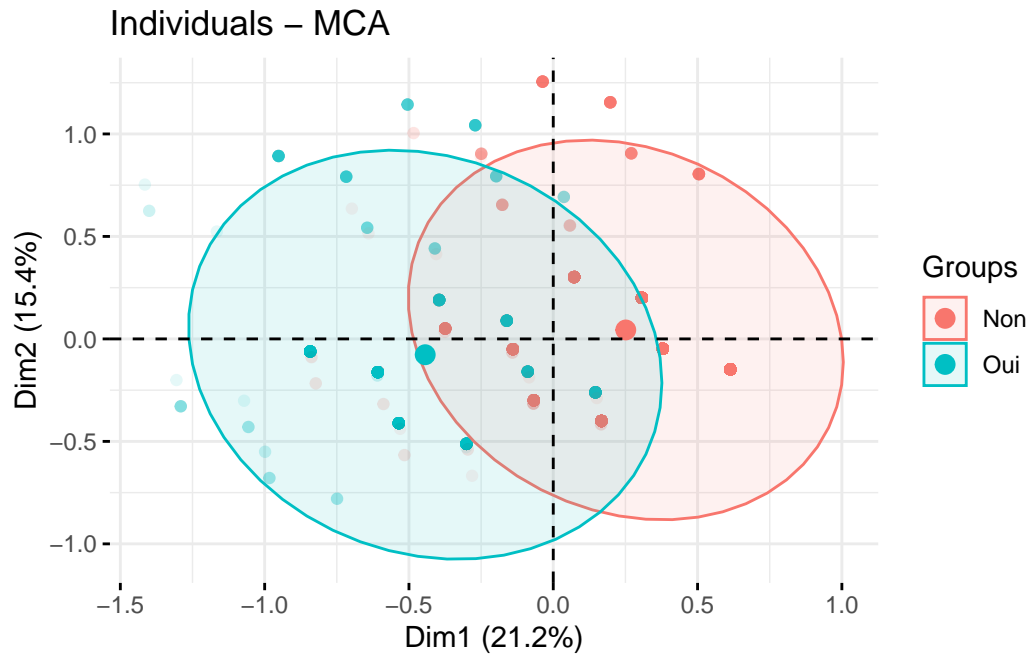
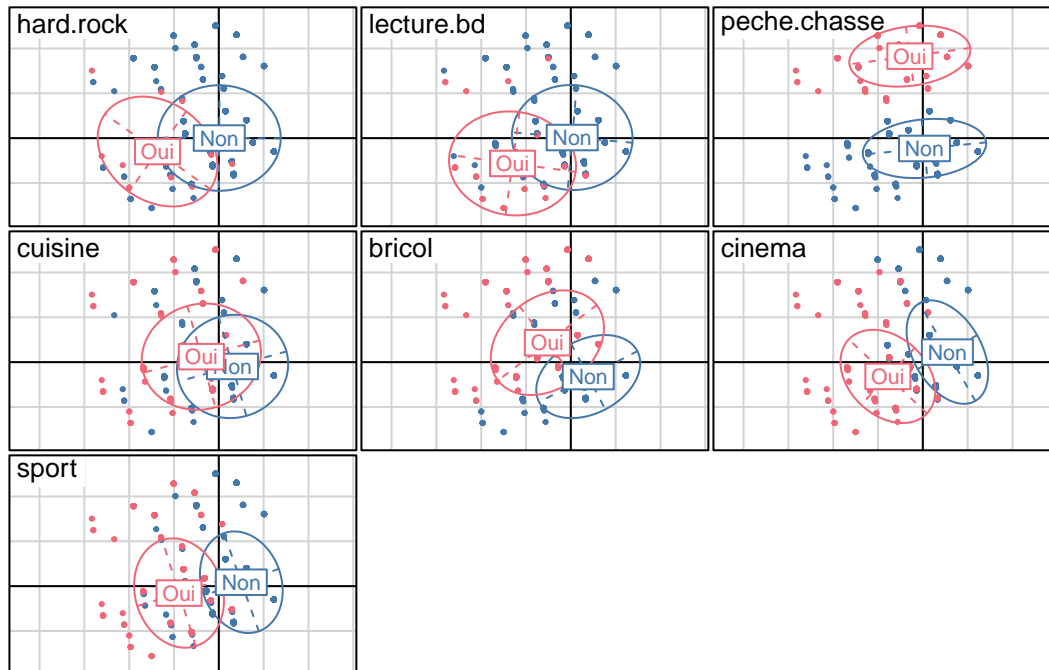


Figure 39.8: Projection des individus dans le plan factoriel selon la pratique d'un sport

💡 Astuce

Si l'on a réalisé l'ACM avec `{ade4}`, nous pouvons utiliser la fonction `ade4::scatter()` pour réaliser ce même type de graphique avec l'ensemble des variables incluses dans l'ACM. Afin de rendre le graphique plus lisible, nous passons à la fonction une palette de couleur obtenue avec `khroma::colour()` (cf. Section 16.3.2).

```
acm1_ad |>
  ade4::scatter(col = khroma::colour("bright")(2))
```

39.2.8 Récupérer les coordonnées des individus / des variables

Dans certaines situations (par exemple pour créer un score à partir de la position sur le premier axe), on peut avoir besoin de récupérer les données brutes de l'analyse factorielle.

On pourra utiliser les fonctions `get_*`() de `{factoextra}`. Par exemple, pour les individus dans le cadre d'une ACM, on utilisera `factoextra::get_mca_ind()`.

```
res <- acm1_ad |>
  factoextra::get_mca_ind()
print(res)
```

Multiple Correspondence Analysis Results for individuals

```
=====
Name      Description
1 "$coord" "Coordinates for the individuals"
2 "$cos2"  "Cos2 for the individuals"
3 "$contrib" "contributions of the individuals"
```

Le résultat obtenu est une liste avec trois tableaux de données. Pour accéder aux coordonnées des individus, il suffit donc d'exécuter la commande ci-dessous.

```
as_tibble(res$coord)
```

```
# A tibble: 2,000 x 7
  Dim.1 Dim.2 Dim.3 Dim.4 Dim.5 Dim.6 Dim.7
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  0.380 -0.0481 -0.130 -0.0757 -0.554 -0.118 -0.0382
2 -0.301 -0.512  0.0125  0.221  0.421 -0.340 -0.0402
3  0.146 -0.261  0.0784  0.152  0.279 -0.252 -0.578
4 -0.842 -0.0612  0.00886  0.247 -0.232  0.149 -0.0297
5  0.614 -0.149  0.0190 -0.0259  0.139 -0.0253 -0.0474
6 -0.301 -0.512  0.0125  0.221  0.421 -0.340 -0.0402
7  0.270  0.905 -0.214 -0.198 -0.267 -0.732  0.144
8 -0.0369 1.26 -0.0688 -0.122 -0.227 -0.150  0.146
9  0.167 -0.400 -0.0469  0.0425  0.282 -0.114  0.490
10 -0.301 -0.512  0.0125  0.221  0.421 -0.340 -0.0402
# i 1,990 more rows
```

39.3 Ajout de variables / d'observations additionnelles

Dans le cadre d'une analyse factorielle, on peut souhaiter ajouter des variables ou des observations additionnelles, qui ne participeront donc pas au calcul de l'analyse factorielle (et seront donc sans effet sur les axes de l'analyse). Ces variables / observations additionnelles seront simplement projetées dans le nouvel espace factoriel.

Reprenons notre exemple et calculons des groupes d'âges.

```
hdv2003 <- hdv2003 |>
  mutate(
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      )
  )
```

Ajoutons maintenant le sexe, le groupe d'âges et le niveau d'étude comme variables additionnelles.

Avec `{FactoMineR}`, cela se fait directement au moment du calcul de l'ACM, en indiquant l'index (ordre de la colonne dans le tableau) des variables supplémentaires à `quali.sup` pour les variables catégorielles et à `quant1.sup` pour les variables continues. De même, `ind.sup` peut-être utilisé pour indiquer les observations additionnelles.

```
d2 <- hdv2003 |>
  select(sexe, groupe_ages, nivetud, hard.rock:sport)
acm2_fm <- d2 |>
  FactoMineR::MCA(
    ncp = Inf,
    graph = FALSE,
    quali.sup = 1:3
  )
```

Avec `{ade4}`, la manipulation est légèrement différente. Le calcul de l'ACM se fait comme précédemment, uniquement avec les variables et les observations incluses dans l'analyse, puis on pourra projeter dans l'espace factoriel les variables / observations additionnelles à l'aide de `ade4::supcol()` et `ade4::suprow()`. Si pour ajouter des observations additionnelles il suffit de passer l'ACM de base et un tableau de données des observations additionnelles à `ade4::suprow()`, c'est un peu plus compliqué pour des variables additionnelles. Il faut déjà réaliser une ACM sur ces variables additionnelles, en extraire le sous objet `$tab` et passer ce dernier à `ade4::supcol()`.

```
acm_add <- hdv2003 |>
  select(sexe, groupe_ages, nivetud) |>
  ade4::dudi.acm(scannf = FALSE, nf = Inf)

acm_supp <- ade4::supcol(
  acm1_ad,
  acm_add$tab
)
```

Si l'on veut pouvoir utiliser `explor::explor()` avec ces variables additionnelles, il faudra enregistrer le résultat de `ade4::supcol()` dans un sous-objet `$supv` de l'ACM principale.

```
acm2_ad <- acm1_ad
acm2_ad$supv <- acm_supp
```

Pour des représentations graphiques avec `{factoextra}`, on privilégiera ici le calcul avec `{FactoMineR}` (les variables additionnelles calculées avec `{ade4}` n'étant pas gérées par `{factoextra}`). En effet, si l'ACM a été calculée avec `{FactoMineR}`, `factoextra::fviz_mca_var()` affiche par défaut les variables ad

```
acm2_fm |>
  factoextra::fviz_mca_var(repel = TRUE, labelsize = 2)
```

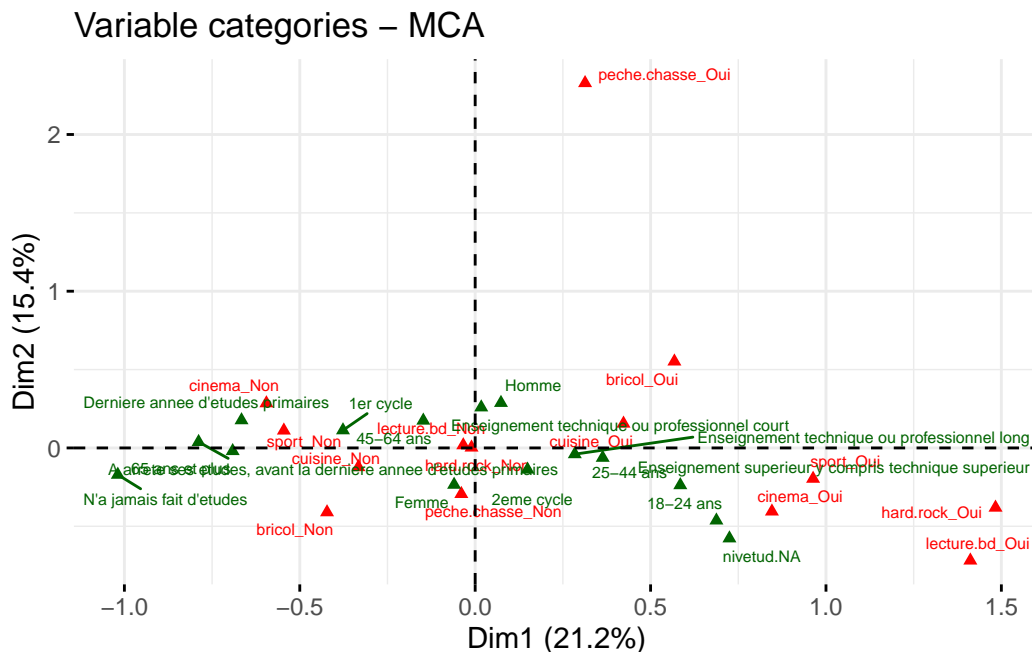


Figure 39.9: Projection des modalités dans le plan factoriel (incluant les variables additionnelles)

39.4 Gestion des valeurs manquantes

Pour ce troisième exemple, nous allons maintenant inclure les variables sexe, groupe d'âges et niveau d'étude dans l'ACM, non pas comme variables additionnelles mais comme variables de l'ACM (qui vont donc contribuer au calcul des axes).

```
d3 <- hdv2003 |>
  select(sexe, groupe_ages, nivetud, hard.rock:sport)
```

Calculons maintenant l'ACM avec {ade4} et {FactoMineR}.

```
acm3_ad <- d3 |>
  ade4::dudi.acm(scannf = FALSE, nf = Inf)
acm3_fm <- d3 |>
  FactoMineR::MCA(ncp = Inf, graph = FALSE)
```

Regardons les valeurs propres et l'inertie expliquée et positionnons les deux graphiques côte à côte (cf. Section 17.6 sur la combinaison de graphiques).

```
p_ad <- acm3_ad |>
  factoextra::fviz_screepplot(choice = "eigenvalue") +
  ggplot2::ggtitle("ACM3 avec ade4")
p_fm<- acm3_fm |>
  factoextra::fviz_screepplot(choice = "eigenvalue") +
  ggplot2::ggtitle("ACM3 avec FactoMineR")
patchwork::wrap_plots(p_ad, p_fm) &
  ggplot2::ylim(0, .3)
```

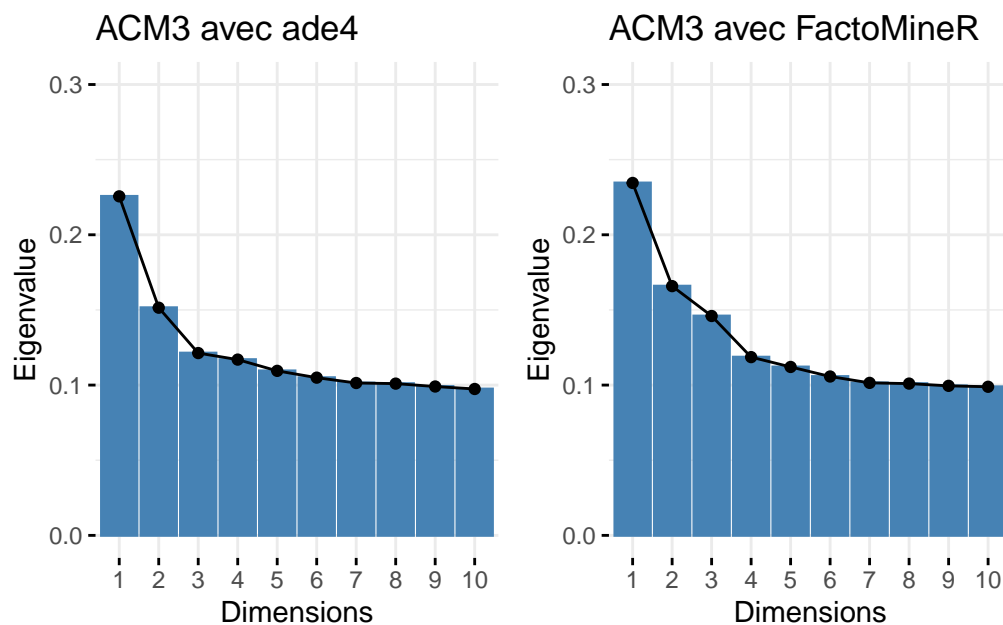


Figure 39.10: Inertie expliquée par axe (comparaison des ACM réalisées avec ade4 et FactoMineR)

Comme nous pouvons le voir, cette fois-ci, l'inertie expliquée par axe diffère entre les deux ACM. Cela est dû à la présence de valeurs manquantes pour la variable *nivetud*.

```
d3 |> questionr::freq.na()
```

missing %

nivetud	112 6
sexe	0 0
groupe_ages	0 0
hard.rock	0 0
lecture.bd	0 0
peche.chasse	0 0
cuisine	0 0
bricol	0 0
cinema	0 0
sport	0 0

Or, les deux packages ne traitent pas les valeurs manquantes de la même manière : `{ade4}` exclue les valeurs manquantes tandis que `{FactoMineR}` les considère comme une modalité additionnelle.

Pour éviter toute ambiguïté, il est préférable de traiter soi-même les valeurs manquantes (NA) en amont des deux fonctions.

Pour convertir les valeurs manquantes d'un facteur en une modalité en soi, on utilisera `forcats::fct_na_value_to_level()`. Il est possible d'appliquer cette fonction à tous les facteurs d'un tableau de données avec `dplyr::across()` (cf. Chapitre 38).

```
d3_modalite_manquant <- d3 |>
  mutate(
    across(
      where(is.factor),
      fct_na_value_to_level,
      level = "(manquant)"
    )
  )
```

Warning: There was 1 warning in `mutate()`.

i In argument: `across(where(is.factor), fct_na_value_to_level, level = "(manquant)")`.

Caused by warning:

! The `...` argument of `across()` is deprecated as of dplyr 1.1.0.

Supply arguments directly to `.fns` through an anonymous function instead.

Previously

```
across(a:b, mean, na.rm = TRUE)
```

Now

```
across(a:b, \(x) mean(x, na.rm = TRUE))
```

```
d3_modalite_manquant |> nrow()
```

```
[1] 2000
```

```
d3_modalite_manquant |> questionr::freq.na()
```

	missing %
sexe	0 0
groupe_ages	0 0
nivetud	0 0
hard.rock	0 0
lecture.bd	0 0
peche.chasse	0 0
cuisine	0 0
bricol	0 0
cinema	0 0
sport	0 0

Pour ne conserver que l'ensemble des observations complètes (i.e. sans valeur manquante), on pourra avoir recours `tidyr::drop_na()`.

```
d3_obs_completes <- d3 |>  
  tidyr::drop_na()  
d3_obs_completes |> nrow()
```

```
[1] 1888
```

```
d3_obs_completes |> questionr::freq.na()
```

	missing %
sexe	0 0
groupe_ages	0 0
nivetud	0 0
hard.rock	0 0
lecture.bd	0 0
peche.chasse	0 0
cuisine	0 0
bricol	0 0
cinema	0 0
sport	0 0

Une alternative est offerte par le package `{GDAtools}` qui implémente une *ACM spécifique* permettant de neutraliser certaines modalités dans la construction de l'espace factoriel, tout en conservant l'ensemble des individus. Les valeurs manquantes sont automatiquement considérées comme des modalités à ne pas tenir compte. Mais il est également possible d'indiquer d'autres modalités à ignorer (voir le [tutoriel du package](#)).

```
acm3_spe <- GDAtools::speMCA(d3)
```

Si les fonctions de `{factorextra}` ne sont pas compatibles avec `{GDAtools}`, on peut tout à fait utiliser `{explor}`. De plus, `{GDAtools}` fournit directement plusieurs outils de visualisation avancée et d'aide à l'interprétation des résultats.

39.5 webin-R

L'analyse factorielle est présentée sur YouTube dans le [webin-R #11](#) (*Analyse des Correspondances Multiples (ACM)*).

<https://youtu.be/fIZblrfenz0>

39.6 Lectures additionnelles

- [Visualiser une analyse géométrique des données avec ggplot2 \(R/RStudio\)](#) par Anton Perdoncin
- [Exploration interactive de résultats d'ACP/ACM avec explor](#) par Julien Barnier
- [Analyse des correspondances multiples \(ACM ou AFCM\) avec FactoMineR](#) (Vidéo YouTube) par François Husson (l'un des auteurs de `FactoMineR`)
- [L'analyse géométrique des données avec GDAtools](#) par Nicolas Robette
- [Tuto@Mate #32 : Les Analyses Factorielles Multiples \(AFM\)](#) par Maelle Amand

40 Classification ascendante hiérarchique

Il existe de nombreuses techniques statistiques visant à partitionner une population en différentes classes ou sous-groupes. La *classification ascendante hiérarchique* (CAH) est l'une d'entre elles. On cherche à ce que les individus regroupés au sein d'une même classe (homogénéité intra-classe) soient le plus semblables possibles tandis que les classes soient le plus dissemblables (hétérogénéité inter-classe).

Le principe de la CAH est de rassembler des individus selon un critère de ressemblance défini au préalable qui s'exprimera sous la forme d'une matrice de distances, exprimant la distance existant entre chaque individu pris deux à deux. Deux observations identiques auront une distance nulle. Plus les deux observations seront dissemblables, plus la distance sera importante. La CAH va ensuite rassembler les individus de manière itérative afin de produire un dendrogramme ou arbre de classification. La classification est *ascendante* car elle part des observations individuelles ; elle est *hiérarchique* car elle produit des classes ou groupes de plus en plus vastes, incluant des sous-groupes en leur sein. En découpant cet arbre à une certaine hauteur choisie, on produira la partition désirée.

40.1 Calculer une matrice des distances

La notion de ressemblance entre observations est évaluée par une distance entre individus. Plusieurs type de distances existent selon les données utilisées.

Il existe de nombreuses distances mathématiques pour les variables quantitatives (euclidiennes, Manhattan...) que nous n'aborderons pas ici¹. La plupart peuvent être calculées avec la fonction `stats::dist()`.

Usuellement, pour un ensemble de variables qualitatives, on aura recours à la distance du Φ^2 qui est celle utilisée pour l'analyse des correspondances multiples (cf. Chapitre 39). Avec l'extension `{ade4}`, la distance du Φ^2 s'obtient avec la fonction `ade4::dist.dudi()`². Le cas particulier de la CAH avec l'extension `{FactoMineR}` sera abordée un peu plus loin.

¹Pour une présentation des propriétés mathématiques des distances et des distances les plus courantes, on pourra se référer à la [page Wikipedia](#) correspondance.

²Cette même fonction peut aussi être utilisée pour calculer une distance après une analyse en composantes principales ou une analyse mixte de Hill et Smith.

Nous évoquerons également la distance de Gower qui peut s'appliquer à un ensemble de variables à la fois qualitatives et quantitatives et qui se calcule avec la fonction `cluster::daisy()` de l'extension `{cluster}`.

Il existe bien entendu d'autres types de distance. Par exemple, dans le chapitre sur l'analyse de séquences, nous verrons comment calculer une distance entre séquences, permettant ainsi de réaliser une classification ascendante hiérarchique.

40.1.1 Distance de Gower

En 1971, Gower a proposé un indice de similarité qui porte son nom³. L'objectif de cet indice consiste à mesurer dans quelle mesure deux individus sont semblables. L'indice de Gower varie entre 0 et 1. Si l'indice vaut 1, les deux individus sont identiques. À l'opposé, s'il vaut 0, les deux individus considérés n'ont pas de point commun. Si l'on note S_g l'indice de similarité de Gower, la distance de Gower D_g s'obtient simplement de la manière suivante : $D_g = 1 - S_g$. Ainsi, la distance sera nulle entre deux individus identiques et elle sera égale à 1 entre deux individus totalement différents. Cette distance s'obtient sous **R** avec la fonction `cluster::daisy()` du package `{cluster}`.

L'indice de similarité de Gower entre deux individus x_1 et x_2 se calcule de la manière suivante :

$$S_g(x_1, x_2) = \frac{1}{p} \sum_{j=1}^p s_{12j}$$

p représente le nombre total de caractères (ou de variables) descriptifs utilisés pour comparer les deux individus⁴. s_{12j} représente la similarité partielle entre les individus 1 et 2 concernant le descripteur j . Cette similarité partielle se calcule différemment s'il s'agit d'une variable qualitative ou quantitative :

- **variable qualitative** : s_{12j} vaut 1 si la variable j prend la même valeur pour les individus 1 et 2, et vaut 0 sinon. Par exemple, si 1 et 2 sont tous les deux « grand », alors s_{12j} vaudra 1. Si 1 est « grand » et 2 « petit », s_{12j} vaudra 0.
- **variable quantitative** : la différence absolue entre les valeurs des deux variables est tout d'abord calculée, soit $|y_{1j} - y_{2j}|$. Puis l'écart maximum observé sur l'ensemble du fichier est déterminé et noté R_j . Dès lors, la similarité partielle vaut $s_{12j} = 1 - |y_{1j} - y_{2j}|/R_j$.

Dans le cas où l'on n'a que des variables qualitatives, la valeur de l'indice de Gower correspond à la proportion de caractères en commun. Supposons des individus 1 et 2 décrits ainsi :

³Voir Gower, J. (1971). A General Coefficient of Similarity and Some of Its Properties. *Biometrics*, 27(4), 857-871. doi:10.2307/2528823 (<http://www.jstor.org/stable/2528823>).

⁴Pour une description mathématique plus détaillée de cette fonction, notamment en cas de valeur manquante, se référer à l'article original de Gower précédemment cité.

1. homme / grand / blond / étudiant / urbain
2. femme / grande / brune / étudiante / rurale

Sur les 5 variables utilisées pour les décrire, 1 et 2 ont deux caractéristiques communes : ils sont grand(e)s et étudiant(e)s. Dès lors, l'indice de similarité de Gower entre 1 et 2 vaut $2/5 = 0,4$ (soit une distance de $1 - 0,4 = 0,6$).

Plusieurs approches peuvent être retenues pour traiter les valeurs manquantes :

- supprimer tout individu n'étant pas renseigné pour toutes les variables de l'analyse ;
- considérer les valeurs manquantes comme une modalité en tant que telle ;
- garder les valeurs manquantes en tant que valeurs manquantes.

Le choix retenu modifiera les distances de Gower calculées. Supposons que l'on ait :

1. homme / grand / blond / étudiant / urbain
2. femme / grande / brune / étudiante / manquant

Si l'on supprime les individus ayant des valeurs manquantes, 2 est retirée du fichier d'observations et aucune distance n'est calculée.

Si l'on traite les valeurs manquantes comme une modalité particulière, 1 et 2 partagent alors 2 caractères sur les 5 analysés, la distance de Gower entre eux est alors de $1 - 2/5 = 1 - 0,4 = 0,6$.

Si on garde les valeurs manquantes, l'indice de Gower est dès lors calculé sur les seuls descripteurs renseignés à la fois pour 1 et 2. La distance de Gower sera calculée dans le cas présent uniquement sur les 4 caractères renseignés et vaudra $1 - 2/4 = 0,5$.

40.1.2 Distance du Φ^2

Il s'agit de la distance utilisée dans les analyses de correspondance multiples (ACM). C'est une variante de la distance du χ^2 . Nous considérons ici que nous avons Q questions (soit Q variables initiales de type facteur). À chaque individu est associé un patron c'est-à-dire une certaine combinaison de réponses aux Q questions. La distance entre deux individus correspond à la distance entre leurs deux patrons. Si les deux individus présentent le même patron, leur distance sera nulle. La distance du Φ^2 peut s'exprimer ainsi :

$$d_{\Phi^2}^2(L_i, L_j) = \frac{1}{Q} \sum_k \frac{(\delta_{ik} - \delta_{jk})^2}{f_k}$$

où L_i et L_j sont deux patrons, Q le nombre total de questions. δ_{ik} vaut 1 si la modalité k est présente dans le patron L_i , 0 sinon. f_k est la fréquence de la modalité k dans l'ensemble de la population.

Exprimé plus simplement, on fait la somme de l'inverse des fréquences des modalités non communes aux deux patrons, puis on divise par le nombre total de question. Si nous reprenons notre exemple précédent :

1. homme / grand / blond / étudiant / urbain
2. femme / grande / brune / étudiante / rurale

Pour calculer la distance entre 1 et 2, il nous faut connaître la proportion des différentes modalités dans l'ensemble de la population étudiée. En l'occurrence :

- hommes : 52 % / femmes : 48 %
- grand : 30 % / moyen : 45 % / petit : 25 %
- blond : 15 % / châtain : 45 % / brun : 30 % / blanc : 10 %
- étudiant : 20 % / salariés : 65 % / retraités : 15 %
- urbain : 80 % / rural : 20 %

Les modalités non communes entre les profils de 1 et 2 sont : homme, femme, blond, brun, urbain et rural. La distance du Φ^2 entre 1 et 2 est donc la suivante :

$$d_{\Phi^2}^2(L_1, L_2) = \frac{1}{5} \left(\frac{1}{0,52} + \frac{1}{0,48} + \frac{1}{0,15} + \frac{1}{0,30} + \frac{1}{0,80} + \frac{1}{0,20} \right) = 4,05$$

Cette distance, bien que moins intuitive que la distance de Gower évoquée précédemment, est la plus employée pour l'analyse d'enquêtes en sciences sociales. Il faut retenir que la distance entre deux profils est dépendante de la distribution globale de chaque modalité dans la population étudiée. Ainsi, si l'on recalcule les distances entre individus à partir d'un sous-échantillon, le résultat obtenu sera différent. De manière générale, les individus présentant des caractéristiques rares dans la population vont se retrouver éloignés des individus présentant des caractéristiques fortement représentées.

40.1.3 Illustration

Nous allons reprendre l'exemple utilisé au chapitre précédent sur l'analyse factorielle (cf. Chapitre 39) et portant sur les loisirs pratiqués par les répondants à l'enquête *histoire de vie* de 2003.

```
library(tidyverse)
data("hdv2003", package = "questionr")
d <- hdv2003 |>
  select(hard.rock:sport)
```

Calculons maintenant une matrice de distances. Il s'agit d'une grande matrice carrée, avec autant de lignes et de colonnes que d'observations et indiquant la distance entre chaque individus pris deux à deux.

La distance de Gower se calcule avec `cluster::daisy()`.

```
md_gower <- d |>
  cluster::daisy(metric = "gower")
```

Pour la distance du Φ^2 , nous allons d'abord réaliser une ACM avec `ade4::dudi.acm()` puis appeler `ade4::dist.dudi()`.

```
acm_ad <- d |>
  ade4::dudi.acm(scannf = FALSE)
md_phi2 <- acm_ad |>
  ade4::dist.dudi()
```

Astuce

La distance du Φ^2 peut être calculée entre les observations (ce que nous venons de faire) afin de créer ensuite une typologie d'individus, mais il est également possible de calculer une distance du Φ^2 entre les modalités des variables afin de créer une typologie de variables. Dans ce cas-là, on appellera `ade4::dist.dudi()` avec l'option `amongrow = FALSE`.

40.2 Calcul du dendrogramme

Il faut ensuite choisir une méthode d'agrégation pour construire le dendrogramme. De nombreuses solutions existent (saut minimum, distance maximum, moyenne, méthode de Ward...). Chacune d'elle produira un dendrogramme différent. Nous ne détaillerons pas ici ces différentes techniques⁵.

Cependant, à l'usage, on privilégiera le plus souvent la méthode de Ward⁶. De manière simplifiée, cette méthode cherche à minimiser l'inertie intra-classe et à maximiser l'inertie inter-classe afin d'obtenir des classes les plus homogènes possibles. Cette méthode est souvent incorrectement présentée comme une méthode de minimisation de la variance alors qu'au sens strict

⁵Les méthodes *single*, *complete*, *centroid*, *average* et *Ward* sont présentées succinctement dans le document *Hierarchical Clustering* par Fatih Karabiber.

⁶Ward, J. (1963). Hierarchical Grouping to Optimize an Objective Function. *Journal of the American Statistical Association*, 58(301), 236-244. doi:10.2307/2282967. (<http://www.jstor.org/stable/2282967>)

Ward vise l'augmentation minimum de la somme des carrés (*"minimum increase of sum-of-squares (of errors)"*)⁷.

En raison de la variété des distances possibles et de la variété des techniques d'agrégation, on pourra être amené à réaliser plusieurs dendrogrammes différents sur un même jeu de données jusqu'à obtenir une classification qui fait « sens ».

La fonction de base pour le calcul d'un dendrogramme est `stats::hclust()` en précisant le critère d'agrégation avec `method`. Dans notre cas, nous allons opter pour la méthode de Ward appliquée au carré des distances (ce qu'on indique avec `method = "ward.D2"`)⁸ :

```
arbre_phi2 <- md_phi2 |>
  hclust(method = "ward.D2")
```

Astuce

Le temps de calcul d'un dendrogramme peut être particulièrement important sur un gros fichier de données. Le package `{fastcluster}` permet de réduire significativement ce temps de calcul. Elle propose une version optimisée de `hclust()` (les arguments sont identiques).

Il suffira donc de charger `{fastcluster}` pour surcharger la fonction `hclust()`, ou bien d'appeler explicitement `fastcluster::hclust()`.

```
arbre_gower <- md_gower |>
  fastcluster::hclust(method = "ward.D2")
```

Note

Le dendrogramme peut également être calculé avec la fonction `cluster::agnes()`. Cependant, à l'usage, le temps de calcul peut être plus long qu'avec `hclust()`.

Les noms des arguments sont légèrement différents. Pour la méthode de Ward appliquée au carré de la matrice des distance, on précisera à `cluster::agnes()` l'option `method = "ward"`.

Le résultat obtenu n'est pas au même format que celui de `stats::hclust()`. Il est possible de transformer un objet `cluster::agnes()` au format `stats::hclust()` avec `cluster::as.hclust()`.

⁷Voir par exemple la discussion, en anglais, sur Wikipedia concernant la page présentant la méthode Ward : https://en.wikipedia.org/wiki/Talk:Ward%27s_method

⁸L'option `method = "ward.D"` correspondant à la méthode de Ward sur la matrice des distances simples (i.e. sans la passer au carré). Mais il est à noter que la méthode décrite par Ward dans son article de 1963 correspond bien à `method = "ward.D2"`.

40.2.1 Représentation graphique du dendrogramme

Pour une représentation graphique rapide du dendrogramme, on peut directement `plot()`. Lorsque le nombre d'individus est important, il peut être utile de ne pas afficher les étiquettes des individus avec `labels = FALSE`.

```
arbre_gower |>
  plot(labels = FALSE, main = "Dendrogramme (distance de Gower)")
```

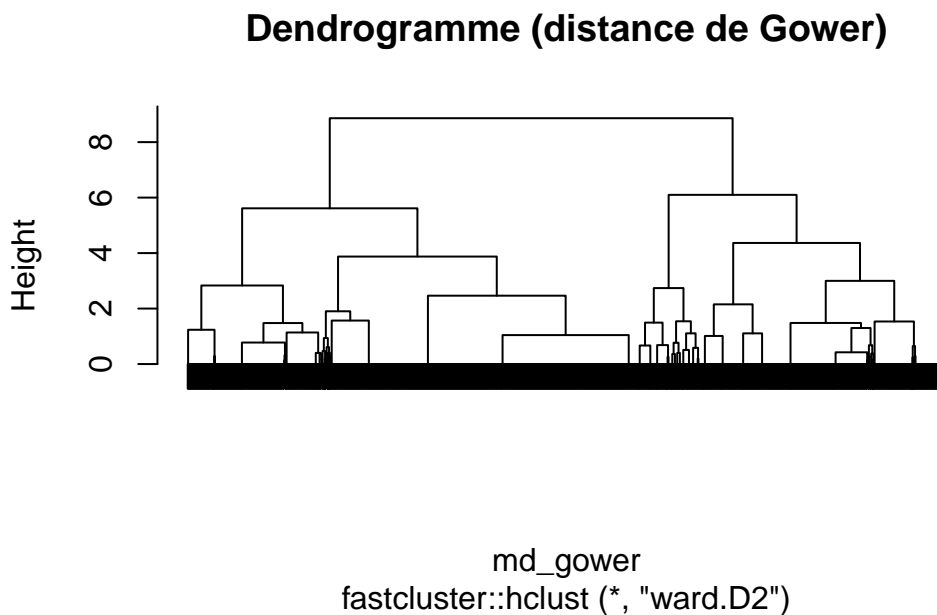


Figure 40.1: Représentation graphique du dendrogramme avec `plot()`

Pour une représentation graphique un peu plus propre (et avec plus d'options que nous verrons plus loin), nous pouvons avoir recours à `factoextra::fviz_dend()` du package `{factoextra}`. Le temps de calcul du graphique est par contre sensible plus long.

```
arbre_gower |>
  factoextra::fviz_dend(show_labels = FALSE) +
  ggplot2::ggtitle("Dendrogramme (distance de Gower)")
```

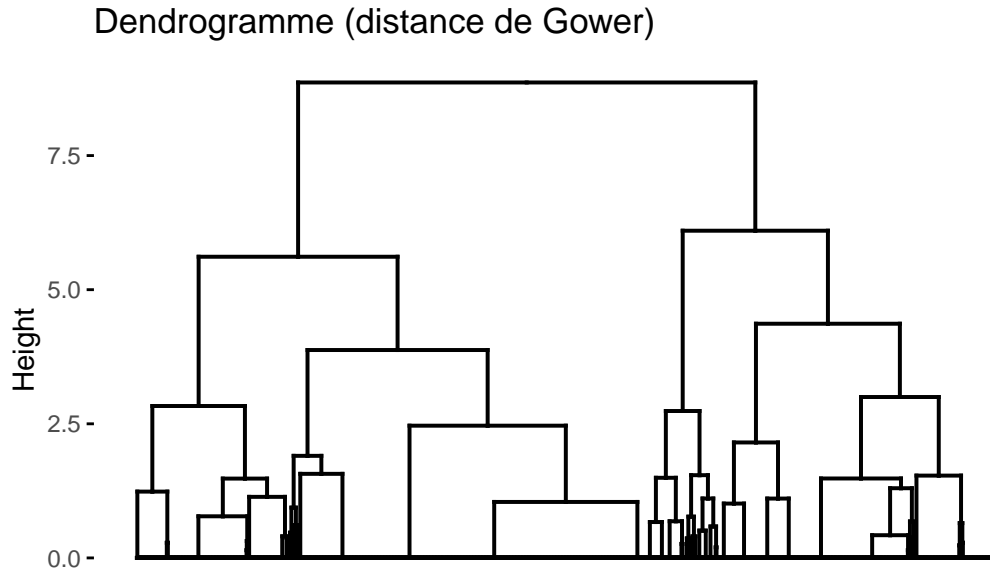


Figure 40.2: Représentation graphique du dendrogramme avec `fviz_dend()`

Il existe d'autres packages offrant des visualisations avancées pour les dendrogrammes. Citons notamment le package `{ggdendro}` et surtout `{dendextend}` qui est très complet.

40.3 Découper le dendrogramme

Pour obtenir une partition de la population, il suffit de découper le dendrogramme obtenu à une certaine hauteur. Cela aura pour effet de découper l'échantillon en plusieurs groupes, i.e. en plusieurs classes.

40.3.1 Classes obtenues avec la distance de Gower

En premier lieu, il est toujours bon de prendre le temps d'observer de la forme des branches du dendrogramme. Reprenons le dendrogramme obtenu avec la distance de Gower (Figure 40.2). Nous recherchons des branches qui se distinguent clairement, c'est-à-dire avec un saut marqué sous la branche. Ici, nous avons tout d'abord deux groupes bien distincts qui apparaissent. Chacune des deux premières branches se sépare ensuite en deux branches bien visibles, suggérant une possible classification en 4 groupes.

Nous pouvons l'impact d'un découpage avec `factoextra::fviz_dend()` en précisant `k = 4` pour lui indiquer de colorer un découpage en 4 classes. On peut optionnellement ajouter `rect = TRUE` pour dessiner des rectangles autour de chaque classe.

```
arbre_gower |>
  factoextra::fviz_dend(
    show_labels = FALSE,
    k = 4,
    rect = TRUE
  ) +
  ggplot2::ggtitle("Dendrogramme découpé en 4 classes (distance de Gowver)")
```

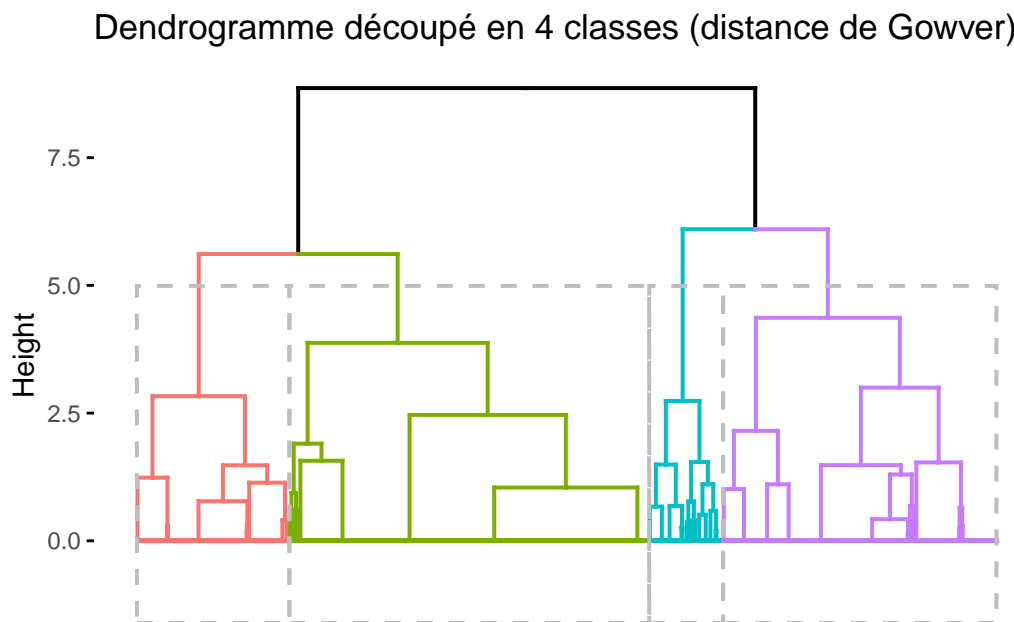


Figure 40.3: Représentation graphique du dendrogramme avec `fviz_dend()`

Pour nous aider dans l'analyse du dendrogramme, il est possible de représenter graphiquement les sauts d'inertie (i.e. de la hauteur des branches) au fur-et-à-mesure que l'on découpe l'arbre en un nombre de classes plus important. Nous pouvons également considérer la perte absolue d'inertie (l'écart de hauteur entre le découpage précédent et le découpage considéré) voire la perte relative (i.e. la perte absolue exprimée en pourcentage de la hauteur précédente). `{FactoMineR}` (que nous aborderons un peu plus loin) suggère par défaut la partition correspondant à la plus grande perte relative d'inertie (*the one with the higher relative loss of inertia*).

Pour faciliter les choses, voici deux petites fonctions que vous pouvez recopier / adapter dans vos scripts. `get_inertia_from_tree()` calcule l'inertie à chaque niveau, ainsi que les pertes absolues et relatives. `plot_inertia_from_tree()` en propose une représentation graphique.

```
get_inertia_from_tree <- function(tree, k_max = 15) {
  if (inherits(tree, "HCPC"))
    tree <- tree$call$t$tree
  if (!inherits(tree, "hclust"))
    tree <- as.hclust(tree)
  inertia <- tree$height |>
    sort(decreasing = TRUE) |>
    head(k_max)
  prev_inertia <- dplyr::lag(inertia)
  dplyr::tibble(
    k = seq_along(inertia),
    inertia = inertia,
    absolute_loss = inertia - prev_inertia,
    relative_loss = (inertia - prev_inertia) / prev_inertia
  )
}

plot_inertia_from_tree <- function(tree, k_max = 15) {
  d <- get_inertia_from_tree(tree, k_max)
  p_inertia <-
    ggplot2::ggplot(d) +
    ggplot2::aes(x = k, y = inertia) +
    ggplot2::geom_step() +
    ggplot2::ylab("Inertia")
  p_absolute <-
    ggplot2::ggplot(d) +
    ggplot2::aes(x = k, y = absolute_loss) +
    ggplot2::geom_bar(stat = "identity", fill = "#4477AA") +
    ggplot2::ylab("Absolute loss")
  p_relative <-
    ggplot2::ggplot(d) +
    ggplot2::aes(x = k, y = relative_loss) +
    ggplot2::geom_line(color = "#AA3377") +
    ggplot2::geom_point(size = 3, color = "#AA3377") +
    ggplot2::scale_y_continuous(label = scales::percent) +
    ggplot2::ylab("Relative loss")
  patchwork::wrap_plots(
    p_inertia,
    p_absolute,

```

```

    p_relative,
    ncol = 1
) &
  ggplot2::theme_light() &
  ggplot2::xlab("Number of clusters") &
  ggplot2::scale_x_continuous(
    breaks = d$k,
    minor_breaks = NULL,
    limits = c(1, k_max)
  )
}

```

Voyons ce qu'on obtient.

```

arbre_gower |>
  plot_inertia_from_tree()

```

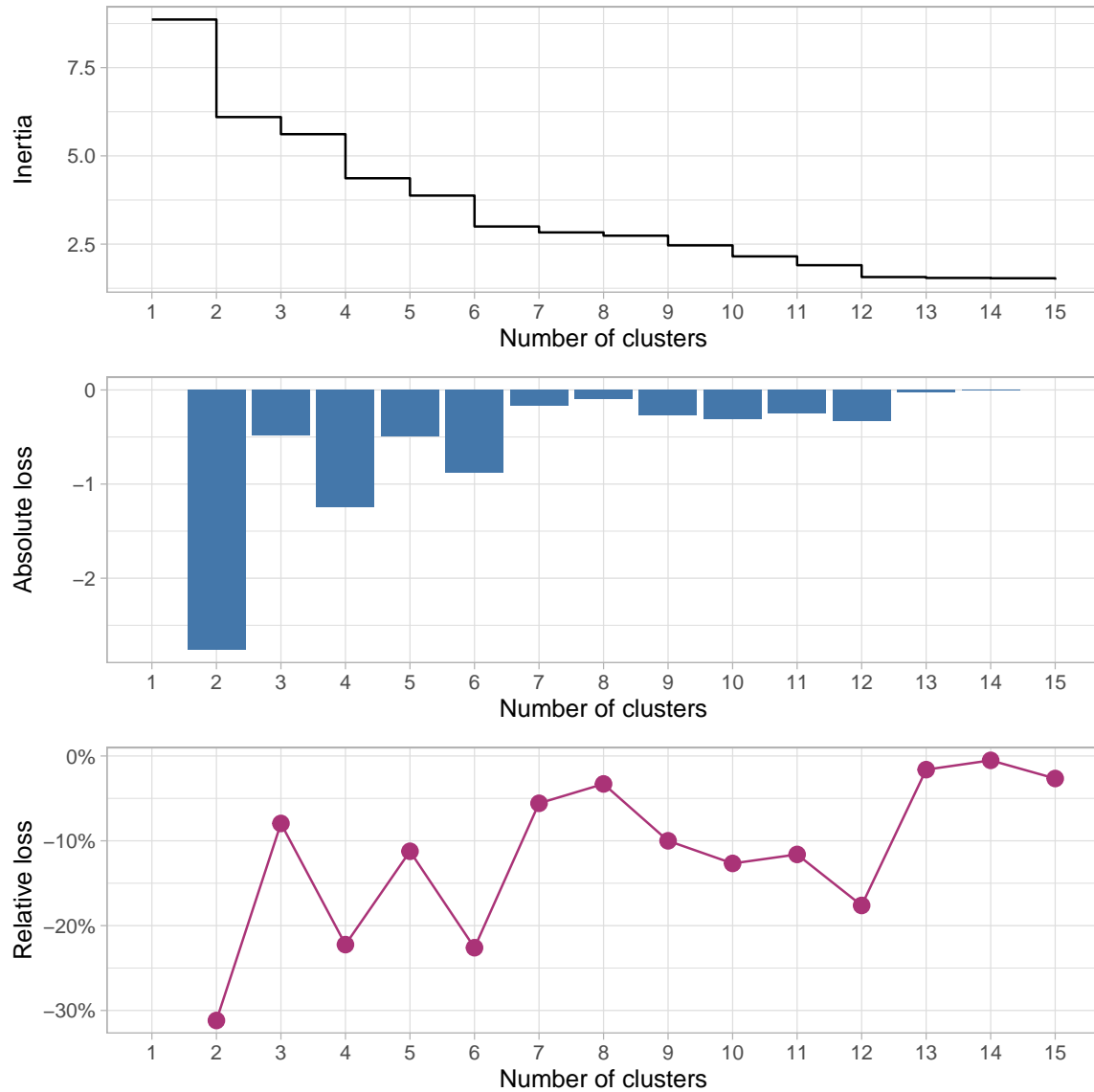


Figure 40.4: Inertie, perte absolue et perte relative d'inertie pour différents découpages en classes (distance de Gower)

Dans cet exemple, nous pouvons voir qu'un découpage en deux classes maximise la perte absolue et la perte relative d'inertie. Mais, pour les besoins de l'analyse, nous pouvons souhaiter un nombre de classe un peu plus élevé (plus de classes permet une analyse plus fine, trop de classes rend l'interprétation des résultats compliqués). Un découpage en 4 classes apparaît sur ce graphique comme une bonne alternative, voir un découpage en 6 classes (une lecture

du dendrogramme nous permet de voir que, dans cet exemple, un découpage en 6 classes reviendrait à couper en deux les 2 classes les plus larges du découpage en 4 classes).

Pour découper notre dendrogramme et récupérer la classification, nous appliquerons la fonction `cutree()` au dendrogramme, en indiquant le nombre de classes souhaitées⁹.

```
hdv2003$typo_gower_4classes <- arbre_gower |>
  cutree(4)
```

Nous pouvons rapidement faire un tri à plat avec `gtsummary::tbl_summary()`.

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",")
```

```
hdv2003 |>
  tbl_summary(include = typo_gower_4classes)
```

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

Table 40.1: Distribution de la typologie en 4 classes obtenues à partir de la distance de Gower

Caractéristique	N = 2 000
typo_gower_4classes	
1	837 (42%)
2	636 (32%)
3	172 (8,6%)
4	355 (18%)

Nous obtenons deux classes principales regroupant chacune plus du tiers de l'échantillon, une troisième classe regroupant presque un cinquième et une dernière classe avec un peu moins de 9 % des individus.

⁹Ici, nous pouvons ajouter le résultat obtenu directement à notre tableau de données `hdv2003` dans la mesure où, depuis le début de l'analyse, l'ordre des lignes n'a jamais changé à aucune étape de l'analyse.

40.3.2 Classes obtenues à partir de l'ACM (distance du Φ^2)

Pour découper l'arbre obtenu à partir de l'ACM, nous allons procéder de la même manière. D'abord, jetons un œil au dendrogramme.

```
arbre_phi2 |>
  factoextra::fviz_dend(show_labels = FALSE) +
  ggplot2::ggtitle("Dendrogramme (distance du  $\Phi^2$ )")
```

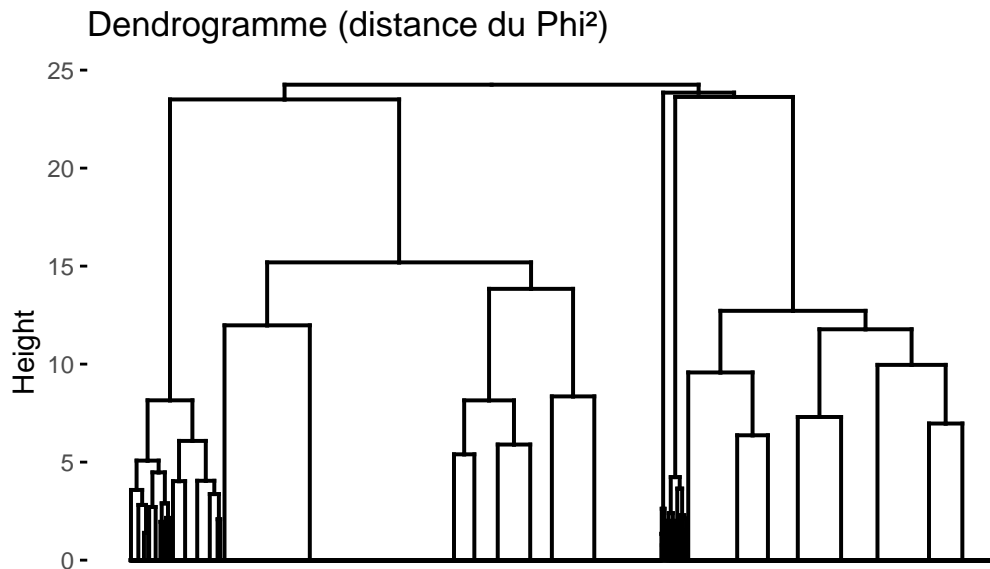


Figure 40.5: Représentation graphique du dendrogramme issu de l'ACM

i Note

Le dendrogramme obtenu est ici bien différent. Cela est lié au fait que les deux distances traitent différemment les modalités atypiques. En effet, la distance du Φ^2 prend en compte la fréquence de chaque modalité dans l'ensemble de l'échantillon. De fait, les modalités très peu représentées dans l'échantillon se retrouvent très éloignées des autres et la CAH aura tendance à isoler les individus atypiques. À l'inverse, la distance de Gower est indépendante de la fréquence de chaque modalité dans l'échantillon. De fait, plutôt que d'isoler les individus atypiques, une CAH basée sur la distance de Gower aura plutôt tendance à les associer aux autres à partir de leurs autres caractéristiques, aboutissant à des classes plus équilibrées.

Il n'y a pas une approche meilleure que l'autre. Tout dépend des questions de recherche que l'on se pose et de ce que l'on souhaite faire émerger.

Comme nous pouvons le voir, dès le début du dendrogramme, l'arbre se divise rapidement en 5 branches puis il y a un saut relativement marqué. Nous pouvons confirmer cela avec `plot_inertia_from_tree()`.

```
arbre_phi2 |>  
  plot_inertia_from_tree()
```

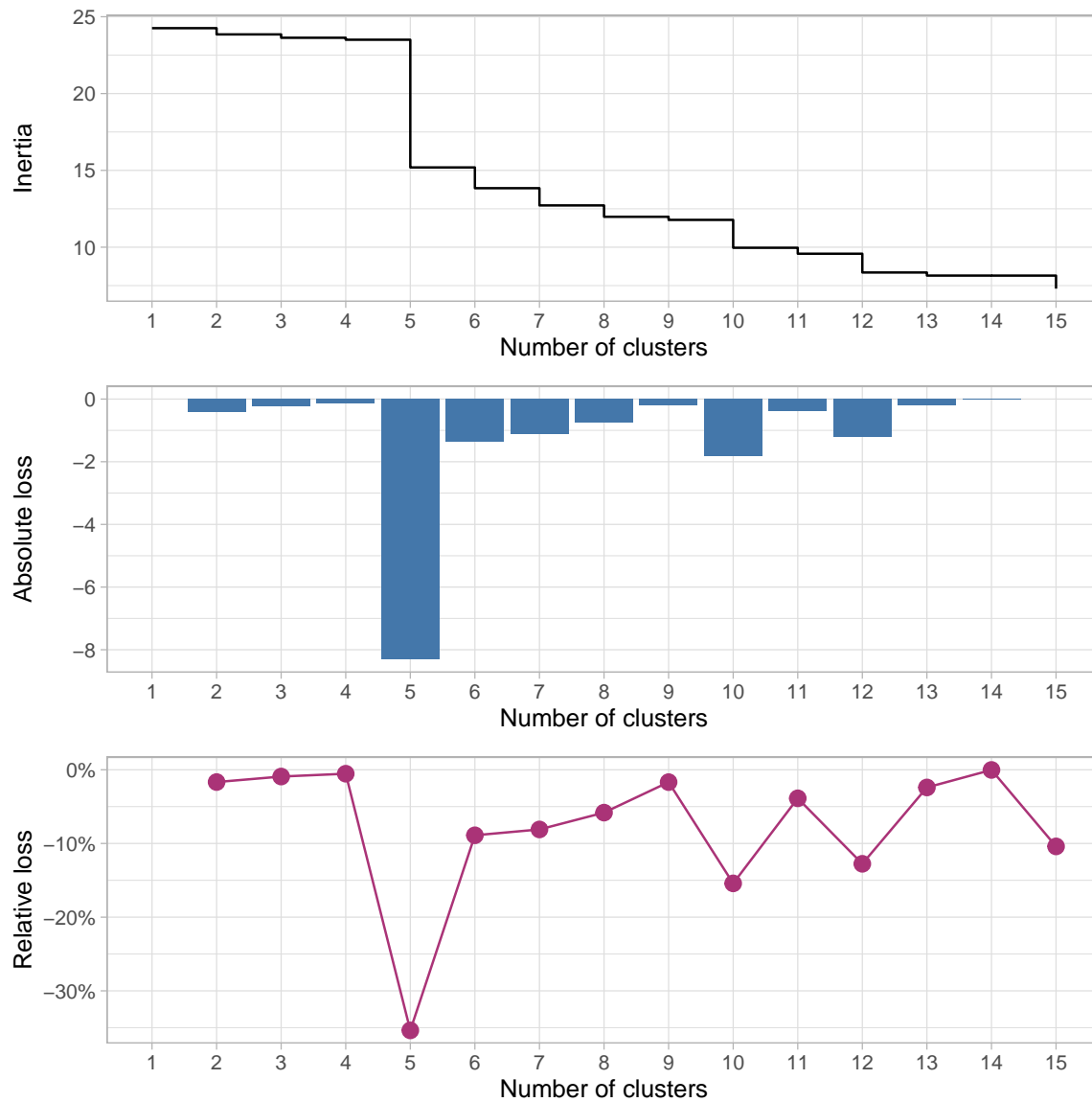


Figure 40.6: Inertie, perte absolue et perte relative d'inertie pour différents découpages en classes (distance de Φ^2)

Cela confirme un découpage optimal en 5 classes. Regardons la distribution de cette typologie.

```
hdv2003$typo_phi2_5classes <- arbre_phi2 |>
  cutree(5)
```



```
hdv2003 |>
  tbl_summary(include = typo_phi2_5classes)
```

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

Table 40.2: Distribution de la typologie en 5 classes obtenues à partir de la distance du Φ^2

Caractéristique	N = 2 000
typo_phi2_5classes	
1	1 010 (51%)
2	713 (36%)
3	216 (11%)
4	14 (0,7%)
5	47 (2,4%)

Sur les 5 classes, deux sont très atypiques puisqu'elles ne réunissent que 0,7 % et 2,4 % de l'échantillon. À voir si cela est problématique pour la suite de l'analyse. Au besoin, nous pourrions envisager de fusionner les classes 4 et 5 avec la classe 2 avec laquelle elles sont plus proches selon le dendrogramme.

Il est possible de visualiser la répartition de la typologie dans le plan factoriel avec `factoextra::fviz_mca_ind()` et en passant la typologie à `habillage`.

```
acm_ad |>
  factoextra::fviz_mca_ind(
    habillage = hdv2003$typo_phi2_5classes,
    addEllipses = TRUE,
    geom.ind = "point",
    alpha.ind = 0.1
  )
```

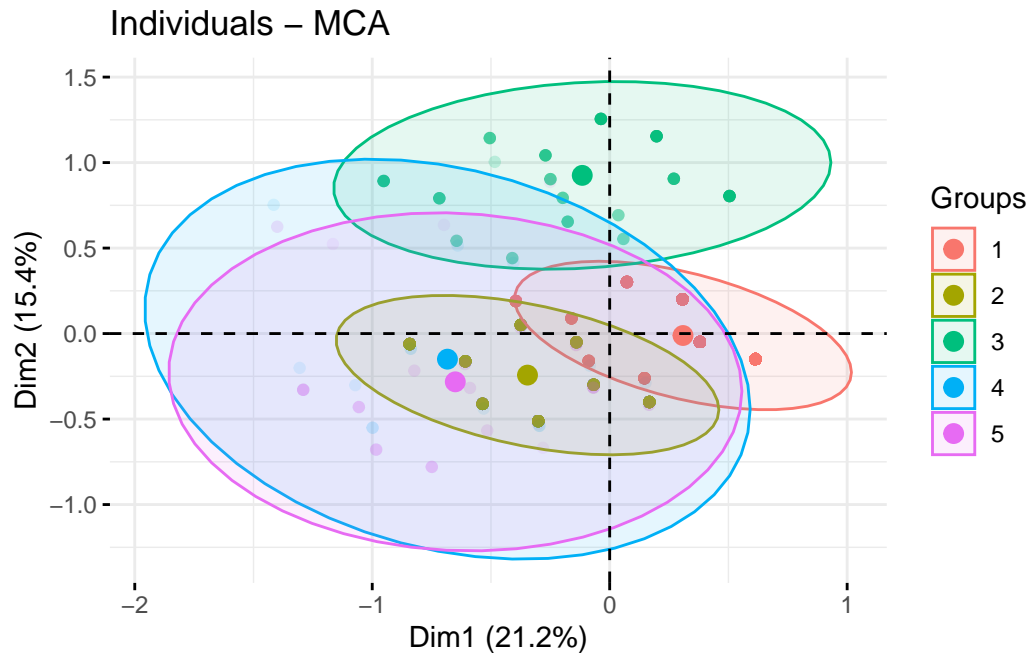


Figure 40.7: Projection de la typologie dans le plan factoriel de l'ACM

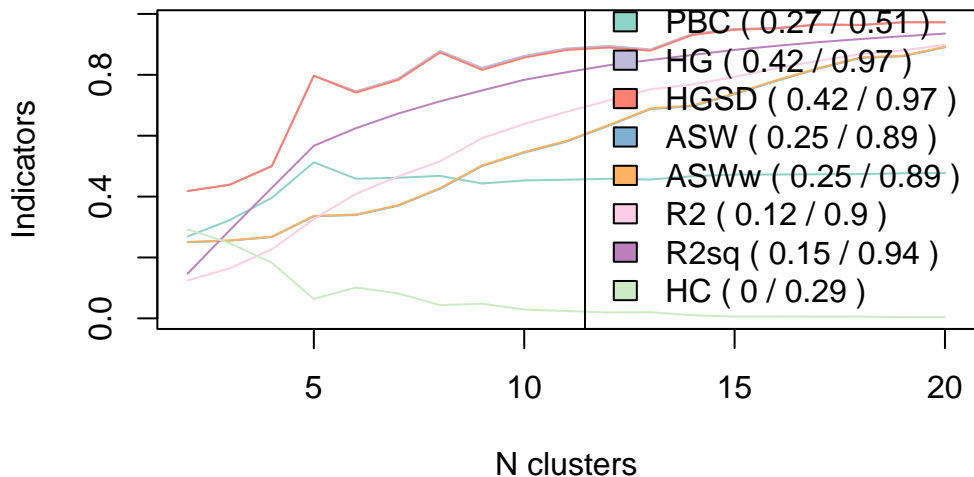
Nous voyons que les classes 1, 2 et 3 séparent bien les individus au niveau du plan factoriel. Les deux classes atypiques 4 et 5, quant à elles, sont diffuses sur les deux premiers axes, suggérant qu'elles capturent des différences observables sur des axes de niveau supérieur.

i Note

Il existe de multiples autres indicateurs statistiques cherchant à mesurer la qualité de chaque partition. Pour cela, on pourra par exemple avoir recours à la fonction `WeightedCluster::as.clustrange()` de l'extension `{WeightedCluster}`. Pour plus d'informations, voir le [manuel de la librairie WeightedCluster](#), chapitre 7.

```
WeightedCluster::as.clustrange(arbre_phi2, md_phi2) |> plot()
```

```
Registered S3 method overwritten by 'vegan':
  method      from
  rev.hclust  dendextend
```



On pourra également lire [Determining The Optimal Number Of Clusters: 3 Must Know Methods](#) par Alboukadel Kassambara, l'un des auteurs du package `{factoextra}`.

40.4 Calcul de l'ACM et de la CAH avec FactoMineR

Le package `{FactoMineR}` permet lui aussi de réaliser une CAH à partir d'une ACM via la fonction `FactoMineR::HCPC()` qui réalise les différentes opérations en une seule fois.

`FactoMineR::HCPC()` réalise à la fois le calcul de la matrice des distances, du dendrogramme et le partitionnement de la population en classes. Par défaut, `FactoMineR::HCPC()` calcule le dendrogramme à partir du carré des distances du Φ^2 et avec la méthode de Ward. Si l'on ne précise rien, `FactoMineR::HCPC()` détermine une partition optimale selon le critère de la plus perte relative d'inertie évoqué plus haut¹⁰. La fonction prend en entrée une analyse factorielle réalisée avec `{FactoMineR}`. Le paramètre `min` permet d'indiquer un nombre minimum de classes.

```
acm_fm <- FactoMineR::MCA(d, graph = FALSE)
cah_fm <- FactoMineR::HCPC(acm_fm, graph = FALSE, min = 3)
```

¹⁰Plus précisément si `graph = FALSE` ou si `nb.clust = -1`. Si `graph = TRUE` et `nb.clust = 0` (valeurs par défaut), la fonction affichera un dendrogramme interactif et l'utilisateur devra cliquer au niveau de la hauteur où il souhaite réaliser la découpe.

Nous pouvons directement passer le résultat de `FactoMineR::HCPC()` à `factoextra::fviz_dend()` pour visualiser le dendrogramme, qui sera d'ailleurs automatiquement colorié à partir de la partition recommandée par `FactoMineR::HCPC()`.

```
cah_fm |>
  factoextra::fviz_dend(show_labels = FALSE)
```

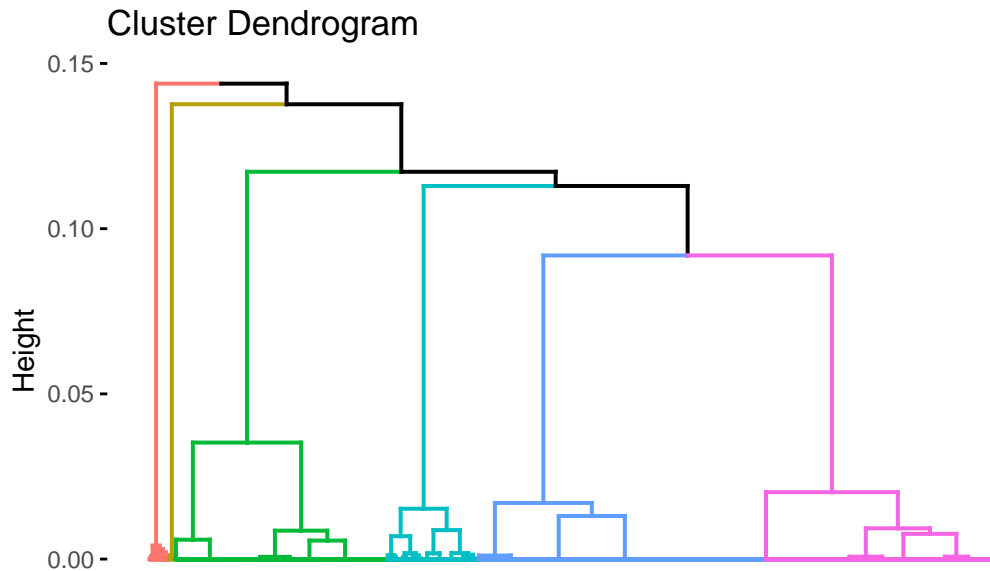


Figure 40.8: Représentation graphique du dendrogramme issu de l'ACM réalisée avec FactoMineR

Nous obtenons ici un découpage en 6 classes qui correspond bien à la plus grande perte relative d'inertie comme nous pouvons le vérifier avec `plot_inertia_from_tree()` qui accepte également en entrée un objet produit par `FactoMineR::HCPC()`.

```
cah_fm |>
  plot_inertia_from_tree()
```

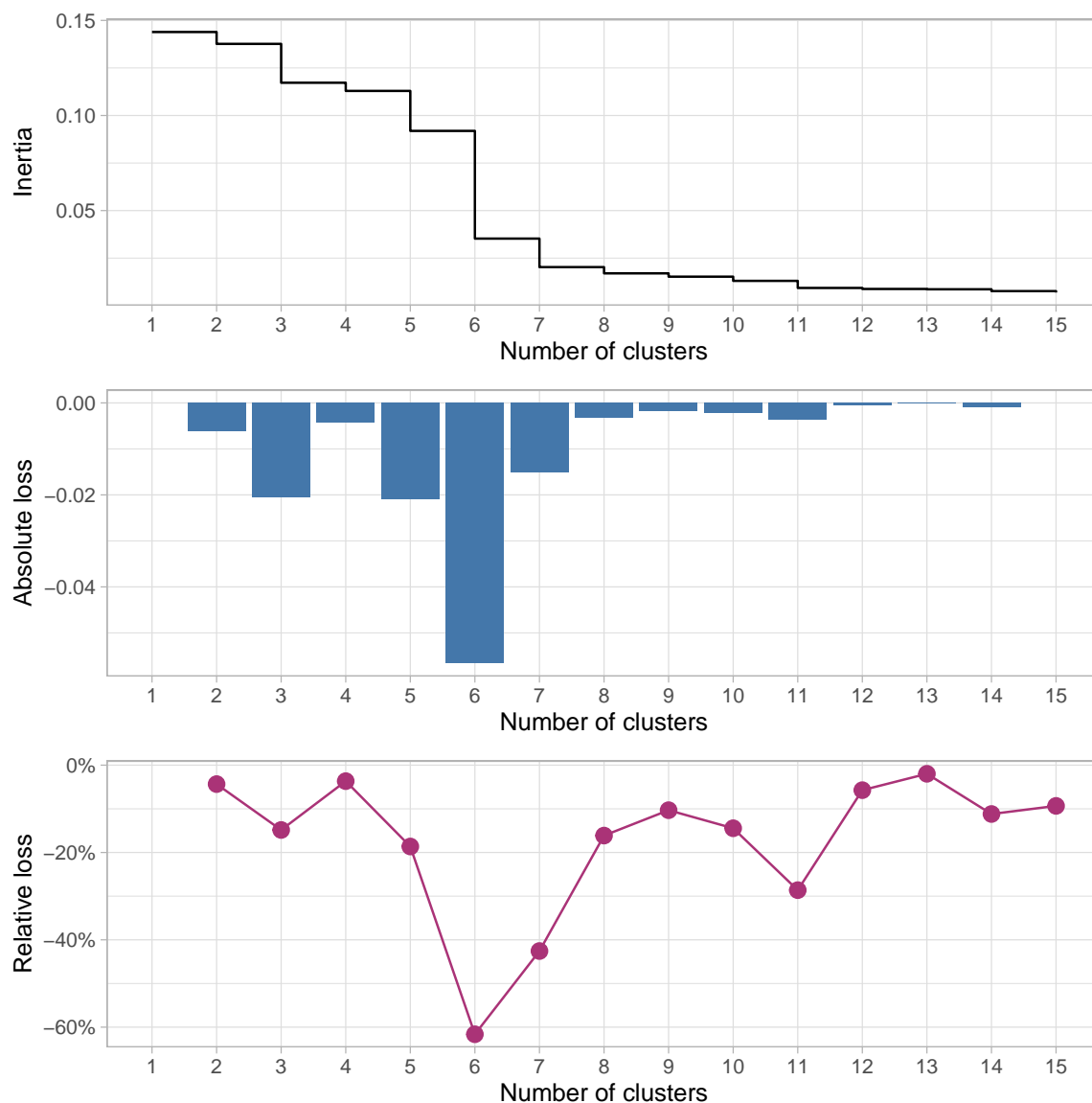


Figure 40.9: Inertie, perte absolue et perte relative d'inertie pour différents découpages en classes à partir du résultat de HCPC()

Par contre, le dendrogramme obtenu diffère de celui que nous avons eu précédemment avec `{ade4}` (cf. Figure 40.5). Cela est dû au fait que `FactoMineR::HCPC()` procède différemment pour calculer la matrice des distances en ne prenant en compte que les axes retenus dans le cadre de l'ACM.

Pour rappel, par défaut, `FactoMineR::MCA()` ne retient que les 5 premiers axes de l'espace

factoriel. `FactoMineR::HCPC()` n'a donc pris en compte que ces 5 premiers axes pour calculer les distances entre les individus, considérant que les autres axes n'apportent que du « bruit » rendant la classification instable. Comme le montre `summary(acm_fm)`, nos cinq premiers axes n'expliquent que 78 % de la variance. On considère usuellement préférable de garder un plus grand nombre d'axes afin de couvrir au moins 80 à 90 % de la variance.

De son côté, `ade4::dist.dudi()` prends en compte l'ensemble des axes pour calculer la matrice des distances. On peut reproduire cela avec `{FactoMineR}` en indiquant `ncp = Inf` lors du calcul de l'ACM.

```
acm_fm2 <- FactoMineR::MCA(d, graph = FALSE, ncp = Inf)
cah_fm2 <- FactoMineR::HCPC(acm_fm2, graph = FALSE, min = 3)
```

```
cah_fm2 |>
  factoextra::fviz_dend(show_labels = FALSE)
```

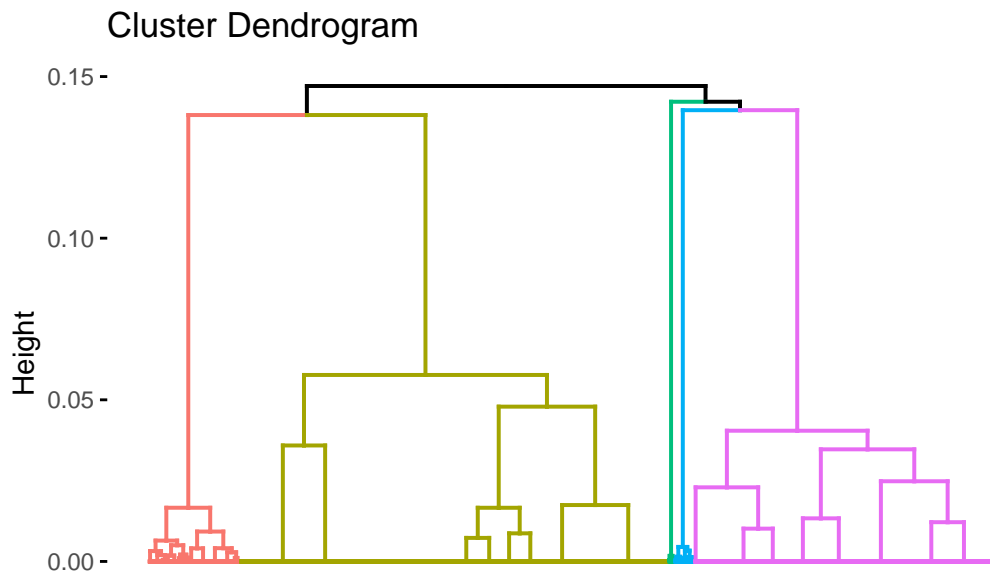


Figure 40.10: Représentation graphique du dendrogramme issu de l'ACM réalisée avec `FactoMineR` avec prise en compte de l'ensemble des axes

Nous retrouvons alors le même résultat que celui obtenu avec `{ade4}` et un découpage en 5 classes.

À noter que `{FactoMineR}` propose une visualisation en 3 dimensions du dendrogramme projeté sur le plan factoriel.

```
cah_fm2 |>
  plot()
```

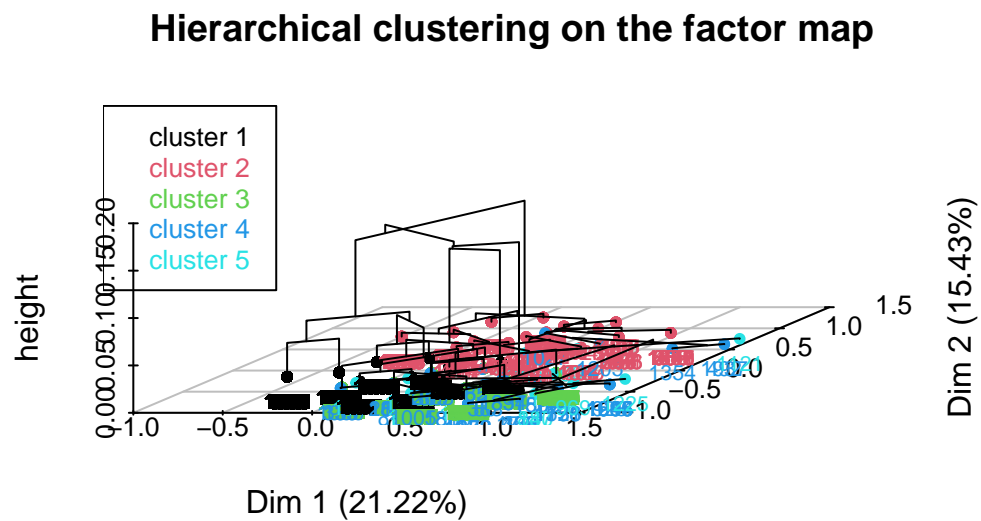


Figure 40.11: Représentation en 3 dimensions du dendrogramme sur le plan factoriel

Notons également l'option `choice = "tree"` qui propose une représentation du dendrogramme, avec des rectangles indiquant le découpage optimal et une vignette présentant l'inertie à chaque découpage.

```
cah_fm2 |>
  plot(choice = "tree")
```

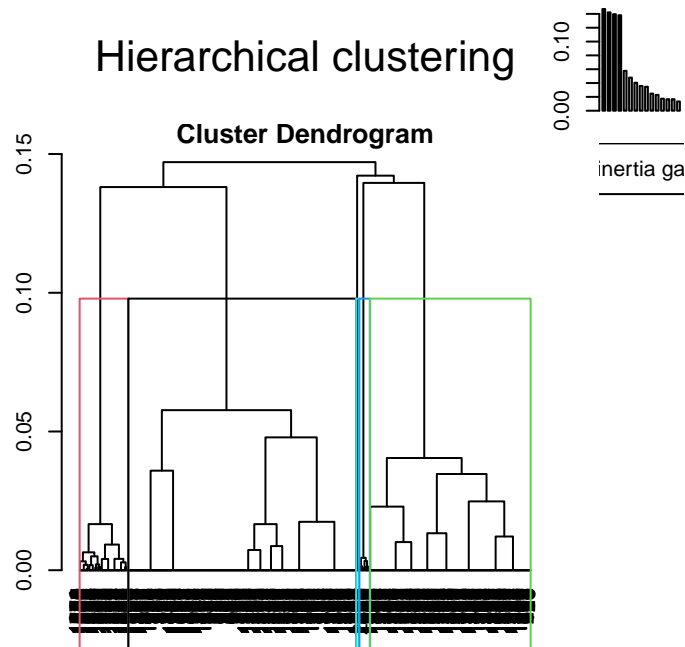


Figure 40.12: Représentation du dendrogramme et de l'inertie

Pour récupérer la classification, on pourra récupérer la colonne `$clust` du sous-objet `$data.clust` du résultat renvoyé par `FactoMineR::HCPC()`.

```
hdv2003$typo_cah_fm2 <- cah_fm2$data.clust$clust
```

Si l'on a besoin de découper le dendrogramme à un autre endroit, nous pouvons récupérer le dendrogramme via le sous-objet `$call$t$tree` puis lui appliquer `cutree()`.

```
hdv2003$typo_alternative <-  
  cah_fm2$call$t$tree |>  
  cutree(3)
```

40.5 Caractériser la typologie

Reste le travail le plus important (et parfois le plus difficile) qui consiste à catégoriser la typologie obtenue et le cas échéant à nommer les classes.

En premier lieu, on peut croiser la typologie obtenue avec les différentes variables incluses dans l'ACM. Le plus simple est d'avoir recours à `gtsummary::tbl_summary()`. Par exemple, pour la typologie obtenue avec la distance de Gower.


```
hdv2003 |>
  tbl_summary(
    include = hard.rock:sport,
    by = typo_gower_4classes
  ) |>
  bold_labels()
```

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

Table 40.3: Description de la typologie en 4 classes à partir de la distance de Gower

Caractéristique	1, N = 837	2, N = 636	3, N = 172	4, N = 355
hard.rock				
Non	837 (100%)	632 (99%)	171 (99%)	346 (97%)
Oui	0 (0%)	4 (0,6%)	1 (0,6%)	9 (2,5%)
lecture.bd				
Non	813 (97%)	613 (96%)	172 (100%)	355 (100%)
Oui	24 (2,9%)	23 (3,6%)	0 (0%)	0 (0%)
peche.chasse				
Non	806 (96%)	633 (100%)	0 (0%)	337 (95%)
Oui	31 (3,7%)	3 (0,5%)	172 (100%)	18 (5,1%)
cuisine				
Non	532 (64%)	316 (50%)	79 (46%)	192 (54%)
Oui	305 (36%)	320 (50%)	93 (54%)	163 (46%)
bricol				
Non	570 (68%)	317 (50%)	47 (27%)	213 (60%)
Oui	267 (32%)	319 (50%)	125 (73%)	142 (40%)
cinema				
Non	831 (99%)	226 (36%)	115 (67%)	2 (0,6%)
Oui	6 (0,7%)	410 (64%)	57 (33%)	353 (99%)
sport				
Non	834 (100%)	2 (0,3%)	86 (50%)	355 (100%)
Oui	3 (0,4%)	634 (100%)	86 (50%)	0 (0%)

Pour une représentation plus visuelle, on peut également avoir recours à `GGally::ggtable()` de `{GGally}` pour représenter les résidus du χ^2 et mieux repérer les différences. La couleur bleue indique que la modalité est sur-représentée et la couleur rouge qu'elle est sous-représentée.

```

library(GGally)
hdv2003$typo_gower_4classes <- factor(hdv2003$typo_gower_4classes)
ggtable(
  hdv2003,
  columnsX = "typo_gower_4classes",
  columnsY = names(d),
  cells = "col.prop",
  fill = "std.resid"
) +
  labs(fill = "Résidus standardizés du Chi2") +
  theme(legend.position = "bottom")

```

typo_gower_4classes					
Oui	0.0%	0.6%	0.6%	2.5%	hard/rock
Non	100.0%	99.4%	99.4%	97.5%	
Oui	2.9%	3.6%	0.0%	0.0%	lecture/bd
Non	97.1%	96.4%	100.0%	100.0%	
Oui	3.7%	0.5%	100.0%	5.1%	peche/chasse
Non	96.3%	99.5%	0.0%	94.9%	
Oui	36.4%	50.3%	54.1%	45.9%	cuisine
Non	63.6%	49.7%	45.9%	54.1%	
Oui	31.9%	50.2%	72.7%	40.0%	bricol
Non	68.1%	49.8%	27.3%	60.0%	
Oui	0.7%	64.5%	33.1%	99.4%	cinema
Non	99.3%	35.5%	66.9%	0.6%	
Oui	0.4%	99.7%	50.0%	0.0%	sport
Non	99.6%	0.3%	50.0%	100.0%	
1234					

Figure 40.13: Distribution de la typologie en 4 classes et résidus du Chi²

Une première lecture nous indique que :

- la première classe rassemble des individus qui n'ont pas (ou peu) de loisirs ;
- la seconde classe réunit des personnes pratiquant un sport et ayant souvent une autre activité telle que le cinéma, la bricolage ou la cuisine ;
- la troisième classe réunit spécifiquement les individus pratiquant la chasse ou la pêche ;
- la quatrième classe correspond à des personnes ne pratiquant pas de sport mais allant au cinéma.

Bien sûr, l'interprétation fine des catégories nécessite un peu plus d'analyse, de croiser avec la littérature et les hypothèses des questions de recherche posées, et de croiser la typologie avec d'autres variables de l'enquête.

Pour la typologie réalisée à partir d'une ACM, nous pourrions procéder de la même manière. Cependant, si la CAH a été réalisée avec `FactoMineR::HCPC()`, l'objet retourné contient directement un sous-objet `$desc.var` donnant une description de la typologie obtenue.

```
cah_fm2$desc.var
```

Link between the cluster variable and the categorical variables (chi-square test)

```
=====
                p.value df
hard.rock      0.000000e+00 4
lecture.bd     0.000000e+00 4
peche.chasse   0.000000e+00 4
cinema         0.000000e+00 4
sport          1.122743e-38 4
bricol         4.121835e-11 4
cuisine        8.681152e-04 4
```

Description of each cluster by the categories

```
=====
$`1`
```

	Cla/Mod	Mod/Cla	Global	p.value	v.test
cinema=cinema_Non	86.03066	100.00000	58.70	0.000000e+00	Inf
peche.chasse=peche.chasse_Non	56.86937	100.00000	88.80	2.188843e-75	18.372312
sport=sport_Non	61.39389	77.62376	63.85	6.009360e-39	13.054257
lecture.bd=lecture.bd_Non	51.71531	100.00000	97.65	2.518852e-15	7.912690
bricol=bricol_Non	57.10549	64.85149	57.35	6.993497e-12	6.857788
cuisine=cuisine_Non	54.60232	60.49505	55.95	3.566198e-05	4.133925
hard.rock=hard.rock_Non	50.85599	100.00000	99.30	5.060286e-05	4.052825

hard.rock=hard.rock_Oui	0.00000	0.00000	0.70	5.060286e-05	-4.052825
cuisine=cuisine_Oui	45.28944	39.50495	44.05	3.566198e-05	-4.133925
bricol=bricol_Oui	41.61782	35.14851	42.65	6.993497e-12	-6.857788
lecture.bd=lecture.bd_Oui	0.00000	0.00000	2.35	2.518852e-15	-7.912690
sport=sport_Oui	31.25864	22.37624	36.15	6.009360e-39	-13.054257
peche.chasse=peche.chasse_Oui	0.00000	0.00000	11.20	2.188843e-75	-18.372312
cinema=cinema_Oui	0.00000	0.00000	41.30	0.000000e+00	-Inf

\$`2`

	Cla/Mod	Mod/Cla	Global	p.value
peche.chasse=peche.chasse_Oui	96.428571	100.00000	11.20	2.257673e-282
bricol=bricol_Oui	14.536928	57.40741	42.65	4.149911e-06
lecture.bd=lecture.bd_Non	11.059908	100.00000	97.65	4.347616e-03
cinema=cinema_Non	12.265758	66.66667	58.70	1.129267e-02
cinema=cinema_Oui	8.716707	33.33333	41.30	1.129267e-02
lecture.bd=lecture.bd_Oui	0.000000	0.00000	2.35	4.347616e-03
bricol=bricol_Non	8.020924	42.59259	57.35	4.149911e-06
peche.chasse=peche.chasse_Non	0.000000	0.00000	88.80	2.257673e-282
	v.test			
peche.chasse=peche.chasse_Oui	35.908415			
bricol=bricol_Oui	4.603730			
lecture.bd=lecture.bd_Non	2.851773			
cinema=cinema_Non	2.533509			
cinema=cinema_Oui	-2.533509			
lecture.bd=lecture.bd_Oui	-2.851773			
bricol=bricol_Non	-4.603730			
peche.chasse=peche.chasse_Non	-35.908415			

\$`3`

	Cla/Mod	Mod/Cla	Global	p.value	v.test
cinema=cinema_Oui	86.31961	100.00000	41.30	0.000000e+00	Inf
peche.chasse=peche.chasse_Non	40.14640	100.00000	88.80	6.154083e-47	14.388011
sport=sport_Oui	52.97372	53.71669	36.15	1.222033e-33	12.088015
lecture.bd=lecture.bd_Non	36.50794	100.00000	97.65	7.397824e-10	6.157337
bricol=bricol_Oui	40.21102	48.10659	42.65	2.493205e-04	3.662957
cuisine=cuisine_Oui	39.38706	48.66760	44.05	2.003812e-03	3.089667
hard.rock=hard.rock_Non	35.90131	100.00000	99.30	2.035491e-03	3.085005
hard.rock=hard.rock_Oui	0.00000	0.00000	0.70	2.035491e-03	-3.085005
cuisine=cuisine_Non	32.70777	51.33240	55.95	2.003812e-03	-3.089667
bricol=bricol_Non	32.25806	51.89341	57.35	2.493205e-04	-3.662957
lecture.bd=lecture.bd_Oui	0.00000	0.00000	2.35	7.397824e-10	-6.157337
sport=sport_Non	25.84182	46.28331	63.85	1.222033e-33	-12.088015
peche.chasse=peche.chasse_Oui	0.00000	0.00000	11.20	6.154083e-47	-14.388011

```
cinema=cinema_Non          0.00000  0.00000  58.70 0.000000e+00      -Inf
```

```
$`4`
```

	Cla/Mod	Mod/Cla	Global	p.value	v.test
lecture.bd=lecture.bd_Oui	100.000000	100.00000	2.35	3.168360e-96	20.814956
cinema=cinema_Oui	3.510896	61.70213	41.30	4.803168e-03	2.819946
sport=sport_Oui	3.319502	51.06383	36.15	3.601926e-02	2.096710
sport=sport_Non	1.801096	48.93617	63.85	3.601926e-02	-2.096710
cinema=cinema_Non	1.533220	38.29787	58.70	4.803168e-03	-2.819946
lecture.bd=lecture.bd_Non	0.000000	0.00000	97.65	3.168360e-96	-20.814956

```
$`5`
```

	Cla/Mod	Mod/Cla	Global	p.value	v.test
hard.rock=hard.rock_Oui	100.0000000	100.00000	0.7	5.569208e-36	12.523271
cinema=cinema_Oui	1.4527845	85.71429	41.3	9.122223e-04	3.316287
cinema=cinema_Non	0.1703578	14.28571	58.7	9.122223e-04	-3.316287
hard.rock=hard.rock_Non	0.0000000	0.00000	99.3	5.569208e-36	-12.523271

Une représentation graphique indiquant les modalités contribuant le plus à chaque axe est même directement disponible. La couleur bleue indique que la modalité est sous-représentée dans la classe et la couleur rouge qu'elle est sur-représentée¹¹.

```
cah_fm2$desc.var |> plot()
```

¹¹ **Attention à l'interprétation** : ce code couleur est l'inverse de celui utilisé par `GGally::ggtable()`.

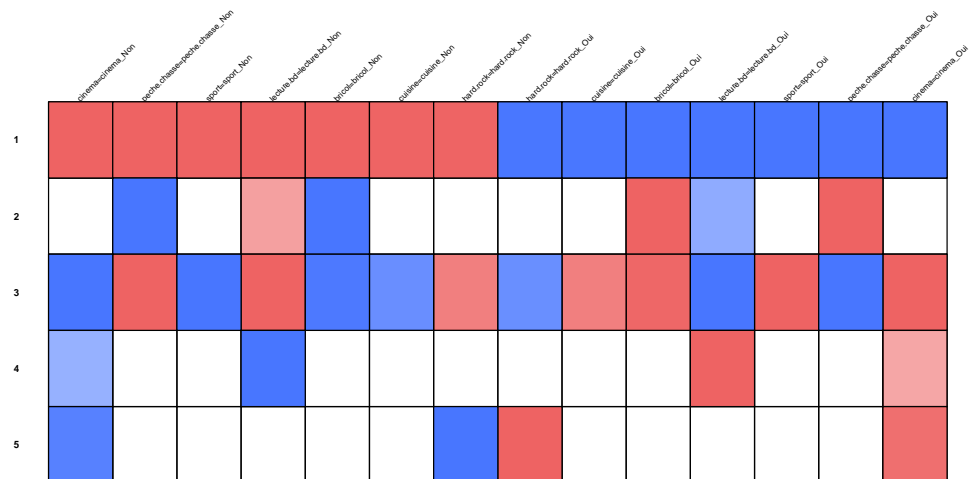


Figure 40.14: Représentation graphique des modalités contribuant le plus

40.6 webin-R

La CAH est présentée sur YouTube dans le [webin-R #12](#) (*Classification Ascendante Hiérarchique*).

<https://youtu.be/Q8adb64NzUI>

41 Régression logistique multinomiale

La régression logistique multinomiale est une extension de la régression logistique binaire (cf. Chapitre 22) aux variables qualitatives à trois modalités ou plus. Dans ce cas de figure, chaque modalité de la variable d'intérêt sera comparée à une modalité de référence. Les *odds ratio* seront donc exprimés par rapport à cette dernière.

41.1 Données d'illustration

Pour illustrer la régression logistique multinomiale, nous allons reprendre le jeu de données `hdv2003` du package `{questionr}` et portant sur l'enquête *histoires de vie 2003* de l'Insee.

```
library(tidyverse)
library(labelled)
data("hdv2003", package = "questionr")
d <- hdv2003
```

Nous allons considérer comme variable d'intérêt la variable *trav.satisf*, à savoir la satisfaction ou l'insatisfaction au travail.

```
questionr::freq(d$trav.satisf)
```

	n	%	val%
Satisfaction	480	24.0	45.8
Insatisfaction	117	5.9	11.2
Equilibre	451	22.6	43.0
NA	952	47.6	NA

Nous allons choisir comme modalité de référence la position intermédiaire, à savoir l'« équilibre », que nous allons donc définir comme la première modalité du facteur.

```
d$trav.satisf <- d$trav.satisf |> fct_relevel("Equilibre")
```

Nous allons aussi en profiter pour raccourcir les étiquettes de la variable *trav.imp* :


```
levels(d$trav.imp) <- c("Le plus", "Aussi", "Moins", "Peu")
```

Enfin, procédons à quelques recodages additionnels :

```
d <- d |>
mutate(
  sexe = sexe |> fct_relevel("Femme"),
  groupe_ages = age |>
    cut(
      c(18, 25, 45, 99),
      right = FALSE,
      include.lowest = TRUE,
      labels = c("18-24 ans", "25-44 ans",
                 "45 et plus")
    ),
  etudes = nivetud |>
    fct_recode(
      "Primaire" = "N'a jamais fait d'etudes",
      "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
      "Primaire" = "Derniere annee d'etudes primaires",
      "Secondaire" = "1er cycle",
      "Secondaire" = "2eme cycle",
      "Technique / Professionnel" = "Enseignement technique ou professionnel court",
      "Technique / Professionnel" = "Enseignement technique ou professionnel long",
      "Supérieur" = "Enseignement superieur y compris technique superieur"
    ) |>
    fct_na_value_to_level("Non documenté")
) |>
set_variable_labels(
  trav.satisf = "Satisfaction dans le travail",
  sexe = "Sexe",
  groupe_ages = "Groupe d'âges",
  etudes = "Niveau d'études",
  trav.imp = "Importance accordée au travail"
)
```

41.2 Calcul du modèle multinomial

Pour calculer un modèle logistique multinomial, nous allons utiliser la fonction `nnet::multinom()` de l'extension `{nnet}`¹. La syntaxe de `nnet::multinom()` est similaire à celle de `glm()`, le paramètre `family` en moins.

```
reg <- nnet::multinom(  
  trav.satisf ~ sexe + etudes + groupe_ages + trav.imp,  
  data = d  
)
```

```
# weights:  36 (22 variable)  
initial  value 1151.345679  
iter   10 value 977.985279  
iter   20 value 971.187398  
final   value 971.113280  
converged
```

Comme pour la régression logistique binaire, il est possible de réaliser une sélection pas à pas descendante (cf. Chapitre 23) :

```
reg2 <- reg |> step()
```

```
trying - sexe  
trying - etudes  
trying - groupe_ages  
trying - trav.imp  
trying - sexe  
trying - etudes  
trying - trav.imp  
trying - etudes  
trying - trav.imp
```

¹Il existe plusieurs alternatives possibles : la fonction `VGAM::vglm()` avec `family = VGAM::multinomial` ou encore `mlogit::mlogit()`. Ces deux fonctions sont un peu plus complexes à mettre en œuvre. On se référera à la documentation de chaque package. Le support des modèles `mlogit()` et `vglm()` est aussi plus limité dans d'autres packages tels que `{broom.helpers}`, `{gtsummary}`, `{ggstats}` ou encore `{marginaleffects}`.

41.3 Affichage des résultats du modèle

Une des particularités de la régression logistique multinomiale est qu'elle produit une série de coefficients pour chaque modalité de la variable d'intérêt (sauf la modalité de référence). Ici, nous aurons donc une série de coefficients pour celles et ceux qui sont satisfaits au travail (comparés à la modalité Équilibre) et une série de coefficients pour celles et ceux qui sont insatisfaits (comparés aux aussi à la modalité Équilibre).

La fonction `gtsummary::tbl_regression()` peut gérer ce type de modèles, et va afficher les deux séries de coefficients l'une au-dessus de l'autre. Nous allons indiquer `exponentiate = TRUE` car, comme pour la régression logistique binaire, l'exponentielle des coefficients peut s'interpréter comme des *odds ratios*.

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",")
```

```
tbl <- reg2 |>
  tbl_regression(exponentiate = TRUE)
```

i Multinomial models have a different underlying structure than the models gtsummary was designed for. Other gtsummary functions designed to work with tbl_regression objects may yield unexpected results.

```
tbl
```

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
To suppress this message, include ``message = FALSE`` in code chunk header.

Table 41.1: Tableau des odds ratio de la régression logistique multinomiale

Outcome	Caractéristique	OR	95% IC	p-valeur
Satisfaction	Niveau d'études	—	—	
	Primaire	—	—	
	Secondaire	1,05	0,63 – 1,76	0,9
	Technique / Professionnel	1,08	0,67 – 1,73	0,7
	Supérieur	2,01	1,24 – 3,27	0,005
	Non documenté	0,58	0,18 – 1,86	0,4
	Importance accordée au travail			
	Le plus	—	—	

Table 41.1: Tableau des odds ratio de la régression logistique multinomiale

Outcome	Caractéristique	OR	95% IC	p-valeur
Insatisfaction	Aussi	1,29	0,56 – 2,98	0,5
	Moins	0,84	0,37 – 1,88	0,7
	Peu	0,55	0,18 – 1,64	0,3
	Niveau d'études			
	Primaire	—	—	
	Secondaire	0,91	0,41 – 1,99	0,8
	Technique / Professionnel	1,09	0,54 – 2,19	0,8
	Supérieur	1,08	0,51 – 2,29	0,8
	Non documenté	0,96	0,18 – 4,97	>0,9
	Importance accordée au travail			
	Le plus	—	—	
	Aussi	0,80	0,24 – 2,69	0,7
	Moins	0,59	0,18 – 1,88	0,4
	Peu	3,82	1,05 – 13,9	0,042

L'*odds ratio* du niveau d'étude supérieur pour la modalité *satisfaction* est de 2,01, indiquant que les personnes ayant un niveau d'étude supérieur ont plus de chances d'être satisfait au travail que d'être à l'équilibre que les personnes de niveau primaire. Par contre, l'OR est de seulement 1,08 (et non significatif) pour la modalité *Insatisfait* indiquant que ces personnes n'ont ni plus ni moins de chance d'être insatisfaite que d'être à l'équilibre.

On notera au passage un message d'avertissement de `{gtsummary}` sur le fait que les modèles multinomiaux n'ont pas la même structure que d'autres modèles et donc que certaines fonctionnalités ne sont pas disponibles. C'est le cas par exemple de `gtsummary::add_global_p()` pour le calcul des p-valeurs globales des variables. Pour tester l'effet globale d'une variable dans le modèle, on aura directement recours à `car::Anova()`.

```
reg2 |> car::Anova()
```

Analysis of Deviance Table (Type II tests)

Response: trav.satisf

	LR	Chisq	Df	Pr(>Chisq)
etudes	24.211	8	0.002112	**
trav.imp	48.934	6	7.687e-09	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

💡 Astuce

La fonction `gtsummary::tbl_regression()` affiche le tableau des coefficients dans un format long. Or, il est souvent plus lisible de présenter les coefficients dans un format large, avec les coefficients pour chaque modalité côte à côte.

Cela n'est pas possible nativement avec `{gtsummary}` mais on pourra éventuellement utiliser la fonction `multinom_pivot_wider()` proposée sur [GitHub Gist](#). Voici son code à recopier dans son script.

```
multinom_pivot_wider <- function(x) {  
  # check inputs match expectatations  
  if (!inherits(x, "tbl_regression") || !inherits(x$model_obj, "multinom")) {  
    stop("`x` must be class 'tbl_regression' summary of a `nnet::multinom()` model.")  
  }  
  
  # create tibble of results  
  df <- tibble::tibble(outcome_level = unique(x$table_body$groupname_col))  
  df$tbl <-  
    purrr::map(  
      df$outcome_level,  
      function(lvl) {  
        gtsummary::modify_table_body(  
          x,  
          ~dplyr::filter(.x, .data$groupname_col %in% lvl) %>%  
            dplyr::ungroup() %>%  
            dplyr::select(-.data$groupname_col)  
        )  
      }  
    )  
  
  tbl_merge(df$tbl, tab_spanner = paste0("**", df$outcome_level, "**"))  
}
```

Il ne reste plus qu'à l'appliquer au tableau généré avec `gtsummary::tbl_regression()`. Attention : cette fonction n'est compatible qu'avec les modèles `nnet::multinom()`.

```
tbl |> multinom_pivot_wider()
```

Warning: Use of .data in tidyselect expressions was deprecated in tidyselect 1.2.0.
i Please use `"groupname_col"` instead of `.data\$groupname_col`

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at

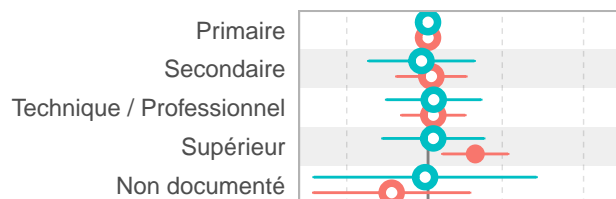
<https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
 To suppress this message, include `message = FALSE` in code chunk header.

Caractéristique	OR	95% IC	p-valeur	OR	95% IC	p-valeur
Niveau d'études						
Primaire	—	—		—	—	
Secondaire	1,05	0,63 – 1,76	0,9	0,91	0,41 – 1,99	0,8
Technique / Professionnel	1,08	0,67 – 1,73	0,7	1,09	0,54 – 2,19	0,8
Supérieur	2,01	1,24 – 3,27	0,005	1,08	0,51 – 2,29	0,8
Non documenté	0,58	0,18 – 1,86	0,4	0,96	0,18 – 4,97	>0,9
Importance accordée au travail						
Le plus	—	—		—	—	
Aussi	1,29	0,56 – 2,98	0,5	0,80	0,24 – 2,69	0,7
Moins	0,84	0,37 – 1,88	0,7	0,59	0,18 – 1,88	0,4
Peu	0,55	0,18 – 1,64	0,3	3,82	1,05 – 13,9	0,042

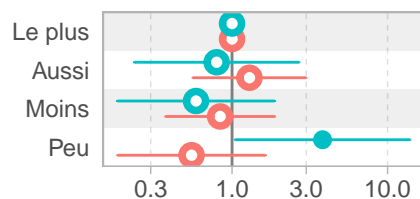
Pour un graphique des coefficients, on ne peut appeler directement `ggstats::ggcoef_model()` en raison de la structure différente du modèle. Heureusement, `{ggstats}` propose une fonction spécifique `ggstats::ggcoef_multinom()` avec trois types de visualisation.

```
reg2 |>
  ggstats::ggcoef_multinom(
    exponentiate = TRUE
  )
```

Niveau d'études



Importance accordée au travail



OR

Satisfaction Insatisfaction

● $p \leq 0.05$ ○ $p > 0.05$

Figure 41.1: Graphique des coefficients du modèle multinomial (type “dodged”)

```
reg2 |>
  ggstats::ggcoef_multinom(
    type = "faceted",
    exponentiate = TRUE
  )
```

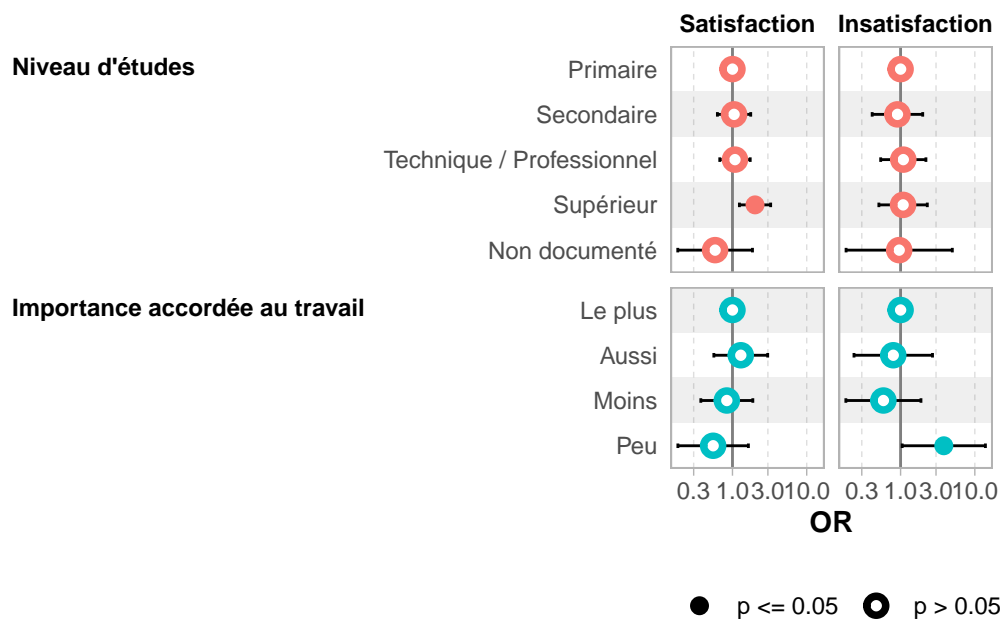
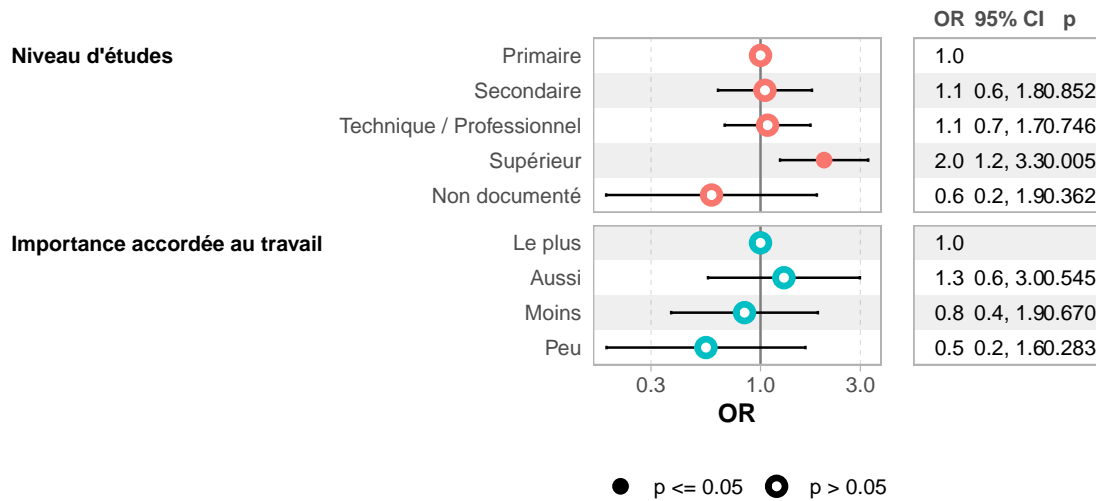


Figure 41.2: Graphique des coefficients du modèle multinomial (type “faceted”)

```
reg2 |>
  ggstats::ggcoef_multinom(
    type = "table",
    exponentiate = TRUE
  )
```


Satisfaction



Insatisfaction

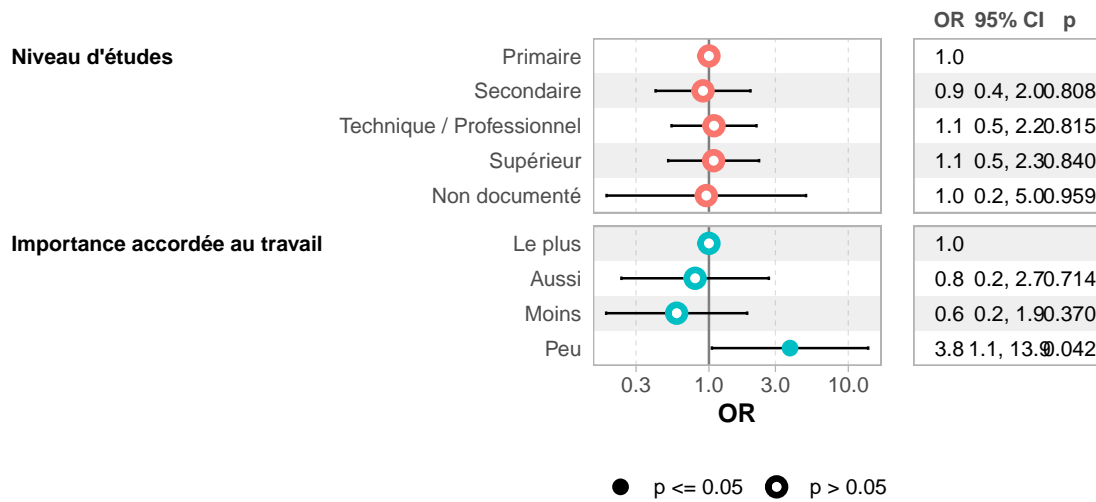


Figure 41.3: Graphique des coefficients du modèle multinomial (type “table”)

Pour faciliter l’interprétation, on pourra représenter les prédictions marginales du modèle (cf. Chapitre 24) avec `broom.helpers::plot_marginal_predictions()`.

```
reg2 |>
  broom.helpers::plot_marginal_predictions() |>
  patchwork::wrap_plots(ncol = 1) &
```

```
scale_y_continuous(labels = scales::percent, limits = c(0, .8)) &
coord_flip()
```

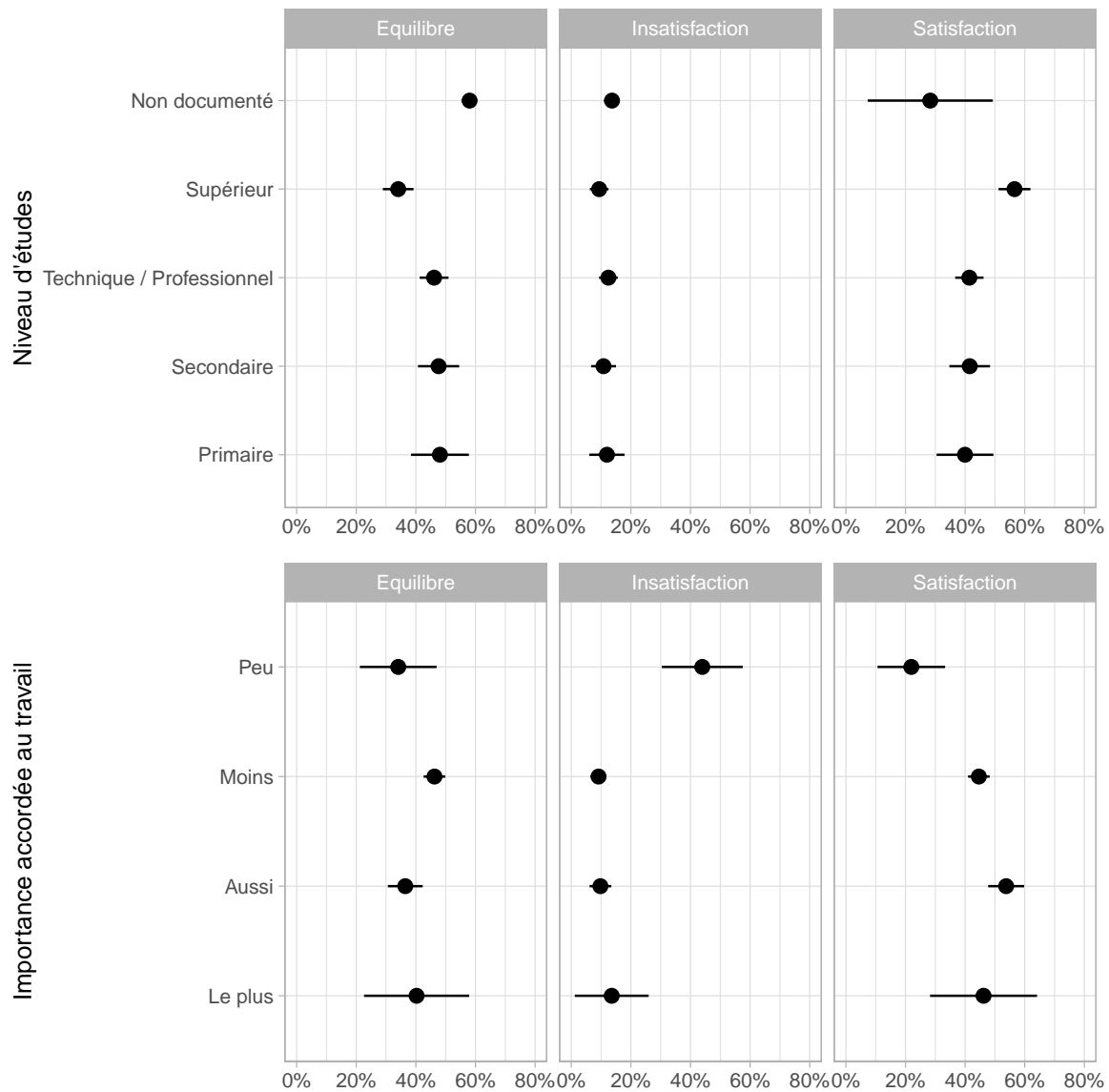


Figure 41.4: Prédictions marginales du modèle multinomial

Astuce

Dans certaines situations, il peut être plus simple de réaliser plusieurs modèles logistiques binaires séparés plutôt qu'une régression multinomiale. Si la variable à expliquer a trois niveaux (A, B et C), on pourra réaliser un modèle binaire B vs A, et un modèle binaire C vs A. Cette approche est appelée approximation de Begg et Gray. On trouvera, en anglais, plus d'explications et des références bibliographiques sur [StackOverflow](#).

41.4 Données pondérées

L'extension `{survey}` (cf. Chapitre 28) ne fournit pas de fonction adaptée aux régressions multinomiales. Cependant, il est possible d'en réaliser une en ayant recours à des poids de réplication, comme suggéré par Thomas Lumley dans son ouvrage *Complex Surveys: A Guide to Analysis Using R*. Thomas Lumley est par ailleurs l'auteur de l'extension `{survey}`.

41.4.1 avec `svrepmisc::svymultinom()`

L'extension `{svrepmisc}` disponible sur [GitHub](#) fournit quelques fonctions facilitant l'utilisation des poids de réplication avec `{survey}`. Pour l'installer, on utilisera le code ci-dessous :

```
remotes::install_github("carlganz/svrepmisc")
```

En premier lieu, il faut définir le design de notre tableau de données puis calculer des poids de réplication.

```
library(survey)
library(srvyr)
dw_rep <- d |>
  as_survey(weights = poids) |>
  as_survey_rep(type = "bootstrap", replicates = 25)
```

Il faut prévoir un nombre de `replicates` suffisant pour calculer ultérieurement les intervalles de confiance des coefficients. Plus ce nombre est élevé, plus précise sera l'estimation de la variance et donc des valeurs p et des intervalles de confiance. Cependant, plus ce nombre est élevé, plus le temps de calcul sera important. Pour gagner en temps de calcul, nous avons ici pris une valeur de 25, mais l'usage est de considérer au moins 1000 réplifications.

`{svrepmisc}` fournit une fonction `svrepmisc::svymultinom()` pour le calcul d'une régression multinomiale avec des poids de réplication.

```
library(svrepmisc)
regm <- svymultinom(
  trav.satisf ~ sexe + etudes + trav.imp,
  design = dw_rep
)
```

{svrepmisc} fournit également des méthodes `svrepmisc::confint()` et `svrepmisc::tidy()`. Nous pouvons donc calculer et afficher les *odds ratio* et leur intervalle de confiance.

```
regm
```

	Coefficient	SE	t value
Satisfaction.(Intercept)	-0.116149	0.539876	-0.2151
Insatisfaction.(Intercept)	-1.547056	0.857410	-1.8043
Satisfaction.sexeHomme	-0.041405	0.186923	-0.2215
Insatisfaction.sexeHomme	0.221849	0.269817	0.8222
Satisfaction.etudesSecondaire	0.115722	0.344703	0.3357
Insatisfaction.etudesSecondaire	0.418476	0.554183	0.7551
Satisfaction.etudesTechnique / Professionnel	0.220702	0.300445	0.7346
Insatisfaction.etudesTechnique / Professionnel	0.529317	0.380713	1.3903
Satisfaction.etudesSupérieur	0.905852	0.328634	2.7564
Insatisfaction.etudesSupérieur	0.584499	0.454961	1.2847
Satisfaction.etudesNon documenté	-0.323293	0.669188	-0.4831
Insatisfaction.etudesNon documenté	0.646168	8.071336	0.0801
Satisfaction.trav.impAussi	-0.027506	0.708846	-0.0388
Insatisfaction.trav.impAussi	-0.375642	0.891773	-0.4212
Satisfaction.trav.impMoins	-0.220703	0.603956	-0.3654
Insatisfaction.trav.impMoins	-0.694337	0.924771	-0.7508
Satisfaction.trav.impPeu	-0.069034	0.733182	-0.0942
Insatisfaction.trav.impPeu	1.584747	0.973010	1.6287
	Pr(> t)		
Satisfaction.(Intercept)	0.83579		
Insatisfaction.(Intercept)	0.11416		
Satisfaction.sexeHomme	0.83102		
Insatisfaction.sexeHomme	0.43806		
Satisfaction.etudesSecondaire	0.74692		
Insatisfaction.etudesSecondaire	0.47481		
Satisfaction.etudesTechnique / Professionnel	0.48647		
Insatisfaction.etudesTechnique / Professionnel	0.20703		
Satisfaction.etudesSupérieur	0.02824 *		
Insatisfaction.etudesSupérieur	0.23977		
Satisfaction.etudesNon documenté	0.64376		

Insatisfaction.etudesNon documenté	0.93843
Satisfaction.trav.impAussi	0.97013
Insatisfaction.trav.impAussi	0.68622
Satisfaction.trav.impMoins	0.72558
Insatisfaction.trav.impMoins	0.47724
Satisfaction.trav.impPeu	0.92762
Insatisfaction.trav.impPeu	0.14740

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```
regm |> confint()
```

	2.5 %	97.5 %
Satisfaction.(Intercept)	-1.3927521	1.1604537
Insatisfaction.(Intercept)	-3.5745084	0.4803955
Satisfaction.sexeHomme	-0.4834084	0.4005975
Insatisfaction.sexeHomme	-0.4161679	0.8598657
Satisfaction.etudesSecondaire	-0.6993713	0.9308161
Insatisfaction.etudesSecondaire	-0.8919574	1.7289101
Satisfaction.etudesTechnique / Professionnel	-0.4897374	0.9311423
Insatisfaction.etudesTechnique / Professionnel	-0.3709257	1.4295606
Satisfaction.etudesSupérieur	0.1287558	1.6829483
Insatisfaction.etudesSupérieur	-0.4913121	1.6603108
Satisfaction.etudesNon documenté	-1.9056717	1.2590849
Insatisfaction.etudesNon documenté	-18.4395090	19.7318445
Satisfaction.trav.impAussi	-1.7036599	1.6486476
Insatisfaction.trav.impAussi	-2.4843488	1.7330652
Satisfaction.trav.impMoins	-1.6488321	1.2074252
Insatisfaction.trav.impMoins	-2.8810726	1.4923987
Satisfaction.trav.impPeu	-1.8027336	1.6646649
Insatisfaction.trav.impPeu	-0.7160553	3.8855497

```
regm |>
  broom::tidy(exponentiate = TRUE, conf.int = TRUE)
```

	term	estimate	std.error
1	Insatisfaction.(Intercept)	0.2128737	0.8574098
2	Insatisfaction.etudesNon documenté	1.9082140	8.0713359
3	Insatisfaction.etudesSecondaire	1.5196444	0.5541827
4	Insatisfaction.etudesSupérieur	1.7940926	0.4549608
5	Insatisfaction.etudesTechnique / Professionnel	1.6977731	0.3807130

```

6             Insatisfaction.sexeHomme 1.2483828 0.2698174
7             Insatisfaction.trav.impAussi 0.6868483 0.8917726
8             Insatisfaction.trav.impMoins 0.4994055 0.9247709
9             Insatisfaction.trav.impPeu 4.8780580 0.9730098
10            Satisfaction.(Intercept) 0.8903423 0.5398756
11            Satisfaction.etudesNon documenté 0.7237615 0.6691880
12            Satisfaction.etudesSecondaire 1.1226842 0.3447033
13            Satisfaction.etudesSupérieur 2.4740390 0.3286341
14            Satisfaction.etudesTechnique / Professionnel 1.2469523 0.3004451
15            Satisfaction.sexeHomme 0.9594401 0.1869231
16            Satisfaction.trav.impAussi 0.9728687 0.7088457
17            Satisfaction.trav.impMoins 0.8019545 0.6039558
18            Satisfaction.trav.impPeu 0.9332946 0.7331817
      statistic    p.value    conf.low    conf.high
1  -1.80433726 0.11415740 2.802920e-02 1.616714e+00
2   0.08005710 0.93843255 9.813479e-09 3.710489e+08
3   0.75512352 0.47480992 4.098527e-01 5.634510e+00
4   1.28472459 0.23976534 6.118231e-01 5.260946e+00
5   1.39033201 0.20703101 6.900952e-01 4.176863e+00
6   0.82221868 0.43805864 6.595695e-01 2.362843e+00
7  -0.42123052 0.68621929 8.337983e-02 5.657970e+00
8  -0.75082051 0.47723630 5.607458e-02 4.447752e+00
9   1.62870635 0.14739899 4.886761e-01 4.869370e+01
10 -0.21514076 0.83579140 2.483908e-01 3.191381e+00
11 -0.48311294 0.64375930 1.487227e-01 3.522197e+00
12  0.33571603 0.74692269 4.968976e-01 2.536578e+00
13  2.75641496 0.02824154 1.137412e+00 5.381399e+00
14  0.73458478 0.48646597 6.127873e-01 2.537406e+00
15 -0.22151043 0.83101822 6.166779e-01 1.492716e+00
16 -0.03880411 0.97013007 1.820161e-01 5.199943e+00
17 -0.36542976 0.72558301 1.922743e-01 3.344861e+00
18 -0.09415723 0.92762274 1.648476e-01 5.283902e+00

```

Par contre, le support de `gtsummary::tbl_regression()` et `ggstats::ggcoef_model()` est plus limité. Vous pourrez afficher un tableau basique des résultats et un graphiques des coefficients, mais sans les enrichissements usuels (identification des variables, étiquettes propres, identification des niveaux, etc.).

41.4.2 avec `svyVGAM::svy_glm()`

Une alternative possible pour le calcul de la régression logistique multinomiale avec des données pondérées est `svyVGAM::svy_glm()` avec `family = VGAM::multinomial`.

Nous allons commencer par définir le plan d'échantillonnage.

```
library(survey)
library(srvyr)
dw <- d |>
  as_survey(weights = poids)
```

Puis, on appelle `svyVGAM::svy_vglm()` en précisant `family = VGAM::multinomial`. Par défaut, `VGAM::multinomial()` utilise la dernière modalité de la variable d'intérêt comme modalité de référence. Cela est modifiable avec `refLevel`.

```
regm2 <- svyVGAM::svy_vglm(
  trav.satisf ~ sexe + etudes + trav.imp,
  family = VGAM::multinomial(refLevel = "Equilibre"),
  design = dw
)
regm2 |> summary()
```

```
svy_vglm.survey.design(trav.satisf ~ sexe + etudes + trav.imp,
  family = VGAM::multinomial(refLevel = "Equilibre"), design = dw)
Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
- ids: `1`
- weights: poids
Data variables: id (int), age (int), sexe (fct), nivetud (fct), poids (dbl),
  occup (fct), qualif (fct), freres.soeurs (int), clso (fct), relig (fct),
  trav.imp (fct), trav.satisf (fct), hard.rock (fct), lecture.bd (fct),
  peche.chasse (fct), cuisine (fct), bricol (fct), cinema (fct), sport (fct),
  heures.tv (dbl), groupe_ages (fct), etudes (fct)
```

	Coef	SE	z	p
(Intercept):1	-0.116117	0.553242	-0.2099	0.833757
(Intercept):2	-1.547693	0.876195	-1.7664	0.077332
sexeHomme:1	-0.041412	0.171351	-0.2417	0.809029
sexeHomme:2	0.221930	0.272669	0.8139	0.415693
etudesSecondaire:1	0.115688	0.341830	0.3384	0.735034
etudesSecondaire:2	0.418102	0.563205	0.7424	0.457868
etudesTechnique / Professionnel:1	0.220662	0.310123	0.7115	0.476754
etudesTechnique / Professionnel:2	0.529020	0.501080	1.0558	0.291079
etudesSupérieur:1	0.905798	0.314513	2.8800	0.003977
etudesSupérieur:2	0.584320	0.525633	1.1116	0.266289
etudesNon documenté:1	-0.323271	0.662511	-0.4879	0.625587

etudesNon documenté:2	0.646195	0.939745	0.6876	0.491687
trav.impAussi:1	-0.027517	0.511636	-0.0538	0.957109
trav.impAussi:2	-0.374881	0.825214	-0.4543	0.649625
trav.impMoins:1	-0.220706	0.494951	-0.4459	0.655659
trav.impMoins:2	-0.693571	0.792031	-0.8757	0.381200
trav.impPeu:1	-0.069004	0.706959	-0.0976	0.922244
trav.impPeu:2	1.585521	0.866529	1.8297	0.067289

Là encore, le support de `gtsummary::tbl_regression()` et `ggstats::ggcoef_model()` sera limité (seulement des résultats bruts). Pour calculer les *odds ratios* avec leurs intervalles de confiance, on pourra avoir recours à `broom.helpers::tidy_parameters()`.

```
regm2 |> broom.helpers::tidy_parameters(exponentiate = TRUE)
```

	term	estimate	std.error	conf.level	conf.low
1	(Intercept):1	0.8903708	0.4925908	0.95	0.30105802
2	(Intercept):2	0.2127382	0.1864002	0.95	0.03819678
3	sexeHomme:1	0.9594340	0.1643997	0.95	0.68574259
4	sexeHomme:2	1.2484837	0.3404228	0.95	0.73162172
5	etudesSecondaire:1	1.1226456	0.3837540	0.95	0.57448201
6	etudesSecondaire:2	1.5190753	0.8555505	0.95	0.50370762
7	etudesTechnique / Professionnel:1	1.2469025	0.3866926	0.95	0.67897791
8	etudesTechnique / Professionnel:2	1.6972676	0.8504673	0.95	0.63566751
9	etudesSupérieur:1	2.4739057	0.7780767	0.95	1.33557655
10	etudesSupérieur:2	1.7937708	0.9428657	0.95	0.64024629
11	etudesNon documenté:1	0.7237778	0.4795111	0.95	0.19754885
12	etudesNon documenté:2	1.9082659	1.7932841	0.95	0.30250055
13	trav.impAussi:1	0.9728582	0.4977496	0.95	0.35689786
14	trav.impAussi:2	0.6873710	0.5672279	0.95	0.13638549
15	trav.impMoins:1	0.8019528	0.3969271	0.95	0.30398071
16	trav.impMoins:2	0.4997879	0.3958473	0.95	0.10582985
17	trav.impPeu:1	0.9333229	0.6598208	0.95	0.23348961
18	trav.impPeu:2	4.8818349	4.2302506	0.95	0.89328990

	conf.high	statistic	df.error	p.value
1	2.633247	-0.20988509	Inf	0.833757356
2	1.184853	-1.76637893	Inf	0.077332298
3	1.342360	-0.24167835	Inf	0.809029407
4	2.130488	0.81391637	Inf	0.415692863
5	2.193860	0.33843739	Inf	0.735033605
6	4.581209	0.74236198	Inf	0.457868051
7	2.289862	0.71153296	Inf	0.476754032
8	4.531799	1.05575832	Inf	0.291078645

9	4.582448	2.87999791	Inf	0.003976778
10	5.025587	1.11164951	Inf	0.266288875
11	2.651771	-0.48794745	Inf	0.625587062
12	12.037925	0.68762767	Inf	0.491687277
13	2.651888	-0.05378223	Inf	0.957108667
14	3.464290	-0.45428375	Inf	0.649624613
15	2.115688	-0.44591425	Inf	0.655659186
16	2.360279	-0.87568771	Inf	0.381199829
17	3.730751	-0.09760692	Inf	0.922244427
18	26.679258	1.82973852	Inf	0.067289048

41.5 webin-R

La régression logistique multinomiale est abordée dans le webin-R #20 (*trajectoires de soins : un exemple de données longitudinales (4)*) sur [YouTube](https://www.youtube.com/watch?v=8l70djhwk2E).

<https://www.youtube.com/watch?v=8l70djhwk2E>

42 Régression logistique ordinale

La régression logistique ordinale est une extension de la régression logistique binaire (cf. Chapitre 22) aux variables qualitatives à trois modalités ou plus qui sont ordonnées, par exemple *modéré*, *moyen* et *fort*.

Pour une variable ordonnée, il est possible de réaliser une régression logistique multinomiale (cf. Chapitre 41) comme on le ferait avec une variable non ordonnée. Dans ce cas de figure, chaque modalité de la variable d'intérêt serait comparée à une modalité de référence.

Alternativement, on peut réaliser une **régression logistique ordinale** aussi appelée **modèle cumulatif** ou **modèle logistique à égalité des pentes**. Ce type de modèle est plus simple que la régression multinomiale car il ne renvoie qu'un seul jeu de coefficients.

Supposons une variable d'intérêt à trois modalités A, B et C telles que $A < B < C$. Les *odds ratios* qui seront calculés compareront la probabilité que $Y = B$ par rapport à la probabilité que $Y = A$ (aspect *cumulatif*) et ferons l'hypothèse que ce ratio est le même quand on compare la probabilité que $Y = C$ par rapport à la probabilité que $Y = B$ (*égalité des pentes*).

42.1 Données d'illustration

Pour illustrer la régression logistique multinomiale, nous allons reprendre le jeu de données `hdv2003` du package `{questionr}` et portant sur l'enquête *histoires de vie 2003* de l'Insee et l'exemple utilisé dans le chapitre sur la régression logistique multinomiale (cf. Chapitre 41).

```
library(tidyverse)
library(labelled)
data("hdv2003", package = "questionr")
d <- hdv2003
```

Nous allons considérer comme variable d'intérêt la variable `trav.satisf`, à savoir la satisfaction ou l'insatisfaction au travail.

```
questionr::freq(d$trav.satisf)
```

	n	%	val%
Satisfaction	480	24.0	45.8
Insatisfaction	117	5.9	11.2
Equilibre	451	22.6	43.0
NA	952	47.6	NA

Nous allons devoir ordonner les modalités de la plus faible à la plus forte.

```
d$trav.satisf <- d$trav.satisf |>
  fct_relevel("Insatisfaction", "Equilibre")
```

Et nous allons indiquer qu'il s'agit d'un facteur ordonné.

```
d$trav.satisf <- d$trav.satisf |>
  as.ordered()
```

Nous allons aussi en profiter pour raccourcir les étiquettes de la variable *trav.imp* :

```
levels(d$trav.imp) <- c("Le plus", "Aussi", "Moins", "Peu")
```

Enfin, procédons à quelques recodages additionnels :

```
d <- d |>
  mutate(
    sexe = sexe |> fct_relevel("Femme"),
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45 et plus")
      ),
    etudes = nivetud |>
      fct_recode(
        "Primaire" = "N'a jamais fait d'etudes",
        "Primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
        "Primaire" = "Derniere annee d'etudes primaires",
        "Secondaire" = "1er cycle",
        "Secondaire" = "2eme cycle",
        "Technique / Professionnel" = "Enseignement technique ou professionnel court",
```

```

    "Technique / Professionnel" = "Enseignement technique ou professionnel long",
    "Supérieur" = "Enseignement superieur y compris technique superieur"
  ) |>
  fct_na_value_to_level("Non documenté")
) |>
set_variable_labels(
  trav.satisf = "Satisfaction dans le travail",
  sexe = "Sexe",
  groupe_ages = "Groupe d'âges",
  etudes = "Niveau d'études",
  trav.imp = "Importance accordée au travail"
)

```

42.2 Calcul du modèle ordinal

42.2.1 avec MASS::polr()

Le plus facile pour le calcul d'un modèle ordinal est d'avoir recours à la fonction `MASS::polr()`.

```

rego <- MASS::polr(
  trav.satisf ~ sexe + etudes + groupe_ages + trav.imp,
  data = d
)

```

Nous pouvons aisément simplifier le modèle avec `step()` (cf. Chapitre 23).

```
rego2 <- rego |> step()
```

Start: AIC=1978.29

trav.satisf ~ sexe + etudes + groupe_ages + trav.imp

	Df	AIC
- groupe_ages	2	1977.2
- sexe	1	1978.3
<none>		1978.3
- etudes	4	1991.5
- trav.imp	3	2014.0

Step: AIC=1977.23

```
trav.satisf ~ sexe + etudes + trav.imp
```

	Df	AIC
- sexe	1	1977.0
<none>		1977.2
- etudes	4	1990.6
- trav.imp	3	2013.2

Step: AIC=1976.97

```
trav.satisf ~ etudes + trav.imp
```

	Df	AIC
<none>		1977.0
- etudes	4	1990.6
- trav.imp	3	2011.6

42.2.2 Fonctions alternatives

Un package alternatif pour le calcul de régressions ordinales est le package dédié `{ordinal}` et sa fonction `ordinal::clm()`. Pour les utilisateurs avancés, `{ordinal}` permet également de traiter certains prédicteurs comme ayant un effet nominal (et donc avec un coefficient par niveau) via l'argument `nominal`. Il existe également une fonction `ordinal::clmm()` permet de définir des modèles mixtes avec variables à effet aléatoire.

```
rego3 <- ordinal::clm(  
  trav.satisf ~ sexe + etudes + groupe_ages + trav.imp,  
  data = d  
)
```

Les modèles créés avec `ordinal::clm()` sont plutôt bien traités par des fonctions comme `gtsummary::tbl_regression()` ou `ggstats::ggcoef_model()`.

Enfin, on peut également citer la fonction `VGAM::vgam()` en spécifiant `family = VGAM::cumulative(parallel = TRUE)`. Cette famille de modèles offre des options avancées (par exemple il est possible de calculer des modèles non parallèles, c'est-à-dire avec une série de coefficients pour chaque changement de niveau). Par contre, le support de `gtsummary::tbl_regression()` et `ggstats::ggcoef_model()` sera limité (seulement des résultats bruts).

```
rego4 <- VGAM::vgam(  
  trav.satisf ~ sexe + etudes + groupe_ages + trav.imp,  
  data = d,
```

```
family = VGAM::cumulative(parallel = TRUE)
)
```

! Important

En toute rigueur, il faudrait **tester l'égalité des pentes**. Laurent Rouvière propose un code permettant d'effectuer ce test sous **R** à partir d'un modèle réalisé avec `VGAM::vglm()`, voir la diapositive 33 sur 35 d'un de ses cours de janvier 2015 intitulé *Quelques modèles logistiques polytomiques*.

42.3 Affichage des résultats du modèle

Pour un tableau des coefficients, on peut tout simplement appeler `gtsummary::tbl_regression()`. Nous allons indiquer `exponentiate = TRUE` car, comme pour la régression logistique binaire, l'exponentielle des coefficients peut s'interpréter comme des *odds ratios*. Pour éviter certains messages d'information, nous allons préciser `tidy_fun = broom.helpers::tidy_parameters` (cela implique juste que le tableau des coefficients sera calculé avec le package `{parameters}` plutôt qu'avec le package `{broom}`). Nous pouvons calculer à la volée la p-valeur globale de chaque variable avec `gtsummary::add_global_p()`.

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",")
```

```
rego2 |>
  tbl_regression(
    exponentiate = TRUE,
    tidy_fun = broom.helpers::tidy_parameters
  ) |>
  bold_labels() |>
  add_global_p(keep = TRUE)
```

Réajustement pour obtenir le Hessien

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
To suppress this message, include ``message = FALSE`` in code chunk header.

Table 42.1: Tableau des odds ratio de la régression logistique ordinale

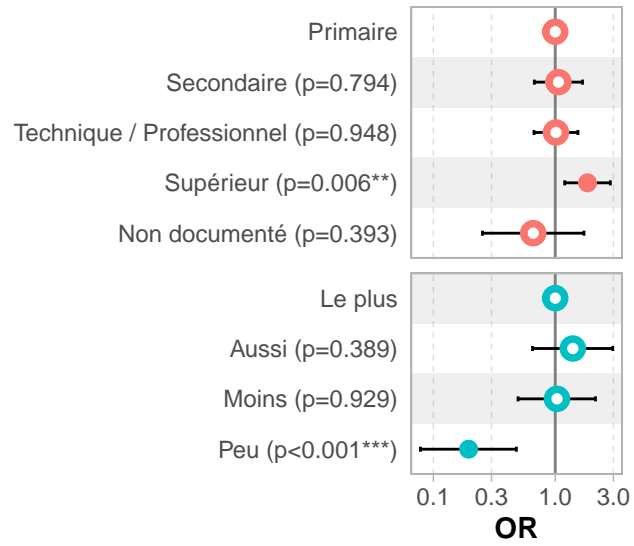
Caractéristique	OR	95% IC	p-valeur
Niveau d'études			<0,001
Primaire	—	—	
Secondaire	1,06	0,67 – 1,67	0,8
Technique / Professionnel	1,01	0,67 – 1,53	>0,9
Supérieur	1,84	1,20 – 2,83	0,006
Non documenté	0,66	0,25 – 1,72	0,4
Importance accordée au travail			<0,001
Le plus	—	—	
Aussi	1,39	0,65 – 2,97	0,4
Moins	1,03	0,50 – 2,14	>0,9
Peu	0,19	0,08 – 0,48	<0,001

La même commande fonctionne avec un modèle créé avec `ordinal::clm()`.

Pour un graphique des coefficients, on va procéder de même avec `ggstats::ggcoef_model()` ou `ggstats::ggcoef_table()`.

```
rego2 |>
  ggstats::ggcoef_model(
    exponentiate = TRUE,
    tidy_fun = broom.helpers::tidy_parameters
  )
```

Niveau d'études



Importance accordée au travail

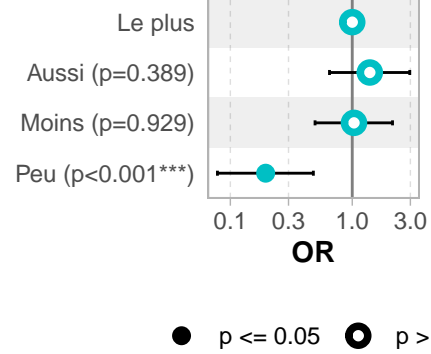


Figure 42.1: Graphique des coefficients du modèle ordinal

```
rego2 |>
  ggstats::ggcoef_table(
    exponentiate = TRUE,
    tidy_fun = broom.helpers::tidy_parameters
  )
```

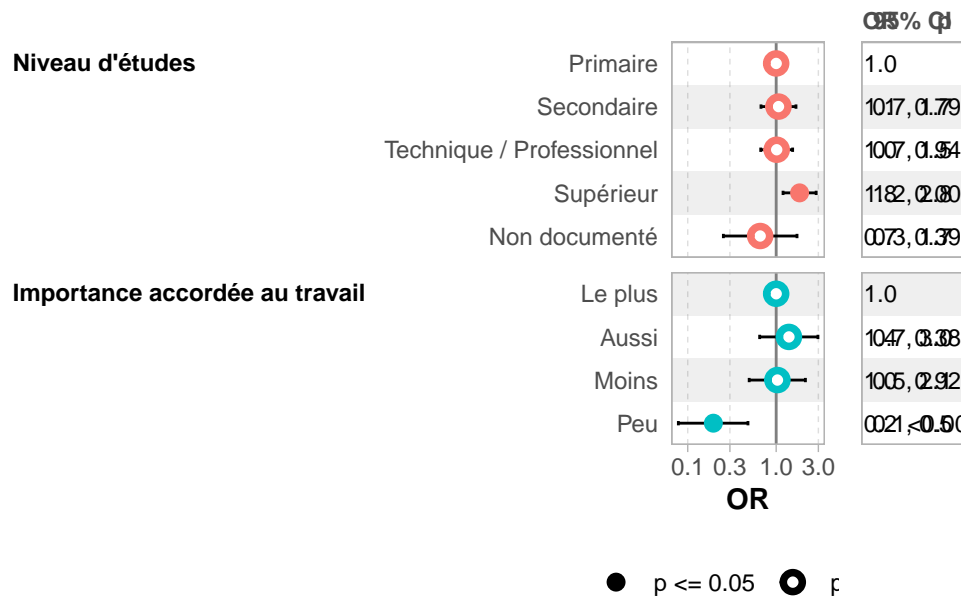



Figure 42.2: Graphique et table des coefficients du modèle ordinal

Pour faciliter l'interprétation, on pourra représenter les prédictions marginales du modèle (cf. Chapitre 24) avec `broom.helpers::plot_marginal_predictions()`.

```
rego2 |>
  broom.helpers::plot_marginal_predictions() |>
  patchwork::wrap_plots(ncol = 1) &
  scale_y_continuous(labels = scales::percent, limits = c(0, .6)) &
  coord_flip()
```

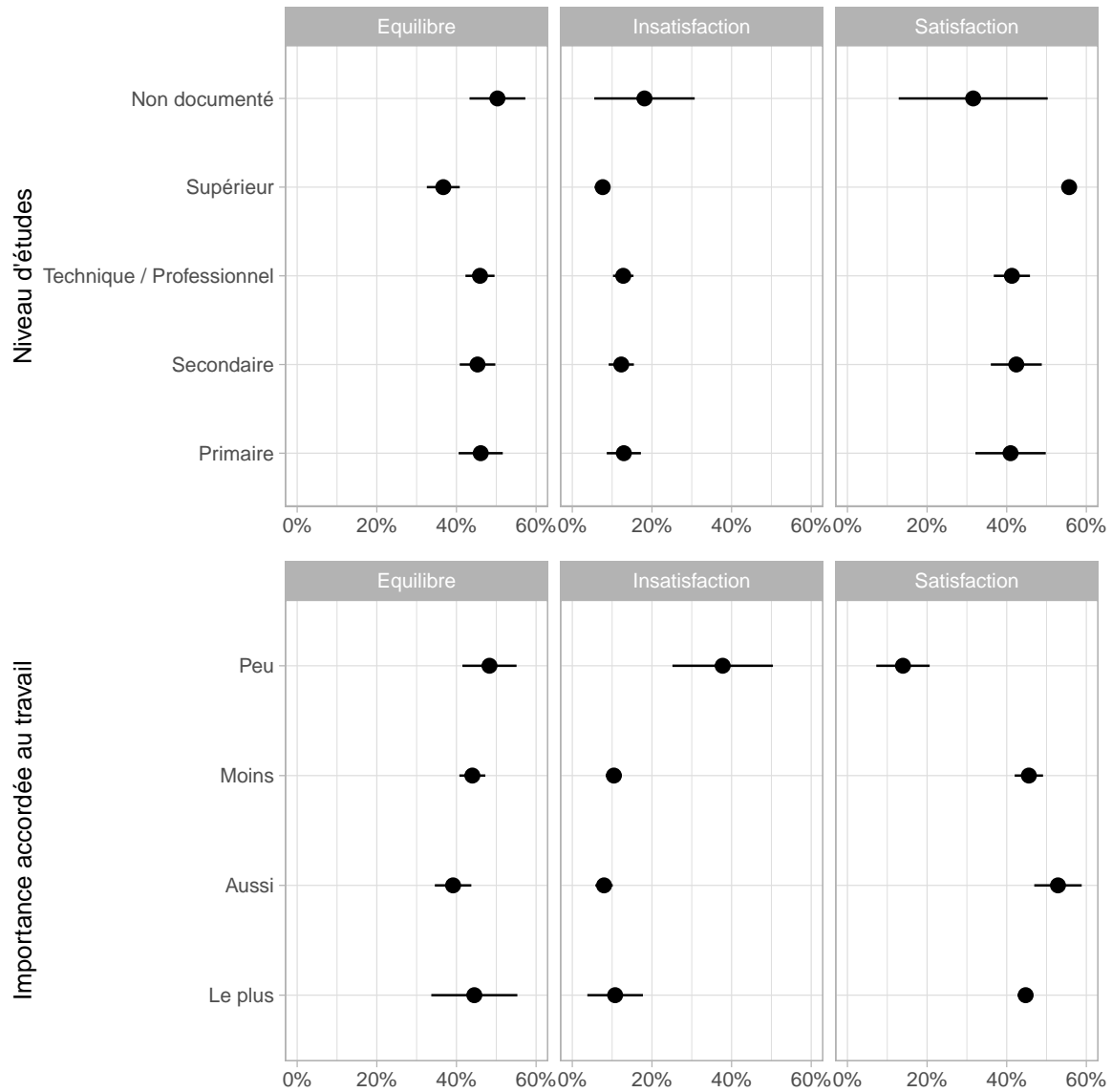


Figure 42.3: Prédictions marginales du modèle ordinal

42.4 Données pondérées

42.4.1 avec `survey::svyolr()`

L'extension `{survey}` (cf. Chapitre 28) propose une fonction native `survey::svyolr()` pour le calcul d'une régression ordinale.

Nous allons commencer par définir le plan d'échantillonnage.

```
library(survey)
library(srvyr)
dw <- d |>
  as_survey(weights = poids)
```

Calculons le modèle.

```
rego5 <- svyolr(
  trav.satisf ~ sexe + etudes + groupe_ages + trav.imp,
  design = dw
)
```

Le résultat peut être visualisé aisément avec `gtsummary::tbl_regression()` ou `ggstats::ggcoef_model()`.

42.4.2 avec `svrepmisc::svymultinom()`

Alternativement, il est possible d'utiliser `ordinal::clm()` en ayant recours à des poids de réplication, comme suggéré par Thomas Lumley dans son ouvrage *Complex Surveys: A Guide to Analysis Using R*.

L'extension `{svrepmisc}` disponible sur [GitHub](#) fournit quelques fonctions facilitant l'utilisation des poids de réplication avec `{survey}`. Pour l'installer, on utilisera le code ci-dessous :

```
remotes::install_github("carlganz/svrepmisc")
```

En premier lieu, il faut définir le design de notre tableau de données puis calculer des poids de réplication.

```
library(survey)
library(srvyr)
dw_rep <- d |>
  as_survey(weights = poids) |>
  as_survey_rep(type = "bootstrap", replicates = 25)
```

Il faut prévoir un nombre de `replicates` suffisant pour calculer ultérieurement les intervalles de confiance des coefficients. Plus ce nombre est élevé, plus précise sera l'estimation de la variance et donc des valeurs p et des intervalles de confiance. Cependant, plus ce nombre est élevé, plus le temps de calcul sera important. Pour gagner en temps de calcul, nous avons ici pris une valeur de 25, mais l'usage est de considérer au moins 1000 réplifications.

{svrepmisc} fournit une fonction `svrepmisc::svyclm()` pour le calcul d'une régression multinomiale avec des poids de réplification.

```
library(svrepmisc)
rego6 <- svyclm(
  trav.satisf ~ sexe + etudes + trav.imp,
  design = dw_rep
)
```

{svrepmisc} fournit également des méthodes `svrepmisc::confint()` et `svrepmisc::tidy()`. Nous pouvons donc calculer et afficher les *odds ratio* et leur intervalle de confiance.

```
rego6
```

	Coefficient	SE	t value	Pr(> t)	
Insatisfaction Equilibre	-2.03917466	0.43957112	-4.6390	0.0003212	***
Equilibre Satisfaction	0.27271800	0.47725404	0.5714	0.5761701	
sexeHomme	-0.13280477	0.12592647	-1.0546	0.3082902	
etudesSecondaire	-0.04271403	0.25651752	-0.1665	0.8699755	
etudesTechnique / Professionnel	0.00042809	0.24959262	0.0017	0.9986541	
etudesSupérieur	0.64247607	0.23150366	2.7752	0.0141490	*
etudesNon documenté	-0.46945975	0.55384978	-0.8476	0.4099657	
trav.impAussi	0.12071087	0.48530748	0.2487	0.8069425	
trav.impMoins	0.05267146	0.50124698	0.1051	0.9177039	
trav.impPeu	-1.53039056	0.52351035	-2.9233	0.0104865	*

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```
rego6 |> confint()
```

		2.5 %	97.5 %
Insatisfaction Equilibre	-2.9760983	-1.1022510	
Equilibre Satisfaction	-0.7445249	1.2899609	
sexeHomme	-0.4012107	0.1356011	
etudesSecondaire	-0.5894682	0.5040401	
etudesTechnique / Professionnel	-0.5315660	0.5324222	
etudesSupérieur	0.1490377	1.1359144	
etudesNon documenté	-1.6499626	0.7110431	
trav.impAussi	-0.9136975	1.1551193	
trav.impMoins	-1.0157112	1.1210541	
trav.impPeu	-2.6462265	-0.4145547	

```
rego6 |>
  broom::tidy(exponentiate = TRUE, conf.int = TRUE)
```

	term	estimate	std.error	statistic
1	Equilibre Satisfaction	1.3135298	0.4772540	0.571431525
2	etudesNon documenté	0.6253400	0.5538498	-0.847630122
3	etudesSecondaire	0.9581854	0.2565175	-0.166515071
4	etudesSupérieur	1.9011825	0.2315037	2.775230755
5	etudesTechnique / Professionnel	1.0004282	0.2495926	0.001715136
6	Insatisfaction Equilibre	0.1301361	0.4395711	-4.639009579
7	sexeHomme	0.8756360	0.1259265	-1.054621545
8	trav.impAussi	1.1282986	0.4853075	0.248730692
9	trav.impMoins	1.0540833	0.5012470	0.105080854
10	trav.impPeu	0.2164511	0.5235103	-2.923324367

	p.value	conf.low	conf.high
1	0.5761700555	0.47495990	3.6326446
2	0.4099656839	0.19205709	2.0361140
3	0.8699754583	0.55462216	1.6553958
4	0.0141489560	1.16071675	3.1140198
5	0.9986541229	0.58768394	1.7030524
6	0.0003211797	0.05099140	0.3321226
7	0.3082902108	0.66950899	1.1452250
8	0.8069424863	0.40103862	3.1744020
9	0.9177039342	0.36214479	3.0680866
10	0.0104864874	0.07091832	0.6606344

Par contre, le support de `gtsummary::tbl_regression()` et `ggstats::ggcoef_model()` est plus limité. Vous pourrez afficher un tableau basique des résultats et un graphique des coefficients, mais sans les enrichissements usuels (identification des variables, étiquettes propres, identification des niveaux, etc.).

42.4.3 avec `svyVGAM::svy_glm()`

Une alternative possible pour le calcul de la régression logistique multinomiale avec des données pondérées est `svyVGAM::svy_vglm()` avec `family = VGAM::multinomial`.

Nous allons commencer par définir le plan d'échantillonnage.

```
library(survey)
library(srvyr)
dw <- d |>
  as_survey(weights = poids)
```

Puis, on appelle `svyVGAM::svy_vglm()` en précisant `family = VGAM::cumulative(parallel = TRUE)`.

```
rego7 <- svyVGAM::svy_vglm(
  trav.satisf ~ sexe + etudes + trav.imp,
  family = VGAM::cumulative(parallel = TRUE),
  design = dw
)
rego7 |> summary()
```

```
svy_vglm.survey.design(trav.satisf ~ sexe + etudes + trav.imp,
  family = VGAM::cumulative(parallel = TRUE), design = dw)
Independent Sampling design (with replacement)
Called via srvyr
Sampling variables:
- ids: `1`
- weights: poids
Data variables: id (int), age (int), sexe (fct), nivetud (fct), poids (dbl),
  occup (fct), qualif (fct), freres.soeurs (int), clso (fct), relig (fct),
  trav.imp (fct), trav.satisf (ord), hard.rock (fct), lecture.bd (fct),
  peche.chasse (fct), cuisine (fct), bricol (fct), cinema (fct), sport (fct),
  heures.tv (dbl), groupe_ages (fct), etudes (fct)
```

	Coef	SE	z	p
(Intercept):1	-2.03917232	0.52467564	-3.8865	0.0001017

(Intercept):2	0.27272042	0.51315926	0.5315	0.5951044
sexeHomme	0.13280416	0.15421876	0.8611	0.3891602
etudesSecondaire	0.04271296	0.28498254	0.1499	0.8808599
etudesTechnique / Professionnel	-0.00043004	0.25720348	-0.0017	0.9986659
etudesSupérieur	-0.64247750	0.26590126	-2.4162	0.0156823
etudesNon documenté	0.46945812	0.52541495	0.8935	0.3715896
trav.impAussi	-0.12071140	0.48525530	-0.2488	0.8035476
trav.impMoins	-0.05267211	0.47029563	-0.1120	0.9108251
trav.impPeu	1.53036737	0.65156164	2.3488	0.0188356

Là encore, le support de `gtsummary::tbl_regression()` et `ggstats::ggcoef_model()` sera limité (seulement des résultats bruts). Pour calculer les *odds ratios* avec leurs intervalles de confiance, on pourra avoir recours à `broom.helpers::tidy_parameters()`.

```
rego7 |> broom.helpers::tidy_parameters(exponentiate = TRUE)
```

	term	estimate	std.error	conf.level	conf.low
1	(Intercept):1	0.1301364	0.06827939	0.95	0.04653653
2	(Intercept):2	1.3135330	0.67405160	0.95	0.48043985
3	sexeHomme	1.1420263	0.17612188	0.95	0.84412131
4	etudesSecondaire	1.0436383	0.29741869	0.95	0.59699738
5	etudesTechnique / Professionnel	0.9995700	0.25709290	0.95	0.60378349
6	etudesSupérieur	0.5259877	0.13986079	0.95	0.31234891
7	etudesNon documenté	1.5991274	0.84020546	0.95	0.57101702
8	trav.impAussi	0.8862897	0.43007677	0.95	0.34239361
9	trav.impMoins	0.9486910	0.44616524	0.95	0.37740568
10	trav.impPeu	4.6198737	3.01013253	0.95	1.28830837

	conf.high	statistic	df.error	p.value
1	0.3639179	-3.886538948	Inf	0.0001016836
2	3.5912276	0.531453766	Inf	0.5951043720
3	1.5450672	0.861141383	Inf	0.3891601794
4	1.8244316	0.149879202	Inf	0.8808599211
5	1.6547990	-0.001671995	Inf	0.9986659418
6	0.8857499	-2.416225817	Inf	0.0156823321
7	4.4783403	0.893499744	Inf	0.3715896094
8	2.2941709	-0.248758549	Inf	0.8035475563
9	2.3847407	-0.111997880	Inf	0.9108250872
10	16.5668669	2.348768359	Inf	0.0188356206

43 Modèles de comptage (Poisson & apparentés)

Une variable de type comptage correspond à un *outcome* (variable à expliquer) entier positif. Le plus souvent, il s'agit du nombre d'occurrences de l'évènement d'intérêt. Pour expliquer une variable de ce type, les modèles linéaires ou les régressions logistiques ne sont pas adaptées. On aura alors recours à des modèles ou régressions de **Poisson**, c'est-à-dire de manière plus spécifique à un modèle linéaire généralisé (GLM en anglais), avec une fonction de lien logarithmique et une distribution statistique de **Poisson**.

Les modèles de **Poisson** font l'hypothèse que la variance est égale à la moyenne, ce qui ne s'observe pas toujours. Auquel cas on aura alors un problème de **surdispersion**. Dans ce cas-là, nous pourrions avoir recours à des modèles apparentés aux modèles de Poisson, à savoir les modèles **quasi-Poisson**, les modèles **binomiaux négatifs**.

Les modèles de comptage peuvent aussi être utilisés pour un *outcome* binaire, en lieu et place d'une régression logistique, afin d'estimer des *prevalence ratios* plutôt que des *odds ratios*.

Enfin, lorsqu'il y a un nombre important de 0 dans les données à expliquer, on peut avoir recours à des modèles *zero-inflated* ou des modèles *hurdle* qui, d'une certaine manière, combinent deux modèles en un (d'une part la probabilité de vivre l'évènement et d'autre part le nombre d'occurrences de l'évènement).

43.1 Modèle de Poisson

Nous allons nous intéresser à un premier exemple démographique en nous intéressant à la descendance atteinte par des femmes à l'âge de 30 ans.

43.1.1 Préparation des données du premier exemple

Pour cet exemple, nous allons considérer le jeu de données **fecondite** fourni par le package `{questionr}`. Ce jeu de données comprends trois tables de données (**menages**, **femmes** et **enfants**) simulant les résultats d'une enquête démographique et de santé (EDS). Chargeons les données et jetons y un œil avec `labelled::look_for()`.


```
library(tidyverse)
library(labelled)
data("fecondite", package = "questionr")
```

```
enfants |> look_for()
```

pos	variable	label	col_type	missing	values
1	id_enfant	Identifiant de l'enfant	dbl	0	
2	id_femme	Identifiant de la mère	dbl	0	
3	date_naissance	Date de naissance	date	0	
4	sexe	Sexe de l'enfant	dbl+lbl	0	[1] masculin [2] féminin
5	survie	L'enfant est-il toujours en~	dbl+lbl	0	[0] non [1] oui
6	age_deces	Age au décès (en mois)	dbl	1442	

Comme nous pouvons le voir, les variables catégorielles sont codées sous la forme de vecteurs numériques labellisés (cf. Chapitre 12), comme cela aurait été le cas si nous avions importé un fichier Stata ou SPSS avec `{haven}`. Première étape, nous allons convertir à la volée ces variables catégorielles en facteurs avec `labelled::unlabelled()`.

```
femmes <-
  femmes |>
  unlabelled()
enfants <-
  enfants |>
  unlabelled()
```

Pour notre analyse, nous allons devoir compter le nombre d'enfants que chaque femme a eu avant l'âge de 30 ans exacts. En premier lieu, nous devons calculer l'âge (cf. Section 34.4) de la mère à la naissance dans le tableau `enfants`, à l'aide de la fonction `lubridate::time_length()`.

```
enfants <-
  enfants |>
  left_join(
    femmes |>
      select(id_femme, date_naissance_mere = date_naissance),
    by = "id_femme"
  ) |>
  mutate(
```

```

    age_mere = time_length(
      date_naissance_mere %--% date_naissance,
      unit = "years"
    )
  )
)

```

Comptons maintenant, par femme, le nombre d'enfants nés avant l'âge de 30 ans et ajoutons cette valeur à la table `femmes`. N'oublions, après la fusion, de recoder les valeurs manquantes NA en 0 avec `tidyr::replace_na()`.

```

femmes <-
  femmes |>
  left_join(
    enfants |>
      filter(age_mere < 30) |>
      group_by(id_femme) |>
      count(name = "enfants_avt_30"),
    by = "id_femme"
  ) |>
  tidyr::replace_na(list(enfants_avt_30 = 0L))

```

Il nous reste à calculer l'âge des femmes au moment de l'enquête. Nous allons en profiter pour recoder (cf. Section 9.3) la variable éducation en regroupant les modalités secondaire et supérieur.

```

femmes <-
  femmes |>
  mutate(
    age = time_length(
      date_naissance %--% date_entretien,
      unit = "years"
    ),
    educ2 = educ |>
      fct_recode(
        "secondaire/supérieur" = "secondaire",
        "secondaire/supérieur" = "supérieur"
      )
  )
)

```

Enfin, pour l'analyse, nous n'allons garder que les femmes âgées d'au moins 30 ans au moment de l'enquête. En effet, les femmes plus jeunes n'ayant pas encore atteint 30 ans, nous ne connaissons pas leur descendance atteinte à cet âge.

```
femmes30p <-
  femmes |>
  filter(age >= 30)
```

43.1.2 Calcul & Interprétation du modèle de Poisson

Les modèles de Poisson font partie de la famille des modèles linéaires généralisés (*GLM* en anglais) et se calculent donc avec la fonction `stats::glm()` en précisant `family = poisson`.

```
mod1_poisson <- glm(
  enfants_avt_30 ~ educ2 + milieu + region,
  family = poisson,
  data = femmes30p
)
```

L'ensemble des fonctions et méthodes applicables aux modèles GLM, que nous avons déjà abordé dans le chapitre sur la régression logistique (cf. Chapitre 22), peuvent s'appliquer également aux modèles de Poisson. Nous pouvons par exemple réduire notre modèle par minimisation de l'AIC avec `stats::step()`.

```
mod1_poisson <- step(mod1_poisson)
```

```
Start:  AIC=1013.81
enfants_avt_30 ~ educ2 + milieu + region
```

		Df	Deviance	AIC
-	region	3	686.46	1010.6
	<none>		683.62	1013.8
-	milieu	1	686.84	1015.0
-	educ2	2	691.10	1017.3

```
Step:  AIC=1010.65
enfants_avt_30 ~ educ2 + milieu
```

		Df	Deviance	AIC
	<none>		686.46	1010.6
-	milieu	1	691.30	1013.5
-	educ2	2	693.94	1014.1

Par défaut, les modèles de Poisson utilisent une fonction de lien logarithmique (*log*). Dès lors, il est fréquent de ne pas présenter directement les coefficients du modèle, mais plutôt l'exponentielle de leurs valeurs qui peut s'interpréter comme des risques relatifs (*relative risks* ou RR)¹. L'exponentielle des coefficients peut aussi être appelée *incidence rate ratio* (IRR) car la régression de Poisson peut également être utilisée pour des modèles d'incidence, qui seront abordés dans le prochain chapitre (cf. Chapitre 44).

Pour un tableau mis en forme des coefficients, on aura tout simplement recours à `{gtsummary}` et sa fonction `gtsummary::tbl_regression()`.

```
library(gtsummary)
theme_gtsummary_language("fr", decimal.mark = ",", big.mark = " ")
```

Setting theme `language: fr`

```
mod1_poisson |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels()
```

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
To suppress this message, include ``message = FALSE`` in code chunk header.

Table 43.1: Tableau des coefficients du modèle de Poisson

Caractéristique	IRR	95% IC	p-valeur
Niveau d'éducation			
aucun	—	—	
primaire	1,25	0,90 – 1,72	0,2
secondaire/supérieur	0,53	0,27 – 0,96	0,052
Milieu de résidence			
urbain	—	—	
rural	1,42	1,04 – 1,98	0,032

Pour une représentation graphique, on pourra avoir recours à `ggstats::ggcoef_model()` ou `ggstats::ggcoef_table()`.

¹À ne pas confondre avec les *odds ratio* ou OR de la régression logistique.

```
library(ggstats)
mod1_poisson |>
  ggcoef_table(exponentiate = TRUE)
```

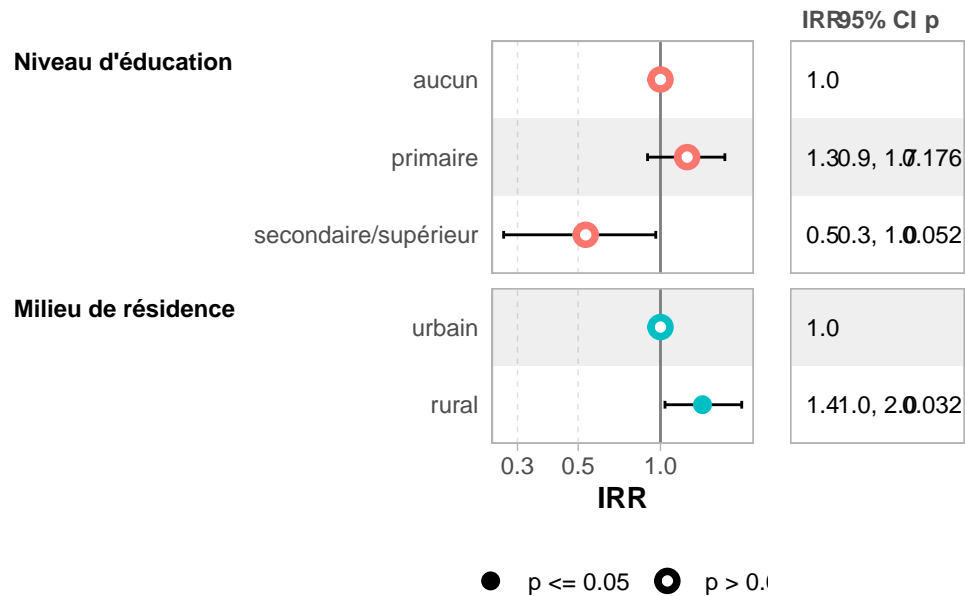


Figure 43.1: Forest plot des coefficients du modèle de Poisson

Pour interpréter les coefficients, il faut se rappeler que ce qui est modéliser dans le modèle de Poisson est le nombre moyen d'événements. Le RR pour la modalité secondaire/supérieur est de 0,5 : cela signifie donc que, indépendamment des autres variables du modèle, la descendance atteinte à 30 ans moyenne des femmes de niveau secondaire ou supérieure est moitié moindre que cela des femmes sans niveau d'éducation (modalité de référence).

Nous pouvons vérifier visuellement ce résultat en réalisant un graphique des prédictions marginales moyennes avec `broom.helpers::plot_marginal_predictions()` (cf. Section 24.3).

```
broom.helpers::plot_marginal_predictions(mod1_poisson) |>
  patchwork::wrap_plots() &
  ggplot2::scale_y_continuous(limits = c(0, .4))
```

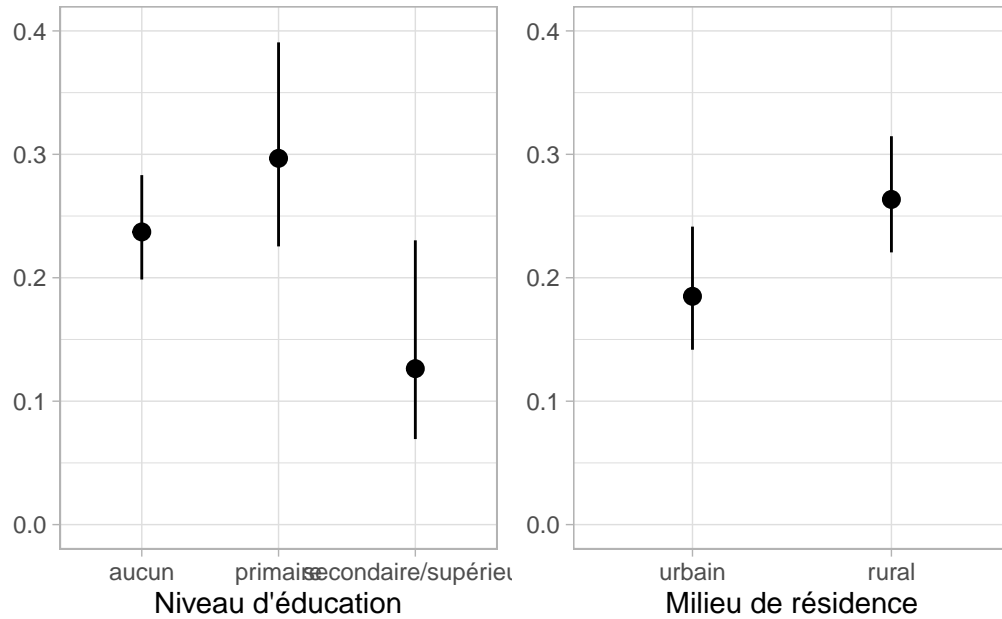


Figure 43.2: Prédictions marginales du modèle de Poisson

43.1.3 Évaluation de la surdispersion

Comme tous les modèles GLM, le modèle de Poisson présuppose que les observations sont indépendantes les unes des autres². Surtout, la distribution de Poisson présuppose que la variance est égale à la moyenne. Or, il arrive fréquemment que la variance soit supérieure à la moyenne, auquel cas on parle alors de **surdispersion**. Si c'est le cas, le modèle de Poisson va sous-estimer la variance et produire des intervalles de confiance trop petit et des p-valeurs trop petites.

En premier lieu, nous pouvons vérifier si la distribution observée des données correspond peu ou prou à la distribution théorique de Poisson pour une moyenne correspondant à la moyenne observée.

La fonction ci-dessous permet justement de comparer la distribution observée avec la distribution théorique d'un modèle.

```
observed_vs_theoretical <- function(model) {
  observed <- model.response(model.frame(model))
```

²Si ce n'était pas le cas, par exemple s'il y a plusieurs observations pour un même individu, il faudrait adopter un autre type de modèle, par exemple un modèle mixte ou un modèle GEE, pour lesquels il est possible d'utiliser une distribution de Poisson.

```

theoretical <- simulate(model, nsim = 1)
theoretical <- theoretical[[1]]
df <- dplyr::tibble(
  status = c(
    rep.int("observed", length(observed)),
    rep.int("theoretical", length(theoretical))
  ),
  values = c(observed, theoretical)
)
if (is.numeric(observed) && any(observed != as.integer(observed))) {
  ggplot2::ggplot(df) +
  ggplot2::aes(x = values, fill = status) +
  ggplot2::geom_density(
    alpha = .5,
    position = "identity"
  ) +
  ggplot2::theme_light() +
  ggplot2::labs(fill = NULL)
} else {
  ggplot2::ggplot(df) +
  ggplot2::aes(x = values, fill = status) +
  ggplot2::geom_bar(
    alpha = .5,
    position = "identity"
  ) +
  ggplot2::theme_light() +
  ggplot2::theme(
    panel.grid.major.x = ggplot2::element_blank(),
    panel.grid.minor.x = ggplot2::element_blank()
  ) +
  ggplot2::labs(fill = NULL)
}
}

```

Appliquons cette fonction à notre modèle de Poisson.

```

mod1_poisson |>
  observed_vs_theoretical()

```

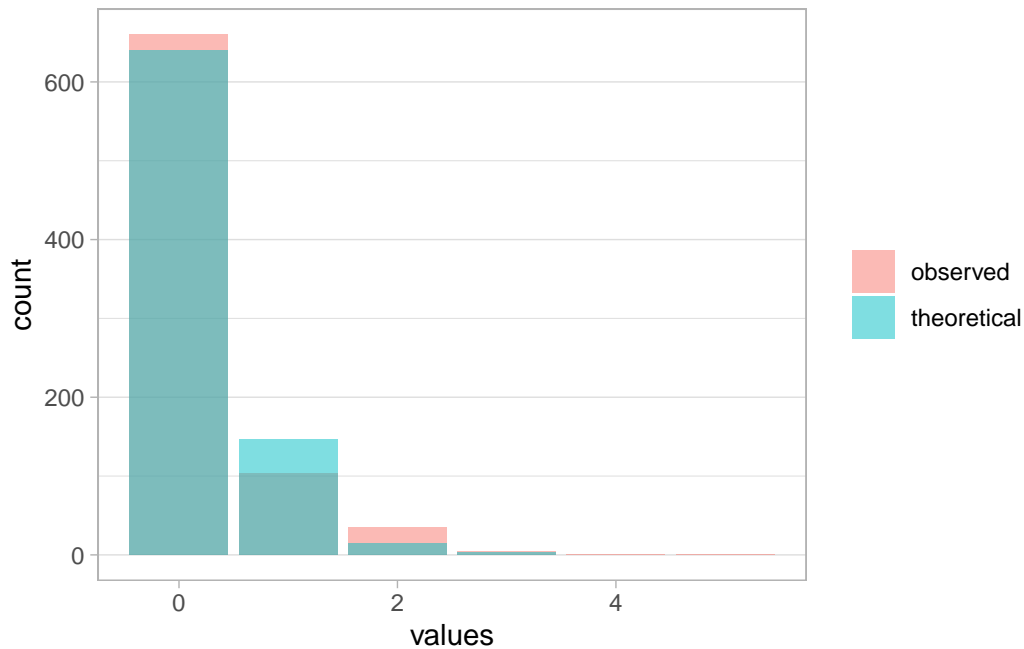


Figure 43.3: Distribution observée vs distribution théorique de la descendance atteinte à 30 ans (modèle de Poisson)

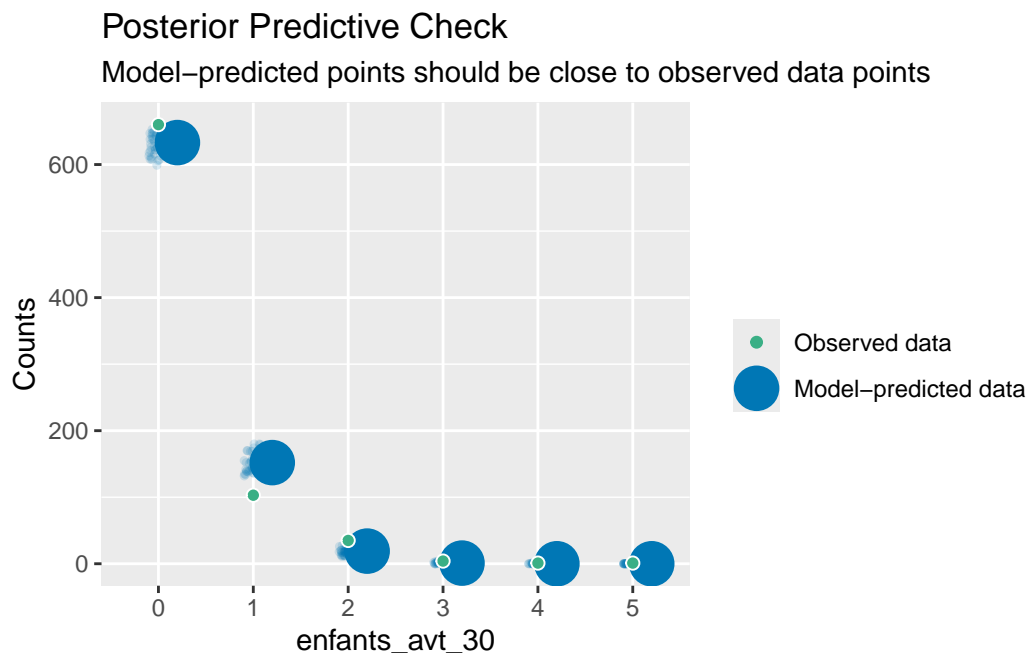
Les deux distributions restent assez proches même si la distribution observée est légèrement décalée vers la droite.

i Note

La fonction `performance::check_predictions()` propose une visualisation un peu plus avancée.

```
mod1_poisson |>
  performance::check_predictions(type = "discrete_both")
```

Warning: Maximum value of original data is not included in the replicated data.
Model may not capture the variation of the data.



Les points verts correspondent au nombre d'individus observés (sur l'axe vertical) pour chaque valeur de notre variable à expliquer (sur l'axe horizontal). La fonction simule ensuite une cinquantaine de jeu de données par réplication et affiche les prédictions du modèle pour ces jeux de données (points bleus avec effet de transparence), ainsi que les prédictions médianes (point bleu sans transparence) et l'intervalle des prédictions (barres verticales).

Dans notre exemple, on voit notamment que le modèle a du mal à prédire correctement le nombre d'individus avec 0, 1 ou 2 enfants.

Le paramètre ϕ qui correspond au ratio entre la variance observée et la variance théorique peut être estimée, en pratique, selon certains auteurs, comme le ratio de la déviance résiduelle sur le nombre de degrés de libertés du modèle. Il s'obtient ainsi :

```
mod1_poisson$deviance / mod1_poisson$df.residual
```

```
[1] 0.8580717
```

Si ce ratio s'écarte de 1, alors il y a un problème de surdispersion. Cependant, en pratique, il n'y a pas de seuil précis à partir duquel nous pourrions conclure qu'il faut rejeter le modèle.

La package `{AER}` propose un test, `AER::dispersiontest()`, pour tester s'il y a un problème de surdispersion. Ce test ne peut s'appliquer qu'à un modèle de Poisson.

```
mod1_poisson |> AER::dispersiontest()
```

Overdispersion test

```
data:  mod1_poisson
z = 3.3367, p-value = 0.0004238
alternative hypothesis: true dispersion is greater than 1
sample estimates:
dispersion
  1.361364
```

Le package `{performance}` propose, quant à lui, un test plus générale de surdispersion via la fonction `performance::check_overdispersion()`.

```
mod1_poisson |>
performance::check_overdispersion()
```

```
# Overdispersion test
```

```
dispersion ratio =    1.371
Pearson's Chi-Squared = 1096.575
p-value =    < 0.001
```

Overdispersion detected.

Dans les deux cas, nous obtenons une p-valeur inférieure à 0,001, indiquant que le modèle de Poisson n'est peut-être pas approprié ici.

43.2 Modèle de quasi-Poisson

Le modèle de **quasi-Poisson** est similaire au modèle de **Poisson** mais autorise plus de souplesse pour la modélisation de la variance qui est alors modélisée comme une relation linéaire de la moyenne. Il se calcule également avec `stats::glm()`, mais en indiquant `family = quasipoisson`. Comme avec le modèle de Poisson, la fonction de lien par défaut est la fonction logarithmique (*log*).

```
mod1_quasi <- glm(
  enfants_avt_30 ~ educ2 + milieu,
  family = quasipoisson,
  data = femmes30p
)
```

! Important

L'AIC (*Akaike information criterion*) n'est pas défini pour ce type de modèle. Il n'est donc pas possible d'utiliser `stats::step()` avec un modèle de quasi-Poisson. Si l'on veut procéder à une sélection pas à pas descendante, on procédera donc en amont avec un modèle de Poisson classique.

Regardons les résultats obtenus :

```
mod1_quasi |>
  tbl_regression(exponentiate = TRUE) |>
  bold_labels()
```

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>. To suppress this message, include ``message = FALSE`` in code chunk header.

Table 43.2: Tableau des coefficients du modèle de quasi-Poisson

Caractéristique	IRR	95% IC	p-valeur
Niveau d'éducation			
aucun	—	—	
primaire	1,25	0,85 – 1,81	0,2
secondaire/supérieur	0,53	0,23 – 1,05	0,10
Milieu de résidence			
urbain	—	—	
rural	1,42	0,99 – 2,10	0,067

Les coefficients sont identiques à ceux obtenus avec le modèle de Poisson (cf. Table 43.1), mais les intervalles de confiance sont plus larges et les p-valeurs plus élevées, traduisant la prise en compte d'une variance plus importante. Cela se voit aisément si l'on compare les coefficients avec `ggstats::ggcoef_compare()`.

```
list(Poisson = mod1_poisson, "quasi-Poisson" = mod1_quasi) |>
  ggcoef_compare(exponentiate = TRUE)
```

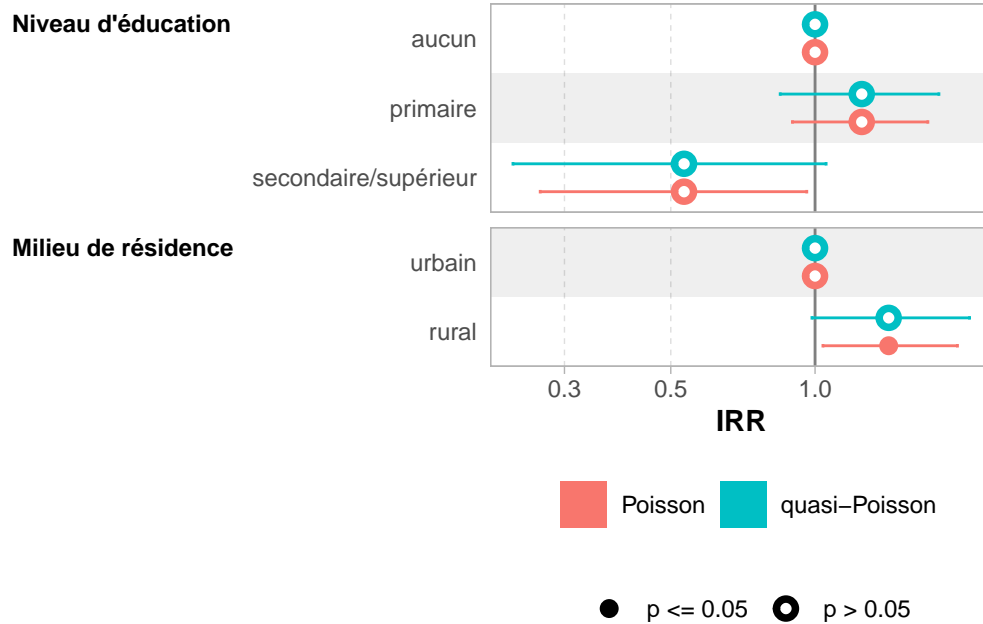


Figure 43.4: Comparaison des coefficients du modèle de Poisson et du modèle de quasi-Poisson

Le passage à un modèle de quasi-Poisson aura-t-il été suffisant pour régler notre problème de surdispersion ? La fonction `performance::check_overdispersion()` peut être appliquée à un modèle de quasi-Poisson.

```
mod1_quasi |>
  performance::check_overdispersion()
```

```
# Overdispersion test
```

```
dispersion ratio = 1.371
Pearson's Chi-Squared = 1096.575
p-value = < 0.001
```

```
Overdispersion detected.
```

Il semble que nous ayons toujours une surdispersion, insuffisamment corrigée par le modèle de quasi-Poisson.

43.3 Modèle binomial négatif

Le modèle binomial négatif (*negative binomial* en anglais) modélise la variance selon une spécification quadratique (i.e. selon le carré de la moyenne). Il est implémenté dans le package {MASS} via la fonction `MASS::glm.nb()`. Les autres paramètres sont identiques à ceux de `stats::glm()`.

```
mod1_nb <- MASS::glm.nb(  
  enfants_avt_30 ~ educ2 + milieu + region,  
  data = femmes30p  
)
```

L'AIC étant défini pour ce type de modèle, nous pouvons procéder à une sélection pas à pas avec `stats::step()`.

```
mod1_nb <- mod1_nb |> step()
```

```
Start:  AIC=979.1  
enfants_avt_30 ~ educ2 + milieu + region
```

	Df	Deviance	AIC
- region	3	462.89	975.01
<none>		460.98	979.10
- milieu	1	463.29	979.41
- educ2	2	466.11	980.22

```
Step:  AIC=975  
enfants_avt_30 ~ educ2 + milieu
```

	Df	Deviance	AIC
<none>		460.14	975.00
- milieu	1	463.37	976.24
- educ2	2	465.54	976.40

La fonction de lien étant toujours logarithmique, nous pouvons donc afficher plutôt l'exponentielle des coefficients qui s'interprètent comme pour un modèle de Poisson.

```
mod1_nb |>  
  tbl_regression(exponentiate = TRUE) |>  
  bold_labels()
```

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
 To suppress this message, include ``message = FALSE`` in code chunk header.

Table 43.3: Tableau des coefficients du modèle binomial négatif

Caractéristique	IRR	95% IC	p-valeur
Niveau d'éducation			
aucun	—	—	
primaire	1,23	0,82 – 1,82	0,3
secondaire/supérieur	0,53	0,25 – 1,03	0,074
Milieu de résidence			
urbain	—	—	
rural	1,41	0,97 – 2,06	0,076

Cette fois-ci, les coefficients sont légèrement différents par rapport au modèle de Poisson, ce qui se voit aisément si l'on compare les trois modèles.

```
list(
  Poisson = mod1_poisson,
  "quasi-Poisson" = mod1_quasi,
  "Binomial négatif" = mod1_nb
) |>
ggcoef_compare(exponentiate = TRUE)
```

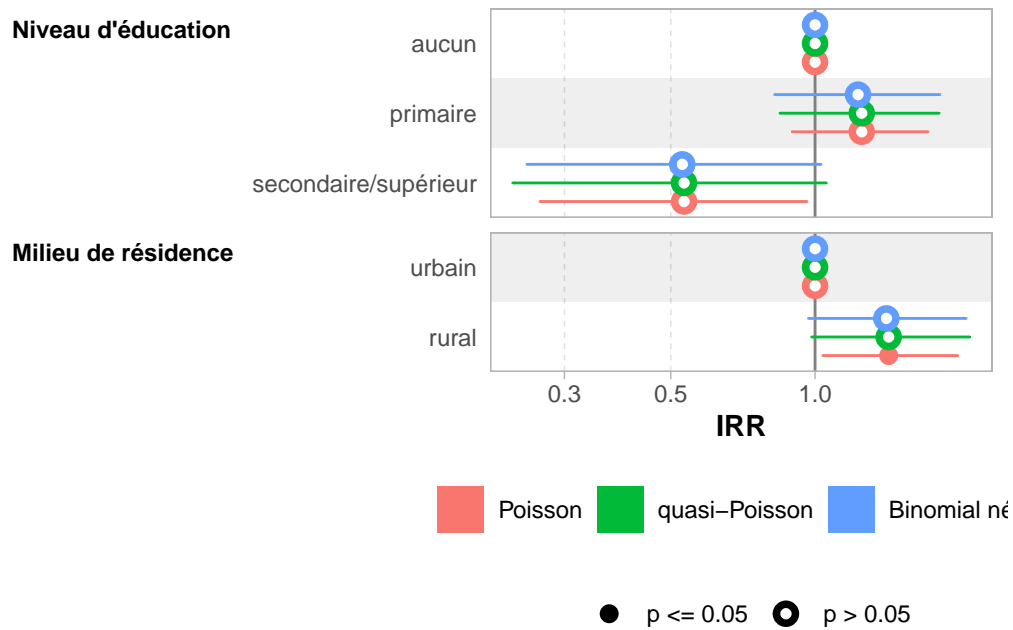


Figure 43.5: Comparaison des coefficients du modèle de Poisson, du modèle de quasi-Poisson et du modèle binomial négatif

Nous pouvons comparer la distribution observée et la distribution théorique et constater que les prédictions sont plus proches des données observées.

```
mod1_nb |>
  observed_vs_theoretical()
mod1_nb |>
  performance::check_predictions(type = "discrete_both")
```

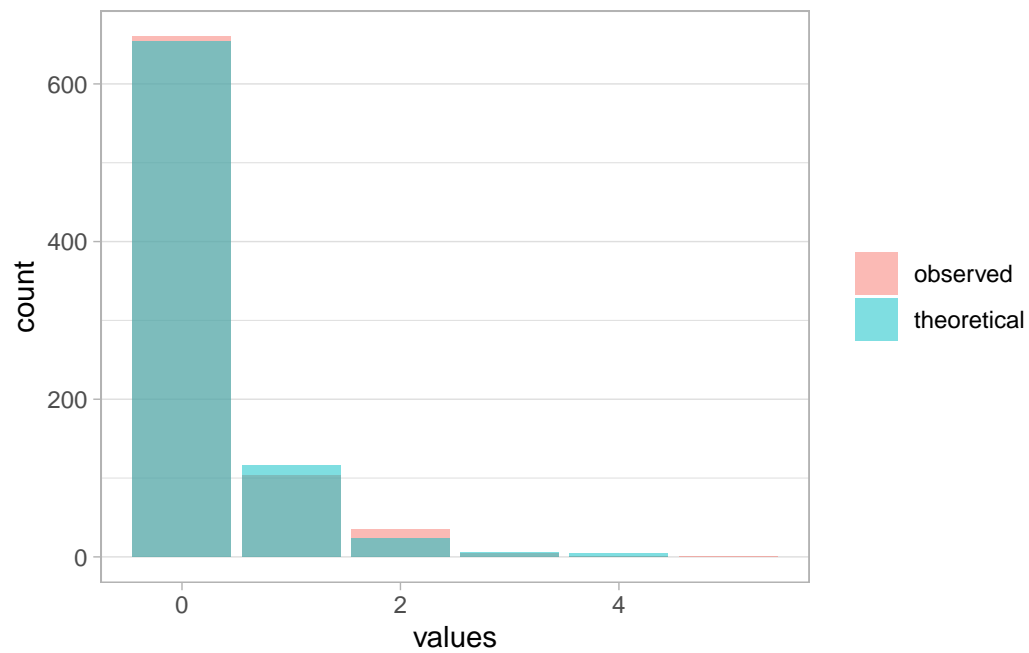


Figure 43.6: Distribution observée vs distribution théorique de la descendance atteinte à 30 ans (modèle négatif binomial)

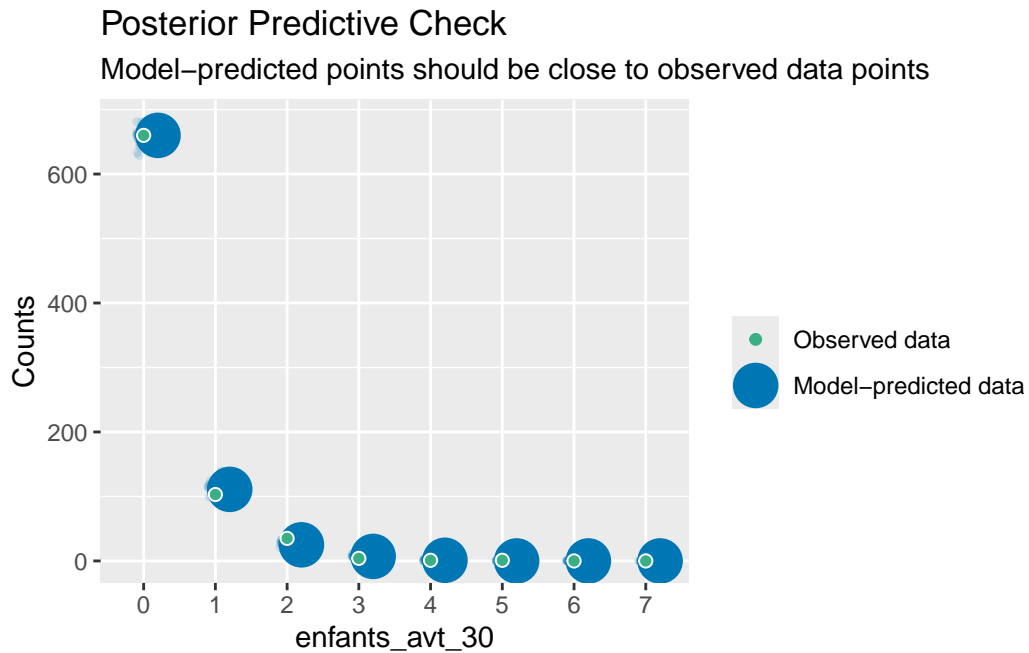


Figure 43.7: Distribution observée vs distribution théorique de la descendance atteinte à 30 ans (modèle négatif binomial)

Nous pouvons vérifier la surdispersion.

```
mod1_nb |>
  performance::check_overdispersion()
```

Overdispersion test

```
dispersion ratio = 0.975
p-value = 0.856
```

No overdispersion detected.

Ouf ! Nous n'avons plus de problème de surdispersion.

Pour celles et ceux intéressé·es, il est possible de comparer la performance des modèles avec la fonction `performance::compare_performance()`.

```
performance::compare_performance(
  mod1_poisson,
  mod1_nb,
  metrics = "common"
)
```

Comparison of Model Performance Indices

Name	Model	AIC (weights)	BIC (weights)	Nagelkerke's R2	RMSE
mod1_poisson	glm	1010.7 (<.001)	1029.4 (<.001)	0.033	0.579
mod1_nb	negbin	977.0 (>.999)	1000.5 (>.999)	0.032	0.579

43.4 Exemple avec une plus grande surdispersion

Pour ce second exemple, nous allons considérer le jeu de données `MASS::quine` qui contient les données de 146 enfants scolarisés en Australie, notamment le nombre de jours d'absence à l'école au cours de l'année passée, le sexe l'enfant et leur vitesse d'apprentissage (dans la moyenne ou lentement).

Préparons les données en francisant les facteurs et en ajoutant des étiquettes de variable.

```
d <- MASS::quine |>
mutate(
  jours = Days,
  sexe = Sex |>
    fct_recode(
      "fille" = "F",
      "garçon" = "M"
    ),
  apprentissage = Lrn |>
    fct_recode(
      "dans la moyenne" = "AL",
      "lentement" = "SL"
    )
) |>
set_variable_labels(
  jours = "Nombre de jours d'absence à l'école",
  sexe = "Sexe de l'enfant",
  apprentissage = "Vitesse d'apprentissage"
)
```

Calculons notre modèle de Poisson.

```
mod2_poisson <- glm(  
  jours ~ sexe + apprentissage,  
  data = d,  
  family = poisson  
)
```

Comparons les données observées avec les données prédites.

```
mod2_poisson |>  
  observed_vs_theoretical()
```

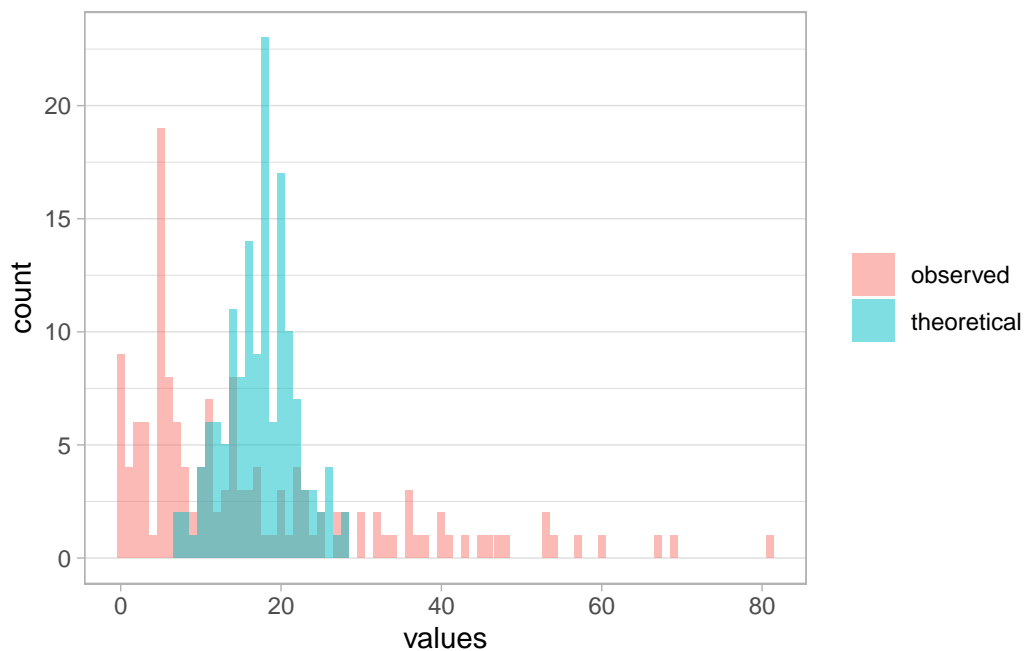


Figure 43.8: Distribution observée vs distribution théorique du nombre de jours d'absence (modèle de Poisson)

Nous voyons très clairement un décalage des deux distributions. Notre modèle de Poisson n'arrive pas à capturer la variabilité des observations. Faisons le test de surdispersion pour vérifier.

```
mod2_poisson |>
  performance::check_overdispersion()
```

```
# Overdispersion test
```

```
      dispersion ratio =    15.895
Pearson's Chi-Squared = 2273.046
      p-value =    < 0.001
```

Overdispersion detected.

Calculons un modèle binomial négatif et voyons si cela améliore la situation.

```
mod2_nb <- MASS::glm.nb(
  jours ~ sexe + apprentissage,
  data = d
)
```

```
mod2_nb |>
  observed_vs_theoretical()
```

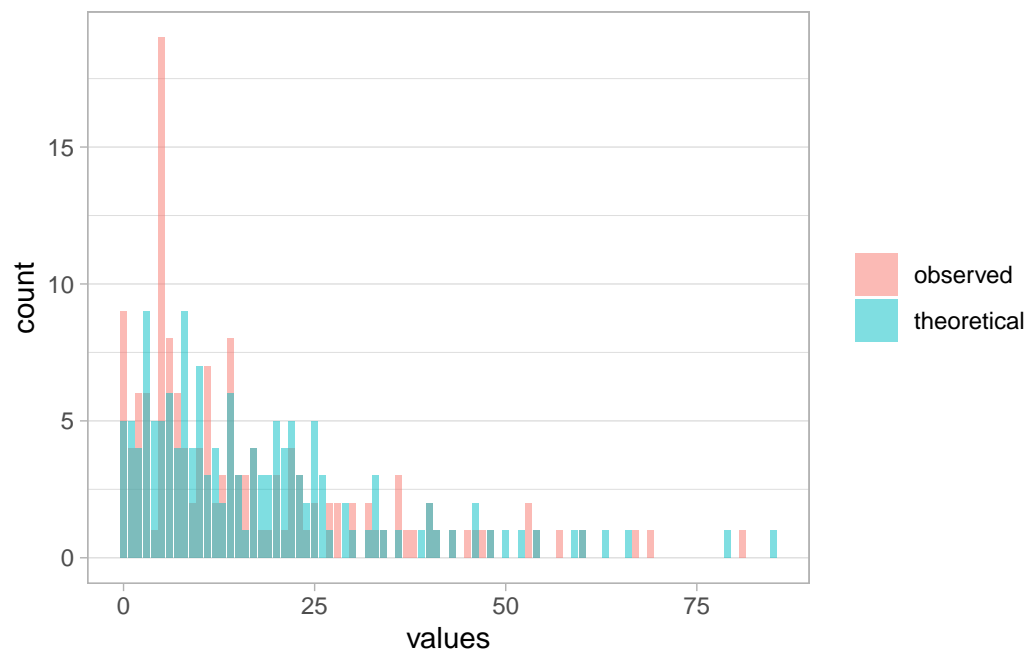


Figure 43.9: Distribution observée vs distribution théorique du nombre de jours d'absence (modèle binomial négatif)

Les deux distributions sont bien plus proches maintenant. Vérifions la surdispersion.

```
mod2_nb |>
  performance::check_overdispersion()
```

```
# Overdispersion test
```

```
dispersion ratio = 0.978
p-value = 0.992
```

No overdispersion detected.

Voilà !

Pour finir, visualisons les coefficients du modèle.

```
mod2_nb |>
  ggcoef_table(exponentiate = TRUE)
```

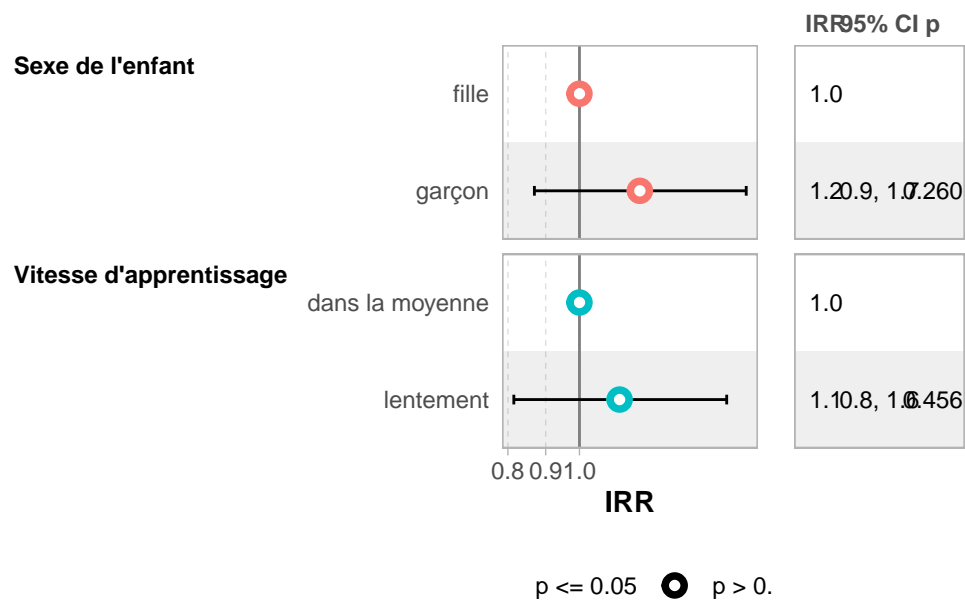


Figure 43.10: Facteurs associés à l'absentéisme scolaire (modèle négatif binomial)

43.5 Modèles de comptage avec une variable binaire

Pour l'analyse d'une variable binaire, les modèles de comptage représentent une alternative à la régression logistique binaire (cf. Chapitre 22). L'intérêt est de pouvoir interpréter les coefficients comme des *prevalence ratio* plutôt que des *odds ratio*.

Reprenons un exemple, que nous avons déjà utilisé à plusieurs reprises, concernant la probabilité de faire du sport, à partir de l'enquête *histoires de vie 2003*. Commençons par charger et recoder les données.

```
data(hdv2003, package = "questionr")

d <-
  hdv2003 |>
  mutate(
    groupe_ages = age |>
      cut(
        c(18, 25, 45, 65, 99),
        right = FALSE,
        include.lowest = TRUE,
        labels = c("18-24 ans", "25-44 ans",
                  "45-64 ans", "65 ans et plus")
      )
  ) |>
  set_variable_labels(
    sport = "Pratique un sport ?",
    sexe = "Sexe",
    groupe_ages = "Groupe d'âges",
    heures.tv = "Heures de télévision / jour"
  )
```

Pour la variable *sexe*, nous allons définir la modalité Femme comme modalité de référence. Pour cela, nous allons utiliser un contraste personnalisé (cf. Chapitre 25).

```
levels(d$sexe)
```

```
[1] "Homme" "Femme"
```

```
contrasts(d$sexe) <- contr.treatment(2, base = 2)
```

Calculons le modèle de régression logistique binaire classique.

```
mod3_binomial <- glm(
  sport ~ sexe + groupe_ages + heures.tv,
  family = binomial,
  data = d
)
```

Nous allons maintenant calculer un modèle de Poisson. Nous devons déjà ré-exprimer notre variable à expliquer sous la forme d'une variable numérique égale à 0 si l'on ne pratique pas de sport et à 1 si l'on pratique un sport.

```
levels(d$sport)
```

```
[1] "Non" "Oui"
```

```
d$sport2 <- as.integer(d$sport == "Oui")
mod3_poisson <- glm(
  sport2 ~ sexe + groupe_ages + heures.tv,
  family = poisson,
  data = d
)
```

Vérifions si nous avons un problème de surdispersion.

```
performance::check_overdispersion(mod3_poisson)
```

```
# Overdispersion test
```

```
      dispersion ratio =      0.635
Pearson's Chi-Squared = 1263.552
      p-value =          1
```

```
No overdispersion detected.
```

Regardons maintenant les résultats de nos deux modèles.

```
mod3_binomial |>
  ggstats::ggcoef_table(exponentiate = TRUE)
```

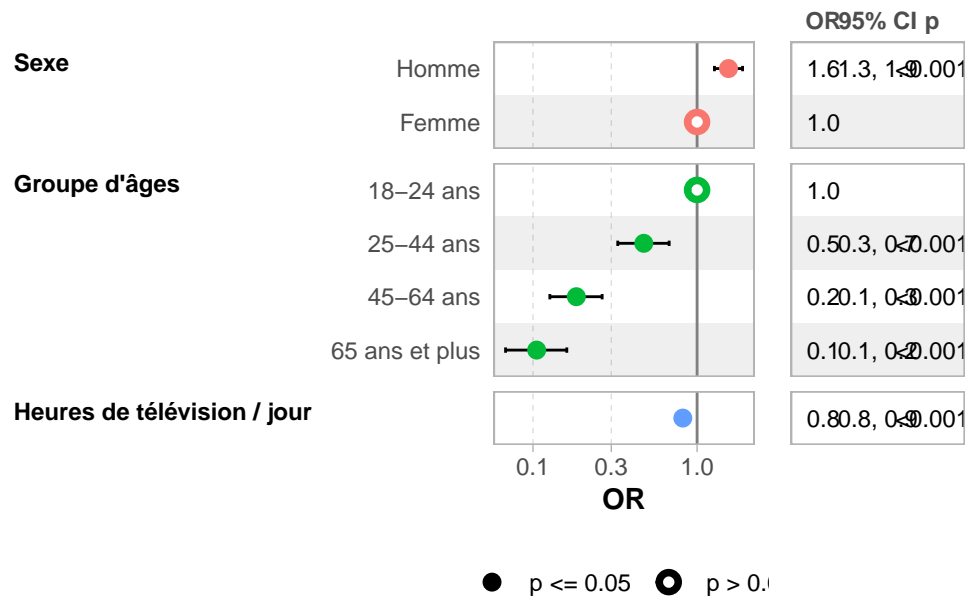


Figure 43.11: Facteurs associés à la pratique d'un sport (régression logistique)

```
mod3_poisson |>
  ggstats::ggcoef_table(exponentiate = TRUE)
```

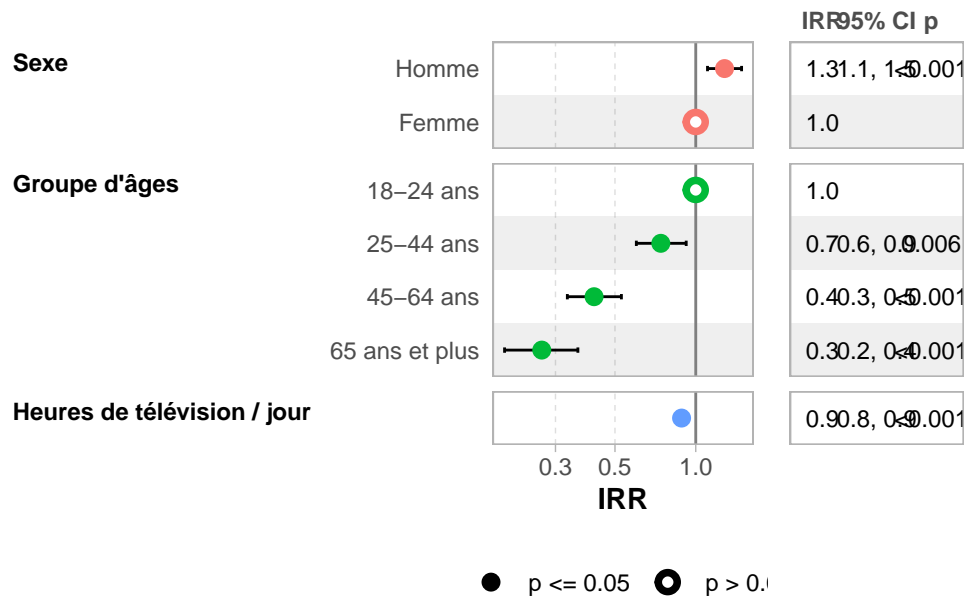



Figure 43.12: Facteurs associés à la pratique d'un sport (régression de Poisson)

Nous pouvons voir ici que les deux modèles fournissent des résultats assez proches. Par contre, les coefficients ne s'interprètent pas de la même manière. Dans le cadre de la régression logistique, il s'agit d'*odds ratios* (ou rapports des côtes) définis comme $OR_{A/B} = \left(\frac{p_A}{1-p_A}\right) / \left(\frac{p_B}{1-p_B}\right)$ où p_A correspond à la probabilité de faire du sport pour la modalité A . Pour la régression de Poisson, il s'agit de *prevalence ratios* (rapports des prévalences) définis comme $PR_{A/B} = p_A / p_B$. Avec un rapport des prévalences de 1,3, nous pouvons donc dire que, selon le modèle, les hommes ont 30% de chance en plus de pratiquer un sport.

Pour mieux comparer les deux modèles, nous pouvons présenter les résultats sous la forme de contrastes marginaux moyens (cf. Section 24.4) qui, pour rappel, sont exprimés dans l'échelle de la variable d'intérêt, soit ici sous la forme d'une différence de probabilité.

```
list(
  "régression logistique" = mod3_binomial,
  "régression de Poisson" = mod3_poisson
) |>
ggcoef_compare(tidy_fun = broom.helpers::tidy_marginal_contrasts) +
scale_x_continuous(labels = scales::percent)
```

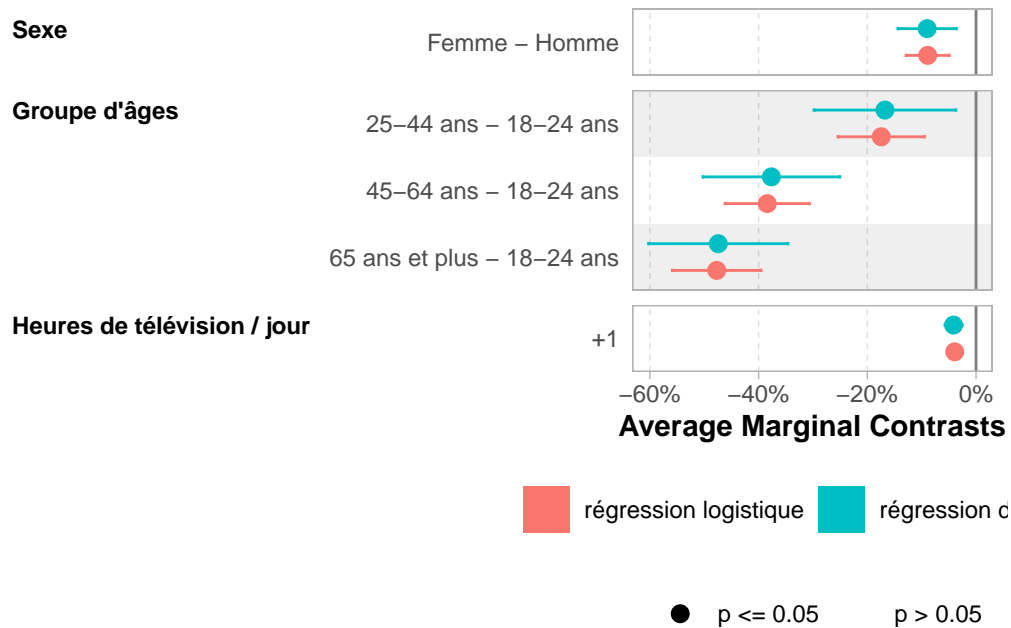


Figure 43.13: Comparaison des contrastes marginaux des deux modèles

Les résultats sont ici très proches. Nous pouvons néanmoins constater que les intervalles de confiance pour la régression de Poisson sont un peu plus large. Nous pouvons comparer les deux modèles avec `performance::compare_performance()` pour constater que, dans notre exemple, la régression de Poisson est un peu moins bien ajustée aux données que la régression logistique binaire. Cependant, en pratique, cela n'est pas ici problématique : le choix entre les deux modèles peut donc se faire en fonction de la manière dont on souhaite présenter et narrer les résultats.

```
performance::compare_performance(
  mod3_binomial,
  mod3_poisson,
  metrics = "common"
)
```

```
Warning: contrasts dropped from factor sexe
Warning: contrasts dropped from factor sexe
Warning: contrasts dropped from factor sexe
Warning: contrasts dropped from factor sexe
```

```
# Comparison of Model Performance Indices
```

Name	Model	AIC (weights)	BIC (weights)	RMSE	Tjur's R2	Nagelkerke's R2
mod3_binomial	glm	2346.3 (>.999)	2379.8 (>.999)	0.447	0.132	
mod3_poisson	glm	2748.8 (<.001)	2782.4 (<.001)	0.447		0.15

Astuce

Lorsque l'on a une variable binaire à expliquer et que l'on souhaite calculer des risques relatifs (RR) ou *prevalence ratio* (PR), une alternative au modèle de Poisson est le modèle log-binomial. Il s'agit d'un modèle binomial avec une fonction de lien logarithme. Il faut noter que ce type de modèles a parfois du mal à converger.

```
mod3_log <- glm(
  sport ~ sexe + groupe_ages + heures.tv,
  family = binomial(link = "log"),
  data = d
)
```

Error: impossible de trouver un jeu de coefficients correct : prière de fournir des valeurs

C'est le cas ici ! Nous allons donc initier le modèle avec les coefficients du modèle de Poisson.

```
mod3_log <- glm(
  sport ~ sexe + groupe_ages + heures.tv,
  family = binomial(link = "log"),
  start = mod3_poisson$coefficients,
  data = d
)
```

Regardons les résultats.

```
mod3_log |>
  ggstats::ggcoef_table(exponentiate = TRUE)
```

Warning: le pas a été tronqué à cause de la divergence

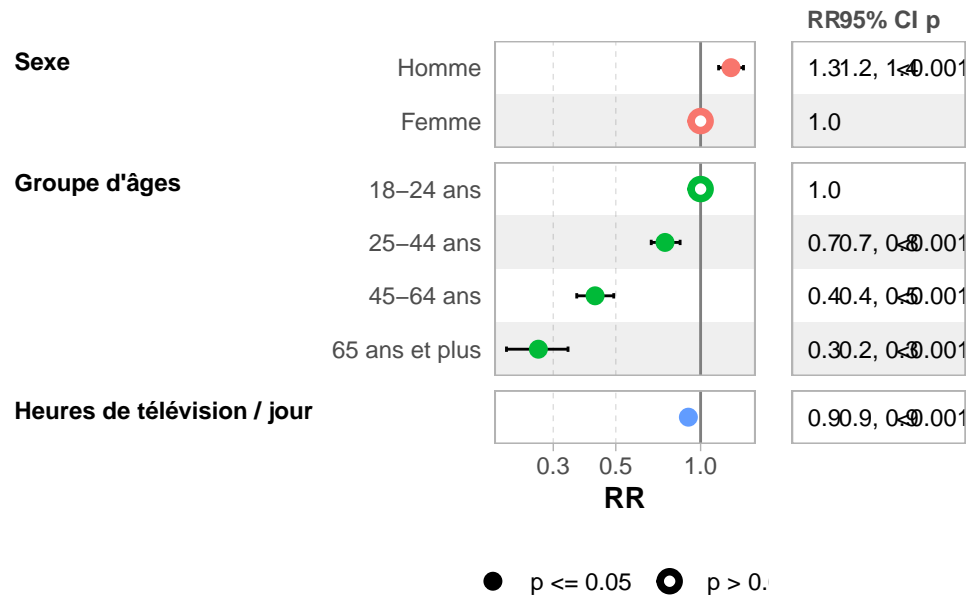
Warning: le pas a été tronqué à cause de la divergence

Warning: glm.fit: l'algorithme n'a pas convergé

Warning: glm.fit: l'algorithme n'a pas convergé

Warning: le pas a été tronqué à cause de la divergence

Warning: glm.fit: l'algorithme n'a pas convergé



Nous obtenons des résultats assez proches de ceux du modèle de Poisson. Notons cependant les différents avis affichés qui nous indiquent que le modèle a eu des difficultés à converger³.

Le package `{logbin}` propose, via `logbin::logbin()`, une implémentation de la régression log-binomiale en proposant des algorithmes de convergence plus stables.

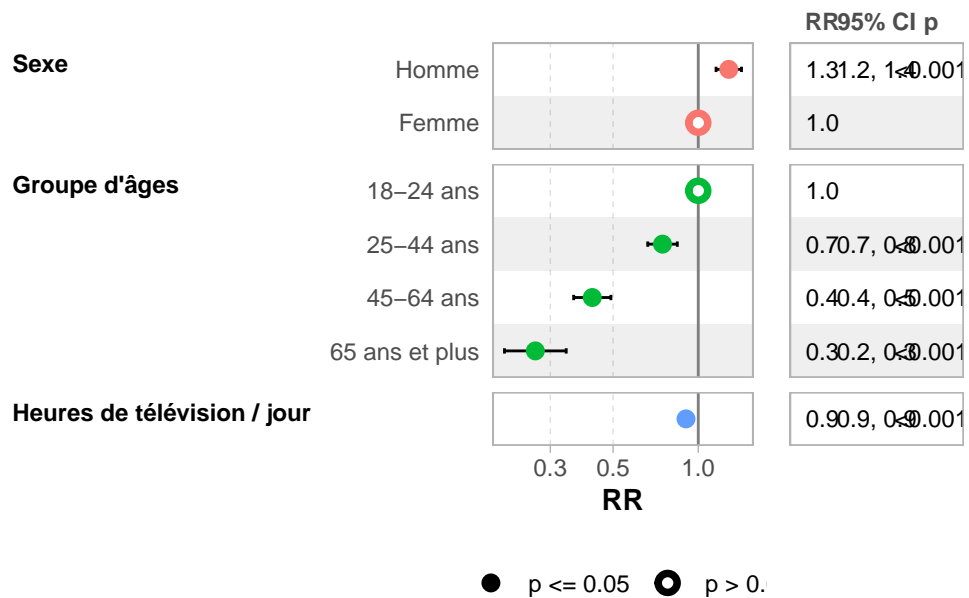
```
mod3_logbin <- logbin::logbin(
  sport ~ sexe + groupe_ages + heures.tv,
  data = d
)
```

Les résultats sont très proches.

```
mod3_logbin |>
  ggstats::ggcoef_table(exponentiate = TRUE)
```

Warning: The ``tidy()`` method for objects of class ``logbin`` is not maintained by the broom team.

This warning is displayed once per session.



43.6 Données pondérées et plan d'échantillonnage

Lorsque l'on a des données pondérées avec prise en compte d'un plan d'échantillonnage (cf. Chapitre 28), on ne peut utiliser directement `stats::glm()` avec un objet `{survey}`. On aura alors recours à `survey::svyglm()` qui est très similaire.

```
library(srvyr)
library(survey)
dp <- d |>
  as_survey_design(weights = poids)
mod4_poisson <- svyglm(
  sport2 ~ sexe + groupe_ages + heures.tv,
  family = poisson,
  design = dp
)
mod4_quasi <- svyglm(
  sport2 ~ sexe + groupe_ages + heures.tv,
```

³Sur ce sujet, on pourra consulter l'article *Log-binomial models: exploring failed convergence* par Tyler Williamson, Misha Eliasziw et Gordon Hilton Fick, DOI: [10.1186/1742-7622-10-14](https://doi.org/10.1186/1742-7622-10-14). On pourra également consulter cet échange sur [StackExchange](#).

```
family = quasipoisson,
design = dp
)
```

Il est tout à fait possible d'appliquer `stats::step()` à ces modèles⁴.

Concernant la régression binomiale négative, il n'y a pas d'implémentation fournie directement par `{survey}`. Cependant, le package `{sjstats}` en propose une via la fonction `sjstats::svyglm.nb()`.

```
mod4_nb <- sjstats::svyglm.nb(
  sport2 ~ sexe + groupe_ages + heures.tv,
  design = dp
)
```

```
Warning in theta.ml(Y, mu, sum(w), w, limit = control$maxit, trace =
control$trace > : nombre limite d'iterations atteint
Warning in theta.ml(Y, mu, sum(w), w, limit = control$maxit, trace =
control$trace > : nombre limite d'iterations atteint
```

i Note

Une alternative possible consiste à utiliser des poids de réplification selon une approche du type *bootstrap*. Il faudra déjà définir des poids de réplification avec `srvyr::as_survey_rep()` puis avoir recours à `survey::withReplicates()`. Pour faciliter cette deuxième étape, on pourra se faciliter la vie avec le package `{svrepmisc}` et sa fonction `svrepmisc::svynb()`. Ce package n'étant pas disponible sur CRAN, on devra l'installer avec la commande `remotes::install_github("carlganz/svrepmisc")`.

Attention : le temps de calcul du modèle avec les poids de réplification est de plusieurs minutes.

```
dp_rep <- dp |>
  as_survey_rep(type = "bootstrap", replicates = 100)
mod4_nb_alt <- svrepmisc::svynb(
  sport2 ~ sexe + groupe_ages + heures.tv,
  design = dp_rep
)
```

⁴Y compris, dans le cas présent, au modèle de quasi-Poisson.

43.7 Lectures complémentaires

- [Tutoriel : GLM sur données de comptage \(régression de Poisson\) avec R](#) par Claire Della Vedova
- [Tutorial: Poisson Regression in R](#) (en anglais) par Hafsa Jabeen
- [Quasi-Poisson vs. negative binomial regression: how should we model overdispersed count data?](#) par Jay M Ver Hoef et Peter L Boveng, *Ecology*. 2007 Nov; 88(11):2766-72. doi: [10.1890/07-0043.1](#). PMID: [18051645](#)

44 Modèles d'incidence / de taux

En épidémiologie, le taux d'incidence rapporte le nombre de nouveaux cas d'une pathologie observés pendant une période donnée à la population exposée pendant cette même période. En démographie, le terme de taux est utilisé pour désigner la fréquence relative d'un évènement au sein d'une population pendant une période de temps donnée (par exemple : taux de natalité).

Si l'ensemble des individus sont observés / exposés pendant une seule unité de temps (par exemple une année), alors cela revient à rapporter le nombre moyen d'évènements à 1 : nous pouvons utiliser un modèle classique de comptage (cf. Chapitre 43).

Cependant, le plus souvent, la durée d'observation / d'exposition varie d'un individu à l'autre. Par exemple, si nous nous intéressons à un taux de divorce, les individus ne sont exposés au risque de divorcer qu'à partir du moment où ils sont mariés. De même, une fois divorcés ou veufs, ils ne sont plus exposés au risque de divorce (sauf s'ils se remarient ultérieurement). Pour chaque individu, il nous faut donc connaître le nombre d'évènements vécus (n_{evts}) et la durée d'exposition (d_{exp}). Ce que l'on cherche à modéliser est donc le ratio n_{evts}/d_{exp} .

Une astuce consiste à modéliser ce taux à l'aide d'un modèle ayant une fonction de lien logarithmique (*log*) comme le modèle de Poisson ou le modèle binomial négatif. En effet, dans ce cas-là, on cherchera donc à modéliser notre variable sous la forme $\log(n_{evts}/d_{exp}) = \beta_i X_i$ où X_i représente les variables explicatives et β_i les coefficients du modèle. Or, $\log(n_{evts}/d_{exp}) = \log(n_{evts}) - \log(d_{exp})$. Nous pouvons donc réécrire l'équation du modèle sous la forme $\log(n_{evts}) = \beta_i X_i + \log(d_{exp})$. Nous retombons sur un modèle de comptage classique, à condition d'ajouter à chaque observation ce qu'on appelle un décalage (*offset* en anglais) de $\log(d_{exp})$. Ce décalage correspond donc en quelque sorte à une variable ajoutée au modèle mais pour laquelle on ne calcule pas de coefficient.

44.1 Premier exemple (données individuelles, évènement unique)

Prenons un premier exemple à partir du jeu de données `gtsummary::trial` qui contient des informations sur 200 patients atteints d'un cancer. Il contient entre autre les variables suivantes :

- *death* : variable binaire (0/1) indiquant si le patient est décédé
- *ttdeath* : le nombre de mois d'observation jusqu'au décès (si décès) ou jusqu'à la fin de l'étude (si survie)

- *stage* : un facteur indiquant le stade T de la tumeur (plus la valeur est élevée, plus la tumeur est grosse)
- *trt* : le traitement reçu par le patient (A ou B)
- *response* : une variable binaire (0/1) indiquant si le traitement a eu un effet sur la tumeur (diminution)

Nous nous intéressons donc aux facteurs associés au taux de mortalité (*death/ttdeath*) : nous allons donc réaliser un modèle de Poisson sur la variable *death* en ajoutant un décalage (*offset*) correspondant à $\log(ttdeath)$. Pour ajouter un décalage, nous avons deux syntaxes équivalentes : soit en ajoutant `offset(log(ttdeath))` directement à l'équation du modèle, soit en passant à `glm()` l'argument `offset = log(ttdeath)`.

```
mod1_poisson <- glm(
  death ~ stage + trt + response + offset(log(ttdeath)),
  family = poisson,
  data = gtsummary::trial
)
mod1_poisson_alt <- glm(
  death ~ stage + trt + response,
  offset = log(ttdeath),
  family = poisson,
  data = gtsummary::trial
)
```

Les deux écritures sont totalement équivalentes.

Vérifions la présence éventuelle de surdispersion.

```
mod1_poisson |>
  performance::check_overdispersion()
```

```
# Overdispersion test
```

```
      dispersion ratio =    0.850
Pearson's Chi-Squared = 159.039
      p-value =    0.932
```

```
No overdispersion detected.
```

Tout est bon. Regardons maintenant les coefficients du modèle.

```
mod1_poisson |>
  ggstats::ggcoef_table(exponentiate = TRUE)
```

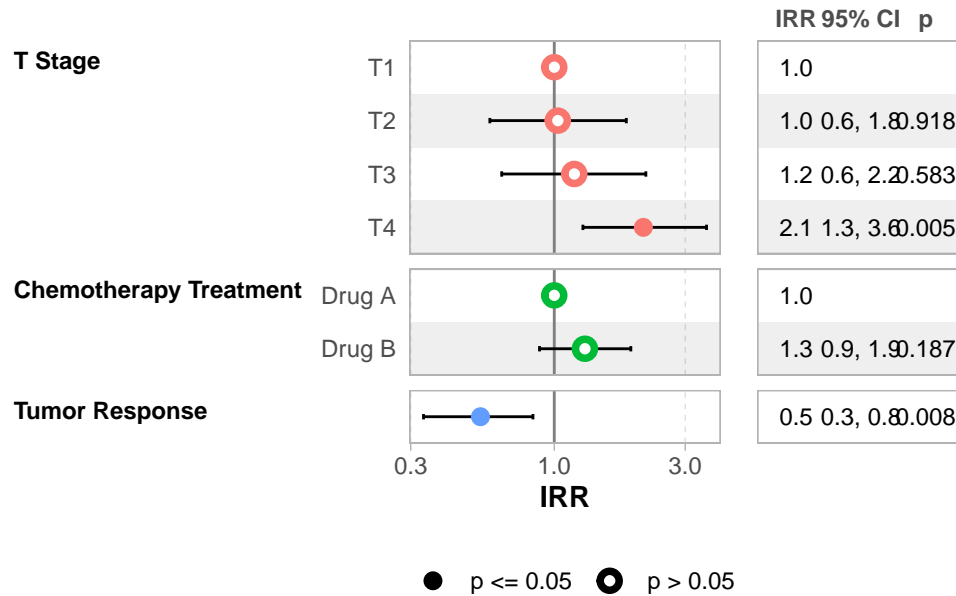


Figure 44.1: Facteurs associés à la mortalité des patients atteints d'un cancer (modèle de Poisson)

Nous avons affiché ici l'exponentielle des coefficients qui s'interprètent ici comme des IRR ou *incidence rate ratio* : le taux de décès est deux fois moindre pour les patients pour lesquels le traitement a eu un effet sur la tumeur (variable *Tumor response*). Sans surprise, le taux de décès est bien plus élevé selon la taille de la tumeur : 2,1 fois plus important pour ceux avec une tumeur au stade T4 par rapport à ceux ayant une tumeur au stade T1.

44.2 Deuxième exemple (données agrégées)

Pour ce second exemple, nous allons considérer le jeu de données **MASS : : Insurance** qui provient d'une compagnie d'assurance américaine et porte sur le troisième trimestre 1973. Il indique le nombre de demande d'indemnisations (*Claims*) parmi les assurés pour leur voiture (*Holders*) en fonction de leur groupe d'âges (*Age*) et de la taille de la cylindrée de la voiture (*Group*). Nous cherchons à identifier les facteurs associés au taux de réclamation. Préparons rapidement les données et définissons notre modèle.

```
d <- MASS::Insurance
d$Age <- factor(d$Age, ordered = FALSE)
d$Group <- factor(d$Group, ordered = FALSE)
mod2_poisson <- glm(
  Claims ~ Age + Group + offset(log(Holders)),
  family = poisson,
  data = d
)
mod2_poisson |>
performance::check_overdispersion()
```

Overdispersion test

```
dispersion ratio = 1.140
Pearson's Chi-Squared = 65.003
p-value = 0.218
```

No overdispersion detected.

Regardons les résultats.

```
mod2_poisson |>
ggstats::ggcoef_table(exponentiate = TRUE)
```

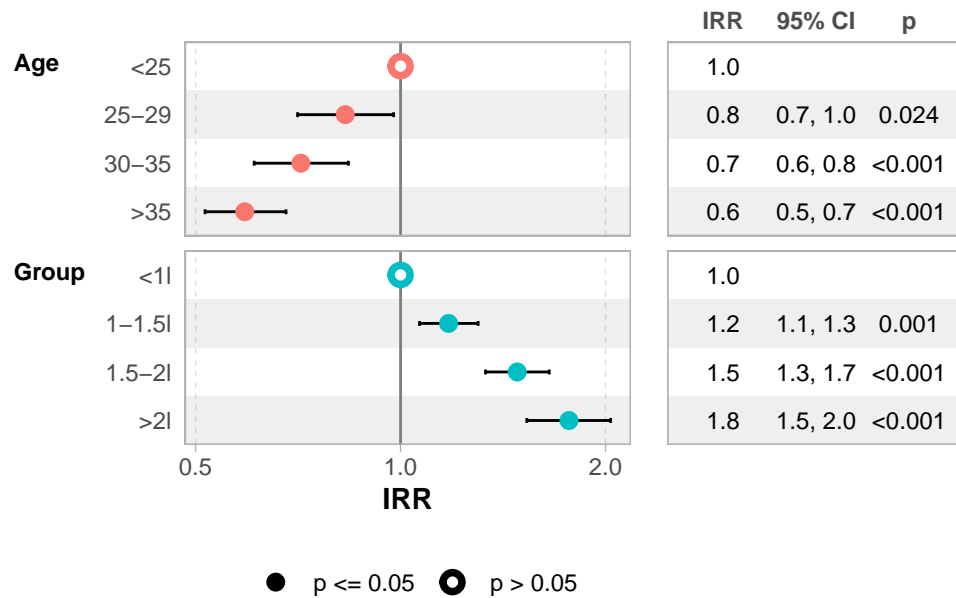


Figure 44.2: Facteurs associés au taux de réclamation (modèle de Poisson)

Le taux de réclamation diminue avec l'âge de l'assuré (il est 40% moindre pour les assurés de plus de 35 ans par rapport à ceux de moins de 25 ans) et augmente avec la cylindrée de la voiture (il est 80% plus élevé pour les véhicules avec une cylindrée de plus de 2 litres par rapport aux véhicules avec une cylindrée de moins d'1 litre).

44.3 Troisième exemple (données individuelles, évènement unique)

Pour notre troisième exemple, nous allons reprendre les données de fécondité présentée au chapitre précédent (cf. Chapitre 43) et venant d'une enquête transversale rétrospective menée auprès de femmes âgées de 15 à 49 ans.

Nous allons nous intéresser au taux de fécondité entre 15 et 24 ans révolus (soit entre 15 et 25 ans exacts) et intégrer à l'analyse les femmes de moins de 25 ans en tenant compte de leur durée d'exposition (différence entre l'âge à l'enquête et 15 ans). Nous allons donc calculer la durée d'exposition comme `exposition = if_else(age <= 25, age - 15, 10)` puisque, pour les femmes de plus de 25 ans à l'enquête, la durée d'exposition entre 15 et 25 ans exacts est de 10 ans.

```

library(tidyverse)
library(labelled)
data("fecondite", package = "questionr")
femmes <-
  femmes |>
  unlabelled() |>
  mutate(
    age = time_length(
      date_naissance %--% date_entretien,
      unit = "years"
    ),
    exposition = if_else(age <= 25, age - 15, 10),
    educ2 = educ |>
      fct_recode(
        "secondaire/supérieur" = "secondaire",
        "secondaire/supérieur" = "supérieur"
      )
  ) |>
  # exclure celles qui viennent juste d'avoir 15 ans
  filter(exposition > 0)

```

Comptons maintenant le nombre de naissances entre 15 et 25 ans exacts.

```

enfants <-
  enfants |>
  unlabelled() |>
  left_join(
    femmes |>
      select(id_femme, date_naissance_mere = date_naissance),
    by = "id_femme"
  ) |>
  mutate(
    age_mere = time_length(
      date_naissance_mere %--% date_naissance,
      unit = "years"
    )
  )
femmes <-
  femmes |>
  left_join(
    enfants |>
      filter(age_mere >= 15 & age_mere < 25) |>

```

```

group_by(id_femme) |>
  count(name = "enfants_15_24"),
  by = "id_femme"
) |>
tidyr::replace_na(list(enfants_15_24 = 0L))

```

Calculons maintenant notre modèle.

```

mod3_poisson <- glm(
  enfants_15_24 ~ educ2 + milieu + offset(log(exposition)),
  family = poisson,
  data = femmes
)

```

Vérifions la surdispersion.

```

mod3_poisson |>
  performance::check_overdispersion()

```

Overdispersion test

```

      dispersion ratio =      1.368
Pearson's Chi-Squared = 2723.252
      p-value =    < 0.001

```

Overdispersion detected.

Le test indique de la surdispersion. Optons donc pour un modèle négatif binomial.

```

mod3_nb <- MASS::glm.nb(
  enfants_15_24 ~ educ2 + milieu + offset(log(exposition)),
  data = femmes
)
mod3_nb |>
  performance::check_overdispersion()

```

Overdispersion test

```

      dispersion ratio = 0.822
      p-value = 0.016

```

Underdispersion detected.

Nous pouvons maintenant regarder les résultats.

```
mod3_nb |>
  ggstats::ggcoef_table(exponentiate = TRUE)
```

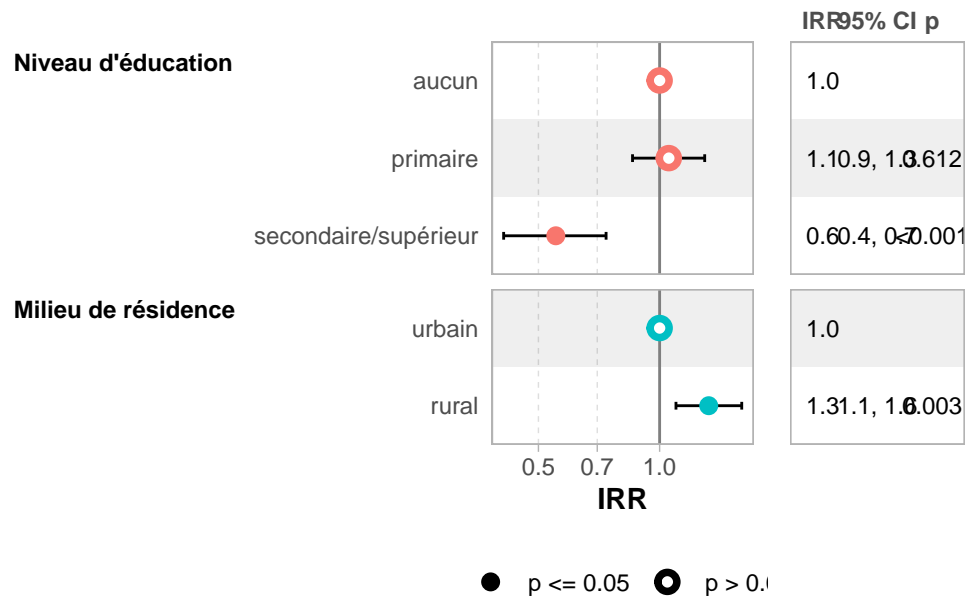


Figure 44.3: Facteurs associés au taux de fécondité entre 15 et 25 ans (modèle négatif binomial)

44.4 Lectures complémentaires

- [Tutoriel : GLM sur données de comptage \(régression de Poisson\) avec R](#) par Claire Della Vedova
- [Zoom sur la Régression de Poisson et l'Incidence Risk Ratio \(IRR\) : exemple du vaccin anti-SarsCov2 d'Oxford](#) par Ihsane Hmamouchi

45 Modèles de comptage *zero-inflated* et *hurdle*

Dans certaines situations, les modèles de comptage (cf. Chapitre 43) ont des difficultés à estimer correctement le nombre d'individus n'ayant pas vécu l'évènement (i.e. le nombre de 0).

45.1 Données d'illustration

Dans ce chapitre, nous allons utiliser un jeu de données issu d'un article de Partha Deb et Pravin K. Trivedi (Deb et Trivedi 1997). Ce jeu de données est disponible dans le package `{MixAll}` sous le nom de `DebTrivedi`. Il porte sur 4406 individus âgés de 66 ans ou plus et couvert par le programme américain *Medicare*.

L'analyse va porter sur la demande de soins, mesurée ici à travers le nombre de visites médicales (*ofp*). Pour les variables explicatives, nous allons considérer le genre du patient (*gender*), le fait de disposer d'une assurance privée (*privins*), la santé perçue (*health*) et le nombre de conditions chroniques de l'assuré.

Chargeons et préparons rapidement les données. Nous allons recoder les variables catégorielles en français (Section 9.3) et ajouter des étiquettes de variables (cf. Chapitre 11).

```
library(labelled)
library(tidyverse)
data("DebTrivedi", package = "MixAll")
d <- DebTrivedi |>
  mutate(
    gender = gender |>
      fct_recode("femme" = "female", "homme" = "male"),
    privins = privins |>
      fct_recode("non" = "no", "oui" = "yes"),
    health = health |>
      fct_recode(
        "pauvre" = "poor",
        "moyenne" = "average",
```



```

    "excellente" = "excellent"
  )
) |>
set_variable_labels(
  ofp = "Nombre de visites médicales",
  gender = "Genre de l'assuré",
  privins = "Dispose d'une assurance privée ?",
  health = "Santé perçue",
  numchron = "Nombre de conditions chroniques"
)
contrasts(d$health) <- contr.treatment(3, base = 2)

```

45.2 Modèles de comptage classique

Commençons tout d'abord par une approche classique (Chapitre 43) : calculons un modèle de Poisson et vérifions la surdispersion.

```

mod_poisson <- glm(
  ofp ~ gender + privins + health + numchron,
  family = poisson,
  data = d
)
mod_poisson |>
  performance::check_overdispersion()

```

```
# Overdispersion test
```

```

      dispersion ratio =      7.103
Pearson's Chi-Squared = 31254.867
      p-value =    < 0.001

```

Overdispersion detected.

Une surdispersion étant détectée, basculons sur un modèle négatif binomial.

```

mod_nb <- MASS::glm.nb(
  ofp ~ gender + privins + health + numchron,
  data = d
)

```

```
mod_nb |>
  performance::check_overdispersion()
```

```
# Overdispersion test
```

```
dispersion ratio = 1.050
p-value = 0.36
```

```
No overdispersion detected.
```

Le modèle négatif binomial ne règle pas notre problème de surdispersion. Comparons les valeurs observées avec les valeurs théoriques avec `performance::check_predictions()`¹. Pour faciliter la lecture du graphique, nous allons zoomer sur les 20 premières valeurs.

```
mod_nb |>
  performance::check_predictions() |>
  plot() +
  xlim(0, 20)
```

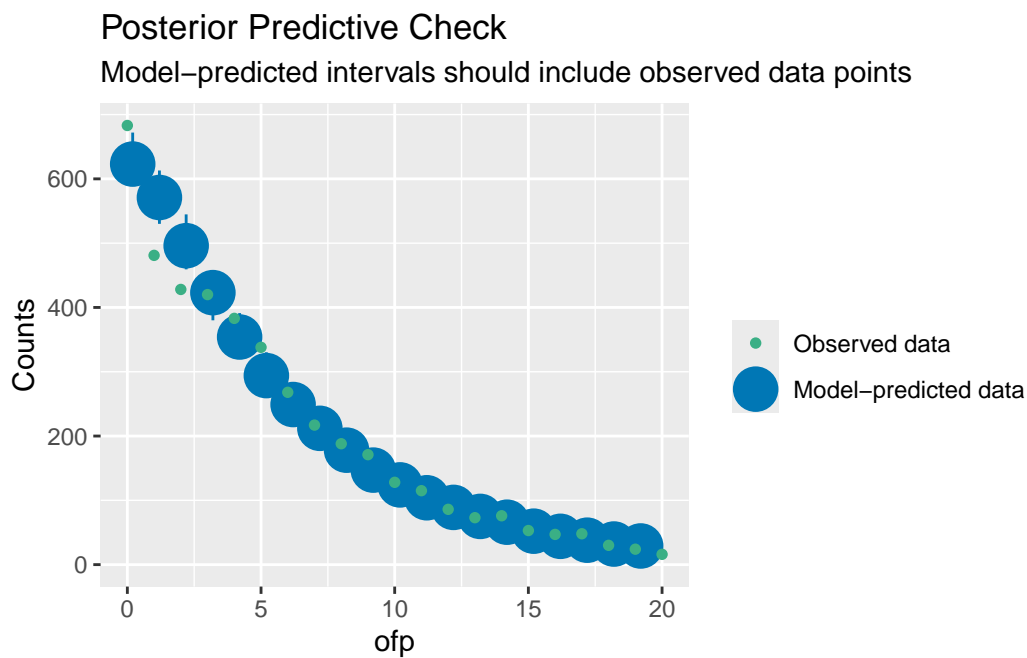


Figure 45.1: Comparaison des valeurs observées et des valeurs théoriques

¹Nous aurions aussi pu utiliser la fonction `observed_vs_theoretical()` présentée dans le chapitre sur les modèles de comptage (cf. Section 43.1.3).

Comme nous pouvons le voir sur ce graphique, le nombre de 0 prédit par le modèle est inférieur à celui observé. Cela signifie que les 0 sont sur-représentés dans nos données par rapport à une distribution négative binomiale. Ces 0 ont tendance à tirer la moyenne vers le bas. Dès lors, le nombre de 1 et de 2 prédits par le modèle sont quant à eux sur-représentés par rapport aux données observées. On dit alors qu'il y a une inflation de zéros dans les données (*zero-inflated* en anglais).

45.3 Modèles *zero-inflated*

Les modèles *zero-inflated* ont justement été prévus pour ce cas de figure. Un modèle de Poisson *zero-inflated* combine deux modèles : un modèle logistique binaire et un modèle de Poisson. Dans un premier temps, on applique le modèle logistique binaire. Si la valeur obtenue est 0, le résultat final est 0. Si la valeur obtenue est 1, alors on applique le modèle de Poisson.

Les modèles de Poisson *zero-inflated* sont notamment implémentés dans le package `{pscl}` via la fonction `pscl::zeroinfl()`.

Calculons un premier modèle de Poisson *zero-inflated*.

```
mod_zip <- pscl::zeroinfl(  
  ofp ~ gender + privins + health + numchron,  
  data = d  
)
```

Regardons les coefficients du modèle (en forçant l'affichage des *intercepts*). Comme il s'agit d'un modèle à plusieurs composantes, nous aurons recours à `ggstats::ggcoef_multicomponents()`.

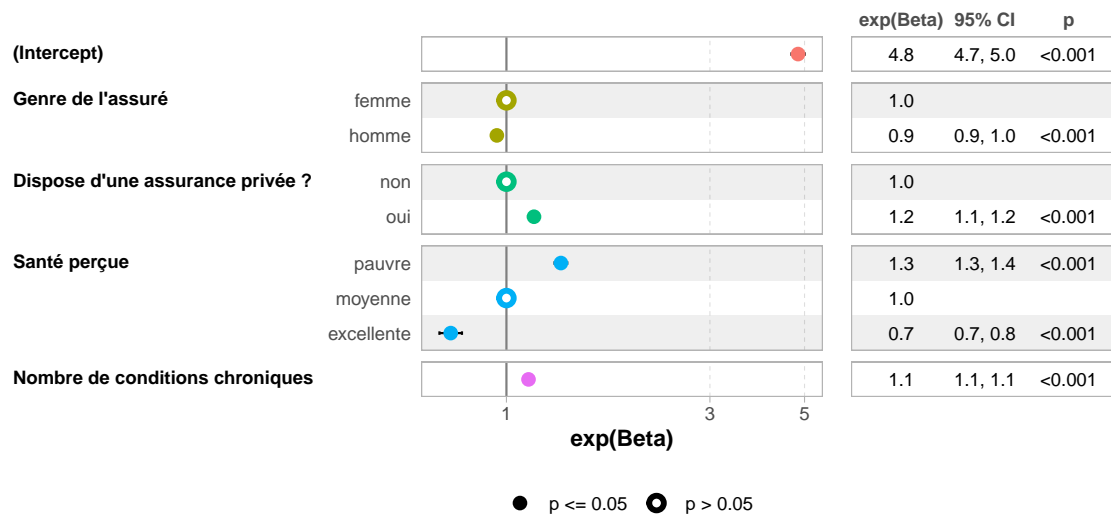
```
mod_zip |>  
  ggstats::ggcoef_multicomponents(  
    type = "table",  
    exponentiate = TRUE,  
    intercept = TRUE  
  )
```

```
i <zeroinfl> model detected.
```

```
v `tidy_zeroinfl()` used instead.
```

```
i Add `tidy_fun = broom.helpers::tidy_zeroinfl` to quiet these messages.
```

conditional



zero_inflated

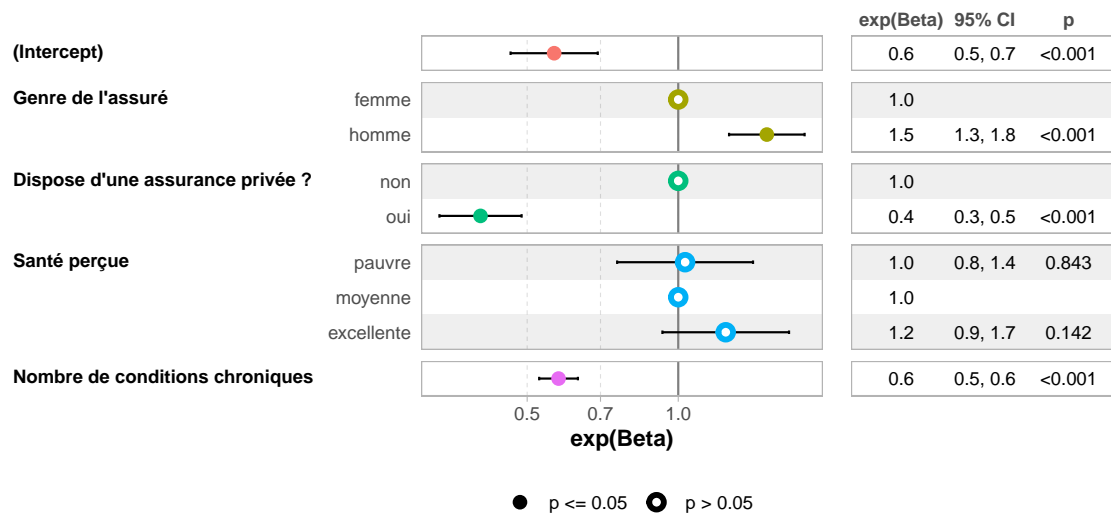


Figure 45.2: Coefficients du modèle de Poisson *zero-inflated*

Nous obtenons deux séries de coefficients : une série *conditional* correspondant au modèle de Poisson et une série *zero_inflated* correspondant au modèle logistique binaire. Nous avons représenté les exponentiels des coefficients, qui s'interprètent donc comme des *risk ratio* pour le modèle de Poisson et des *odds ratio* pour le modèle logistique.

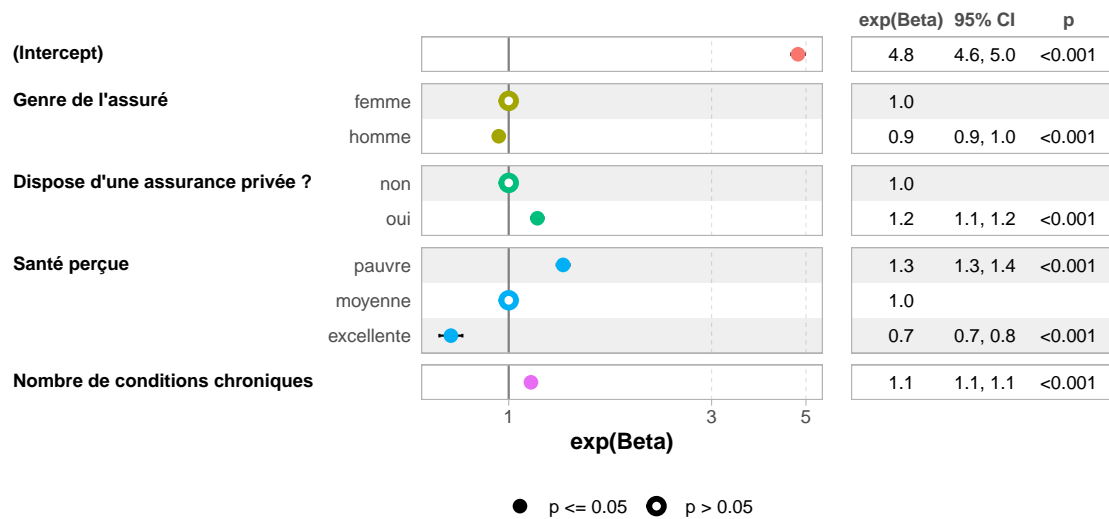
Les variables ayant un effet significatif ne sont pas les mêmes pour les deux composantes du modèle. Il est d'ailleurs possible d'utiliser des variables différentes pour chaque composante,

en écrivant d'abord l'équation du modèle de Poisson, puis celle du modèle logistique et en les séparant avec le symbole `|`. D'ailleurs, la syntaxe `ofp ~ gender + privins + health + numchron` est équivalente à `ofp ~ gender + privins + health + numchron | gender + privins + health + numchron`. Dans la littérature, on trouve fréquemment des modèles de Poisson *zero-inflated* simplifiés où seul un *intercept* est utilisé pour la composante logistique binaire.

```
mod_zip_simple <- pscl::zeroinfl(  
  ofp ~ gender + privins + health + numchron | 1,  
  data = d  
)
```

```
mod_zip_simple |>  
  ggstats::ggcoef_multicomponents(  
    type = "table",  
    tidy_fun = broom.helpers::tidy_zeroinfl,  
    exponentiate = TRUE,  
    intercept = TRUE,  
    component_label = c(  
      conditional = "Modèle de Poisson",  
      zero_inflated = "Modèle logistique binaire"  
    )  
  ) +  
  patchwork::plot_layout(heights = c(6, 1))
```

Modèle de Poisson



Modèle logistique binaire

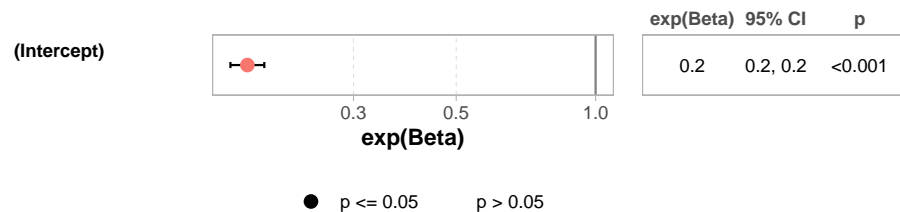


Figure 45.3: Coefficients du modèle de Poisson *zero-inflated* simple

Pour ce jeu de données, cela ne modifie que peu les coefficients de la composante modèle de comptage.

De même, il est possible de préférer un modèle négatif binomial plutôt que Poisson pour la composante modèle de comptage. Il suffit d'ajouter l'argument `dist = "negbin"`.

```
mod_zinb <- pscl::zeroinfl(
  ofp ~ gender + privins + health + numchron,
  dist = "negbin",
  data = d
)
```

Nous pouvons aisément comparer l'AIC de ces différents modèles.

```
performance::compare_performance(
  mod_poisson,
  mod_zip_simple,
  mod_zip,
  mod_nb,
  mod_zinb,
  metrics = "AIC"
)
```

Comparison of Model Performance Indices

Name	Model	AIC (weights)
mod_poisson	glm	36792.1 (<.001)
mod_zip_simple	zeroinfl	33308.5 (<.001)
mod_zip	zeroinfl	33014.2 (<.001)
mod_nb	negbin	24505.7 (<.001)
mod_zinb	zeroinfl	24380.9 (>.999)

Comparons les coefficients de la composante comptage du modèle négatif binomial *zero-inflated* avec ceux du modèle négatif binomial classique.

```
library(gtsummary)
tbl_nb <- mod_nb |>
  tbl_regression(exponentiate = TRUE)
tbl_zinb <- mod_zinb |>
  tbl_regression(
    tidy_fun = broom.helpers::tidy_zeroinfl,
    component = "conditional",
    exponentiate = TRUE
  )
list(tbl_nb, tbl_zinb) |>
  tbl_merge(c("**NB**", "**ZI-NB**")) |>
  bold_labels()
```

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
To suppress this message, include ``message = FALSE`` in code chunk header.

Table 45.1: Coefficients du modèle négatif binomial et de la composante comptage du modèle négatif binomial zero-inflated

Characteristic	IRR	95% CI	p-value	exp(Beta)	95% CI	p-value
Genre de l'assuré						
femme	—	—		—	—	
homme	0.90	0.84, 0.95	<0.001	0.93	0.88, 0.99	0.031
Dispose d'une assurance privée ?						
non	—	—		—	—	
oui	1.39	1.29, 1.50	<0.001	1.23	1.14, 1.33	<0.001
Santé perçue						
pauvre	1.39	1.27, 1.53	<0.001	1.38	1.26, 1.51	<0.001
moyenne	—	—		—	—	
excellente	0.71	0.63, 0.80	<0.001	0.71	0.63, 0.81	<0.001
Nombre de conditions chroniques	1.21	1.19, 1.25	<0.001	1.16	1.13, 1.19	<0.001

Comme nous pouvons le voir, les résultats sont relativement proches.

Si l'interprétation du modèle de comptage reste classique, celle du modèle logistique binaire est parfois un peu plus complexe. En effet, il y a deux sources de 0 dans le modèle *zero-inflated* : si certains sont générés par la composante logistique binaire (dont c'est justement le rôle), le modèle de comptage génère lui aussi des 0. Dès lors, le modèle logistique binaire ne suffit pas à lui seul à identifier les facteurs associés de vivre au moins une fois l'évènement.

Si l'objectif de l'analyse est avant d'identifier les facteurs associés avec le nombre moyen d'évènement, on pourra éventuellement se contenter d'un modèle *zero-inflated* simple, c'est-à-dire avec seulement un *intercept* pour la composante *zero-inflated* afin de corriger la sur-représentation des zéros dans nos données.

Alternativement, on pourra se tourner vers un modèle avec saut qui distingue les valeurs nulles des valeurs positives : les modèles *hurdle* en anglais.

45.4 Modèles *hurdle*

Les modèles *hurdle* se distinguent des modèles *zero-inflated* dans le sens où l'on combine un modèle logistique binomial pour déterminer si les individus ont vécu au moins une fois l'évènement et un modèle de comptage tronqué (qui n'accepte que des valeurs strictement positives) qui détermine le nombre d'évènements vécus uniquement pour ceux l'ayant vécu au moins une fois.

Les modèles *zero-inflated* et *hurdle* diffèrent par leur conceptualisation des zéros et l'interprétation des paramètres du modèle (C. X. Feng 2021).

Un modèle *zero-inflated* suppose que les comptes nuls résultent d'un mélange de deux distributions, l'une où les sujets produisent toujours des comptes nuls, souvent appelés “zéros structurels” ou “zéros excessifs”. Les sujets qui sont exposés au résultat mais qui n'ont pas ou n'ont pas rapporté l'expérience du résultat au cours de la période d'étude sont appelés “zéros d'échantillonnage”. La différenciation des zéros en deux groupes se justifie par le fait que les zéros excessifs sont souvent dus à l'existence d'une sous-population de sujets qui ne sont pas exposés à certains résultats au cours de la période d'étude. Par exemple, lors de la modélisation du nombre de comportements à haut risque, certains participants peuvent obtenir un score de zéro parce qu'ils ne sont pas exposés à un tel comportement à risque pour la santé ; il s'agit des zéros structurels puisqu'ils ne peuvent pas présenter de tels comportements à haut risque. D'autres participants à risque peuvent obtenir un score de zéro parce qu'ils n'ont pas manifesté de tels comportements à risque au cours de la période étudiée. La probabilité d'appartenir à l'une ou l'autre population est estimée à l'aide d'une composante de probabilité à inflation nulle, tandis que les effectifs de la seconde population du groupe d'utilisateurs sont modélisés par une distribution de comptage ordinaire, telle qu'une distribution de Poisson ou binomiale négative.

En revanche, un modèle *hurdle* suppose que toutes les données nulles proviennent d'une source “structurelle”, une partie du modèle étant un modèle binaire pour modéliser si la variable de réponse est nulle ou positive, et une autre partie utilisant un modèle tronqué, pour les données positives. Par exemple, dans les études sur l'utilisation des soins de santé, la partie zéro implique la décision de rechercher des soins, et la composante positive détermine la fréquence de l'utilisation au sein du groupe de l'utilisateur.

Une autre différence importante entre les modèles *hurdle* et *zero-inflated* est leur capacité à gérer la déflation zéro (moins de zéros que prévu par le processus de génération des données). Les modèles *zero-inflated* ne sont pas en mesure de gérer la déflation zéro, quel que soit le niveau d'un facteur, et donneront des estimations de paramètres de l'ordre de l'infini pour la composante logistique, alors que les modèles *hurdle* peuvent gérer la déflation zéro (Min et Agresti 2005).

Les modèles *hurdle* peuvent être calculés avec la fonction `pscl::hurdle()` dont la syntaxe est similaire à `pscl::zeroinfl()`.

```

mod_hurdle_poisson <- pscl::hurdle(
  ofp ~ gender + privins + health + numchron,
  data = d
)
mod_hurdle_nb <- pscl::hurdle(
  ofp ~ gender + privins + health + numchron,
  dist = "negbin",
  data = d
)

```

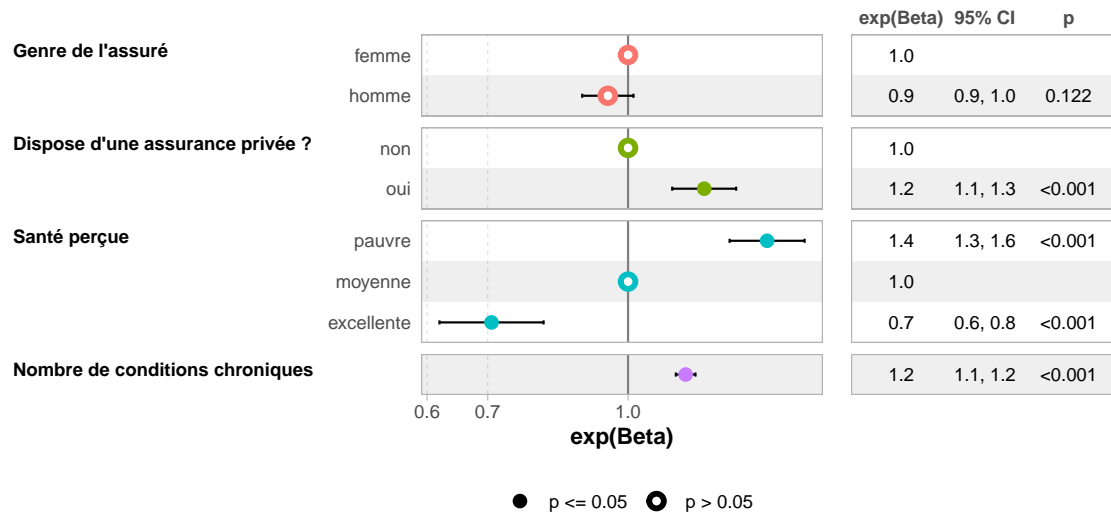
Regardons les coefficients obtenus.

```

mod_hurdle_nb |>
  ggstats::ggcoef_multicomponents(
    type = "table",
    tidy_fun = broom.helpers::tidy_zeroinfl,
    exponentiate = TRUE,
    component_label = c(
      conditional = "Facteurs associés au nombre d'évènements",
      zero_inflated = "Facteurs associés au fait d'avoir vécu l'évènement"
    )
  )

```

Facteurs associés au nombre d'évènements



Facteurs associés au fait d'avoir vécu l'évènement

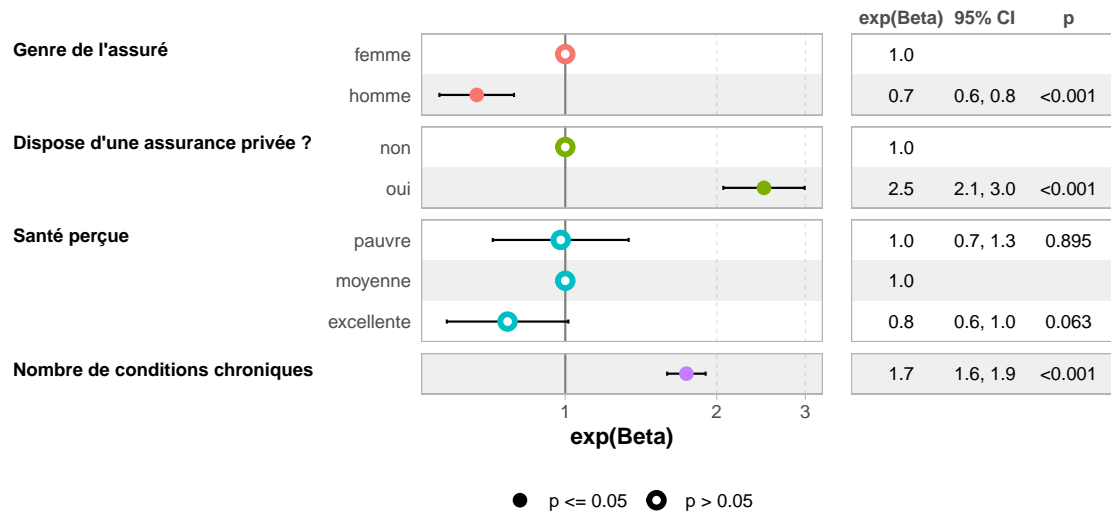


Figure 45.4: Coefficients du modèle négatif binomial *hurdle*

Avec un tel modèle, on cherche à répondre à deux questions :

- Quels sont les facteurs associés au fait d'avoir vécu l'évènement au moins une fois ?
- Si l'on a vécu l'évènement au moins une fois, quels sont les facteurs associés à la fréquence de l'évènement ?

Dans notre exemple, le fait d’avoir une assurance privée joue à la fois sur le fait d’aller consulter un médecin et sur le nombre de consultations. Par contre, la santé perçue n’a pas d’effet sur le fait d’aller consulter mais, si l’on va consulter, cela va influencer fortement sur le nombre de consultations. À l’inverse, le sexe de l’assuré a un effet sur le fait d’aller consulter (les hommes consultent moins que les femmes) mais, si l’on consulte, ne joue pas sur le nombre de consultations.

45.5 Modèles de taux *zero-inflated* ou *hurdle*

Il est tout à fait possible de réaliser un modèle de taux ou d’incidence (cf. Chapitre 44) *zero-inflated* ou *hurdle*. Pour cela, on rajoutera comme avec un modèle classique un décalage (*offset*) correspondant au logarithme de la durée d’exposition.

Ce décalage s’ajoute *a minima* à la composante comptage du modèle *zero-inflated* ou du modèle *hurdle*. Toutefois, la probabilité de ne pas vivre l’évènement (donc de zéro) peut elle-même être influencée par la durée d’exposition, auquel cas il pourrait être pertinent d’ajouter également l’*offset* à la composante inflation des zéros du modèle. Certains auteurs suggèrent même d’inclure le logarithme de la durée d’exposition non pas sous forme d’un *offset* mais directement comme une variable explicative du modèle (C. Feng 2022).

45.6 Lectures complémentaires

- *Regression Models for Count Data in R* par Achim Zeileis, Christian Kleiber et Simon Jackman
- *Too many zeros and/or highly skewed? A tutorial on modelling health behaviour as count data with Poisson and negative binomial regression* par James A. Green. DOI : [10.1080/21642850.2021.1920416](https://doi.org/10.1080/21642850.2021.1920416)
- *A comparison of zero-inflated and hurdle models for modeling zero-inflated count data* par Cindy Xin Feng. DOI : [10.1186/s40488-021-00121-4](https://doi.org/10.1186/s40488-021-00121-4)
- *Zero-inflated models for adjusting varying exposures: a cautionary note on the pitfalls of using offset* by Cindy Xin Feng. DOI : [10.1080/02664763.2020.1796943](https://doi.org/10.1080/02664763.2020.1796943)

46 Quel modèle choisir ?

La tableau synthétique ci-dessous permet d'identifier rapidement quelle fonction utiliser en fonction du type de variable à expliquer, du type de distribution et de la famille de modèles (modèle de base, modèle avec prise en compte du plan d'échantillonnage, modèle mixte ou modèle GEE).

Ce tableau est trop grand pour un rendu au format PDF. Merci de le consulter en ligne à l'adresse https://larmarange.github.io/guide-R/analyses_avancees/choix-modele.html.

partie VII

Pour aller plus loin

47 Ressources documentaires

Dans ce dernier chapitre, nous listons quelques ressources documentaires, notamment sur des domaines non couverts par *guide-R*.

47.1 Ressources génériques

- [Introduction à R et au tidyverse](#) de Julien Barnier
- [Awesome R](#) : cette liste recense des ressources sur différents domaines
- [Rzine](#) : un site collaboratif et interdisciplinaire de référencement et de partage de documentation sur la pratique de **R** en sciences humaines et sociales
- [Rseek](#) : un moteur de recherche restreignant les résultats à un corpus de sources sur **R**
- [R for data science](#) par Hadley Wickham, Mine Çetinkaya-Rundel et Garret Grolemund
- [UtilitR](#), une documentation sur **R** née à l'INSEE
- [Notes de cours de R](#) par Ewan Gallic
- [Programmer en R](#), un wikibook collaboratif
- [Introduction à la programmation en R](#) par Vincent Goulet
- [Happy Git and GitHub for the useR](#) par Jennifer Bryan
- [Modern Data Science with R](#) par Benjamin S. Baumer, Daniel T. Kaplan, and Nicholas J. Horton
- [The Epidemiologist R Handbook: R for applied epidemiology and public health](#) porté par un collectif d'auteurs
- [R for Non-Programmers: A Guide for Social Scientists](#) par Daniel Dauber

47.2 Analyse de réseaux

- *L'analyse de réseau en sciences sociales. Petit guide pratique* par Laurent Beauguitte, HAL : [hal-04052709](https://hal.archives-ouvertes.fr/hal-04052709)
- *Network visualization with R* par Katherine Ognyanova
- *Awesome Network Analysis list* portée par François Briatte
- *Cours d'introduction à l'analyse de réseaux avec R* par Hugues Pecout et Laurent Beauguitte

47.3 Analyse spatiale & Cartographie

- *Cartographie avec R* et *Géomatique avec R* par Timothée Giraud et Hugues Pecout
- *Données géospatiales et cartographie avec R* par Nicolas Roelandt
- *Geocomputation with R* par Robin Lovelace, Jakub Nowosad et Jannes Muenchow
- *Spatial Modelling for Data Scientists* par Francisco Rowe et Dani Arribas-Bel
- *Spatial Data Science with R and "terra"* par Robert J. Hijmans

47.4 Analyse textuelle

- *Le Descriptoire : Recueil et analyse de texte* avec R par Lise Vaudor
- *Text mining with R* par Julia Silge et David Robinson

Deb, Partha, et Pravin K. Trivedi. 1997. « Demand for Medical Care by the Elderly: A Finite Mixture Approach ». *Journal of Applied Econometrics* 12 (3): 313-36. [https://doi.org/10.1002/\(SICI\)1099-1255\(199705\)12:3%3C313::AID-JAE440%3E3.0.CO;2-G](https://doi.org/10.1002/(SICI)1099-1255(199705)12:3%3C313::AID-JAE440%3E3.0.CO;2-G).

Feng, Cindy. 2022. « Zero-inflated models for adjusting varying exposures: a cautionary note on the pitfalls of using offset ». *Journal of Applied Statistics* 49 (1): 1-23. <https://doi.org/10.1080/02664763.2020.1796943>.

Feng, Cindy Xin. 2021. « A comparison of zero-inflated and hurdle models for modeling zero-inflated count data ». *Journal of Statistical Distributions and Applications* 8 (1): 8. <https://doi.org/10.1186/s40488-021-00121-4>.

Min, Yongyi, et Alan Agresti. 2005. « Random Effect Models for Repeated Measures of Zero-Inflated Count Data ». *Statistical Modelling* 5 (1): 1-19. <https://doi.org/10.1191/1471082X05st084oa>.