Les statistiques avec R

Marc-André Désautels

2024-06-09

Table des matières

Préface

Ceci est un livre Quarto.

Pour en apprendre davantage sur Quarto, visitez https://quarto.org/docs/books.

partie I

Le tidyverse

1 Le tidyverse

Dans ce document, nous utiliserons l'extension `tidyverse`. Ce chapitre permettra d'introduire l'extension `tidyverse` mais surtout les principes qui la sous-tendent.

```
library(tidyverse)
```

Cette commande va en fait charger plusieurs extensions qui constituent le **coeur** du `tidyverse`, à savoir :

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

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

La liste complète des extensions se trouve sur le site officiel du `tidyverse` https://www.tidyverse.org/packages/.

1.1 Extensions

Le terme *tidyverse* est une contraction de *tidy* (qu'on pourrait traduire par *bien rangé*) et de *universe*. En allant visiter le site internet de ces extensions https://www.tidyverse.org/, voici ce que nous pouvons trouver sur la première page du site:

> The tidyverse is an opinionated collection of R packages designed for data science.
> All packages share an underlying design philosophy, grammar, and data structures.

que nous pourrions traduire par:

> Le tidyverse est une collection dogmatique d'extensions pour le langage R conçues pour la science des données. Toutes les extensions partagent une philosphie sous-jacente de design, de grammaire et de structures de données.

Ces extensions abordent un très grand nombre d'opérations courantes dans `R`. L'avantage d'utiliser le `tidyverse` c'est qu'il permet de simplifier plusieurs opérations fréquentes et il introduit le concept de **tidy data**. De plus, la grammaire du `tidyverse` étant cohérente entre toutes ses extensions, en apprenant comment utiliser l'une de ces extensions, vous serez en monde connu lorsque viendra le temps d'apprendre de nouvelles extensions.

Nous utiliserons le `tidyverse` pour:

- Le concept de **tidy data**
- L'importation et/ou l'exportation de données
- La manipulation de variables
- La visualisation

Le `tidyverse` permet aussi de:

- Travailler avec des chaînes de caractères (du texte par exemple)
- Programmer
- Remettre en forme des données
- Extraire des données du Web
- Etc.

Pour en savoir plus, nous invitons le lecteur à se rendre au site du `tidyverse` https://www.tidyverse.org/. Le `tidyverse` est en grande partie issu des travaux de Hadley Wickham.

1.2 Les tidy data

Le `tidyverse` est en partie fondé sur le concept de *tidy data*, développé à l'origine par Hadley Wickham dans un article du *Journal of Statistical Software*. Nous pourrions traduire ce concept par *données bien rangées*.

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. Dans ce livre, nous travaillerons toujours avec des *tidy data*. En réalité, la plupart des données rencontrées par les chercheurs ne sont pas *tidy*. Il existe une extension du `tidyverse` qui permet de faciliter la transformation de données *non tidy* en données *tidy*, l'extension `tidyr`. Nous ne verrons pas comment l'utiliser dans ce livre.

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 valeur doit être dans une cellule différente

La Figure 1.1 montre ces règles de façon visuelle.

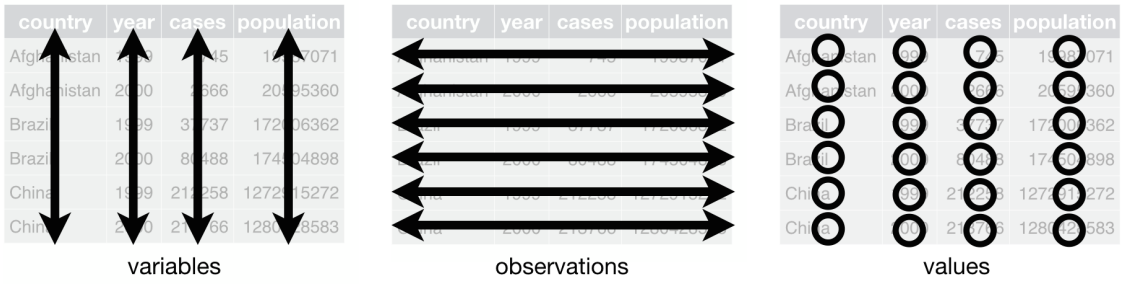


Figure 1.1: Suivre les trois principes rend les données tidy: les variables sont en colonnes, les observations sont sur des lignes, et chaques valeurs sont dans des cellules différentes.

Pourquoi s'assurer que vos données sont *tidy*? Il y a deux avantages importants:

1. Un avantage général de choisir une seule façon de conserver vos données. Si vous utilisez une structure de données consitante, il est plus facile d'apprendre à utiliser les outils qui fonctionneront avec ce type de structure, étant donné que celles-ci possède une uniformité sous-jacente.

2. Un avantage spécifique de placer les variables en colonnes car ceci permet de *vectoriser* les opérations dans `R`. Ceci implique que vos fonctions seront plus rapides lorsque viendra le temps de les exécuter.

Voici un exemple de données *tidy* qui sont accessibles dans la librairie `tidyverse`.

```
diamonds
#> # A tibble: 53,940 x 10
#>    carat cut       color clarity depth table price     x     y     z
#>    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#>  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
#>  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
#>  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
#>  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
#>  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
#>  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
#>  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
#>  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
```

```
#>  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
#> 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
#> # i 53,930 more rows
```

1.2.1 La base de données flights

Pour visualiser facilement une base de données sous forme **tibble**, il suffit de taper son nom dans la console. Nous allons utiliser la base de données flights. Par exemple:

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

Nous allons décortiquer la sortie console:

- `A tibble: 336,776 x 19`: un **tibble** est une façon de représenter une base de données en R. Cette base de données possède:
  - `336 776` lignes
  - `19` colonnes correspondant aux 19 variables décrivant chacune des observations
- `year month day dep_time sched_dep_time dep_delay arr_time` sont différentes colonnes, en d'autres mots des variables, de cette base de données.
- Nous avons ensuite 10 lignes d'obervations correspondant à 10 vols
- `... with 336,766 more rows, and 12 more variables:` nous indique que 336 766 lignes et 12 autres variables ne pouvaient pas être affichées à l'écran.

Malheureusement cette sortie écran ne nous permet pas d'explorer les données correctement. Nous verrons à la section Section 1.2.2 comment explorer des **tibbles**.

1.2.2 Comment explorer des "tibbles"

Voici les façons les plus communes de comprendre les données se trouvant à l'intérieur d'un **tibble**:

- En utilisant la fonction `View()` de RStudio.C'est la commande que nous utiliserons le plus fréquemment.
- En utilisant la fonction `glimpse()`.
- En utilisant l'opérateur `$` pour étudier une seule variable d'une base de données.

Voici comment utiliser ces fonctions.

1. `View()`: Éxécutez `View(flights)` dans la console de RStudio et explorez la base de données obtenue.

Nous remarquons que chaque colonnes représentent une variable différente et que ces variables peuvent être de différents types. Certaines de ces variables, comme `distance`, `day` et `arr_delay` sont des variables dites quantitatives. Ces variables sont numériques par nature. D'autres variables sont dites qualitatives.

Si vous regardez la colonne à l'extrême-gauche de la sortie de `View(flights)`, vous verrez une colonne de nombres. Ces nombres représentent les numéros de ligne de la base de données. Si vous vous promenez sur une ligne de même nombre, par exemple la ligne 5, vous étudiez une unité statistique.

2. `glimpse`:

La seconde façon d'explorer une base de données est d'utiliser la fonction `glimpse()`. Cette fonction nous donne la majorité de l'information précédente et encore plus.

```
glimpse(flights)
#> Rows: 336,776
#> Columns: 19
#> $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
#> $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
#> $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
#> $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
#> $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
#> $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
#> $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
#> $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
```

```
#> $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
#> $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
#> $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
#> $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
#> $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
#> $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
#> $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
#> $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
#> $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
#> $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
#> $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0~
```

3. L'opérateur `$`:

Finalement, l'opérateur `$` nous permet d'explorer une seule variable à l'intérieur d'une base de données. Par exemple, si nous désirons étudier la variable `name` de la base de données `airlines`, nous obtenons:

```
airlines$name
#>  [1] "Endeavor Air Inc."           "American Airlines Inc."
#>  [3] "Alaska Airlines Inc."        "JetBlue Airways"
#>  [5] "Delta Air Lines Inc."        "ExpressJet Airlines Inc."
#>  [7] "Frontier Airlines Inc."      "AirTran Airways Corporation"
#>  [9] "Hawaiian Airlines Inc."      "Envoy Air"
#> [11] "SkyWest Airlines Inc."       "United Air Lines Inc."
#> [13] "US Airways Inc."             "Virgin America"
#> [15] "Southwest Airlines Co."      "Mesa Airlines Inc."
```

2 Manipuler les données

`dplyr` est une extension facilitant le traitement et la manipulation de données contenues dans une ou plusieurs tables. Elle propose une syntaxe claire et cohérente, sous formes de verbes, pour la plupart des opérations de ce type.

`dplyr` part du principe que les données sont organisées selon le modèle des *tidy data* (voir Section 1.2). Les fonctions de l'extension peuvent s'appliquer à des tableaux de type `data.frame` ou `tibble`, et elles retournent systématiquement un `tibble`).

Le code présent dans ce document nécessite d'avoir installé la version 1.0 de `dplyr` (ou plus récente).

2.1 Préparation

`dplyr` fait partie du coeur du *tidyverse*, elle est donc chargée automatiquement avec :

```
library(tidyverse)
```

On peut également la charger individuellement.

```
library(dplyr)
```

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

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

On va charger les trois tables du jeu de données :

```r
library(nycflights13)
## Chargement des trois tables
data(flights)
data(airports)
data(airlines)
```

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

2.2 Les verbes de `dplyr`

La manipulation de données avec **`dplyr`** se fait en utilisant un nombre réduit de verbes, qui correspondent chacun à une action différente appliquée à un tableau de données.

2.2.1 `slice`

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

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

```r
slice(airports, 345)
#> # A tibble: 1 x 8
#>   faa   name               lat   lon   alt    tz dst   tzone
#>   <chr> <chr>            <dbl> <dbl> <dbl> <dbl> <chr> <chr>
#> 1 CYF   Chefornak Airport  60.1 -164.    40    -9 A     America/Anchorage
```

Si on veut sélectionner les 5 premières lignes :

```r
slice(airports, 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~
```

`slice` propose plusieurs variantes utiles, dont `slice_head` et `slice_tail`, qui permettent de sélectionner les premières ou les dernières lignes du tableau (on peut spécifier le nombre de lignes souhaitées avec `n`, ou la proportion avec `prop`).

```
slice_tail(airports, n = 3)
#> # A tibble: 3 x 8
#>   faa   name                       lat   lon   alt    tz dst   tzone
#>   <chr> <chr>                    <dbl> <dbl> <dbl> <dbl> <chr> <chr>
#> 1 ZWI   Wilmington Amtrak Station  39.7 -75.6     0    -5 A     America/New_York
#> 2 ZWU   Washington Union Station   38.9 -77.0    76    -5 A     America/New_York
#> 3 ZYP   Penn Station               40.8 -74.0    35    -5 A     America/New_York
```

```
slice_head(airlines, prop = 0.2)
#> # A tibble: 3 x 2
#>   carrier name
#>   <chr>   <chr>
#> 1 9E      Endeavor Air Inc.
#> 2 AA      American Airlines Inc.
#> 3 AS      Alaska Airlines Inc.
```

Autres variantes utiles, `slice_min` et `slice_max` permettent de sélectionner les lignes avec les valeurs les plus grandes ou les plus petite d'une variable donnée. Ainsi, la commande suivante sélectionne le vol ayant le retard au départ le plus faible.

```
slice_min(flights, dep_delay)
#> # A tibble: 1 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
#> # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

On peut aussi spécifier le nombre de lignes souhaitées, par exemple la commande suivante retourne les 5 aéroports avec l'altitude la plus élevée (en cas de valeurs ex-aequo, il se peut que le nombre de lignes retournées soit plus élevé que celui demandé).

```
slice_max(airports, alt, n = 5)
#> # A tibble: 5 x 8
#>   faa   name                          lat   lon   alt    tz dst   tzone
#>   <chr> <chr>                       <dbl> <dbl> <dbl> <dbl> <chr> <chr>
```

```
#> 1 TEX   Telluride                      38.0 -108.  9078    -7 A     America/D~
#> 2 TVL   Lake Tahoe Airport             38.9 -120.  8544    -8 A     America/L~
#> 3 ASE   Aspen Pitkin County Sardy Field 39.2 -107.  7820    -7 A     America/D~
#> 4 GUC   Gunnison - Crested Butte       38.5 -107.  7678    -7 A     America/D~
#> 5 BCE   Bryce Canyon                   37.7 -112.  7590    -7 A     America/D~
```

2.2.2 `filter`

`filter` sélectionne des lignes d'une table selon une condition. On lui passe en paramètre un test, et seules les lignes pour lesquelles ce test renvoie `TRUE` (vrai) sont conservées. Pour plus d'informations sur les tests et leur syntaxe, voir Chapitre 9.

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 <dttm>
```

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

```
filter(flights, dep_delay >= 10 & dep_delay <= 15)
#> # A tibble: 14,919 x 19
#>     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
```

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

Si on passe plusieurs arguments à `filter`, celui-ci rajoute automatiquement une condition *et* entre les conditions. La commande précédente peut donc être écrite de la manière suivante, avec le même résultat :

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

On peut également placer des fonctions dans les tests, qui nous permettent par exemple de sélectionner les vols ayant une distance supérieure à la distance médiane :

```
filter(flights, distance > median(distance))
#> # A tibble: 167,133 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      555            600        -5      913            854
#>  6  2013     1     1      557            600        -3      838            846
#>  7  2013     1     1      558            600        -2      849            851
#>  8  2013     1     1      558            600        -2      853            856
#>  9  2013     1     1      558            600        -2      924            917
#> 10  2013     1     1      558            600        -2      923            937
#> # i 167,123 more rows
#> # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

2.2.3 `select` et `rename`

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

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

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

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

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

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

La syntaxe `colonne1:colonne2` permet de sélectionner toutes les colonnes situées entre `colonne1` et `colonne2` incluses[1].

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

`select` propose de nombreuses autres possibilités de sélection qui sont décrites dans la documentation de l'extension `tidyselect`.

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

Une variante de `select` est `rename`[2], qui permet de renommer 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` :

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

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

```
tmp <- rename(
    flights,
    "retard départ" = dep_delay,
    "retard arrivée" = arr_delay
)
select(tmp, `retard départ`, `retard arrivée`)
#> # A tibble: 336,776 x 2
#>    `retard départ` `retard arrivée`
#>              <dbl>            <dbl>
#>  1               2               11
#>  2               4               20
#>  3               2               33
#>  4              -1              -18
#>  5              -6              -25
#>  6              -4               12
#>  7              -5               19
#>  8              -3              -14
```

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

```
#>  9              -3               -8
#> 10              -2                8
#> # i 336,766 more rows
```

2.2.4 `arrange`

`arrange` réordonne les lignes d'un tableau selon une ou plusieurs colonnes.

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

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

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

```
arrange(flights, month, dep_delay)
#> # A tibble: 336,776 x 19
#>     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#>  1  2013     1    11     1900           1930       -30     2233           2243
#>  2  2013     1    29     1703           1730       -27     1947           1957
#>  3  2013     1    12     1354           1416       -22     1606           1650
#>  4  2013     1    21     2137           2159       -22     2232           2316
#>  5  2013     1    20      704            725       -21     1025           1035
```

```
#>  6  2013     1    12     2050           2110       -20     2310           2355
#>  7  2013     1    12     2134           2154       -20        4             50
#>  8  2013     1    14     2050           2110       -20     2329           2355
#>  9  2013     1     4     2140           2159       -19     2241           2316
#> 10  2013     1    11     1947           2005       -18     2209           2230
#> # i 336,766 more rows
#> # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

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

```
arrange(flights, desc(dep_delay))
#> # A tibble: 336,776 x 19
#>     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#>  1  2013     1     9      641            900      1301     1242           1530
#>  2  2013     6    15     1432           1935      1137     1607           2120
#>  3  2013     1    10     1121           1635      1126     1239           1810
#>  4  2013     9    20     1139           1845      1014     1457           2210
#>  5  2013     7    22      845           1600      1005     1044           1815
#>  6  2013     4    10     1100           1900       960     1342           2211
#>  7  2013     3    17     2321            810       911      135           1020
#>  8  2013     6    27      959           1900       899     1236           2226
#>  9  2013     7    22     2257            759       898      121           1026
#> 10  2013    12     5      756           1700       896     1058           2020
#> # i 336,766 more rows
#> # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

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

```
tmp <- arrange(flights, desc(dep_delay))
slice(tmp, 1:3)
#> # A tibble: 3 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     9      641            900      1301     1242           1530
#> 2  2013     6    15     1432           1935      1137     1607           2120
```

```
#> 3  2013     1    10     1121           1635      1126     1239           1810
#> # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

2.2.5 `mutate`

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

Par exemple, la table `flights` contient la durée du vol en minutes.. Si on veut créer une nouvelle variable `duree_h` avec cette durée en heures, on peut faire :

```
flights <- mutate(flights, duree_h = air_time / 60)

select(flights, air_time, duree_h)
#> # A tibble: 336,776 x 2
#>    air_time duree_h
#>       <dbl>   <dbl>
#>  1      227   3.78
#>  2      227   3.78
#>  3      160   2.67
#>  4      183   3.05
#>  5      116   1.93
#>  6      150   2.5
#>  7      158   2.63
#>  8       53   0.883
#>  9      140   2.33
#> 10      138   2.3
#> # i 336,766 more rows
```

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

```
flights <- mutate(
    flights,
    duree_h = air_time / 60,
    distance_km = distance / 0.62137,
```

```
    vitesse = distance_km / duree_h
)

select(flights, air_time, duree_h, distance, distance_km, vitesse)
#> # A tibble: 336,776 x 5
#>    air_time duree_h distance distance_km vitesse
#>       <dbl>   <dbl>    <dbl>       <dbl>   <dbl>
#>  1      227   3.78      1400       2253.    596.
#>  2      227   3.78      1416       2279.    602.
#>  3      160   2.67      1089       1753.    657.
#>  4      183   3.05      1576       2536.    832.
#>  5      116   1.93       762       1226.    634.
#>  6      150   2.5        719       1157.    463.
#>  7      158   2.63      1065       1714.    651.
#>  8       53   0.883      229        369.    417.
#>  9      140   2.33       944       1519.    651.
#> 10      138   2.3        733       1180.    513.
#> # i 336,766 more rows
```

À noter que `mutate` est évidemment parfaitement compatible avec les fonctions vues **?@sec-vectorfactor** sur les recodages : `fct_recode`, `ifelse`, `case_when`…

L'avantage d'utiliser `mutate` est double. D'abord il permet d'éviter d'avoir à saisir le nom du tableau de données dans les conditions d'un `ifelse` ou d'un `case_when` :

```
flights <- mutate(
    flights,
    type_retard = case_when(
        dep_delay > 0 & arr_delay > 0 ~ "Retard départ et arrivée",
        dep_delay > 0 & arr_delay <= 0 ~ "Retard départ",
        dep_delay <= 0 & arr_delay > 0 ~ "Retard arrivée",
        TRUE ~ "Aucun retard"
    )
)
```

Ensuite, il permet aussi d'intégrer ces recodages dans un *pipeline* de traitement de données, concept présenté dans la section suivante.

2.3 Enchaîner les opérations avec le *pipe*

Quand on manipule un tableau de données, il est très fréquent d'enchaîner plusieurs opérations. On va par exemple extraire une sous-population avec `filter`, sélectionner des colonnes avec

`select` puis trier selon une variable avec `arrange`, etc.

Quand on veut enchaîner des opérations, on peut le faire de différentes manières. La première est d'effectuer toutes les opérations en une fois en les "emboîtant" :

```
arrange(select(filter(flights, dest == "LAX"), dep_delay, arr_delay), dep_delay)
```

Cette notation a plusieurs inconvénients :

- elle est peu lisible
- les opérations apparaissent dans l'ordre inverse de leur réalisation. Ici on effectue d'abord le `filter`, puis le `select`, puis le `arrange`, alors qu'à la lecture du code c'est le `arrange` qui apparaît en premier.
- Il est difficile de voir quel paramètre se rapporte à quelle fonction

Une autre manière de faire est d'effectuer les opérations les unes après les autres, en stockant les résultats intermédiaires dans un objet temporaire :

```
tmp <- filter(flights, dest == "LAX")
tmp <- select(tmp, dep_delay, arr_delay)
arrange(tmp, dep_delay)
```

C'est nettement plus lisible, l'ordre des opérations est le bon, et les paramètres sont bien rattachés à leur fonction. Par contre, ça reste un peu "verbeux", et on crée un objet temporaire `tmp` dont on n'a pas réellement besoin.

Pour simplifier et améliorer encore la lisibilité du code, on va utiliser un nouvel opérateur, baptisé *pipe*[3]. Le *pipe* se note `%>%`, et son fonctionnement est le suivant : si j'exécute `expr %>% f`, alors le résultat de l'expression `expr`, à gauche du *pipe*, sera passé comme premier argument à la fonction `f`, à droite du *pipe*, ce qui revient à exécuter `f(expr)`.

Ainsi les deux expressions suivantes sont rigoureusement équivalentes :

```
filter(flights, dest == "LAX")
```

```
flights %>% filter(dest == "LAX")
```

Ce qui est particulièrement intéressant, c'est qu'on va pouvoir enchaîner les *pipes*. Plutôt que d'écrire :

[3]Le *pipe* a été introduit à l'origine par l'extension `magrittr`, et repris par `dplyr`

```
select(filter(flights, dest == "LAX"), dep_delay, arr_delay)
```

On va pouvoir faire :

```
flights %>% filter(dest == "LAX") %>% select(dep_delay, arr_delay)
```

À chaque fois, le résultat de ce qui se trouve à gauche du *pipe* est passé comme premier argument à ce qui se trouve à droite : on part de l'objet `flights`, qu'on passe comme premier argument à la fonction `filter`, puis on passe le résultat de ce `filter` comme premier argument du `select`.

Le résultat final est le même avec les deux syntaxes, mais avec le *pipe* l'ordre des opérations correspond à l'ordre naturel de leur exécution, et on n'a pas eu besoin de créer d'objet intermédiaire.

Si la liste des fonctions enchaînées est longue, on peut les répartir sur plusieurs lignes à condition que l'opérateur `%>%` soit en fin de ligne :

```
flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay) %>%
  arrange(dep_delay)
```

> **Note**
>
> On appelle une suite d'instructions de ce type un *pipeline*.

Évidemment, il est naturel de vouloir récupérer le résultat final d'un *pipeline* pour le stocker dans un objet. On peut stocker le résultat du *pipeline* ci-dessus dans un nouveau tableau `delay_la` de la manière suivante :

```
delay_la <- flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay) %>%
  arrange(dep_delay)
```

Dans ce cas, `delay_la` contiendra le tableau final, obtenu après application des trois instructions `filter`, `select` et `arrange`.

Cette notation n'est pas forcément très intuitive au départ : il faut bien comprendre que c'est le résultat final, une fois application de toutes les opérations du *pipeline*, qui est renvoyé et stocké dans l'objet en début de ligne.

Une manière de le comprendre peut être de voir que la notation suivante :

```r
delay_la <- flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay)
```

est équivalente à :

```r
delay_la <- (flights %>% filter(dest == "LAX") %>% select(dep_delay, arr_delay))
```

> **Note**
>
> L'utilisation du *pipe* n'est pas obligatoire, mais elle rend les scripts plus lisibles et plus rapides à saisir. On l'utilisera donc dans ce qui suit.

> **Avertissement**
>
> Depuis la version 4.1, R propose un *pipe* "natif", qui fonctionne partout, même si on n'utilise pas les extensions du *tidyverse*. Celui-ci est noté `|>`.
> Il s'utilise de la même manière que `%>%` :
> `flights |> filter(dest == "LAX")`
> Ce *pipe* natif est à la fois un peu plus rapide et un peu moins souple. Par exemple, il est possible avec `%>%` d'appeler une fonction sans mettre de parenthèses :
> `df %>% View`
> Ce n'est pas possible d'omettre les parenthèses avec `|>`, on doit obligatoirement faire :
> `df |> View()`
> Dans la suite de ce document on privilégiera (pour l'instant) le *pipe* du *tidyverse* `%>%`, pour des raisons de compatibilité avec des versions de R moins récentes.

2.4 Opérations groupées

2.4.1 `group_by`

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

```r
flights %>% group_by(month)
#> # A tibble: 336,776 x 22
#> # Groups:   month [12]
#>     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
```

```
#>    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#>  1  2013     1     1      517            515         2      830            819
#>  2  2013     1     1      533            529         4      850            830
#>  3  2013     1     1      542            540         2      923            850
#>  4  2013     1     1      544            545        -1     1004           1022
#>  5  2013     1     1      554            600        -6      812            837
#>  6  2013     1     1      554            558        -4      740            728
#>  7  2013     1     1      555            600        -5      913            854
#>  8  2013     1     1      557            600        -3      709            723
#>  9  2013     1     1      557            600        -3      838            846
#> 10  2013     1     1      558            600        -2      753            745
#> # i 336,766 more rows
#> # i 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>
```

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

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

```
flights %>% group_by(month) %>% slice(1)
#> # A tibble: 12 x 22
#> # Groups:   month [12]
#>     year month   day dep_time sched_dep_time dep_delay arr_time 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 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>
```

Plus utile, en utilisant une variante comme `slice_min` ou `slice_max`, on peut sélectionner les lignes ayant les valeurs les plus grandes ou les plus petites *pour chaque groupe*. Par exemple la commande suivant sélectionne, pour chaque mois de l'année, le vol ayant eu le retard le plus important.

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

Idem pour `mutate` : les opérations appliquées lors du calcul des valeurs des nouvelles colonnes sont appliquées groupe de lignes par groupe de lignes. Dans l'exemple suivant, on ajoute une nouvelle colonne qui contient le retard moyen *pour chaque compagnie aérienne*. Cette valeur est donc différente d'une compagnie à une autre, mais identique pour tous les vols d'une même compagnie :

```
flights %>%
  group_by(carrier) %>%
  mutate(mean_delay_carrier = mean(dep_delay, na.rm = TRUE)) %>%
```

```
  select(dep_delay, mean_delay_carrier)
#> Adding missing grouping variables: `carrier`
#> # A tibble: 336,776 x 3
#> # Groups:   carrier [16]
#>    carrier dep_delay mean_delay_carrier
#>    <chr>       <dbl>              <dbl>
#>  1 UA              2              12.1
#>  2 UA              4              12.1
#>  3 AA              2               8.59
#>  4 B6             -1              13.0
#>  5 DL             -6               9.26
#>  6 UA             -4              12.1
#>  7 B6             -5              13.0
#>  8 EV             -3              20.0
#>  9 B6             -3              13.0
#> 10 AA             -2               8.59
#> # i 336,766 more rows
```

Ceci peut permettre, par exemple, de déterminer si un retard donné est supérieur ou inférieur au retard médian de la compagnie :

```
flights %>%
  group_by(carrier) %>%
  mutate(
      median_delay = median(dep_delay, na.rm = TRUE),
      delay_carrier = ifelse(
          dep_delay > median_delay,
          "Supérieur",
          "Inférieur ou égal"
      )
  ) %>%
  select(dep_delay, median_delay, delay_carrier)
#> Adding missing grouping variables: `carrier`
#> # A tibble: 336,776 x 4
#> # Groups:   carrier [16]
#>    carrier dep_delay median_delay delay_carrier
#>    <chr>       <dbl>        <dbl> <chr>
#>  1 UA              2            0 Supérieur
#>  2 UA              4            0 Supérieur
#>  3 AA              2           -3 Supérieur
#>  4 B6             -1           -1 Inférieur ou égal
#>  5 DL             -6           -2 Inférieur ou égal
```

```
#>  6 UA             -4            0 Inférieur ou égal
#>  7 B6             -5           -1 Inférieur ou égal
#>  8 EV             -3           -1 Inférieur ou égal
#>  9 B6             -3           -1 Inférieur ou égal
#> 10 AA             -2           -3 Supérieur
#> # i 336,766 more rows
```

`group_by` peut aussi être utile avec `filter`, par exemple pour sélectionner *pour chaque mois* les vols avec un retard au départ plus élevé que le retard moyen ce mois-ci.

```
flights %>%
  group_by(month) %>%
  filter(dep_delay >= mean(dep_delay, na.rm = TRUE))
#> # A tibble: 78,164 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      611            600        11      945            931
#>  2  2013     1     1      623            610        13      920            915
#>  3  2013     1     1      632            608        24      740            728
#>  4  2013     1     1      732            645        47     1011            941
#>  5  2013     1     1      743            730        13     1107           1100
#>  6  2013     1     1      743            730        13     1059           1056
#>  7  2013     1     1      749            710        39      939            850
#>  8  2013     1     1      811            630       101     1047            830
#>  9  2013     1     1      826            715        71     1136           1045
#> 10  2013     1     1      848           1835       853     1001           1950
#> # i 78,154 more rows
#> # i 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>
```

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

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

```r
flights %>%
  group_by(month) %>%
  arrange(desc(dep_delay))
#> # A tibble: 336,776 x 22
#> # Groups:   month [12]
#>     year month   day dep_time sched_dep_time dep_delay arr_time 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 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>
```

```r
flights %>%
  group_by(month) %>%
  arrange(desc(dep_delay), .by_group = TRUE)
#> # A tibble: 336,776 x 22
#> # Groups:   month [12]
#>     year month   day dep_time sched_dep_time dep_delay arr_time 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 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
```

```
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>
```

2.4.2 `summarise` et `count`

`summarise` permet d'agréger les lignes du tableau en effectuant une opération "résumée" sur une ou plusieurs colonnes. Par exemple, si on souhaite connaître les retards moyens au départ et à l'arrivée pour l'ensemble des vols du tableau `flights` :

```
flights %>%
  summarise(
      retard_dep = mean(dep_delay, na.rm = TRUE),
      retard_arr = mean(arr_delay, na.rm = TRUE)
  )
#> # A tibble: 1 x 2
#>   retard_dep retard_arr
#>        <dbl>      <dbl>
#> 1       12.6       6.90
```

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

```
flights %>%
  group_by(month) %>%
  summarise(
      max_delay = max(dep_delay, na.rm = TRUE),
      min_delay = min(dep_delay, na.rm = TRUE),
      mean_delay = mean(dep_delay, na.rm = TRUE)
  )
#> # A tibble: 12 x 4
#>    month max_delay min_delay mean_delay
#>    <int>     <dbl>     <dbl>      <dbl>
#>  1     1      1301       -30       10.0
#>  2     2       853       -33       10.8
#>  3     3       911       -25       13.2
#>  4     4       960       -21       13.9
#>  5     5       878       -24       13.0
#>  6     6      1137       -21       20.8
#>  7     7      1005       -22       21.7
```

```
#>  8     8       520       -26      12.6
#>  9     9      1014       -24       6.72
#> 10    10       702       -25       6.24
#> 11    11       798       -32       5.44
#> 12    12       896       -43      16.6
```

`summarise` dispose d'un opérateur spécial, `n()`, qui retourne le nombre de lignes du groupe. Ainsi si on veut le nombre de vols par destination, on peut utiliser :

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

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

À noter que quand on veut compter le nombre de lignes par groupe, il est plus simple d'utiliser directement la fonction `count`. Ainsi le code suivant est identique au précédent :

```
flights %>%
  count(dest)
#> # A tibble: 105 x 2
#>    dest      n
#>    <chr> <int>
#>  1 ABQ     254
#>  2 ACK     265
#>  3 ALB     439
#>  4 ANC       8
#>  5 ATL   17215
```

```
#>  6 AUS    2439
#>  7 AVL     275
#>  8 BDL     443
#>  9 BGR     375
#> 10 BHM     297
#> # i 95 more rows
```

2.4.3 Grouper selon plusieurs variables

On peut grouper selon plusieurs variables à la fois, il suffit de les indiquer dans la clause du `group_by`. Le *pipeline* suivant calcule le retard moyen au départ pour chaque mois et pour chaque destination, et trie le résultat par retard décroissant :

```
flights %>%
  group_by(month, dest) %>%
  summarise(retard_moyen = mean(dep_delay, na.rm = TRUE)) %>%
  arrange(desc(retard_moyen))
#> `summarise()` has grouped output by 'month'. You can override using the
#> `.groups` argument.
#> # A tibble: 1,113 x 3
#> # Groups:   month [12]
#>    month dest  retard_moyen
#>    <int> <chr>        <dbl>
#>  1    12 BZN           75
#>  2     7 TUL           72.6
#>  3     3 DSM           71.0
#>  4     7 CAE           69.4
#>  5    11 SBN           67.5
#>  6     7 BHM           64.6
#>  7     7 TYS           60.6
#>  8     6 BHM           57.2
#>  9     1 TUL           55.2
#> 10     1 SAV           54.8
#> # i 1,103 more rows
```

On peut également utiliser `count` sur plusieurs variables. Les commandes suivantes comptent le nombre de vols pour chaque couple aéroport de départ / aéroport d'arrivée, et trie le résultat par nombre de vols décroissant. Ici la colonne qui contient le nombre de vols, créée par `count`, s'appelle `n` par défaut :

```r
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 on souhaite déterminer le couple aéroport de départ / aéroport d'arrivée ayant le retard moyen au départ le plus élevé pour chaque mois de l'année, on devra procéder en deux étapes :

- d'abord grouper selon mois, aéroports d'origine et d'arrivée pour calculer le retard moyen
- puis grouper uniquement selon le mois pour sélectionner le mois avec le retard moyen maximal.

Au final, on obtient le code suivant :

```r
flights %>%
  group_by(month, origin, dest) %>%
  summarise(retard_moyen = mean(dep_delay, na.rm = TRUE)) %>%
  group_by(month) %>%
  slice_max(retard_moyen)
#> `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  retard_moyen
#>    <int> <chr>  <chr>        <dbl>
#>  1     1 EWR    TUL           55.2
#>  2     2 EWR    DSM           48.6
#>  3     3 EWR    DSM           71.0
```

```
#>  4     4 EWR    OKC           47.0
#>  5     5 EWR    TYS           60.6
#>  6     6 EWR    TYS           68.2
#>  7     7 EWR    CAE           81.5
#>  8     8 LGA    GSO           50.1
#>  9     9 LGA    MSN           24.7
#> 10    10 EWR    CAE           50.1
#> 11    11 LGA    SBN           67.5
#> 12    12 EWR    BZN           75
```

2.4.4 Dégroupage

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

```
flights %>%
  group_by(month, origin, dest) %>%
  summarise(retard_moyen = mean(dep_delay, na.rm = TRUE))
#> `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  retard_moyen
#>    <int> <chr>  <chr>        <dbl>
#>  1     1 EWR    ALB          41.4
#>  2     1 EWR    ATL           8.07
#>  3     1 EWR    AUS           6.67
#>  4     1 EWR    AVL          25.5
#>  5     1 EWR    BDL          21.1
#>  6     1 EWR    BNA          16.3
#>  7     1 EWR    BOS           8.99
#>  8     1 EWR    BQN          12.3
#>  9     1 EWR    BTV          20.5
#> 10     1 EWR    BUF          23.1
#> # i 2,303 more rows
```

`dplyr` nous le signale d'ailleurs via un message d'avertissement : `summarise() has grouped output by 'month', 'origin'`.

Ce dégroupage progressif peut permettre "d'enchaîner" les opérations groupées. Dans l'exemple suivant on calcule le retard moyen au départ par destination et on conserve les trois retards les plus importants *pour chaque mois*.

```
flights %>%
  group_by(month, dest) %>%
  summarise(retard_moyen = mean(dep_delay, na.rm = TRUE)) %>%
  slice_max(retard_moyen, n = 3)
#> `summarise()` has grouped output by 'month'. You can override using the
#> `.groups` argument.
#> # A tibble: 36 x 3
#> # Groups:   month [12]
#>    month dest  retard_moyen
#>    <int> <chr>        <dbl>
#>  1     1 TUL           55.2
#>  2     1 SAV           54.8
#>  3     1 DSM           42.2
#>  4     2 DSM           48.6
#>  5     2 TUL           34.2
#>  6     2 GSP           32.4
#>  7     3 DSM           71.0
#>  8     3 PVD           47.5
#>  9     3 CAE           46.9
#> 10     4 OKC           47.0
#> # i 26 more rows
```

On peut à tout moment "dégrouper" un tableau à l'aide de `ungroup`. C'est nécessaire, dans l'exemple précédent, si on veut seulement récupérer les trois retards les plus importants pour l'ensemble des couples mois / destination.

```
flights %>%
  group_by(month, dest) %>%
  summarise(retard_moyen = mean(dep_delay, na.rm = TRUE)) %>%
  ungroup() %>%
  slice_max(retard_moyen, n = 3)
#> `summarise()` has grouped output by 'month'. You can override using the
#> `.groups` argument.
#> # A tibble: 3 x 3
#>   month dest  retard_moyen
#>   <int> <chr>        <dbl>
#> 1    12 BZN           75
#> 2     7 TUL           72.6
#> 3     3 DSM           71.0
```

On peut aussi spécifier précisément le comportement de dégroupage de `summarise` en lui fournissant un argument supplémentaire `.groups` qui peut prendre notamment les valeurs suivantes :

- `"drop_last"` : dégroupe seulement de la dernière variable de groupage
- `"drop"` : dégroupe totalement le tableau résultat (équivaut à l'application d'un `ungroup`)
- `"keep"` : conserve toutes les variables de groupage

Ce concept de dégroupage successif peut être un peu déroutant de prime abord. Il est donc utile de faire attention aux avertissements affichés par ces opérations, et il ne faut pas hésiter à ajouter un `ungroup` en fin de pipeline si on sait qu'on ne souhaite pas utiliser les groupes encore existants par la suite.

À noter que la fonction `count`, de son côté, renvoie un tableau non groupé.

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

2.5 Autres fonctions utiles

`dplyr` contient beaucoup d'autres fonctions utiles pour la manipulation de données.

2.5.1 `slice_sample`

Ce verbe permet de sélectionner aléatoirement un nombre de lignes (avec l'argument `n`) ou une fraction des lignes (avec l'argument `prop`) d'un tableau.

Ainsi si on veut choisir 5 lignes au hasard dans le tableau `airports` :

```
airports %>% slice_sample(n = 5)
#> # A tibble: 5 x 8
#>   faa   name                        lat    lon   alt    tz dst   tzone
#>   <chr> <chr>                     <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
#> 1 MMV   Mc Minnville Muni          45.2 -123.    163    -8 A     America/Los_Ang~
#> 2 FST   Fort Stockton Pecos Co     30.9 -103.   3011    -6 A     America/Chicago
#> 3 RIC   Richmond Intl              37.5  -77.3   167    -5 A     America/New_York
#> 4 OG6   Williams County Airport    41.5  -84.5   730    -5 A     America/New_York
#> 5 VYS   Illinois Valley Regional   41.4  -89.2   654    -6 A     America/Chicago
```

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

```
flights %>% slice_sample(prop = 0.1)
#> # A tibble: 33,677 x 22
#>     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     5     8     1847           1635       132     2046           1814
#>  2  2013     9    16      852            850         2     1035           1045
#>  3  2013    10    11      817            820        -3     1043           1052
#>  4  2013     6     4      900            900         0     1130           1128
#>  5  2013     5     3     1933           1935        -2     2147           2155
#>  6  2013     6    15      606            600         6      850            900
#>  7  2013    11    17     1215           1219        -4     1410           1416
#>  8  2013     9     5      938            939        -1     1143           1208
#>  9  2013     5    22     1628           1625         3     1811           1835
#> 10  2013    12    21     2107           1959        68     2343           2303
#> # i 33,667 more rows
#> # i 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>
```

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

2.5.2 `lead` et `lag`

`lead` et `lag` permettent de décaler les observations d’une variable d’un cran vers l’arrière (pour `lead`) ou vers l’avant (pour `lag`).

```
lead(1:5)
#> [1]  2  3  4  5 NA
lag(1:5)
#> [1] NA  1  2  3  4
```

Ceci peut être utile pour des données de type "séries temporelles". Par exemple, on peut facilement calculer l'écart entre le retard au départ de chaque vol et celui du vol précédent :

```
flights %>%
  mutate(
      dep_delay_prev = lag(dep_delay),
      dep_delay_diff = dep_delay - dep_delay_prev
  ) %>%
  select(dep_delay_prev, dep_delay, dep_delay_diff)
#> # A tibble: 336,776 x 3
#>    dep_delay_prev dep_delay dep_delay_diff
#>             <dbl>     <dbl>          <dbl>
#>  1             NA         2             NA
#>  2              2         4              2
#>  3              4         2             -2
#>  4              2        -1             -3
#>  5             -1        -6             -5
#>  6             -6        -4              2
#>  7             -4        -5             -1
#>  8             -5        -3              2
#>  9             -3        -3              0
#> 10             -3        -2              1
#> # i 336,766 more rows
```

2.5.3 `distinct` et `n_distinct`

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

```
flights %>%
  select(day, month) %>%
  distinct()
#> # A tibble: 365 x 2
#>      day month
#>    <int> <int>
#>  1     1     1
```

```
#>  2     2     1
#>  3     3     1
#>  4     4     1
#>  5     5     1
#>  6     6     1
#>  7     7     1
#>  8     8     1
#>  9     9     1
#> 10    10     1
#> # 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, `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 22
#>     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 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>
```

La fonction `n_distinct`, elle, renvoie le nombre de valeurs distinctes d'un vecteur. On peut notamment l'utiliser dans un `summarise`.

Dans l'exemple qui suit on calcule, pour les trois aéroports de départ de la table `flights` le nombre de valeurs distinctes de l'aéroport d'arrivée :

```
flights %>%
  group_by(origin) %>%
  summarise(n_dest = n_distinct(dest))
#> # A tibble: 3 x 2
#>   origin n_dest
#>   <chr>   <int>
#> 1 EWR        86
#> 2 JFK        70
#> 3 LGA        68
```

2.5.4 `relocate`

`relocate` peut être utilisé pour réordonner les colonnes d'une table. Par défaut, si on lui passe un ou plusieurs noms de colonnes, `relocate` les place en début de tableau.

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

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

Les arguments supplémentaires `.before` et `.after` permettent de préciser à quel endroit déplacer la ou les colonnes indiquées.

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

2.6 Tables multiples

Le jeu de données `nycflights13` est un exemple de données réparties en plusieurs tables. Ici on en a trois : les informations sur les vols dans `flights`, celles sur les aéroports dans `airports` et celles sur les compagnies aériennes dans `airlines`.

`dplyr` propose différentes fonctions permettant de travailler avec des données structurées de cette manière.

2.6.1 Concaténation : `bind_rows` et `bind_cols`

Les fonctions `bind_rows` et `bind_cols` permettent d'ajouter des lignes (respectivement des colonnes) à une table à partir d'une ou plusieurs autres tables.

L'exemple suivant (certes très artificiel) montre l'utilisation de `bind_rows`. On commence par créer trois tableaux `t1`, `t2` et `t3` :

```
t1 <- airports %>%
  select(faa, name, lat, lon) %>%
  slice(1:2)
t1
#> # A tibble: 2 x 4
#>   faa   name                            lat   lon
#>   <chr> <chr>                         <dbl> <dbl>
#> 1 04G   Lansdowne Airport              41.1 -80.6
#> 2 06A   Moton Field Municipal Airport  32.5 -85.7
```

```
t2 <- airports %>%
  select(faa, name, lat, lon) %>%
  slice(5:6)

t2
#> # A tibble: 2 x 4
#>   faa   name                            lat   lon
#>   <chr> <chr>                         <dbl> <dbl>
#> 1 09J   Jekyll Island Airport          31.1 -81.4
#> 2 0A9   Elizabethton Municipal Airport  36.4 -82.2
```

```
t3 <- airports %>%
  select(faa, name) %>%
  slice(100:101)
t3
#> # A tibble: 2 x 2
#>   faa   name
#>   <chr> <chr>
#> 1 ADW   Andrews Afb
#> 2 AET   Allakaket Airport
```

On concaténe ensuite les trois tables avec `bind_rows` :

```
bind_rows(t1, t2, t3)
#> # A tibble: 6 x 4
#>   faa   name                              lat   lon
#>   <chr> <chr>                           <dbl> <dbl>
#> 1 04G   Lansdowne Airport                41.1 -80.6
#> 2 06A   Moton Field Municipal Airport    32.5 -85.7
#> 3 09J   Jekyll Island Airport            31.1 -81.4
#> 4 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 `NA` sont automatiquement insérées.

Il peut être utile, quand on concatène des lignes, de garder une trace du tableau d'origine de chacune des lignes dans le tableau final. C'est possible grâce à l'argument `.id` de `bind_rows`. On passe à cet argument le nom d'une colonne qui contiendra l'indicateur d'origine des lignes :

```
bind_rows(t1, t2, t3, .id = "source")
#> # A tibble: 6 x 5
#>   source faa   name                              lat   lon
#>   <chr>  <chr> <chr>                           <dbl> <dbl>
#> 1 1      04G   Lansdowne Airport                41.1 -80.6
#> 2 1      06A   Moton Field Municipal Airport    32.5 -85.7
#> 3 2      09J   Jekyll Island Airport            31.1 -81.4
#> 4 2      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 `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
```

`bind_cols` permet de concaténer des colonnes et fonctionne de manière similaire :

```
t1 <- flights %>% slice(1:5) %>% select(dep_delay, dep_time)
t2 <- flights %>% slice(1:5) %>% select(origin, dest)
t3 <- flights %>% slice(1:5) %>% select(arr_delay, arr_time)
bind_cols(t1, t2, t3)
#> # A tibble: 5 x 6
#>   dep_delay dep_time origin dest  arr_delay arr_time
#>       <dbl>    <int> <chr>  <chr>     <dbl>    <int>
#> 1         2      517 EWR    IAH          11      830
#> 2         4      533 LGA    IAH          20      850
#> 3         2      542 JFK    MIA          33      923
#> 4        -1      544 JFK    BQN         -18     1004
#> 5        -6      554 LGA    ATL         -25      812
```

À noter que `bind_cols` associe les lignes uniquement *par position*. Les lignes des différents tableaux associés doivent donc correspondre (et leur nombre doit être identique). Pour associer des tables *par valeur*, on doit utiliser des jointures.

2.6.2 Jointures

2.6.2.1 Clés implicites

Très souvent, les données relatives à une analyse sont réparties dans plusieurs tables différentes. Dans notre exemple, on peut voir que la table `flights` contient le code de la compagnie aérienne du vol dans la variable `carrier` :

```
flights %>% select(carrier)
#> # A tibble: 336,776 x 1
#>    carrier
#>    <chr>
#>  1 UA
#>  2 UA
#>  3 AA
#>  4 B6
#>  5 DL
#>  6 UA
#>  7 B6
```

```
#>  8 EV
#>  9 B6
#> 10 AA
#> # i 336,766 more rows
```

Et que par ailleurs la table `airlines` contient une information supplémentaire relative à ces compagnies, à savoir le nom complet.

```
airlines
#> # A tibble: 16 x 2
#>    carrier name
#>    <chr>   <chr>
#>  1 9E      Endeavor Air Inc.
#>  2 AA      American Airlines Inc.
#>  3 AS      Alaska Airlines Inc.
#>  4 B6      JetBlue Airways
#>  5 DL      Delta Air Lines Inc.
#>  6 EV      ExpressJet Airlines Inc.
#>  7 F9      Frontier Airlines Inc.
#>  8 FL      AirTran Airways Corporation
#>  9 HA      Hawaiian Airlines Inc.
#> 10 MQ      Envoy Air
#> 11 OO      SkyWest Airlines Inc.
#> 12 UA      United Air Lines Inc.
#> 13 US      US Airways Inc.
#> 14 VX      Virgin America
#> 15 WN      Southwest Airlines Co.
#> 16 YV      Mesa Airlines Inc.
```

Il est donc naturel de vouloir associer les deux, ici pour ajouter les noms complets des compagnies à la table `flights`. Pour cela on va effectuer une *jointure* : les lignes d'une table seront associées à une autre en se basant non pas sur leur position, mais sur les valeurs d'une ou plusieurs colonnes. Ces colonnes sont appelées des *clés*.

Pour faire une jointure de ce type, on va utiliser la fonction `left_join` :

```
left_join(flights, airlines)
```

Pour faciliter la lecture, on va afficher seulement certaines colonnes du résultat :

```
left_join(flights, airlines) %>%
  select(month, day, carrier, name)
#> Joining with `by = join_by(carrier)`
#> # A tibble: 336,776 x 4
#>    month   day carrier name
#>    <int> <int> <chr>   <chr>
#>  1     1     1 UA      United Air Lines Inc.
#>  2     1     1 UA      United Air Lines Inc.
#>  3     1     1 AA      American Airlines Inc.
#>  4     1     1 B6      JetBlue Airways
#>  5     1     1 DL      Delta Air Lines Inc.
#>  6     1     1 UA      United Air Lines Inc.
#>  7     1     1 B6      JetBlue Airways
#>  8     1     1 EV      ExpressJet Airlines Inc.
#>  9     1     1 B6      JetBlue Airways
#> 10     1     1 AA      American Airlines Inc.
#> # i 336,766 more rows
```

On voit que la table résultat est bien la fusion des deux tables d'origine selon les valeurs des deux colonnes clés `carrier`. On est parti de la table `flights`, et pour chaque ligne de celle-ci 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**
>
> À noter qu'on peut tout à fait utiliser le *pipe* avec les fonctions de jointure : `flights %>% left_join(airlines)`.

Nous sommes ici dans le cas le plus simple concernant les clés de jointure : les deux clés sont uniques et portent le même nom dans les deux tables. Par défaut, si on ne lui spécifie pas explicitement les clés, `dplyr` fusionne en utilisant l'ensemble des colonnes communes aux deux tables. On peut d'ailleurs voir dans cet exemple qu'un message a été affiché précisant que la jointure s'est bien faite sur la variable `carrier`.

2.6.2.2 Clés explicites

La table `airports`, contient des informations supplémentaires sur les aéroports : nom complet, altitude, position géographique, etc. Chaque aéroport est identifié par un code contenu dans la colonne `faa`.

Si on regarde la table `flights`, on voit que le code d'identification des aéroports apparaît à deux endroits différents : pour l'aéroport de départ dans la colonne `origin`, et pour celui d'arrivée dans la colonne `dest`. On a donc deux clés de jointure possibles, et qui portent un nom différent de la clé de `airports`.

On va commencer par fusionner les données concernant l'aéroport de départ. Pour simplifier l'affichage des résultats, on va se contenter d'un sous-ensemble des deux tables :

```
flights_ex <- flights %>% select(month, day, origin, dest)
airports_ex <- airports %>% select(faa, alt, name)
```

Si on se contente d'un `left_join` comme à l'étape précédente, on obtient un message d'erreur car aucune colonne commune ne peut être identifiée comme clé de jointure :

```
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 `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"))
#> # A tibble: 336,776 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
#> # i 336,766 more rows
```

On constate que les deux nouvelles colonnes `name` et `alt` contiennent bien les données correspondant à l'aéroport de départ.

On va stocker le résultat de cette jointure dans la table `flights_ex` :

```
flights_ex <- flights_ex %>%
  left_join(airports_ex, by = c("origin" = "faa"))
```

Supposons qu'on souhaite maintenant fusionner à nouveau les informations de la table `airports`, mais cette fois pour les aéroports d'arrivée de notre nouvelle table `flights_ex`. Les deux clés sont donc désormais `dest` dans la première table, et `faa` dans la deuxième. La syntaxe est donc la suivante :

```
flights_ex %>%
  left_join(airports_ex, by = c("dest" = "faa"))
#> # A tibble: 336,776 x 8
#>    month   day origin dest  alt.x name.x              alt.y name.y
#>    <int> <int> <chr>  <chr> <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
#> # i 336,766 more rows
```

Cela fonctionne, les informations de l'aéroport d'arrivée ont bien été ajoutées, mais on constate que les colonnes ont été renommées. En effet, ici les deux tables fusionnées contenaient toutes les deux des colonnes `name` et `alt`. Comme on ne peut pas avoir deux colonnes avec le même nom dans un tableau, `dplyr` a renommé les colonnes de la première table en `name.x` et `alt.x`, et celles de la deuxième en `name.y` et `alt.y`.

C'est pratique, mais pas forcément très parlant. On pourrait renommer manuellement les colonnes avec `rename` avant de faire la jointure pour avoir des intitulés plus explicites, mais on peut aussi utiliser l'argument `suffix` de `left_join`, qui permet d'indiquer les suffixes à ajouter aux colonnes.

```
flights_ex %>%
  left_join(
    airports_ex,
    by = c("dest" = "faa"),
```

```
    suffix = c("_depart", "_arrivee")
  )
#> # A tibble: 336,776 x 8
#>    month   day origin dest  alt_depart name_depart      alt_arrivee name_arrivee
#>    <int> <int> <chr>  <chr>      <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~
#> # i 336,766 more rows
```

On obtient ainsi directement des noms de colonnes nettement plus clairs.

2.6.3 Types de jointures

Jusqu'à présent nous avons utilisé la fonction `left_join`, mais il existe plusieurs types de jointures.

Partons de deux tables d'exemple, `personnes` et `voitures` :

```
personnes <- tibble(
    nom = c("Sylvie", "Sylvie", "Monique", "Gunter", "Rayan", "Rayan"),
    voiture = c("Twingo", "Ferrari", "Scenic", "Lada", "Twingo", "Clio")
)
```

nom	voiture
Sylvie	Twingo
Sylvie	Ferrari
Monique	Scenic
Gunter	Lada
Rayan	Twingo
Rayan	Clio

```
voitures <- tibble(
    voiture = c("Twingo", "Ferrari", "Clio", "Lada", "208"),
    vitesse = c("140", "280", "160", "85", "160")
)
```

voiture	vitesse
Twingo	140
Ferrari	280
Clio	160
Lada	85
208	160

2.6.3.1 left_join

Si on fait un `left_join` de `voitures` sur `personnes` :

```
personnes %>% left_join(voitures)
```

```
#> Joining with `by = join_by(voiture)`
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Monique	Scenic	NA
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160

On voit que chaque ligne de `personnes` est bien présente, et qu'on lui a ajouté une ligne de `voitures` correspondante si elle existe. Dans le cas du `Scenic`, il n'y a avait pas de ligne dans `voitures`, donc `vitesse` a été mise à `NA`. Dans le cas de 208, présente dans `voitures` mais pas dans `personnes`, la ligne n'apparaît pas.

Si on fait un `left_join` cette fois de `personnes` sur `voitures`, c'est l'inverse :

```
voitures %>% left_join(personnes)
```

```
#> Joining with `by = join_by(voiture)`
```

voiture	vitesse	nom
Twingo	140	Sylvie
Twingo	140	Rayan
Ferrari	280	Sylvie
Clio	160	Rayan
Lada	85	Gunter
208	160	NA

La ligne `208` est là, mais `nom` est à `NA`. Par contre `Monique` est absente. Et on remarquera que la ligne `Twingo`, présente deux fois dans `personnes`, a été dupliquée pour être associée aux deux lignes de données de `Sylvie` et `Rayan`.

En résumé, quand on fait un `left_join(x, y)`, toutes les lignes de `x` sont présentes, et dupliquées si nécessaire quand elles apparaissent plusieurs fois dans `y`. Les lignes de `y` non présentes dans `x` disparaissent. Les lignes de `x` non présentes dans `y` se voient attribuer des `NA` pour les nouvelles colonnes.

Intuitivement, on pourrait considérer que `left_join(x, y)` signifie "ramener l'information de la table `y` sur la table `x`".

En général, `left_join` sera le type de jointures le plus fréquemment utilisé.

2.6.3.2 right_join

La jointure `right_join` est l'exacte symétrique de `left_join`, c'est-à dire que `right_join(x, y)` est équivalent à `left_join(y, x)` :

```
personnes %>% right_join(voitures)
```

```
#> Joining with `by = join_by(voiture)`
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160

nom	voiture	vitesse
NA	208	160

2.6.3.3 inner_join

Dans le cas de `inner_join(x, y)`, seules les lignes présentes à la fois dans `x` et `y` sont conservées (et si nécessaire dupliquées) dans la table résultat :

```
personnes %>% inner_join(voitures)
```

```
#> Joining with `by = join_by(voiture)`
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160

Ici la ligne `208` est absente, ainsi que la ligne `Monique`, qui dans le cas d'un `left_join` avait été conservée et s'était vue attribuer une `vitesse` à `NA`.

2.6.3.4 full_join

Dans le cas de `full_join(x, y)`, toutes les lignes de `x` et toutes les lignes de `y` sont conservées (avec des `NA` ajoutés si nécessaire) même si elles sont absentes de l'autre table :

```
personnes %>% full_join(voitures)
```

```
#> Joining with `by = join_by(voiture)`
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Monique	Scenic	NA

nom	voiture	vitesse
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160
NA	208	160

2.6.3.5 semi_join et anti_join

`semi_join` et `anti_join` sont des jointures *filtrantes*, c'est-à-dire qu'elles sélectionnent les lignes de `x` sans ajouter les colonnes de `y`.

Ainsi, `semi_join` ne conservera que les lignes de `x` pour lesquelles une ligne de `y` existe également, et supprimera les autres. Dans notre exemple, la ligne `Monique` est donc supprimée :

```
personnes %>% semi_join(voitures)
```

```
#> Joining with `by = join_by(voiture)`
```

nom	voiture
Sylvie	Twingo
Sylvie	Ferrari
Gunter	Lada
Rayan	Twingo
Rayan	Clio

Un `anti_join` fait l'inverse, il ne conserve que les lignes de `x` absentes de `y`. Dans notre exemple, on ne garde donc que la ligne `Monique` :

```
personnes %>% anti_join(voitures)
```

```
#> Joining with `by = join_by(voiture)`
```

nom	voiture
Monique	Scenic

2.7 Ressources

Toutes les ressources ci-dessous sont en anglais...

Le livre *R for data science*, librement accessible en ligne, contient plusieurs chapitres très complets sur la manipulation des données, notamment :

- Data transformation pour les manipulations
- Relational data pour les tables multiples

Le site de l'extension comprend une liste des fonctions et les pages d'aide associées, mais aussi une introduction au package et plusieurs articles dont un spécifiquement sur les jointures.

Enfin, une "antisèche" très synthétique est également accessible depuis RStudio, en allant dans le menu *Help* puis *Cheatsheets* et *Data Transformation with dplyr*.

partie II

Les probabilités et la combinatoire

3 La combinatoire

3.1 La factorielle

Pour calculer la factorielle d'un nombre en `R`, il faut utiliser la commande `factorial`. Par exemple, si nous voulons calculer 6!:

```
factorial(6)
#> [1] 720
```

3.2 Les combinaisons

Pour calculer le nombre de combinaisons lorsque nous choisissons k objets parmi n (**sans** ordre), c'est-à-dire C_k^n, nous utilisons la commande `choose(n,k)`. Par exemple, si nous voulons calculer le nombre de combinaisons possibles au loto 6-49, C_6^{49}, nous avons:

```
choose(49,6)
#> [1] 13983816
```

3.3 Les arrangements

Pour calculer le nombre d'arrangements lorsque nous choisissons k objets parmi n (**avec** ordre), c'est-à-dire A_k^n, nous utilisons les commandes `choose(n,k)` et `factorial`. En effet, nous savons que:

$$A_k^n = C_k^n \cdot k! \tag{3.1}$$

et donc on peut calculer un arrangement en effectuant `choose(n,k)*factorial(k)`. Si nous voulons calculer le nombre de comités de 5 personnes nous pouvons former en choisissant parmi 12 personnes, A_5^{12}, nous avons:

```r
choose(12,5)*factorial(5)
#> [1] 95040
```

4 Les lois de probabilités

Pour être en mesure d'utiliser les lois de probabilités en langage `R`, il faut charger le paquetage `stats`.

```
library(stats)
library(ggplot2)
```

Chaque distribution en `R` possède quatre fonctions qui lui sont associées. Premièrement, la fonction possède un *nom racine*, par exemple le *nom racine* pour la distribution *binomiale* est `binom`. Cette racine est précédée par une de ces quatre lettre:

- `p` pour *probabilité*, qui représente la fonction de répartition
- `q` pour *quantile*, l'inverse de la fonction de répartition
- `d` pour *densité*, la fonction de densité de la distribution
- `r` pour *random*, une variable aléatoire suivant la distribution spécifiée.

Pour la loi binomiale par exemple, ces fonctions sont `pbinom`, `qbinom`, `dbinom` et `rbinom`.

4.1 Les lois de probabilités discrètes

4.1.1 La loi binomiale

Le *nom racine* pour la loi binomiale est `binom`.

Soit X: le nombre de succès en n essais et $X \sim B(n, p)$. Voici la façon de calculer des probabilités pour la loi binomiale à l'aide de `R`:

Probabilités	Commande `R`
$P(X = k)$	`dbinom(k, n, p)`
$P(i \leq X \leq j)$	`sum(dbinom(i:j, n, p))`
$P(X \leq k)$	`pbinom(k, n, p)`
$P(X > k)$	`1-pbinom(k, n, p)`

Soit X la variable aléatoire comptant le nombre de face 2 que nous obtenons en lançant un dé à quatre reprises. Nous avons que $X \sim B(4, \frac{1}{6})$. Si nous voulons calculer $P(X = 3)$, nous aurons:

```
dbinom(3,4,1/6)
#> [1] 0.0154321
```

Nous avons donc une probabilité de 1.5432099% d'obtenir 3 fois la face deux en lançant un dé à quatres reprises.

4.1.2 La loi de Poisson

Le *nom racine* pour la loi de Poisson est `pois`.

Soit X: le nombre d'événements dans un intervalle fixé et $X \sim Po(\lambda)$. Voici la façon de calculer des probabilités pour la loi de Poisson à l'aide de `R`:

Probabilités	Commande `R`
$P(X = k)$	`dpois(k, lambda)`
$P(i \leq X \leq j)$	`sum(dpois(i:j, lambda))`
$P(X \leq k)$	`ppois(k, lambda)`
$P(X > k)$	`1-ppois(k, lambda)`

Soit X le nombre d'erreurs dans une page. Si une page contient en moyenne une demie erreur alors $X \sim Po(1/2)$. Si nous voulons calculer $P(X = 2)$, nous aurons:

```
dpois(2, 1/2)
#> [1] 0.07581633
```

Nous avons donc une probabilité de 7.5816332% d'obtenir deux erreurs sur une page.

4.1.3 La loi géométrique

Le *nom racine* pour la loi géométrique est `geom`.

Soit X: le nombre d'échecs avant d'obtenir un succès et $X \sim G(p)$. Voici la façon de calculer des probabilités pour la loi géométrique à l'aide de `R`:

Probabilités	Commande R
$P(X = k)$	`dgeom(k, p)`
$P(i \leq X \leq j)$	`sum(dgeom(i:j, p))`
$P(X \leq k)$	`pgeom(k, p)`
$P(X > k)$	`1-pgeom(k, p)`

Soit X le nombre d'échecs avant d'avoir un premier succès. Si la probabilité de succès est $\frac{1}{5}$ alors $X \sim G(1/5)$. Si nous voulons calculer $P(X = 6)$, nous aurons:

```
dgeom(6, 1/5)
#> [1] 0.0524288
```

Nous avons donc une probabilité de 5.24288% d'obtenir 6 échecs avant un premier succès.

> Remarque : Pour la loi géométrique, on rencontre parfois cette définition : la probabilité p'(k) est la probabilité, lors d'une succession d'épreuves de Bernoulli indépendantes, d'obtenir k échecs avant un succès. On remarque qu'il ne s'agit que d'un décalage de la précédente loi géométrique. Si X suit la loi p, alors $X + 1$ suit la loi p'.

4.1.4 La loi hypergéométrique

Le *nom racine* pour la loi hypergéométrique est `hyper`.

On tire sans remise n objets d'un ensemble de N objets dont A possèdent une caractéristique particulière (et les autres $B = N - A$ ne la possèdent pas). Soit X le nombre d'objets de l'échantillon qui possèdent la caractéristique. Nous avons que $X \sim H(N, A, n)$.

Voici la façon de calculer des probabilités pour la loi hypergéométrique à l'aide de R:

Probabilités	Commande R
$P(X = k)$	`dhyper(k, A, B, n)`
$P(i \leq X \leq j)$	`sum(dhyper(i:j, A, B, n))`
$P(X \leq k)$	`phyper(k, A, B, n)`
$P(X > k)$	`1-phyper(k, A, B, n)`

Soit X le nombre de boules blanches de l'échantillon de taille 4. Si l'urne contient 5 boules blanches et 8 boules noires, nous avons $X \sim H(13, 5, 4)$. Si nous voulons calculer $P(X = 2)$, nous aurons:

```r
dhyper(2, 5, 8, 4)
#> [1] 0.3916084
```

Nous avons donc une probabilité de 39.1608392% de piger 2 boules blanches dans un échantillon de taille 4.

4.2 Les lois de probabilités continues

4.2.1 La loi normale

Le *nom racine* pour la loi normale est `norm`.

Si X suit une loi normale de moyenne μ et de variance σ^2, nous avons $X \sim N(\mu, \sigma^2)$.

Voici la façon de calculer des probabilités pour la loi normale à l'aide de `R`:

Probabilités	Commande `R`
$P(i \leq X \leq j)$	`pnorm(j, mu, sigma)-pnorm(i, mu, sigma)`
$P(X \leq k)$	`pnorm(k, mu, sigma)`
$P(X > k)$	`1-pnorm(k, mu, sigma)`

Soit $X \sim N(3, 25)$ une variable aléatoire suivant une loi normale de moyenne 3 et de variance 25. Si nous voulons calculer la probabilité $P(1.25 < X < 3.6)$ en `R`, nous pouvons utiliser la commande suivante:

```r
pnorm(3.6, 3, 5) - pnorm(1.25, 3, 5)
#> [1] 0.1845891
```

La probabilité que notre variable aléatoire se trouve entre 1.25 et 3.6 est donc 18.4589077 %.

4.2.2 La loi de Student

Le *nom racine* pour la loi de Student est `t`.

Si X suit une loi de Student à ν degrés de liberté, nous avons $X \sim T_\nu$.

Voici la façon de calculer des probabilités pour la loi de Student à l'aide de `R`:

Probabilités	Commande `R`
$P(i \leq X \leq j)$	`pt(j, nu)-pt(i, nu)`
$P(X \leq k)$	`pt(k, nu)`
$P(X > k)$	`1-pt(k, nu)`

Soit $X \sim T_5$ une variable aléatoire suivant une loi de Student à 5 degrés de liberté. Si nous voulons calculer la probabilité $P(X > 3)$ en `R`, nous pouvons utiliser la commande suivante:

```
1 - pt(3, 5)
#> [1] 0.01504962
```

La probabilité que notre variable aléatoire soit plus grande que 3 est donc 1.5049624 %.

partie III

Les statistiques descriptives

5 Les tableaux

Une fois les données d'un sondage recueillies, il est plus aisé d'analyser ces données si elles sont classées dans un tableau.

Nous utiliserons l'extension `nycflights13` avec les bases de données `planes`, `weather` et `flights` pour montrer la création de tableaux en `R`.

```
library(nycflights13)
data(planes)
data(weather)
data(flights)
```

5.1 Tableau de fréquences à une variable

5.1.1 Les variables qualitatives

Le tableau de fréquences que nous utiliserons est le suivant:

Titre		
Nom de la variable (*Modalités*)	**Nombre d'unités statistiques** (*Fréquences absolues*)	**Pourcentage d'unités statistiques** (%) (*Fréquences relatives*)
Total	n	100%

> Important : Le titre doit toujours être indiqué lors de la construction d'un tableau de fréquence.

Lorsque les données se trouvent dans une `tibble` dans `R`, il est possible d'utiliser la commande `freq` de la librairie `questionr` pour afficher le tableau de fréquences. La commande `freq` prend comme argument la variable dont vous voulez produire le tableau de fréquences. Pour obtenir une sortie adéquate, il faut ajouter trois options à la commande:

- `cum = FALSE`; permet de ne pas afficher les pourcentages cumulés
- `valid = FALSE`; permet de ne pas afficher les données manquantes

- `total = TRUE`; permet d'afficher le total

Dans la base de données `nycflights13::planes`, nous allons afficher la variable `engine`, qui correspond au type de moteur de l'avion.

```
freq(planes$engine,cum = FALSE,valid = FALSE,total = TRUE)
#>                   n     %
#> 4 Cycle           2   0.1
#> Reciprocating    28   0.8
#> Turbo-fan      2750  82.8
#> Turbo-jet       535  16.1
#> Turbo-prop        2   0.1
#> Turbo-shaft       5   0.2
#> Total          3322 100.0
```

5.1.2 Les variables quantitatives discrètes

Le tableau de fréquences que nous utiliserons est le suivant :

Titre			
Nom de la variable (*Valeurs*)	**Nombre d'unités statistiques** (*Fréquences absolues*)	**Pourcentage d'unités statistiques** (%) (*Fréquences relatives*)	**Pourcentage cumulé** (*Fréquences relatives cumulées*)
Total	n	100%	

Le pourcentage cumulé permet de déterminer le pourcentage des répondants qui ont indiqué la valeur correspondante, ou une plus petite. Il sert à donner une meilleure vue d'ensemble.

> Si pour la valeur x_i de la variable A la pourcentage cumulé est de b %, ceci signifie que b % des valeurs de la variable A sont plus petites ou égales à x_i.

La commande `freq` prend comme argument la variable dont vous voulez produire le tableau de fréquences. Pour obtenir une sortie adéquate, il faut ajouter trois options à la commande:

- `cum = TRUE`; permet d'afficher les pourcentages cumulés
- `valid = FALSE`; permet de ne pas afficher les données manquantes
- `total = TRUE`; permet d'afficher le total

Dans la base de données `nycflights13::planes`, nous allons afficher la variable `engines`, qui correspond au nombre de moteurs de l'avion.

```
freq(planes$engines,cum = TRUE,valid = FALSE,total = TRUE)
#>           n     %  %cum
#> 1        27   0.8   0.8
#> 2      3288  99.0  99.8
#> 3         3   0.1  99.9
#> 4         4   0.1 100.0
#> Total  3322 100.0 100.0
```

Dans la base de données `nycflights13::planes`, nous allons afficher la variable `seats`, qui correspond au nombre de sièges de l'avion.

```
freq(planes$seats,cum = TRUE,valid = FALSE,total = TRUE)
#>           n     %  %cum
#> 2        16   0.5   0.5
#> 4         5   0.2   0.6
#> 5         2   0.1   0.7
#> 6         3   0.1   0.8
#> 7         2   0.1   0.8
#> 8         5   0.2   1.0
#> 9         1   0.0   1.0
#> 10        1   0.0   1.1
#> 11        2   0.1   1.1
#> 12        1   0.0   1.1
#> 14        1   0.0   1.2
#> 16        1   0.0   1.2
#> 20       80   2.4   3.6
#> 22        2   0.1   3.7
#> 55      390  11.7  15.4
#> 80       83   2.5  17.9
#> 95      123   3.7  21.6
#> 100     102   3.1  24.7
#> 102       1   0.0  24.7
#> 128       1   0.0  24.7
#> 139       8   0.2  25.0
#> 140     411  12.4  37.4
#> 142     158   4.8  42.1
#> 145      57   1.7  43.8
#> 147       3   0.1  43.9
#> 149     452  13.6  57.5
#> 172      81   2.4  60.0
#> 178     283   8.5  68.5
#> 179     134   4.0  72.5
```

```
#> 182    159   4.8  77.3
#> 189     73   2.2  79.5
#> 191     87   2.6  82.1
#> 199     43   1.3  83.4
#> 200    256   7.7  91.1
#> 222     13   0.4  91.5
#> 255     16   0.5  92.0
#> 260      4   0.1  92.1
#> 269      1   0.0  92.1
#> 275     25   0.8  92.9
#> 290      6   0.2  93.1
#> 292     16   0.5  93.6
#> 300     17   0.5  94.1
#> 330    114   3.4  97.5
#> 375      1   0.0  97.5
#> 377     14   0.4  98.0
#> 379     55   1.7  99.6
#> 400     12   0.4 100.0
#> 450      1   0.0 100.0
#> Total 3322 100.0 100.0
```

Comme nous pouvons le constater, le tableau est très grand car la variable `seats` possède 48 valeurs différentes.

5.1.3 Les variables quantitatives continues

Le tableau de fréquences que nous utiliserons est le suivant :

Titre			
Nom de la variable (*Classes*)	**Nombre d'unités statistiques** (*Fréquences absolues*)	**Pourcentage d'unités statistiques** (%) (*Fréquences relatives*)	**Pourcentage cumulé** (*Fréquences relatives cumulées*)
Total	n	100%	

Pour être en mesure de briser une variable en classes, il faut utiliser la commande `cut`.

Les options de `cut` sont:

- `include.lowest = TRUE` qui permet d'avoir un intervalle fermé à droite et ouvert à gauche;

- `breaks` qui permet d'indiquer à quel endroit on doit créer les classes;
- `seq(from = A, to = B, by = C)` permet de créer un vecteur comportant les valeurs de `A` jusqu'à `B` en faisant des bonds de `C`.

Pour simplifier le code, nous créons en premier lieu une variable `air_time_rec` avec les classes et nous l'affichons ensuite avec `freq`. Remarquons que nous avons ajouté l'option `valid = TRUE` car certaines valeurs sont manquantes. Rappelons que les données manquantes sont représentées par `NA` en `R`. Deux colonnes sont ajoutées:

- `val%`: le pourcentage en omettant les valeurs manquantes
- `val%cum`: le pourcentage cumulé en omettant les valeurs manquantes

Nous obtenons donc:

```
air_time_rec <- cut(flights$air_time,
                    right=FALSE,
                    breaks=seq(from = 0, to = 700, by = 100))
freq(air_time_rec,cum = TRUE,total = TRUE,valid = TRUE)
#>                n     %  val%  %cum val%cum
#> [0,100)   105687  31.4  32.3  31.4    32.3
#> [100,200) 146527  43.5  44.8  74.9    77.0
#> [200,300)  31036   9.2   9.5  84.1    86.5
#> [300,400)  43347  12.9  13.2  97.0    99.8
#> [400,500)     48   0.0   0.0  97.0    99.8
#> [500,600)    132   0.0   0.0  97.0    99.8
#> [600,700)    569   0.2   0.2  97.2   100.0
#> NA          9430   2.8    NA 100.0      NA
#> Total     336776 100.0 100.0 100.0   100.0
```

5.2 Tableau de fréquences à deux variables

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

À noter qu'on va parler ici d'influence ou de lien, mais pas de relation de cause à effet. Les outils présentés permettent de visualiser ou de déterminer une relation, mais la mise en évidence de liens de causalité proprement dit est nettement plus complexe : il faut en effet vérifier que c'est bien telle variable qui influence telle autre et pas l'inverse, qu'il n'y a pas de "variable cachée", etc.

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

5.2.1 Croisement de deux variables qualitatives

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

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

Quand on veut croiser deux variables qualitatives, on fait un *tableau croisé*. Comme pour un tri à plat ceci s'obtient avec la fonction `table` de R, mais à laquelle on passe cette fois deux variables en argument. Par exemple, si on veut croiser la catégorie socio-professionnelle et le sexe des enquêtés :

```
table(d$qualif, d$sexe)
#>
#>                            Homme Femme
#>   Ouvrier specialise          96   107
#>   Ouvrier qualifie           229    63
#>   Technicien                  66    20
#>   Profession intermediaire    88    72
#>   Cadre                      145   115
#>   Employe                     96   498
#>   Autre                       21    37
```

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

Pour calculer les pourcentages ligne :

```
tab <- table(d$qualif, d$sexe)
lprop(tab)
#>
#>                            Homme Femme Total
#>   Ouvrier specialise        47.3  52.7 100.0
#>   Ouvrier qualifie          78.4  21.6 100.0
#>   Technicien                76.7  23.3 100.0
#>   Profession intermediaire  55.0  45.0 100.0
#>   Cadre                     55.8  44.2 100.0
#>   Employe                   16.2  83.8 100.0
```

```
#>   Autre                     36.2  63.8 100.0
#>   All                       44.8  55.2 100.0
```

Et pour les pourcentages colonne :

```
cprop(tab)
#>
#>                           Homme Femme All
#>   Ouvrier specialise       13.0  11.7  12.3
#>   Ouvrier qualifie         30.9   6.9  17.7
#>   Technicien                8.9   2.2   5.2
#>   Profession intermediaire 11.9   7.9   9.7
#>   Cadre                    19.6  12.6  15.7
#>   Employe                  13.0  54.6  35.9
#>   Autre                     2.8   4.1   3.5
#>   Total                   100.0 100.0 100.0
```

6 Les graphiques

Pour produire un graphique, nous utiliserons l'extension `ggplot2` qui est chargée avec le coeur de la librairie `tidyverse`. La grammaire graphique de `ggplot2` peut être décrite de la façon suivante:

> **A statistical graphic is a `mapping` of `data` variables to `aesthetic` attributes of `geometric` objects.**

Plus spécifiquement, nous pouvons briser un graphique en trois composantes essentielles:

1. `data`: la base de données contenant les variables que nous désirons visualiser.
2. `geom`: l'objet géométrique en question. Ceci réfère au type d'objet que nous pouvons observer dans notre graphique. Par exemple, des points, des lignes, des barres, etc.
3. `aes`: les attributs esthétiques (aesthetics) de l'objet géométrique que nous affichons dans notre graphique. Par exemple, la position x/y, la couleur, la forme, la taille. Chaque attribut peut être associé à une variable dans notre base de données.

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

Dans ce qui suit on utilisera le jeu de données issu du recensement de la population de 2018 inclus dans l'extension `questionr` (résultats partiels concernant les communes de plus de 2000 habitants de France métropolitaine). On charge ces données et on en extrait les données de 5 départements (l'utilisation de la fonction `filter` sera expliquée Section 2.2.2) :

```
library(questionr)
data(rp2018)

rp <- filter(
    rp2018,
    departement %in% c("Oise", "Rhône", "Hauts-de-Seine", "Lozère", "Bouches-du-Rhône")
)
```

6.1 Initialisation

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

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

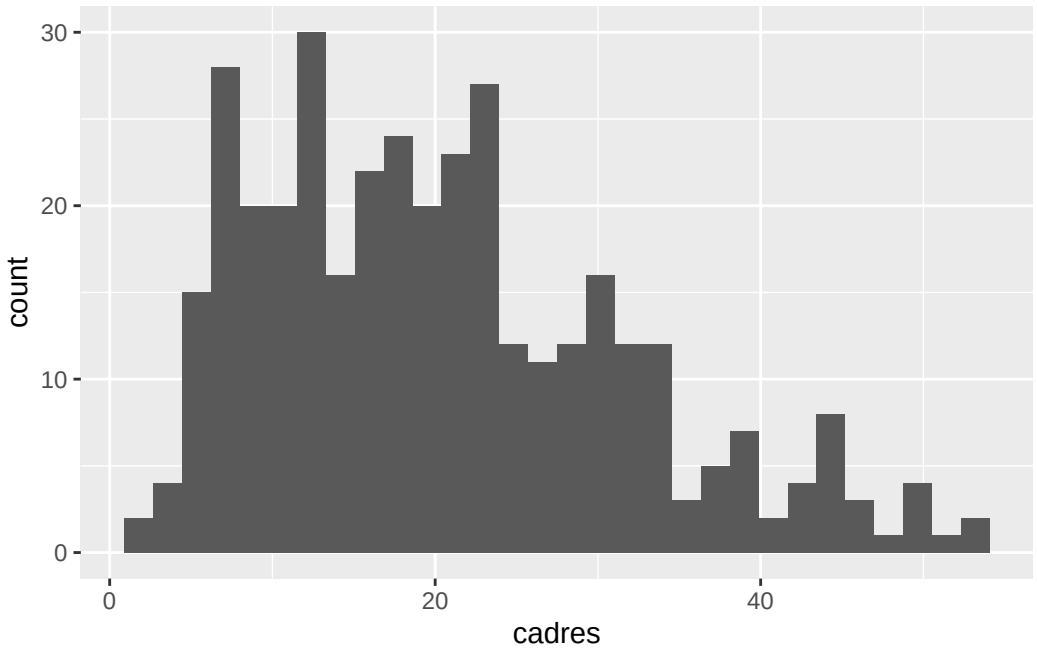
On a défini la source de données, il faut maintenant ajouter des éléments de représentation graphique. Ces éléments sont appelés des `geom`, et on les ajoute à l'objet graphique de base avec l'opérateur `+`.

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

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

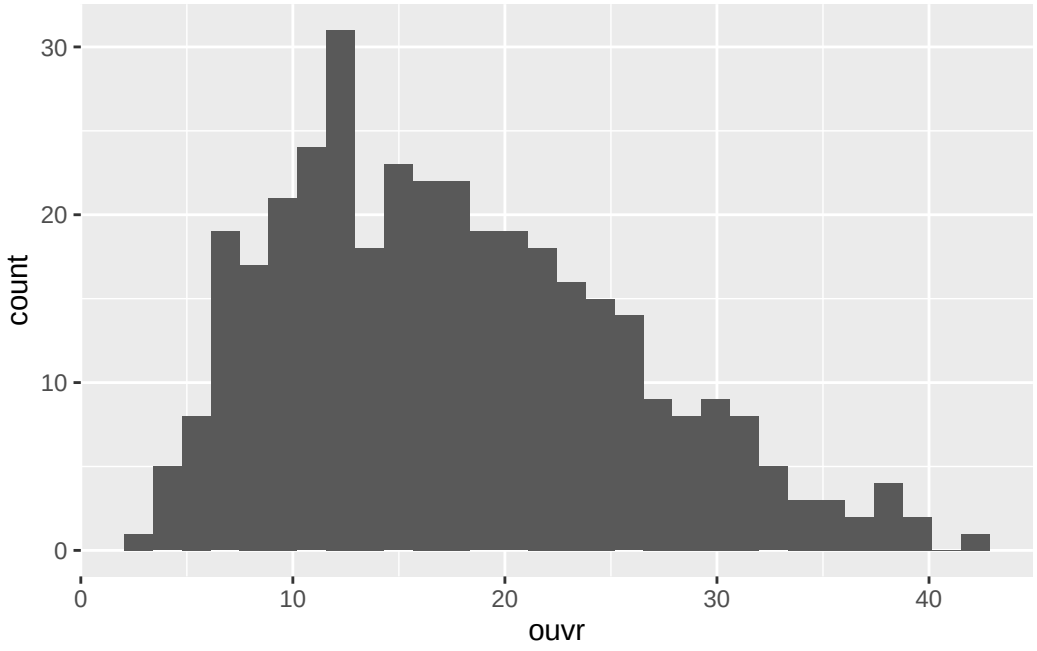
Reste à indiquer quelle donnée nous voulons représenter sous forme d'histogramme. Cela se fait à l'aide d'arguments passés via la fonction `aes()`. Ici nous avons un paramètre à renseigner, `x`, qui indique la variable à représenter sur l'axe des x (l'axe horizontal). Ainsi, si on souhaite représenter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

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



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

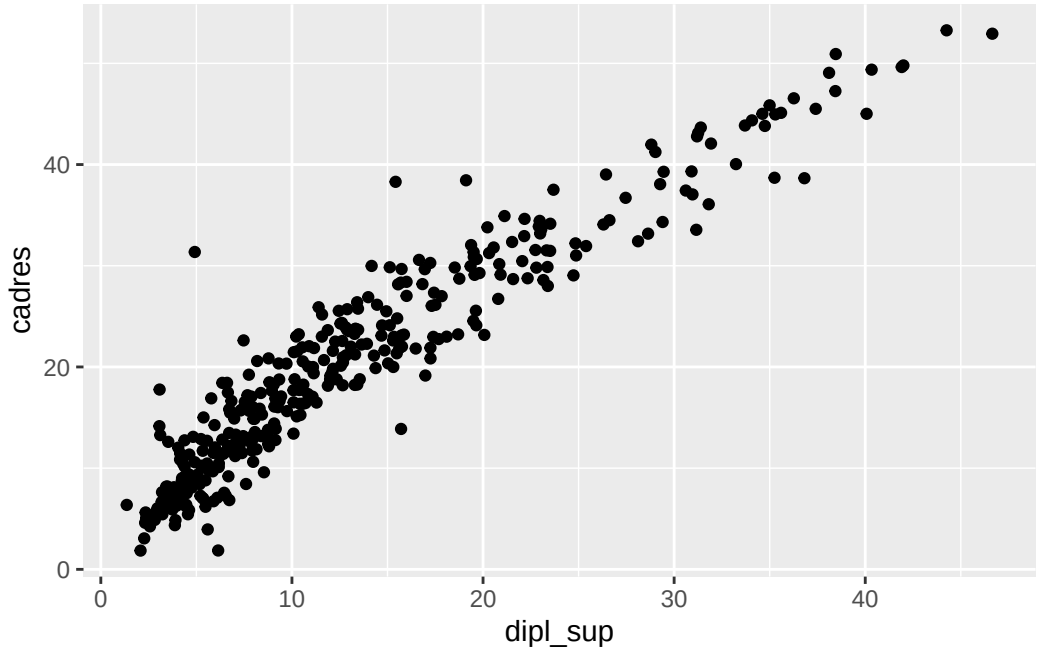
```
ggplot(rp) +
    geom_histogram(aes(x = ouvr))
```



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

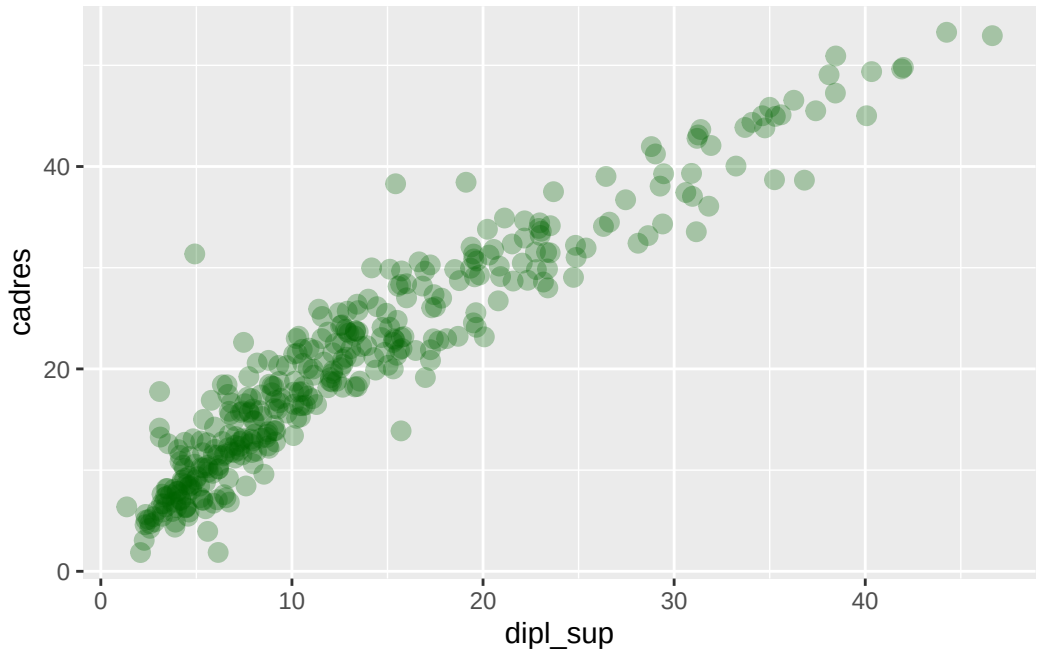
Certains `geom` prennent plusieurs paramètres. Ainsi, si on veut représenter un nuage de points, on peut le faire en ajoutant un `geom_point`. On doit alors indiquer à la fois la position en `x` (la variable sur l'axe horizontal) et en `y` (la variable sur l'axe vertical) de ces points, il faut donc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

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



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

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

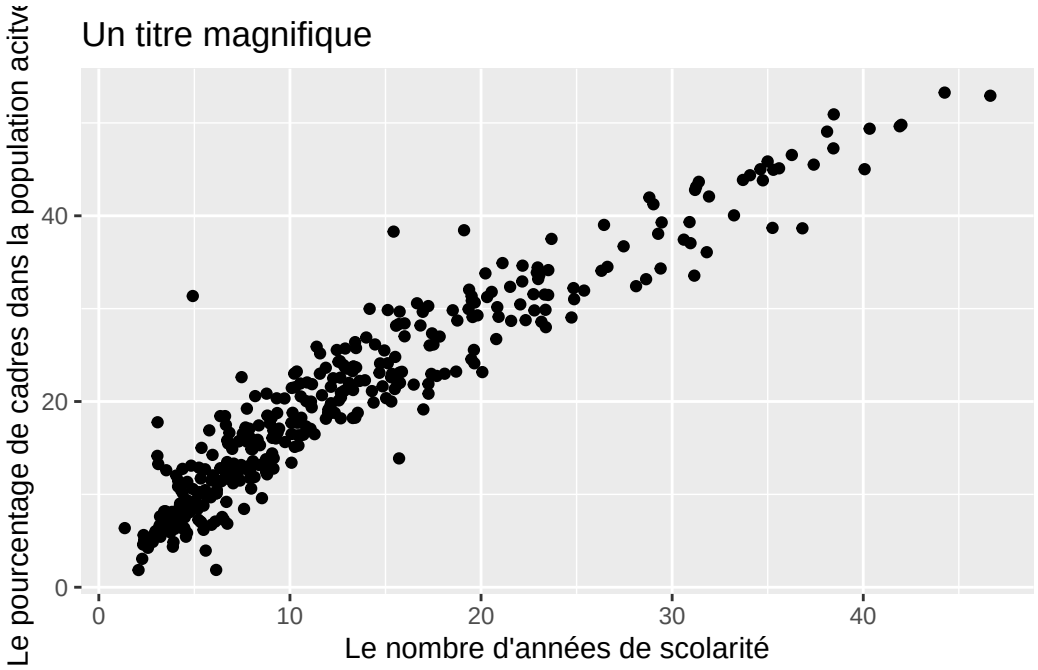


On notera que dans ce cas les arguments sont dans la fonction `geom` mais à l'extérieur du `aes()`. Plus d'explications sur ce point dans quelques instants.

6.2 Les titres

Pour ajouter un titre à votre graphique et pour ajouter des titres à vos axes `x` et `y`, nous utilisons la commande `labs()`.

```
ggplot(rp) +
    geom_point(
        aes(x = dipl_sup, y = cadres)
    ) +
    labs(
        title = "Un titre magnifique",
        x = "Le nombre d'années de scolarité",
        y = "Le pourcentage de cadres dans la population acitve"
    )
```



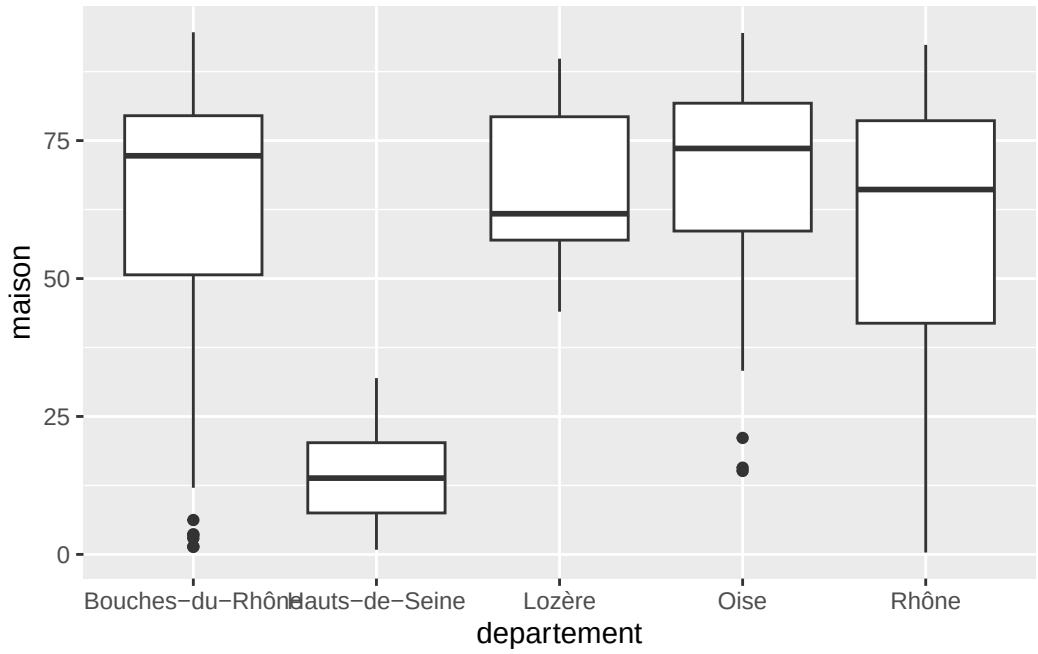
6.3 Exemples de `geom`

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

6.3.1 `geom_boxplot`

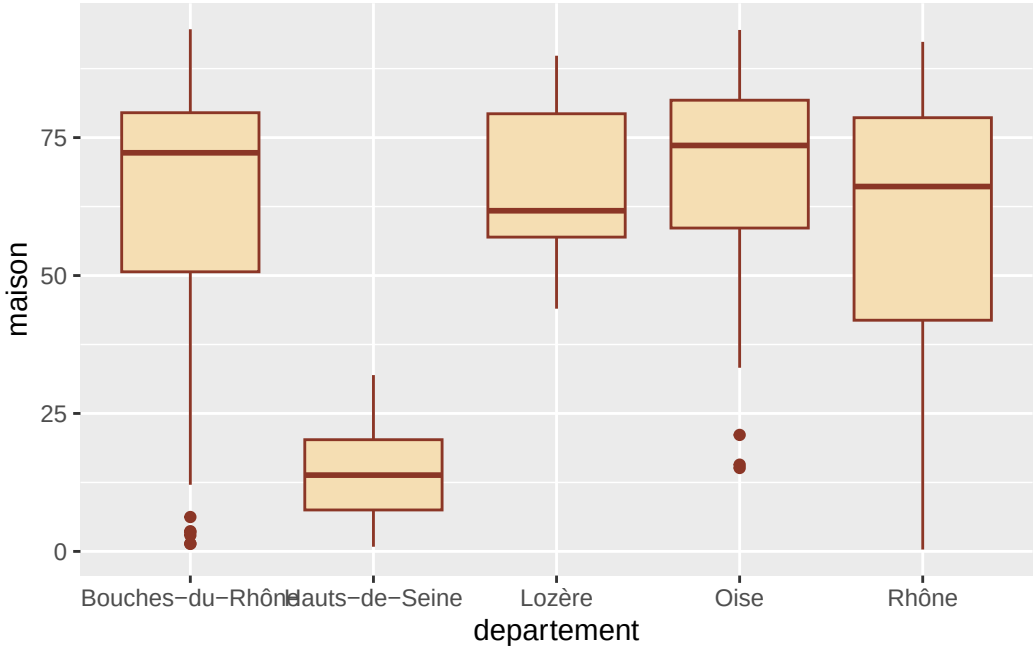
`geom_boxplot` permet de représenter des boîtes à moustaches. On lui passe en `y` la variable numérique dont on veut étudier la répartition, et en `x` la variable qualitative contenant les classes qu'on souhaite comparer. Ainsi, si on veut comparer la répartition du pourcentage de maisons en fonction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

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



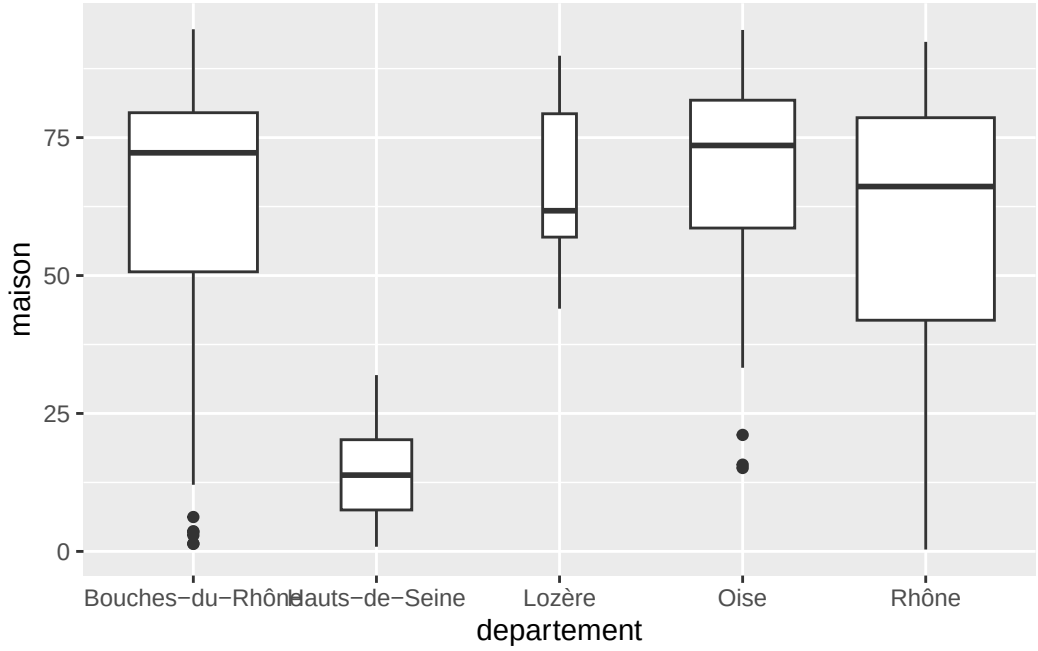
On peut personnaliser la présentation avec différents argument supplémentaires comme `fill` ou `color` :

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



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

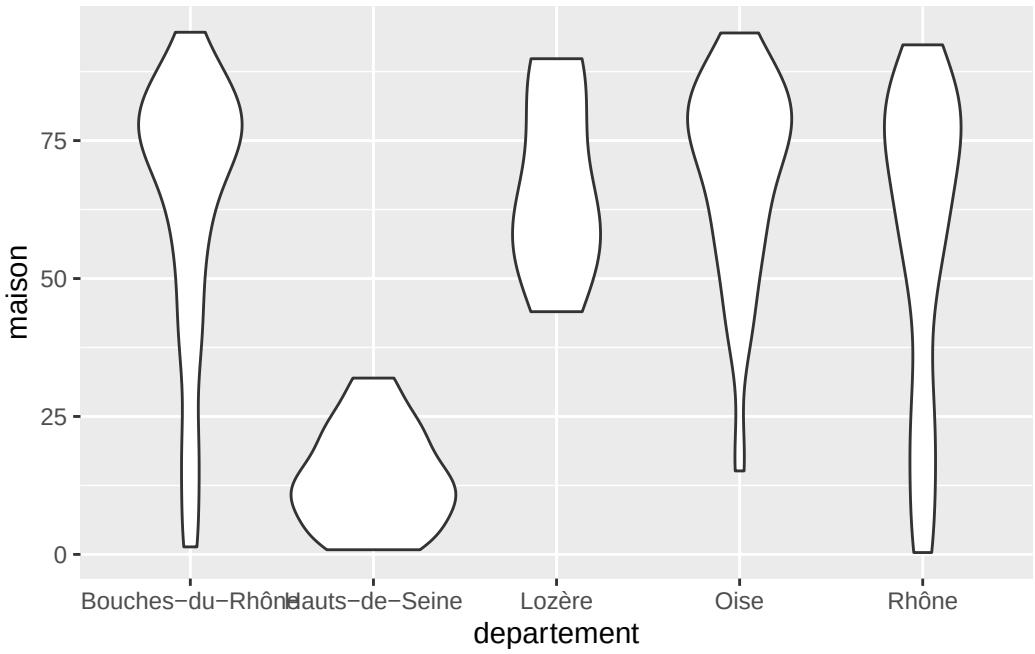
```
ggplot(rp) +
    geom_boxplot(
        aes(x = departement, y = maison),
        varwidth = TRUE)
```



6.3.2 `geom_violin`

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

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



Les graphes en violon peuvent donner une lecture plus fine des différences de distribution selon les classes. Comme pour les graphiques de densité, on peut faire varier le niveau de "détail" de la représentation en utilisant l'argument `bw` (bande passante).

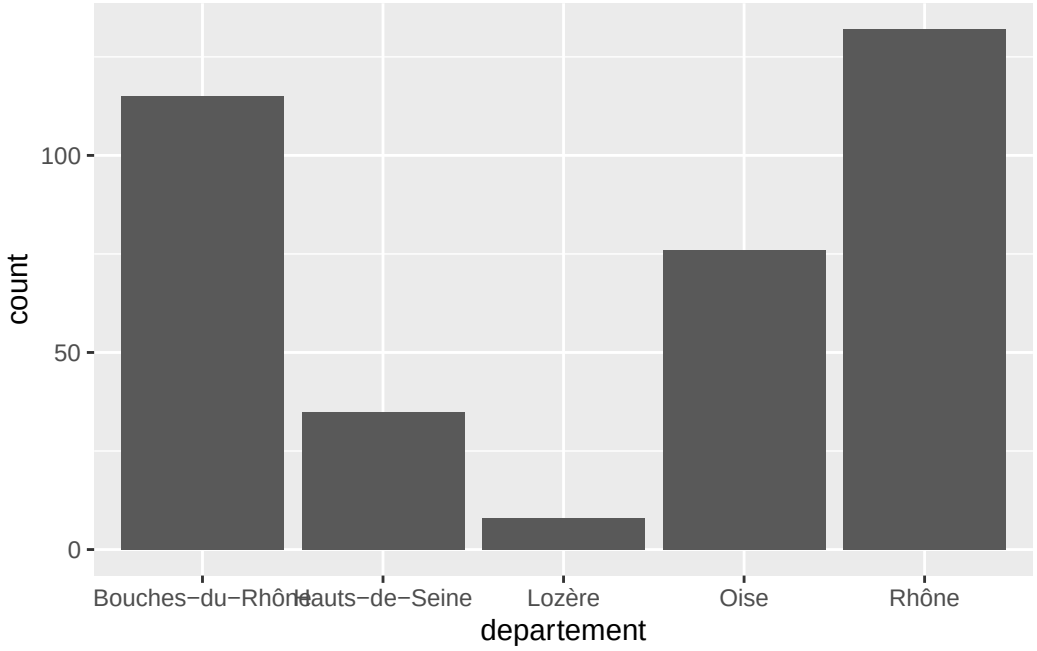
```
ggplot(rp) +
    geom_violin(
        aes(x = departement, y = maison),
        bw = 2
    )
```

6.3.3 `geom_bar` et `geom_col`

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

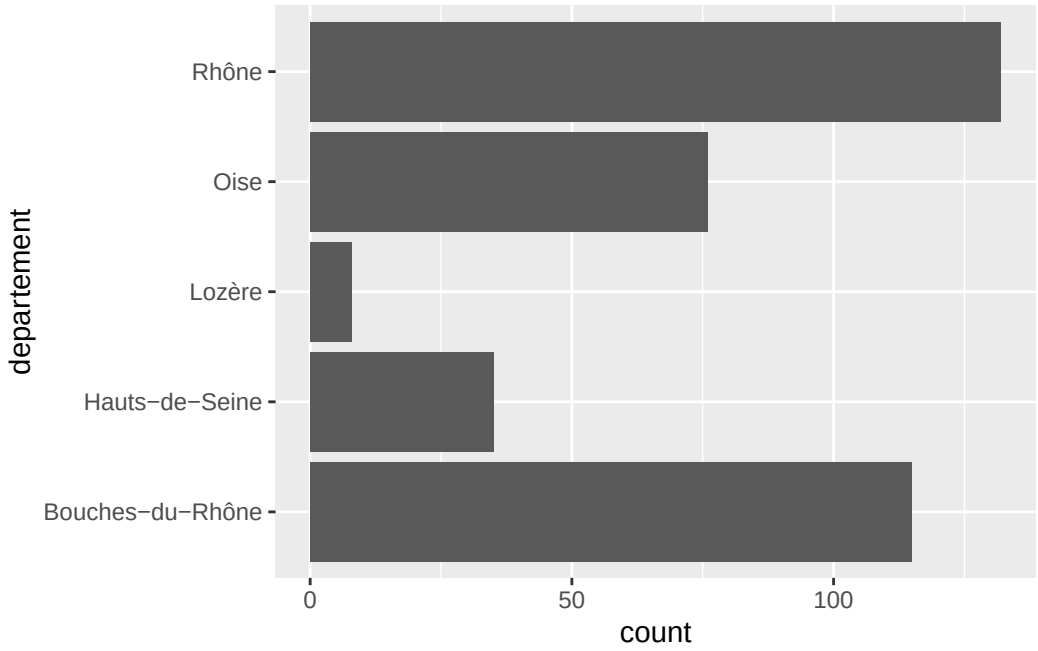
Par exemple, si on veut afficher le nombre de communes de notre jeu de données pour chaque département :

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



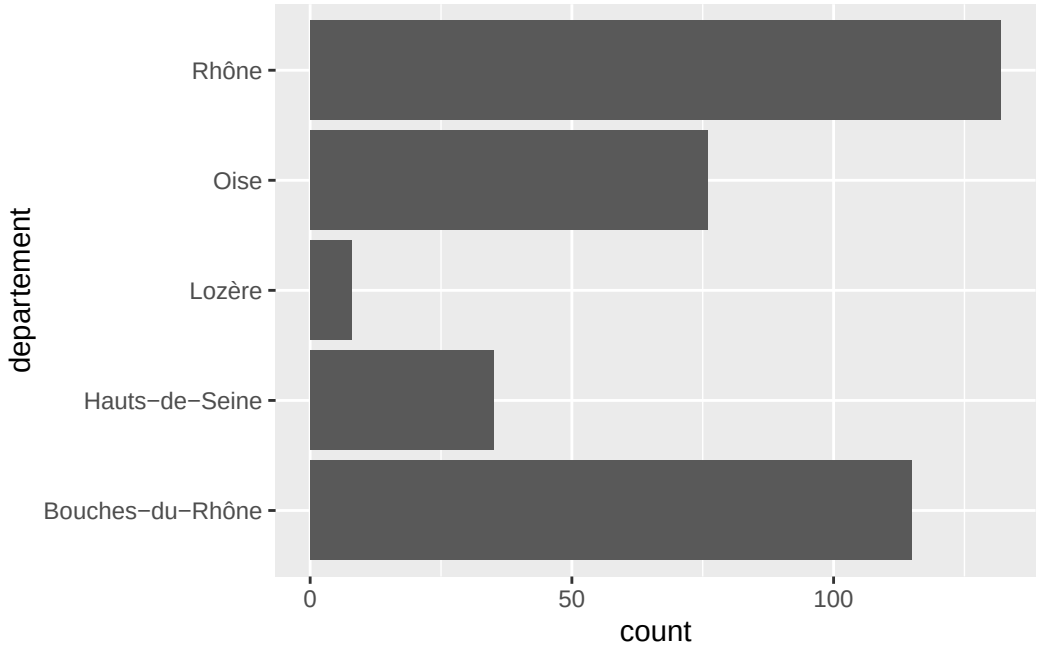
Si on préfère avoir un graphique en barres horizontales, il suffit de passer la variable comme attribut `y` plutôt que `x`.

```
ggplot(rp) +
    geom_bar(aes(y = departement))
```



Une autre possibilité est d'utiliser `coord_flip()`, qui permet d'intervertir l'axe horizontal et l'axe vertical.

```
ggplot(rp) +
    geom_bar(aes(x = departement)) +
    coord_flip()
```

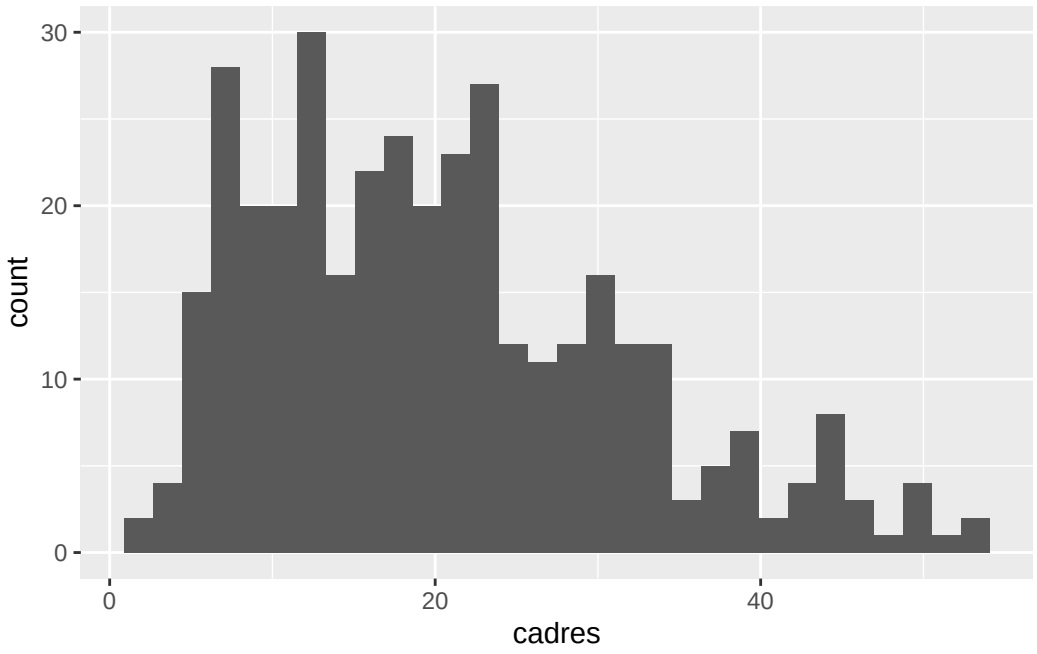


À noter que `coord_flip()` peut s'appliquer à n'importe quel graphique `ggplot2`.

6.3.4 `geom_histogram`

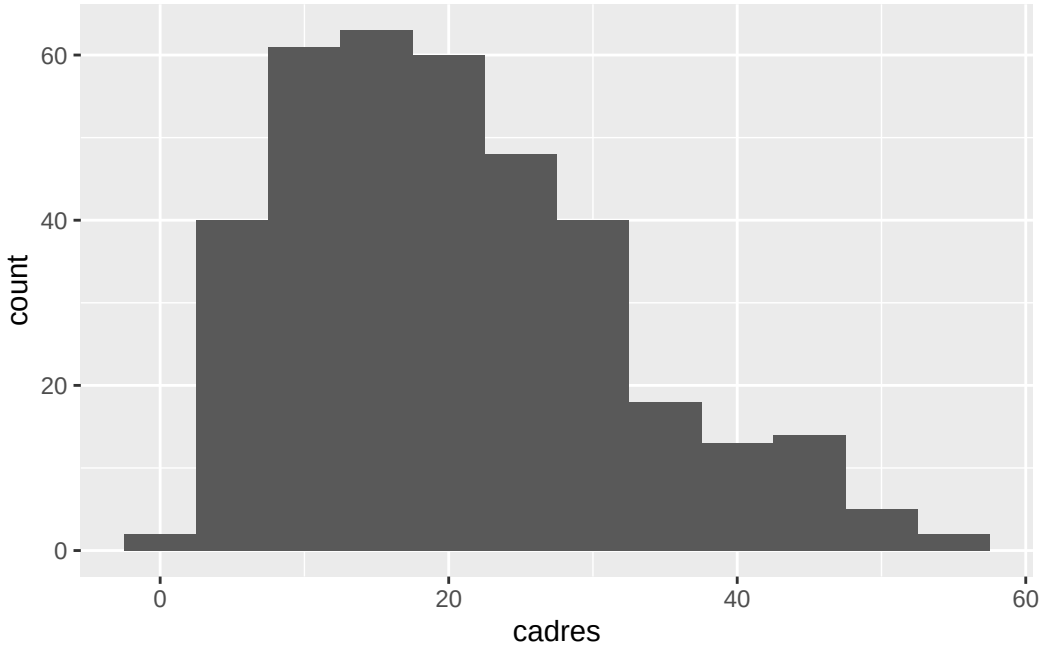
`geom_histogram` permet de représenter des histogrammes. On lui passe en `x` la variable quantitative dont on souhiate étudier la répartition.

```
ggplot(rp) +
    geom_histogram(aes(x = cadres))
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



On peut utiliser différents arguments, comme par exemple `binwidth` pour spécifier la largeur des rectangles de notre histogramme.

```
ggplot(rp) +
    geom_histogram(aes(x = cadres), binwidth = 5)
```

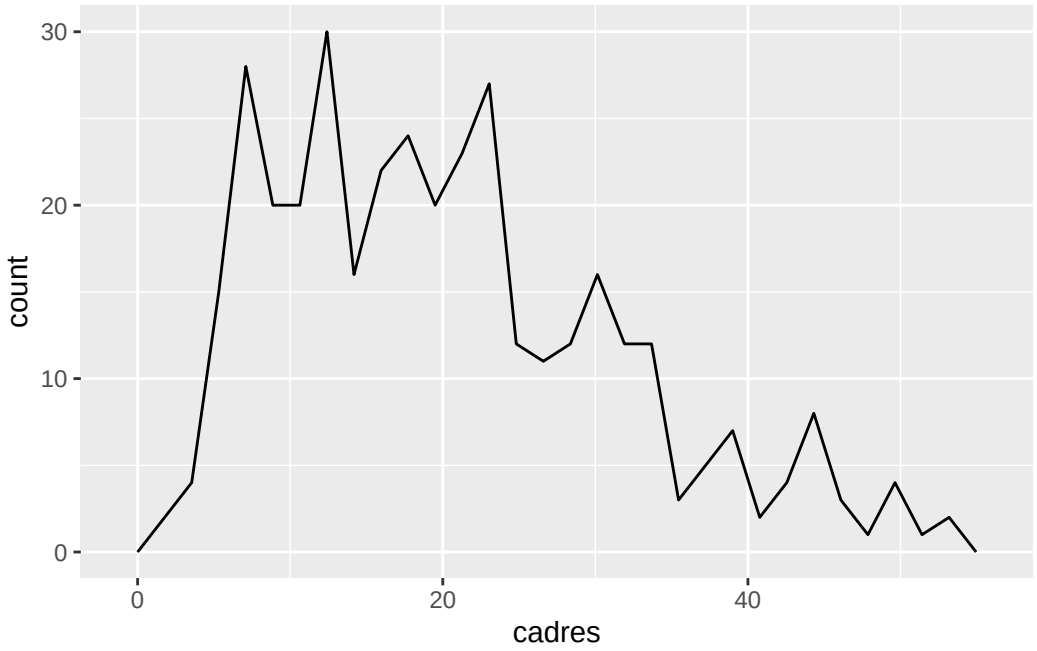


6.3.5 `geom_freqpoly`

`geom_freqpoly` permet d'afficher le polygone de fréquences d'une variable numérique. Son usage est similaire à celui de `geom_histogram`.

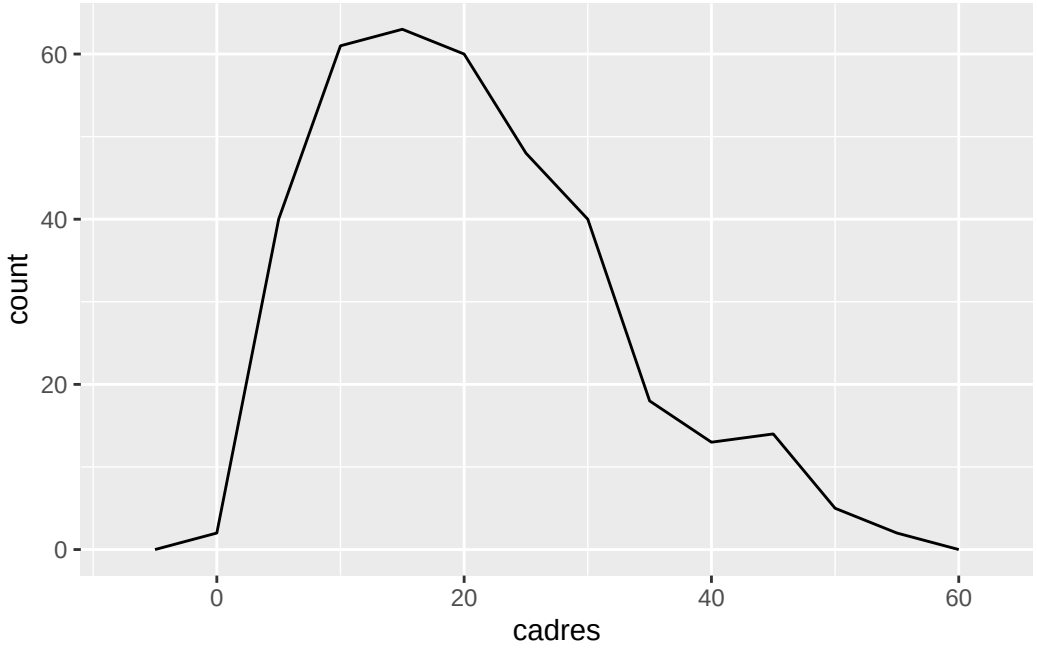
Ainsi, si on veut afficher le polygone de fréquences de la part des cadres dans les communes de notre jeu de données :

```
ggplot(rp) +
    geom_freqpoly(aes(x = cadres))
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



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

```
ggplot(rp) +
    geom_freqpoly(aes(x = cadres), binwidth = 5)
```



6.3.6 `geom_line`

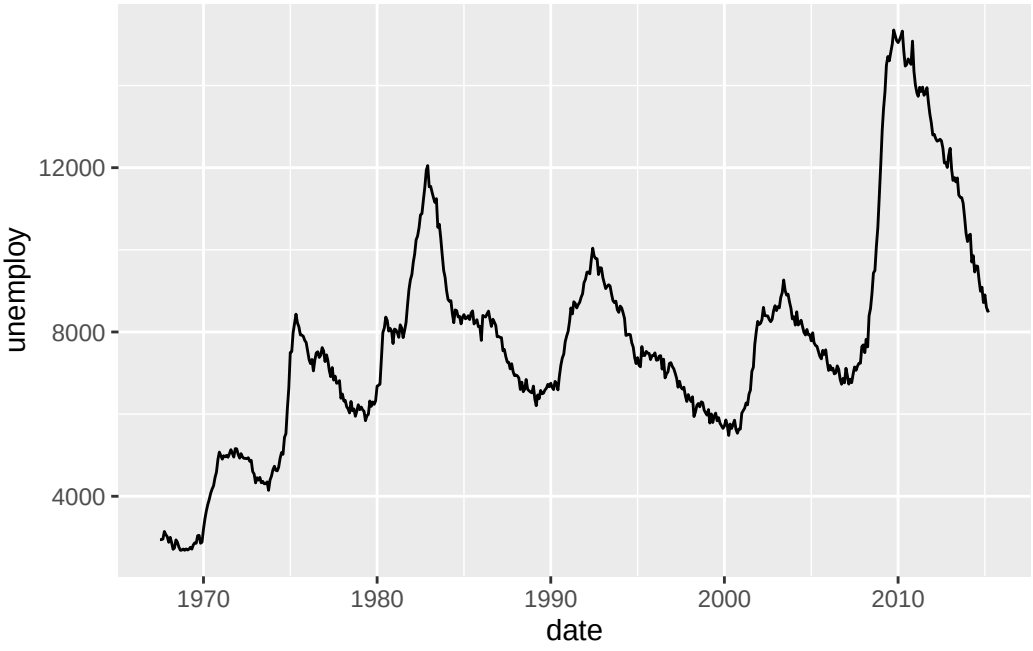
`geom_line` trace des lignes connectant les différentes observations entre elles. Il est notamment utilisé pour la représentation de séries temporelles. On passe à `geom_line` deux paramètres : `x` et `y`. Les observations sont alors connectées selon l'ordre des valeurs passées en `x`.

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

```
data("economics")
economics
#> # A tibble: 574 x 6
#>    date         pce    pop psavert uempmed unemploy
#>    <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
#>  1 1967-07-01  507. 198712    12.6     4.5     2944
#>  2 1967-08-01  510. 198911    12.6     4.7     2945
#>  3 1967-09-01  516. 199113    11.9     4.6     2958
#>  4 1967-10-01  512. 199311    12.9     4.9     3143
#>  5 1967-11-01  517. 199498    12.8     4.7     3066
#>  6 1967-12-01  525. 199657    11.8     4.8     3018
#>  7 1968-01-01  531. 199808    11.7     5.1     2878
```

```
#>  8 1968-02-01  534. 199920    12.3     4.5     3001
#>  9 1968-03-01  544. 200056    11.7     4.1     2877
#> 10 1968-04-01  544  200208    12.3     4.6     2709
#> # i 564 more rows
```

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



6.4 Mappages

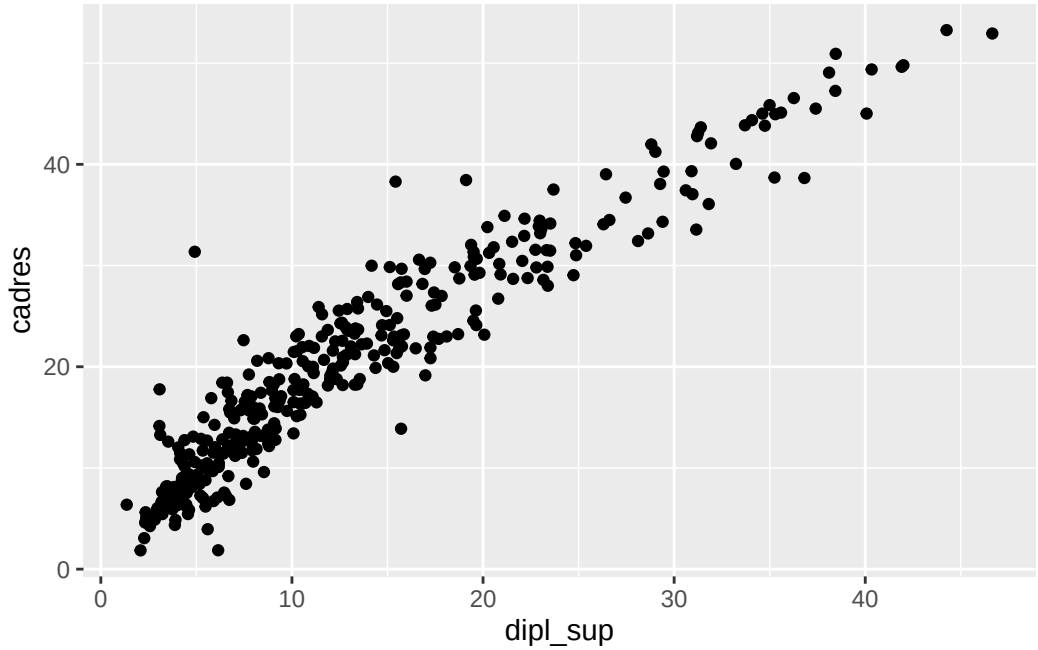
Un *mappage*, dans `ggplot2`, est une mise en relation entre un **attribut graphique** du `geom` (position, couleur, taille…) et une **variable** du tableau de données.

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

6.4.1 Exemples de mappages

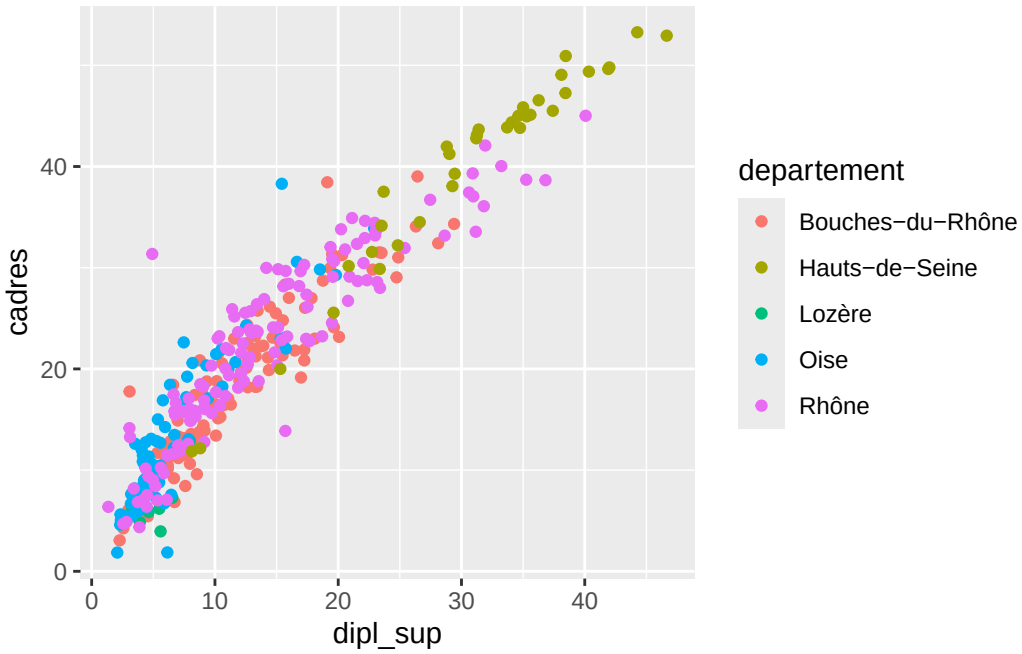
On a déjà vu les mappages `x` et `y` pour un nuage de points. Ceux-ci signifient que la position d'un point donné horizontalement (`x`) et verticalement (`y`) dépend de la valeur des variables passées comme arguments `x` et `y` dans `aes()`.

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



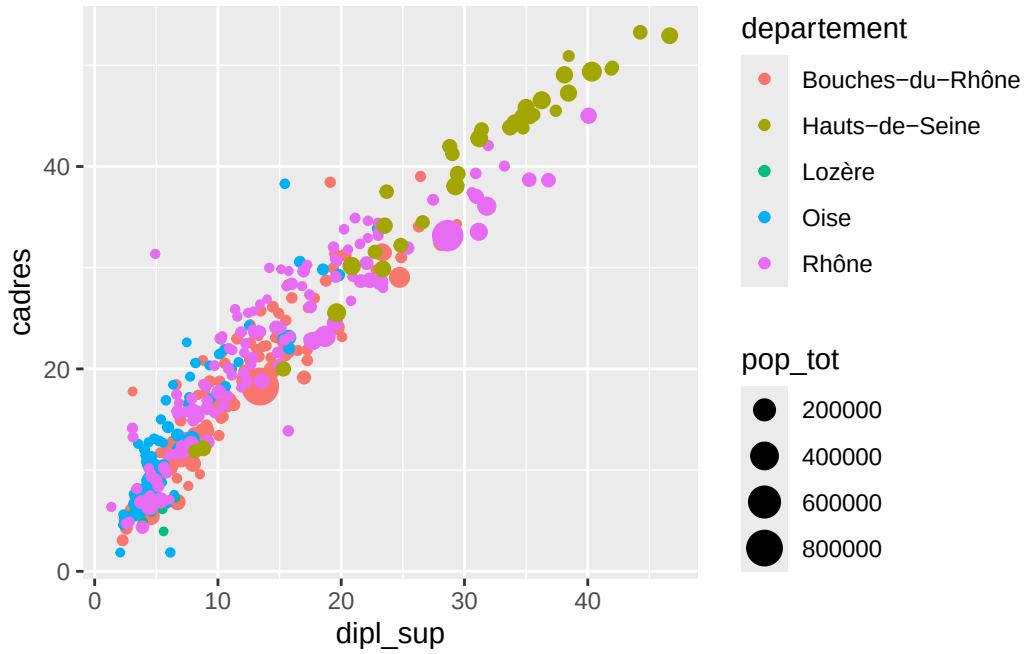
Mais on peut ajouter d'autres mappages. Par exemple, `color` permet de faire varier la couleur des points automatiquement en fonction des valeurs d'une troisième variable. Ainsi, on peut vouloir colorer les points selon le département de la commune correspondante.

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



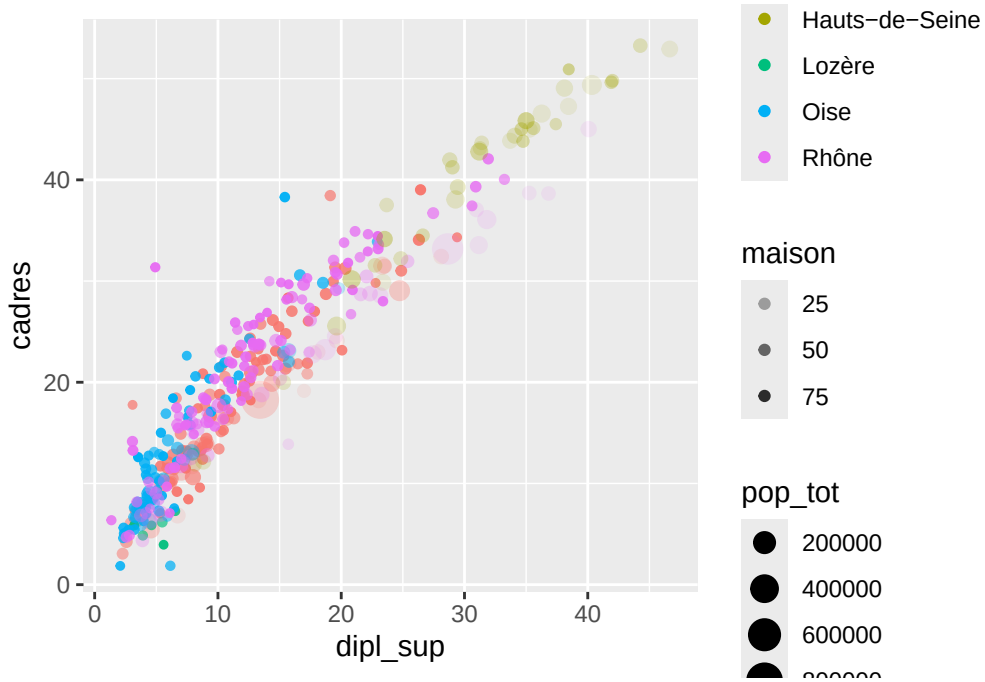
On peut aussi faire varier la taille des points avec `size`. Ici, la taille dépend de la population totale de la commune :

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



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

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

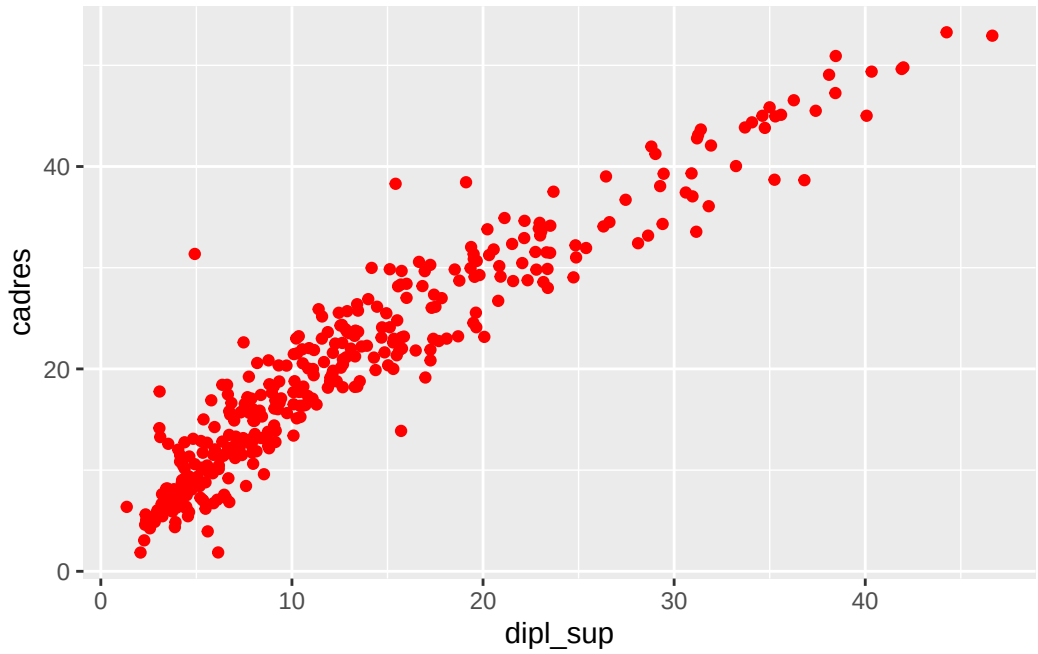


Chaque `geom` possède sa propre liste de mappages.

6.4.2 `aes()` or not `aes()` ?

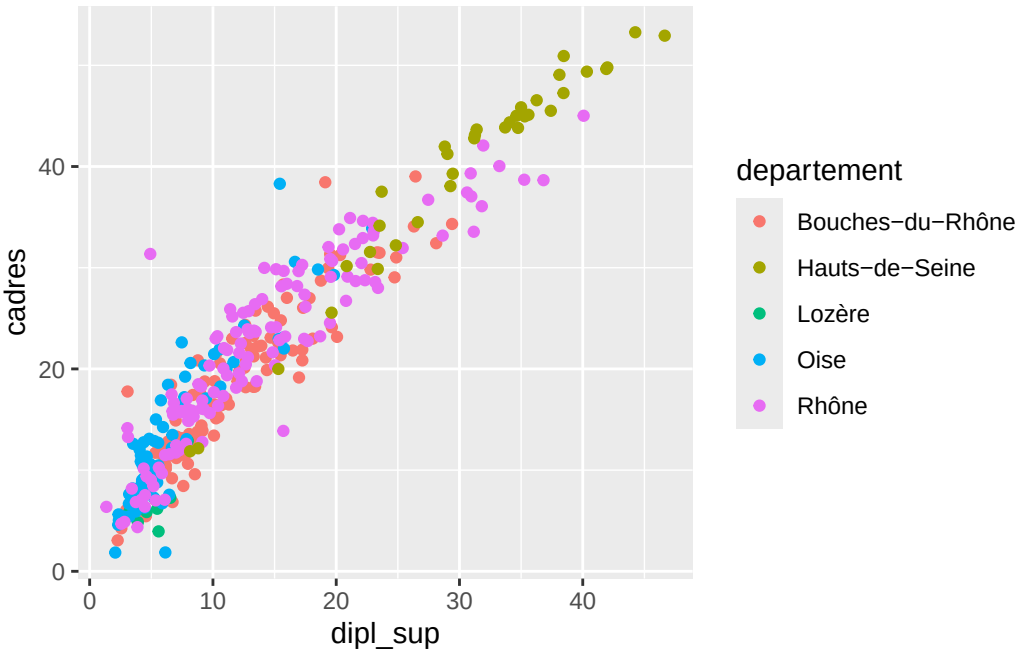
Comme on l'a déjà vu, parfois on souhaite changer un attribut sans le relier à une variable : c'est le cas par exemple si on veut représenter tous les points en rouge. Dans ce cas on utilise toujours l'attribut `color`, mais comme il ne s'agit pas d'un mappage, on le définit **à l'extérieur** de la fonction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

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



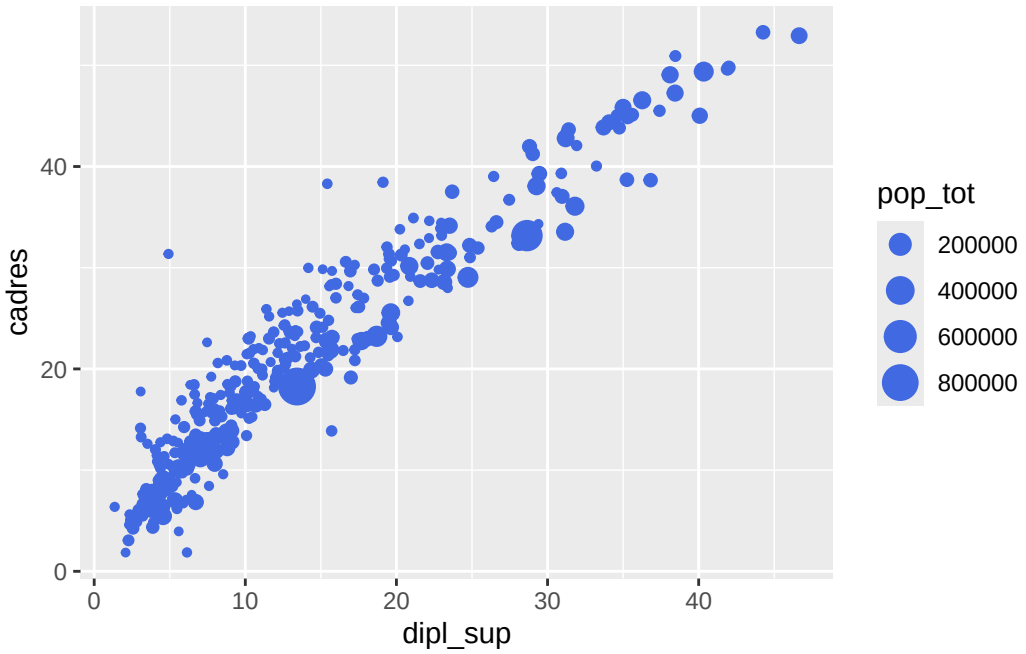
Par contre, si on veut faire varier la couleur en fonction des valeurs prises par une variable, on réalise un mappage, et on doit donc placer l'attribut `color` **à l'intérieur** de `aes()`.

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



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

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



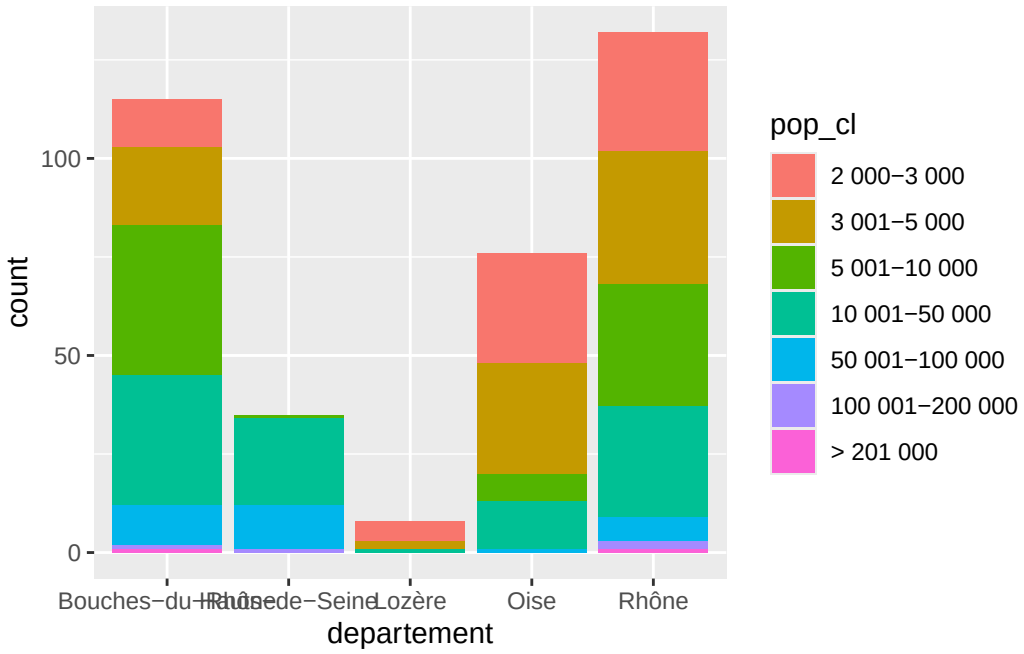
Avertissement

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

6.4.3 `geom_bar` et `position`

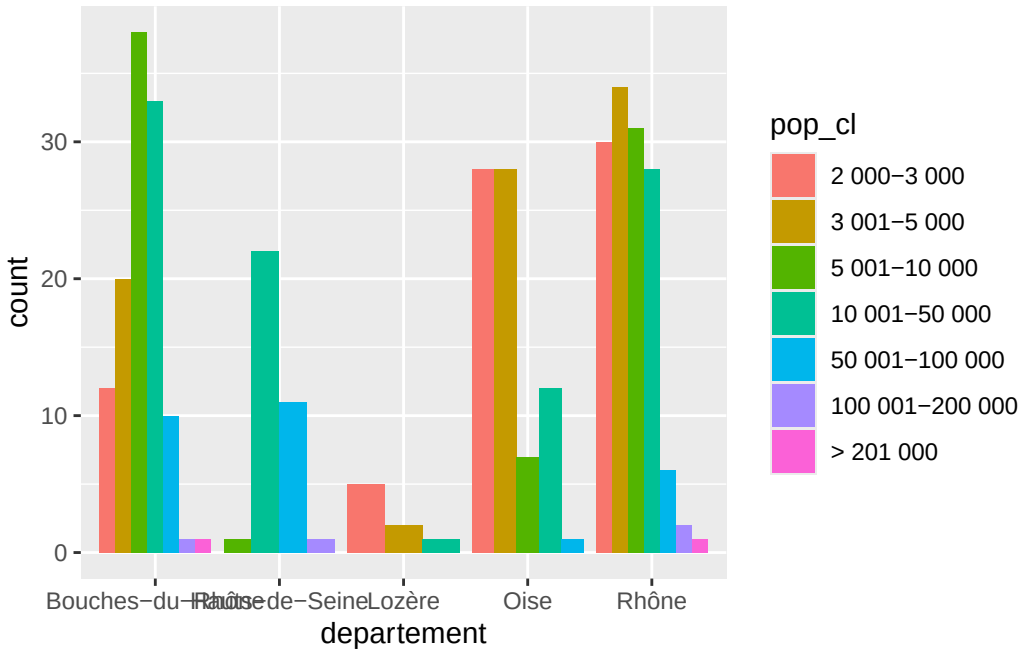
Un des mappages possibles de `geom_bar` est l'attribut `fill`, qui permet de tracer des barres de couleur différentes selon les modalités d'une deuxième variable :

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



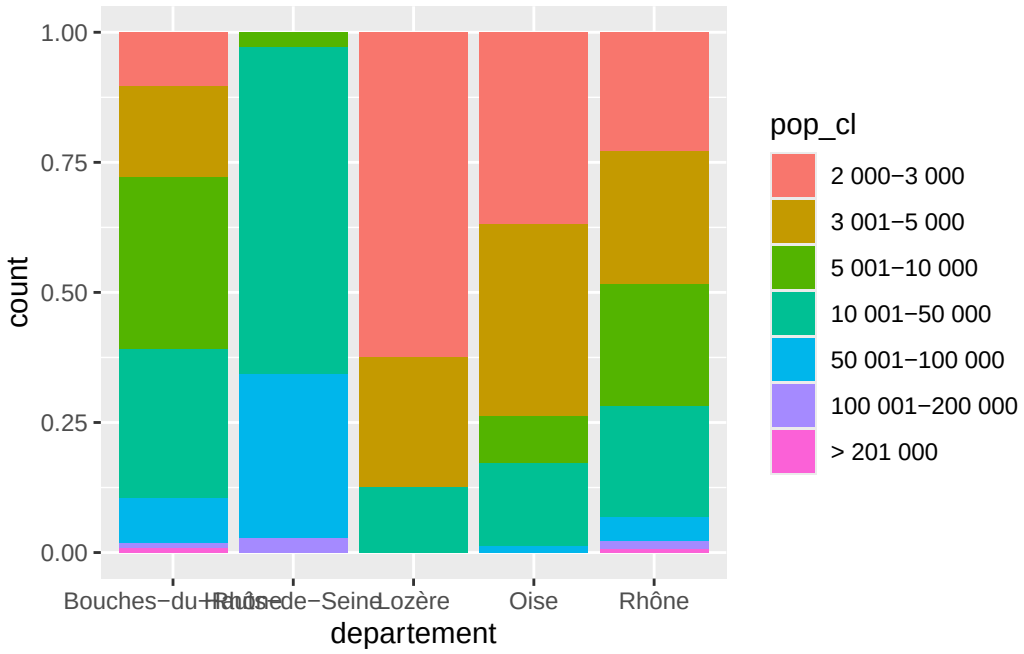
L'attribut `position` de `geom_bar` permet d'indiquer comment les différentes barres doivent être positionnées. Par défaut l'argument vaut `position = "stack"` et elles sont donc "empilées". Mais on peut préciser `position = "dodge"` pour les mettre côte à côte.

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



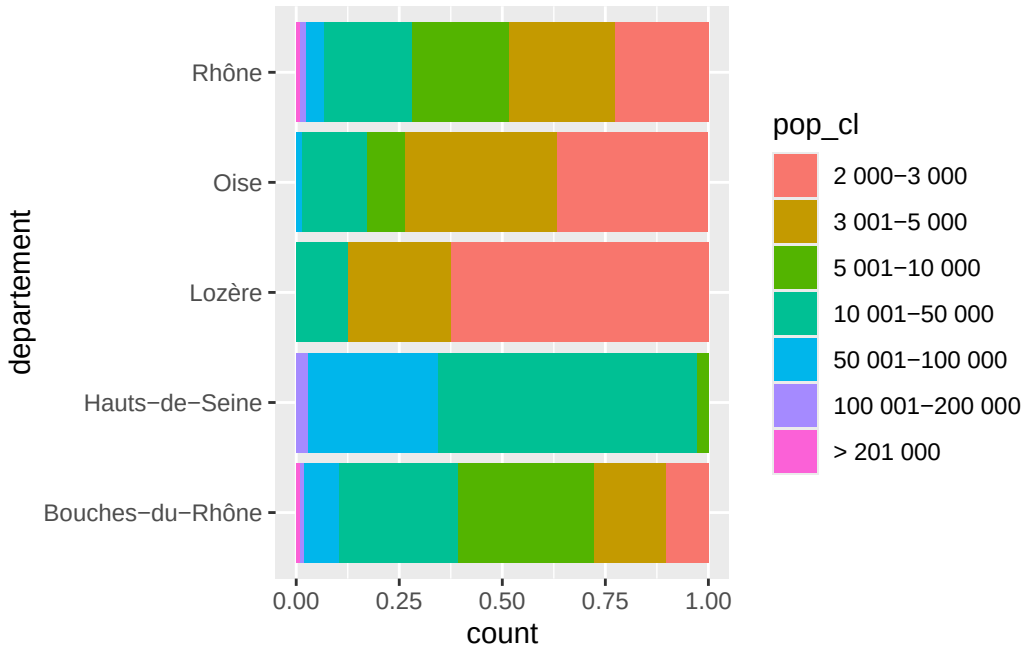
Ou encore `position = "fill"` pour représenter non plus des effectifs, mais des proportions.

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



Là encore, on peut utiliser `coord_flip()` si on souhaite une visualisation avec des barres horizontales.

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

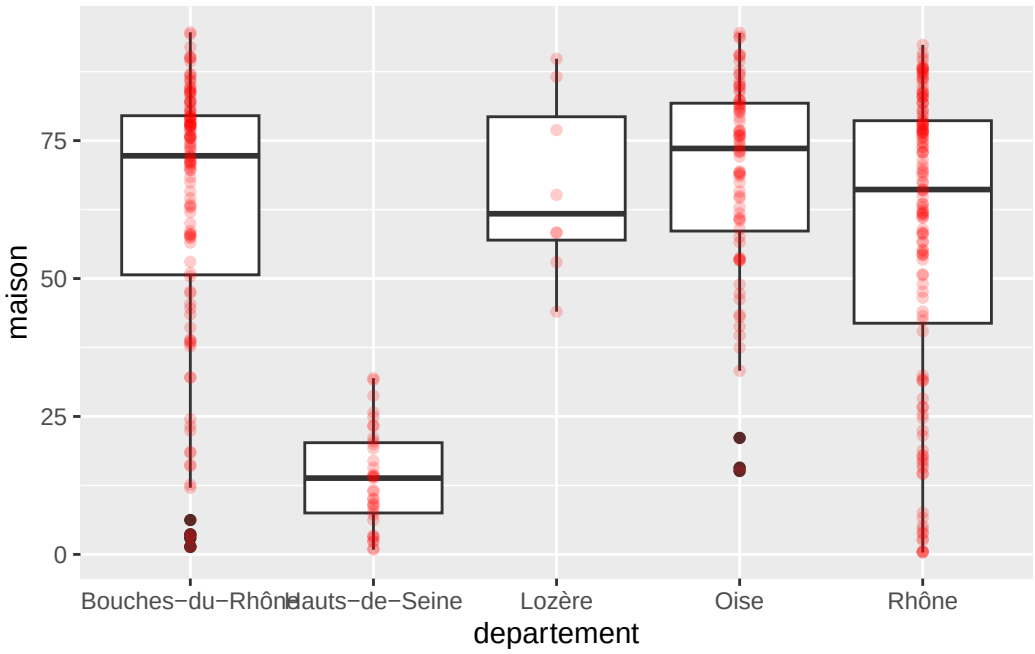


6.5 Représentation de plusieurs `geom`

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

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

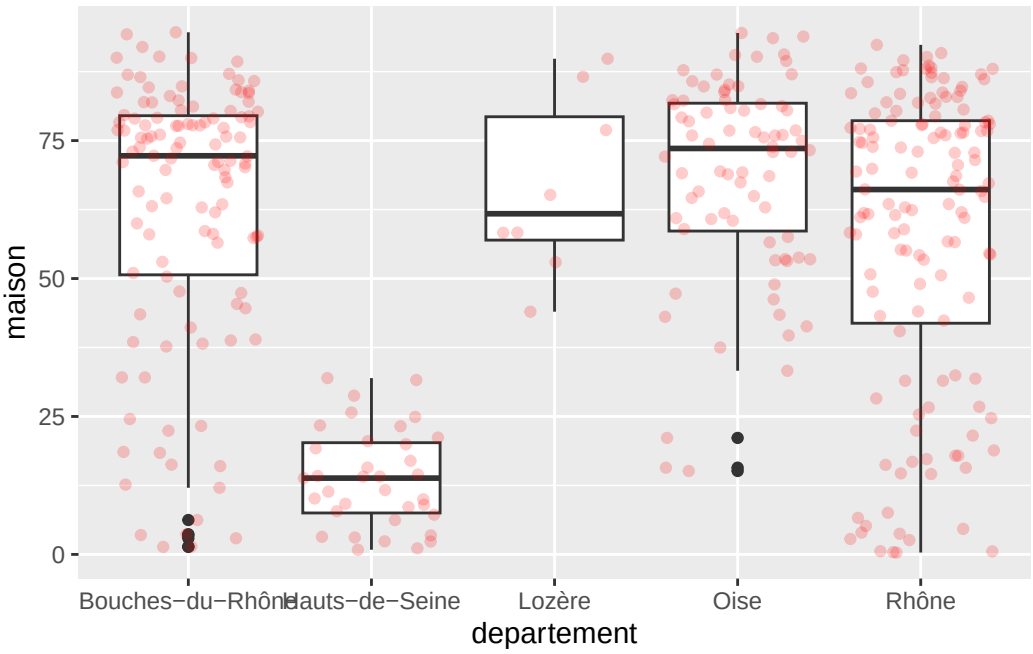
```
ggplot(rp) +
    geom_boxplot(aes(x = departement, y = maison)) +
    geom_point(
        aes(x = departement, y = maison),
        col = "red", alpha = 0.2
    )
```



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

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

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

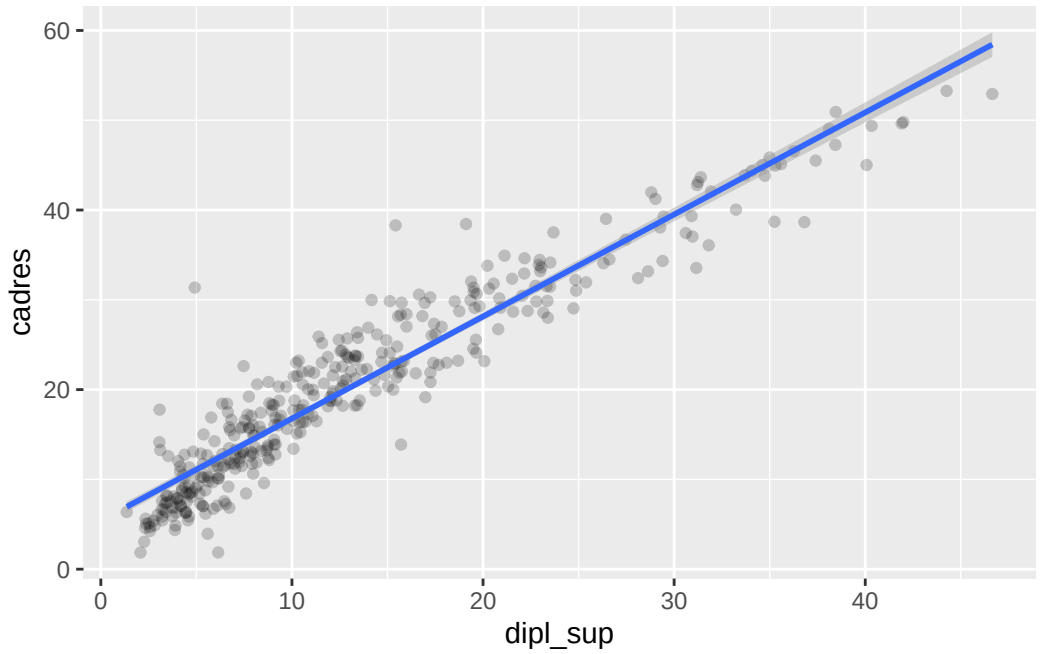


Pour simplifier un peu le code, plutôt que de déclarer les mappages dans chaque `geom`, on peut les déclarer dans l'appel à `ggplot()`. Ils seront automatiquement "hérités" par les `geom` ajoutés (sauf s'ils redéfinissent les mêmes mappages).

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

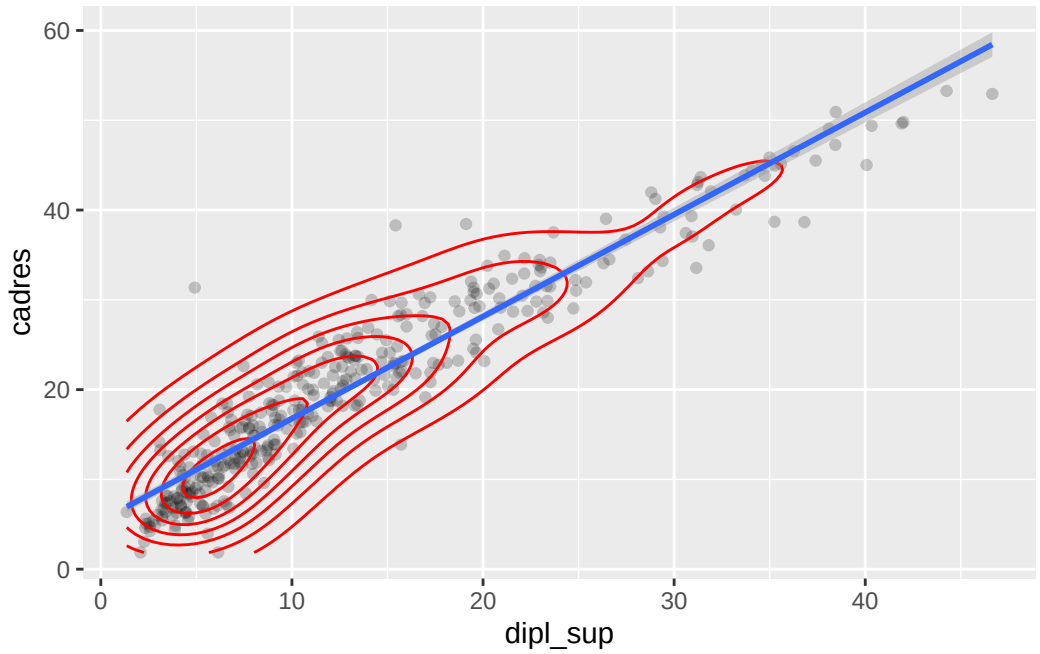
Autre exemple, on peut vouloir ajouter à un nuage de points une ligne de régression linéaire à l'aide de `geom_smooth` :

```
ggplot(rp, aes(x = dipl_sup, y = cadres)) +
    geom_point(alpha = 0.2) +
    geom_smooth(method = "lm")
#> `geom_smooth()` using formula = 'y ~ x'
```



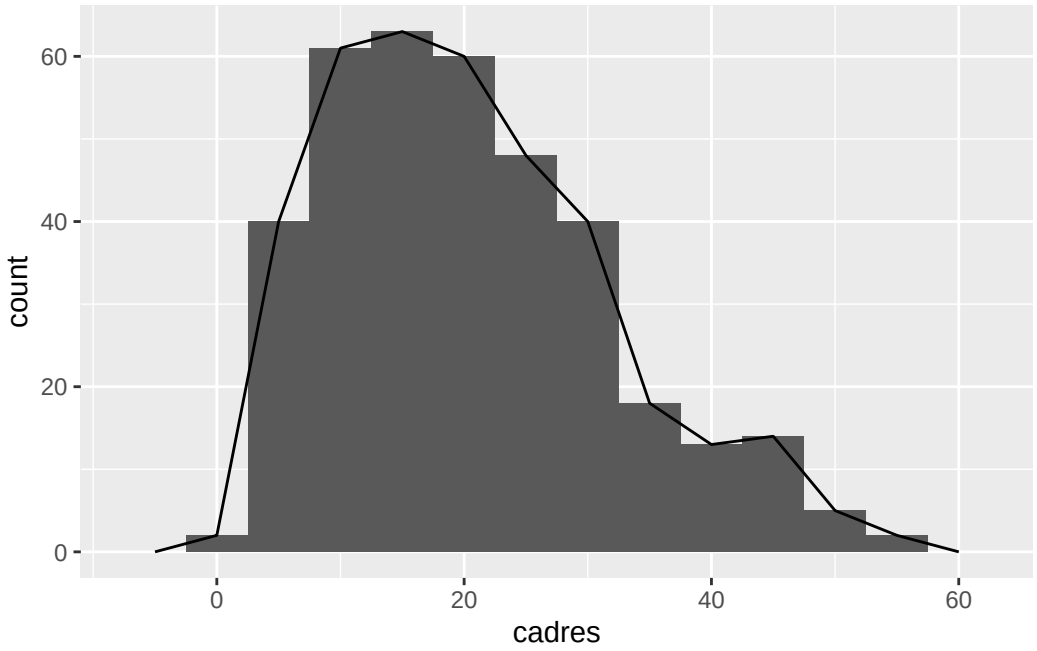
Et on peut même superposer une troisième visualisation de la répartition des points dans l'espace avec `geom_density2d` :

```
ggplot(rp, aes(x = dipl_sup, y = cadres)) +
    geom_point(alpha = 0.2) +
    geom_density2d(color = "red") +
    geom_smooth(method = "lm")
#> `geom_smooth()` using formula = 'y ~ x'
```



On peut enfin superposer l'histogramme ainsi que le polygone de fréquences.

```
ggplot(rp) +
    geom_histogram(aes(x = cadres), binwidth = 5) +
    geom_freqpoly(aes(x = cadres), binwidth = 5)
```

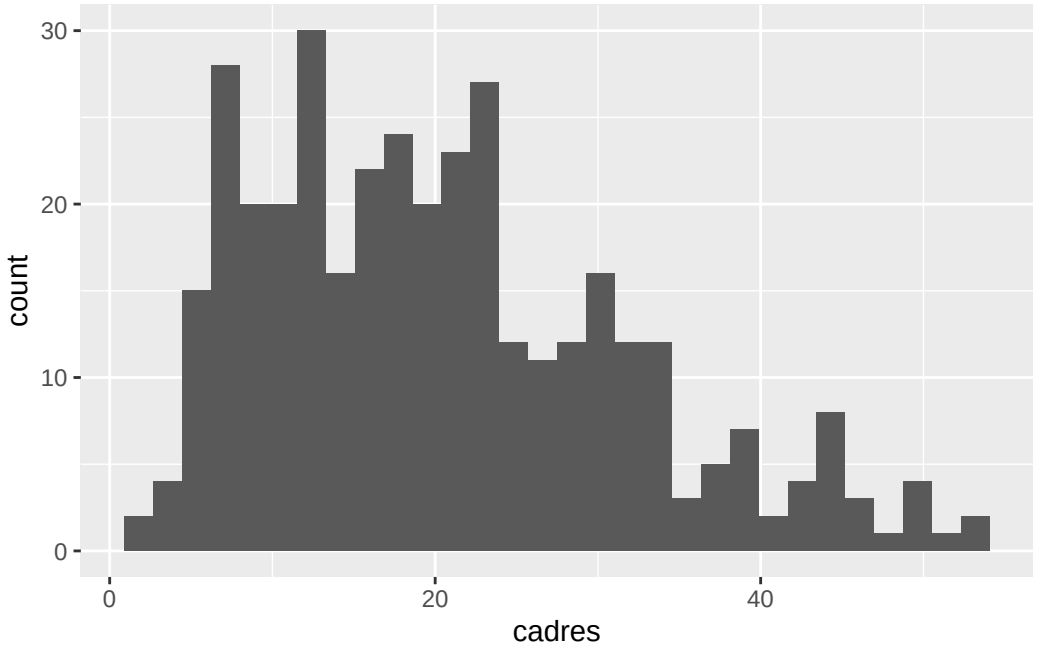


6.6 Faceting

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

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

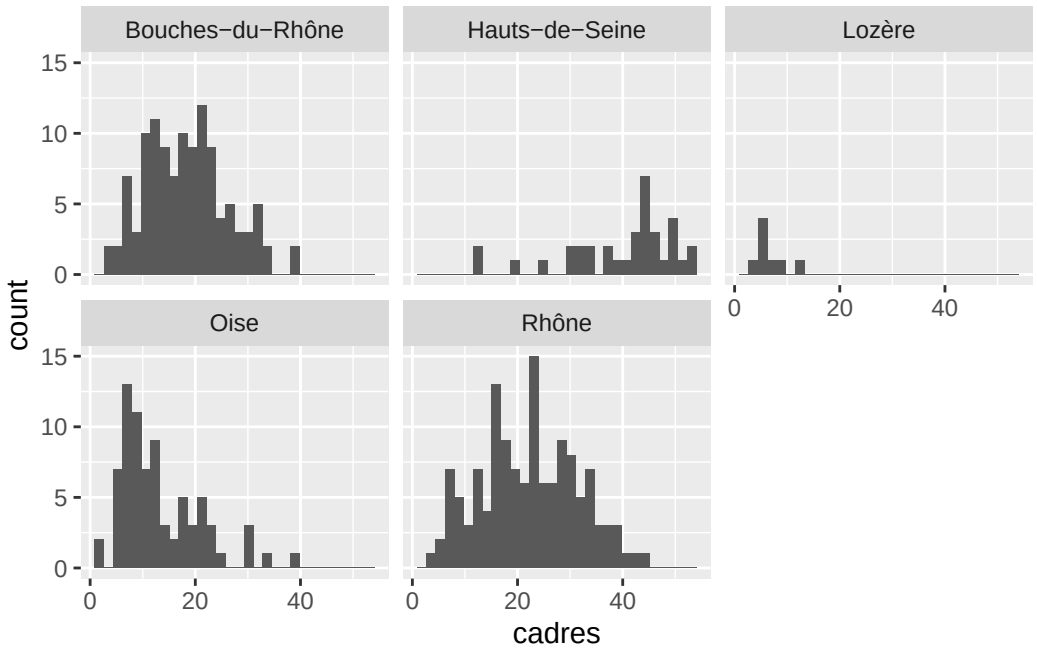
```
ggplot(data = rp) +
    geom_histogram(aes(x = cadres))
```



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

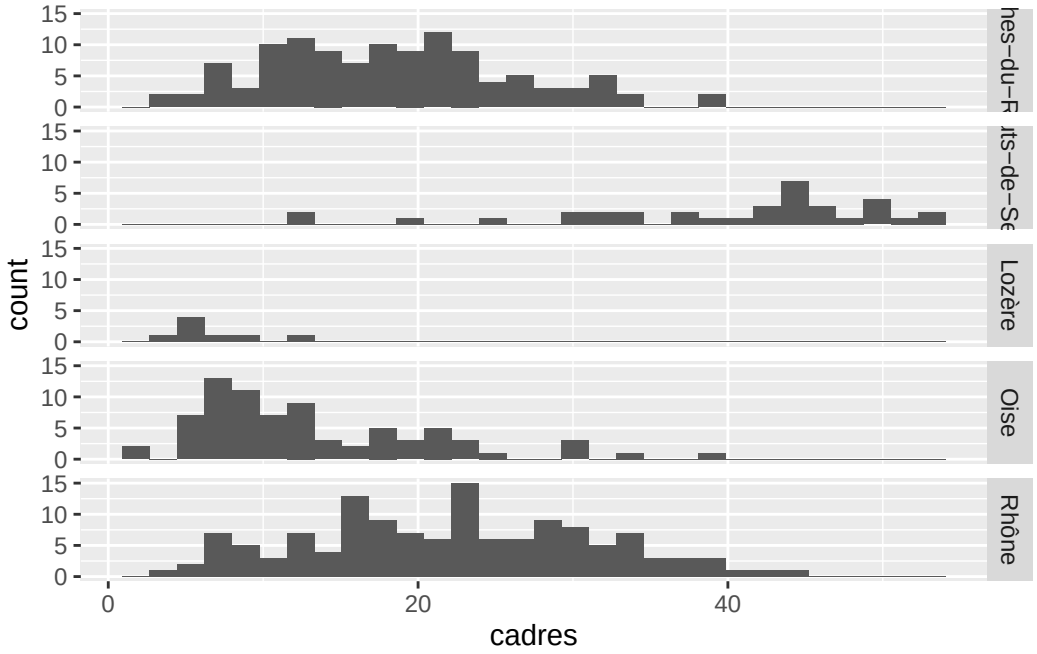
`facet_wrap` prend un paramètre de la forme `vars(variable)`, où `variable` est le nom de la variable en fonction de laquelle on souhaite faire les différents graphiques. Ceux-ci sont alors affichés les uns à côté des autres et répartis automatiquement dans la page.

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



Pour `facet_grid`, les graphiques sont disposés selon une grille. La fonction prend alors deux arguments, `rows` et `cols`, auxquels on passe les variables à afficher en ligne ou en colonne via la fonction `vars()`.

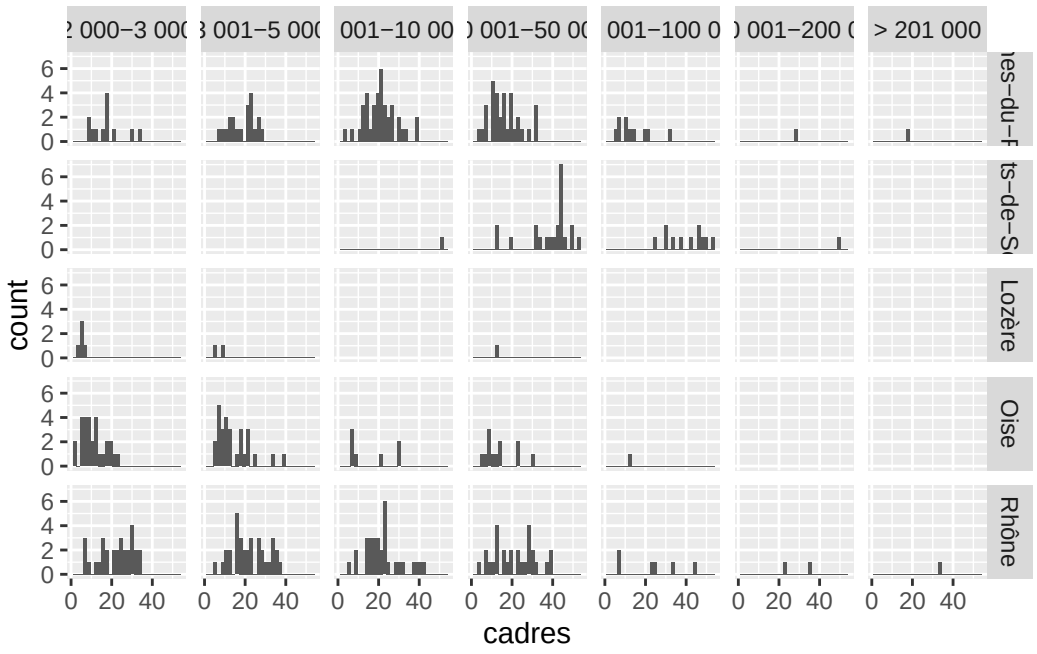
```
ggplot(data = rp) +
    geom_histogram(aes(x = cadres)) +
    facet_grid(rows = vars(departement))
```



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

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

```
ggplot(data = rp) +
    geom_histogram(aes(x = cadres)) +
    facet_grid(
        rows = vars(departement), cols = vars(pop_cl)
    )
```



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

6.7 Ressources

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

Une "antisèche" (en anglais) résumant en deux pages l'ensemble des fonctions et arguments et disponible soit directement depuis RStudio (menu *Help > Cheatsheets > Data visualization with ggplot2*) ou en ligne.

Les parties Data visualisation et Graphics for communication de l'ouvrage en ligne *R for data science*, de Hadley Wickham, sont une très bonne introduction à `ggplot2`.

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

Le site associé à ce dernier ouvrage comporte aussi pas mal d'exemples et d'informations intéressantes.

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

7 Les mesures

Les mesures de tendance centrale permettent de déterminer où se situe le "centre" des données. Les trois mesures de tendance centrale sont le mode, la moyenne et la médiane.

7.1 Les mesures de tendance centrale

7.1.1 Le mode

Le mode est la **modalité**, **valeur** ou **classe** possédant la plus grande fréquence. En d'autres mots, c'est la donnée la plus fréquente.

Puisque le mode se préoccupe seulement de la donnée la plus fréquente, il n'est pas influencé par les valeurs extrêmes.

Lorsque le mode est une classe, il est appelé **classe modale**.

Le mode est noté **Mo**.

Le langage `R` ne possède pas de fonction permettant de calculer le mode. La façon la plus simple de le calculer est d'utiliser la fonction `table` de `R`.

Par exemple, si nous voulons connaître le mode de la variable `marital` de la base de données `gss_cat`:

```
table(gss_cat$marital)
#>
#>     No answer Never married     Separated      Divorced       Widowed
#>            17          5416           743          3383          1807
#>       Married
#>         10117
```

Nous remarquons que le maximum est à la modalité *Married* avec une fréquence de 10117.

Si nous nous intéressons au mode d'une variable quantitative discrète comme `age` de la base de données `gss_cat` nous obtenons:

```
table(gss_cat$age)
#>
#>  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37
#>  91 249 251 278 298 361 344 396 400 385 387 376 433 407 445 425 425 417 428 438
#>  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57
#> 426 415 452 434 405 448 432 404 422 435 424 417 430 390 400 396 387 365 384 321
#>  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77
#> 326 323 338 307 310 292 253 259 231 271 205 201 213 206 189 152 180 179 171 137
#>  78  79  80  81  82  83  84  85  86  87  88  89
#> 150 135 127 119 105  99 100  75  74  54  57 148
```

Nous remarquons que le maximum est à la valeur *40* avec une fréquence de 452.

Dans le cas d'une variable quantitative continue, pour calculer le mode, il faut commencer par séparer les données en classes. Nous utiliserons les mêmes classes utilisées à la section:

```
carat_class = cut(diamonds$carat,
                  breaks = seq(from = 0, to = 6, by = 1),
                  right = FALSE)
table(carat_class)
#> carat_class
#> [0,1) [1,2) [2,3) [3,4) [4,5) [5,6)
#> 34880 16906  2114    34     5     1
```

La classe modale est donc la classe *[0,1)* avec une fréquence de 34880.

7.1.2 La médiane

La médiane, notée **Md**, est la valeur qui sépare une série de données classée en ordre croissant en deux parties égales.

La médiane étant la valeur du milieu, elle est la valeur où le pourcentage cumulé atteint 50%.

Puisque la médiane se préoccupe seulement de déterminer où se situe le centre des données, elle n'est pas influencée par les valeurs extrêmes. Elle est donc une mesure de tendance centrale plus fiable que la moyenne.

> Important : La médiane n'est définie que pour les variables quantitatives. En effet, si vous tentez d'utiliser la médiane pour des données autres que numériques, `R` vous donnera un message d'erreur.

La fonction `median` permet de calculer la médiane en langage `R`.

Par exemple, pour calculer la médiane de la variable `carat` de la base de données `diamonds`, nous avons:

```
median(diamonds$carat)
#> [1] 0.7
```

Ceci signifie que 50% des diamants ont une valeur en carat inférieure ou égale à 0.7 et que 50% des diamants ont une valeur en carat supérieure ou égale à 0.7.

Nous pouvons aussi obtenir que la médiane de la variable `price` de la base de données `diamonds` est donnée par:

```
median(diamonds$price)
#> [1] 2401
```

7.1.3 La moyenne

La moyenne est la valeur qui pourrait remplacer chacune des données d'une série pour que leur somme demeure identique. Intuitivement, elle représente le centre d'équilibre d'une série de données. La somme des distances qui sépare les données plus petites que la moyenne devrait être la même que la somme des distances qui sépare les données plus grandes.

> Important : La moyenne n'est définie que pour les variables quantitatives. En effet, si vous tentez d'utiliser la moyenne pour des données autres que numériques, `R` vous donnera un message d'erreur.

La fonction `mean` permet de calculer la moyenne en langage `R`.

Par exemple, pour calculer la moyenne de la variable `carat` de la base de données `diamonds`, nous avons:

```
mean(diamonds$carat)
#> [1] 0.7979397
```

Nous pouvons aussi obtenir que la moyenne de la variable `price` de la base de données `diamonds` est donnée par:

```
mean(diamonds$price)
#> [1] 3932.8
```

7.2 Les mesures de dispersion

Les mesures de tendance centrale (mode, moyenne et médiane) ne permettent pas de déterminer si une série de données est principalement située autour de son centre, ou si au contraire elle est très dispersée.

Les mesures de dispersion, elles, permettent de déterminer si une série de données est centralisée autour de sa moyenne, ou si elle est au contraire très dispersée.

Les mesures de dispersion sont l'étendue, la variance, l'écart-type et le coefficient de variation.

7.2.1 L'étendue

La première mesure de dispersion, l'étendue, est la différence entre la valeur maximale et la valeur minimale.

L'étendue ne tenant compte que du maximum et du minimum, elle est grandement influencée par les valeurs extrêmes. Elle est donc une mesure de dispersion peu fiable.

La fonction `range` permet de calculer l'étendue d'une variable en langage `R`.

Par exemple, pour calculer l'étendue de la variable `carat` de la base de données `diamonds`, nous avons:

```
range(diamonds$carat)
#> [1] 0.20 5.01
```

Nous pouvons donc calculer l'étendue de la variable `carat` en soustrayant les deux valeurs obtenues par la fonction `range`, c'est-à-dire que l'étendue est 5.01-0.2 = 4.81.

7.2.2 La variance

La variance sert principalement à calculer l'écart-type, la mesure de dispersion la plus connue.

> Attention : Les unités de la variance sont des unités^2.

La fonction `var` permet de calculer la variance d'une variable en langage `R`.

Par exemple, pour calculer la variance de la variable `carat` de la base de données `diamonds`, nous avons:

```
var(diamonds$carat)
#> [1] 0.2246867
```

Ceci signifie que la variance de la variable `carat` est 0.2246867 carat^2.

7.2.3 L'écart-type

L'écart-type est la mesure de dispersion la plus couramment utilisée. Il peut être vu comme la « moyenne » des écarts entre les données et la moyenne.

Puisque l'écart-type tient compte de chacune des données, il est une mesure de dispersion beaucoup plus fiable que l'étendue.

Il est défini comme la racine carrée de la variance.

La fonction `sd` permet de calculer l' 'écart-type d'une variable en langage `R`.

Par exemple, pour calculer l'écart-type de la variable `carat` de la base de données `diamonds`, nous avons:

```
sd(diamonds$carat)
#> [1] 0.4740112
```

Ceci signifie que l'écart-type de la variable `carat` est 0.4740112 carat.

7.2.4 Le coefficient de variation

Le coefficient de variation, noté C. V., est calculé comme suit :

$$C.V. = \frac{\text{ecart-type}}{\text{moyenne}} \times 100\% \tag{7.1}$$

Si le coefficient est inférieur à 15%, les données sont dites **homogènes**. Cela veut dire que les données sont situées près les unes des autres.

Dans le cas contraire, les données sont dites **hétérogènes**. Cela veut dire que les données sont très dispersées.

> Important : Le coefficient de variation ne possède pas d'unité, outre le symbole de pourcentage.

Il n'existe pas de fonctions en `R` permettant de calculer directement le coefficient de variation. Par contre, nous pouvons utiliser en conjonction les fonctions `sd` et `mean` pour le calculer.

Par exemple, pour calculer le coefficient de variation de la variable `carat` de la base de données `diamonds`, nous avons:

```
sd(diamonds$carat)/mean(diamonds$carat)*100
#> [1] 59.40439
```

Le C.V. de la variable `carat` est donc 59.4043906 %, ce qui signifie que les données sont hétérogènes, car le coefficient de variation est plus grand que 15%.

7.3 Les mesures de position

Les mesures de position permettent de situer une donnée par rapport aux autres. Les différentes mesures de position sont la cote Z, les quantiles et les rangs.

Tout comme les mesures de dispersion, celles-ci ne sont définies que pour une variable quantitative.

7.3.1 La cote z

Cette mesure de position se base sur la moyenne et l'écart-type.

La cote Z d'une donnée x est calculée comme suit :

$$Z = \frac{x - \text{moyenne}}{\text{ecart-type}} \tag{7.2}$$

Important : La cote z ne possède pas d'unités.

Une cote Z peut être positive, négative ou nulle.

Cote Z	Interprétation
Z>0	donnée supérieure à la moyenne
Z<0	donnée inférieure à la moyenne
Z=0	donnée égale à la moyenne

Il n'existe pas de fonctions en `R` permettant de calculer directement la cote Z. Par contre, nous pouvons utiliser en conjonction les fonctions `sd` et `mean` pour la calculer.

Par exemple, si nous voulons calculer la cote Z d'un diamant de 3 carats, nous avons:

```
(3-mean(diamonds$carat))/sd(diamonds$carat)
#> [1] 4.645587
```

7.3.2 Les quantiles

Un quantile est une donnée qui correspond à un certain pourcentage cumulé.

Parmi les quantiles, on distingue les quartiles, les quintiles, les déciles et les centiles.

- Les quartiles Q_1, Q_2 et Q_3, séparent les données en quatre parties égales. Environ 25% des données sont inférieures ou égales à Q_1. Environ 50% des données sont inférieures ou égales à Q_2. Environ 75% des données sont inférieures ou égales à Q_3.
- Les quintiles V_1, V_2, V_3 et V_4, séparent les données en cinq parties égales. Environ 20% des données sont inférieures ou égales à V_1. Environ 40% des données sont inférieures ou égales à V_2. Etc.
- Les déciles D_1, D_2, ..., D_8 et D_9, séparent les données en dix parties égales. Environ 10% des données sont inférieures ou égales à D_1. Environ 20% des données sont inférieures ou égales à D_2. Etc.
- Les centiles C_1, C_2, ..., C_{98} et C_{99}, séparent les données en cent parties égales. Environ 1% des données sont inférieures ou égales à C_1. Environ 2% des données sont inférieures ou égales à C_2. Etc.

  Il est utile de noter que certains quantiles se recoupent.

La fonction `quantile` permet de calculer n'importe quel quantile d'une variable en langage `R`. Il suffit d'indiquer la variable étudiée ainsi que le pourcentage du quantile voulu.

Par exemple, si nous voulons calculer D_1 pour la variable `carat`, nous allons utiliser la fonction `quantile` avec une probabilité de 0,1.

```
quantile(diamonds$carat, 0.1)
#>  10%
#> 0.31
```

Ceci implique que 10% des diamants ont une valeur en carat inférieure ou égale à 0.31 carat.

Nous pouvons calculer le troisième quartile Q_3 de la variable `price` en utilisant la fonction `quantile` avec une probabilité de 0,75.

```
quantile(diamonds$price, 0.75)
#>     75%
#> 5324.25
```

Ceci implique que 75% des diamants ont un prix en dollars inférieur ou égal à 5324.25 $.

7.3.3 La commande `summary`

La commande `summary` produit un sommaire contenant six mesures importantes:

1. `Min` : le minimum de la variable
2. `1st Qu.`: Le premier quartile, Q_1, de la variable
3. `Median` : La médiane de la variable
4. `Mean` : La moyenne de la variable
5. `3rd Qu.` : Le troisième quartile, Q_3, de la variable
6. `Max` : Le maximum de la variable

Nous pouvons donc produire le sommaire de la variable `price` de la base de données `diamonds` de la façon suivante:

```
summary(diamonds$price)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     326     950    2401    3933    5324   18823
```

7.3.4 Le rang centile

Un rang centile représente le pourcentage cumulé, *exprimé en nombre entier*, qui correspond à une certaine donnée. Nous déterminerons les rangs centiles pour les variables continues seulement.

Les rangs centiles sont donc exactement l'inverse des centiles.

Il n'existe pas de fonctions dans `R` permettant de trouver directement le rang centile, mais il est facile d'utiliser la fonction `mean` pour le trouver.

Par exemple, si nous voulons trouver le rang centile d'un diamant qui coûte 500$, il suffit d'utiliser la commande suivante. La commande calcule la moyenne de toutes les valeurs en dollars des diamants coûtant 500$ ou moins.

```
floor(mean(diamonds$price<=500)*100)
#> [1] 3
```

Ceci signifie que pour un diamant de 500$, il y a 3 % des diamants qui ont une valeur égale ou inférieure.

partie IV

L'estimation et les tests d'hypothèses

8 L'estimation de paramètres

Nous allons utiliser la librairie `infer` pour faire de l'estimation de paramètres, ainsi que la base de données `gss`, présente dans la librairie.

```
library(infer)
data(gss)
```

8.1 L'intervalle de confiance sur une moyenne

Pour trouver un intervalle de confiance sur une moyenne, nous utilisons la fonction `t_test`.

Les quatres arguments nécessaires sont:

- `x`: la base de données à utiliser, sous forme de *tibble*.
- `response`: la variable quantitative dont on veut connaître l'intervalle de confiance pour la moyenne.
- `alternative`: pour un intervalle de confiance, on utilise toujours la valeur `two-sided`.
- `conf_level`: un niveau de confiance entre 0 et 1.

Par exemple, si on veut trouver un intervalle de confiance à 95% pour la moyenne de la variable `age`, nous utilisons:

```
t_test( x = gss,
        response = age,
        alternative = "two-sided",
        conf_level = 0.95)
#> # A tibble: 1 x 7
#>   statistic  t_df   p_value alternative estimate lower_ci upper_ci
#>       <dbl> <dbl>     <dbl> <chr>          <dbl>    <dbl>    <dbl>
#> 1      67.6   499 2.42e-253 two.sided       40.3     39.1     41.4
```

La borne inférieure de l'intervalle de confiance est donnée par la variable `lower_ci` et la borne supérieure par la variable `upper_ci`. Dans notre test, nous avons donc un intervalle de confiance entre 39.095674 et 41.436326.

8.2 L'intervalle de confiance sur une proportion

Pour trouver un intervalle de confiance sur une proportion, nous utilisons la fonction `prop_test`.

Les cinq arguments nécessaires sont:

- `x`: la base de données à utiliser, sous forme de *tibble*.
- `response`: la variable quantitative dont on veut connaître l'intervalle de confiance pour la proportion.
- `success`: la modalité de la variable que nous considérons comme un succès.
- `alternative`: pour un intervalle de confiance, on utilise toujours la valeur `two-sided`.
- `conf_level`: un niveau de confiance entre 0 et 1.

Par exemple, si on veut trouver un intervalle de confiance à 95% pour la proportion de `female` de la variable `age`, nous utilisons:

```
prop_test(  x = gss,
            response = sex,
            success = "female",
            alternative = "two-sided",
            conf_level = 0.95)
#> No `p` argument was hypothesized, so the test will assume a null hypothesis `p
#> = .5`.
#> # A tibble: 1 x 6
#>   statistic chisq_df p_value alternative lower_ci upper_ci
#>       <dbl>    <int>   <dbl> <chr>          <dbl>    <dbl>
#> 1      1.25        1   0.264 two.sided      0.430    0.519
```

```
#> No `p` argument was hypothesized, so the test will assume a null hypothesis `p
#> = .5`.
```

La borne inférieure de l'intervalle de confiance est donnée par la variable `lower_ci` et la borne supérieure par la variable `upper_ci`. Dans notre test, nous avons donc un intervalle de confiance entre 0.4296103 et 0.5187952.

9 Les tests d'hypothèses

Nous allons utiliser la librairie `infer` pour faire des tests d'hypothèses, ainsi que la base de données `gss`, présente dans la librairie.

```
library(infer)
data(gss)
```

9.1 Les tests d'hypothèses sur une moyenne

Pour effectuer un test d'hypothèses sur une moyenne, nous utilisons la fonction `t_test`.

Les quatres arguments nécessaires sont:

- `x`: la base de données à utiliser, sous forme de *tibble*.
- `response`: la variable dont on veut tester l'hypothèse.
- `mu`: la valeur de la moyenne à l'hypothèse H_0.
- `alternative`:
  - `less`: pour un test unilatéral à gauche
  - `two-sided`: pour un test bilatéral
  - `greater`: pour un test unilatéral à droite

Par exemple, si on veut tester si la moyenne de la variable `age` est plus grande que 39 ans, à un niveau de confiance de 98%, nous utilisons:

```
t_test( x = gss,
        response = age,
        mu = 39,
        alternative = "greater"
)
#> # A tibble: 1 x 7
#>   statistic  t_df p_value alternative estimate lower_ci upper_ci
#>       <dbl> <dbl>   <dbl> <chr>          <dbl>    <dbl>    <dbl>
#> 1      2.13   499  0.0170 greater         40.3     39.3      Inf
```

La variable `p_value` nous permet de conserver ou de rejetter H_0. En effet, dans notre cas, la valeur est de 0.0170242 qui est plus petite que le risque d'erreur de 2% (associé au niveau de confiance de 98%). On rejette donc H_0 et on accepte H_1, l'âge moyen est plus grand que 39 ans.

9.2 Les tests d'hypothèses sur une proportion

Pour effectuer un test d'hypothèses sur une proportion, nous utilisons la fonction `prop_test`.

Les quatres arguments nécessaires sont:

- `x`: la base de données à utiliser, sous forme de *tibble*.
- `response`: la variable dont on veut connaître l'intervalle de confiance pour la proportion.
- `success`: la modalité de la variable que nous considérons comme un succès.
- `alternative`:
  - `less`: pour un test unilatéral à gauche
  - `two-sided`: pour un test bilatéral
  - `greater`: pour un test unilatéral à droite

Par exemple, si on veut tester si la proportion de `female` de la variable `sex` est plus petite que 51%, à un niveau de confiance de 99%, nous utilisons:

```
prop_test(  x = gss,
            response = sex,
            success = "female",
            alternative = "less",
            p = 0.51)
#> # A tibble: 1 x 4
#>   statistic chisq_df p_value alternative
#>       <dbl>    <int>   <dbl> <chr>
#> 1      2.45        1  0.0587 less
```

La variable `p_value` nous permet de conserver ou de rejetter H_0. En effet, dans notre cas, la valeur est de 0.0587257 qui est plus grande que le risque d'erreur de 1% (associé au niveau de confiance de 99%). On conserve donc H_0, la proportion de femmes ne semble pas être plus petite que 51%.

9.3 Les tests d'hypothèses sur une différence de moyennes

```r
t_test(x = gss,
       response = age,
       explanatory = sex,
       alternative = "two-sided",
       mu = 1)
#> Warning: The statistic is based on a difference or ratio; by default, for
#> difference-based statistics, the explanatory variable is subtracted in the
#> order "male" - "female", or divided in the order "male" / "female" for
#> ratio-based statistics. To specify this order yourself, supply `order =
#> c("male", "female")`.
#> # A tibble: 1 x 7
#>   statistic  t_df p_value alternative estimate lower_ci upper_ci
#>       <dbl> <dbl>   <dbl> <chr>          <dbl>    <dbl>    <dbl>
#> 1    -0.213  498.   0.831 two.sided      0.746    -1.59     3.08
```

9.4 Les tests d'hypothèses sur une différence de proportions

```r
prop_test(gss,
          response = college,
          explanatory = sex,
          alternative = "two-sided")
#> Warning: The statistic is based on a difference or ratio; by default, for
#> difference-based statistics, the explanatory variable is subtracted in the
#> order "male" - "female", or divided in the order "male" / "female" for
#> ratio-based statistics. To specify this order yourself, supply `order =
#> c("male", "female")`.
#> # A tibble: 1 x 6
#>   statistic chisq_df p_value alternative lower_ci upper_ci
#>       <dbl>    <dbl>   <dbl> <chr>          <dbl>    <dbl>
#> 1 0.0000204        1   0.996 two.sided    -0.0834   0.0918
```