03 Manipuler les tableaux

Sommaire

Objectifs

Encore quelques outils supplémentaires, qui complètent la série des outils de base. Ce chapitre boucle donc la première partie de ces TPs ; dans les trois chapitres suivants, nous consacrerons l'essentiel du travail aux techniques d'analyse statistique inférentielle.

Donc, tout d'abord, une section pour compléter ce que nous avons vu de `dplyr` dans le chapitre précédent.

La deuxième section est consacrée aux instructions de contrôle de flux dans R. R permet de créer dans les scripts ce qu'on appelle des ***boucles*** et des ***opérations conditionnelles***. Ces possibilités sont relativement peu utilisées parce que la plupart du temps, on peut tirer parti des fonctions vectorielles. Mais il reste quelques situations où les boucles sont indispensables…

La troisième section porte sur les formats de tableaux. Dans le monde des logiciels de traitement de données, il existe deux façons principales d'organiser des données, le format large et le format long. Dans le format long, chaque ligne peut comporter autant de descripteurs que nécessaire, mais ***une seule mesure***. Dans le format large, chaque ligne correspond à une ***unité d'observation*** (par exemple, un participant) et comporte donc plusieurs mesures si plusieurs mesures ont été prélevées pour chaque unité d'observation. Les deux formats ont des avantages. Bref, il faut pouvoir passer de l'un à l'autre.

Et enfin, vous trouverez une section optionnelle consacrée à `ggplot2` un package très riche et puissant pour réaliser des graphiques clairs, informatifs, élegants, agréables à regarder et attractifs… Nous ne considérons pas que cela fait partie des techniques qu'il est nécessaire de maîtriser dans le cadre de l'enseignement OMN. Donc si cela vous amuse, si vous êtes passionné (ou chercheur en puissance), prenez quelques heures de plus pour approfondir cette section.

Au travail !

1. dplyr : Manipulations sur des tableaux de données

Le tableau de données qui nous servira pour les illustrations de cette section est le même que celui utilisé dans le chapitre 2. Vous trouverez ci-dessous les commandes nécessaires pour le créer. Il suffit de copier et de coller les commandes ci-dessous dans la Console. Cela vous permettra de réaliser par vous-même les fonctions et procédures décrites.

```
sexe <- c("M","F","M","M","F","F","F","M","M","M","M",
"F")
niv <- c(2, 3, 2, 4, 4, 5, 4, 3, 4, 3, 1, 5)
f.niv <- factor(niv, lab = c("Prim", "Secd", "Bac",
"Ma", "Doc"), ord = T)
d <- data.frame(Id=1:12, Sexe = sexe, Niveau = f.niv)
d$math <- c(10,8,12,9,10,15,8,12,11,15,11,14)
d$sci <-  c(10,9,14,11,13,18,9,14,11,18,13,14)
d$lang <- c(15,16,16,11,11,15,11,14,16,16,15,12)
d$gén <-  c(12,12,14,10,11,16,9,13,13,16,13,13)
d$total <- (d$math + d$sci + d$lang + d$gén)/4
```

Dans le chapitre précédent, nous avons fait connaissance de la librairie `dplyr`. Nous avons découvert ce que j'avais appelé les *fonctions de base*. En voici un bref rappel :

- `filter()` : filtrer des lignes du tableau sur base de conditions
- `slice()` : extraire des lignes sur base de leur position
- `sample_n()` et `sample_frac()` : échantillonner des lignes
- `arrange()` : trier les lignes du tableau
- `select()` : sélectionner certaines colonnes

- `rename()` : renommer des colonnes
- `mutate()` : créer ou modifier des colonnes
- `summarise()` : extraire des résumés
- `distinct()` : identifier les valeurs distinctes

Et rappelons une fois encore la **règle #1** :

> **Les fonctions de base s'appliquent à un tableau et elles produisent un tableau**

`dplyr` offre beaucoup d'autres possibilités qui en font un outil très puissant et efficace pour manipuler des tableaux. L'une des plus utiles est la fonction de *groupage*, `group_by()`, qui permet de définir des *sous-groupes* dans le tableau. On pourrait considérer que la fonction `group_by()` fait partie des fonctions de base, mais j'ai préféré la présenter séparément, parce qu'elle joue un rôle particulièrement important.

En plus de la fonction de groupage, que nous allons examiner en détail ci-après, `dplyr` offre quelques *fonctions d'agrégation* qui viennent compléter celles qui font partie des statistiques de base et que nous avons déjà rencontrées. Contrairement aux fonctions de base, les fonctions d'agrégation portent sur des *vecteurs* et non sur des tableaux.

1.1. Groupage

La fonction `group_by()` associe à un tableau une organisation en sous-groupes, définie par une ou plusieurs colonnes. Par exemple, pour distinguer les hommes des femmes dans une analyse, il suffira d'introduire la commande

```
group_by(d, Sexe)
```

Pour définir des groupes selon plusieurs critères, on énumèrera simplement les variables qui servent de critères de classification les uns à la suite des autres. Par exemple,

```
group_by(d, Sexe, Niveau)
```

La plupart du temps l'effet de la commande de groupage n'est pas directement visible :

```
d <- group_by(d, Sexe)
d
```

```
Source: local data frame [12 x 8]
Groups: Sexe [2]

      Id   Sexe Niveau  math   sci  lang   gén total
   <int> <fctr>  <ord> <dbl> <dbl> <dbl> <dbl> <dbl>
1      1      M   Secd    10    10    15    12 11.75
2      2      F    Bac     8     9    16    12 11.25
3      3      M   Secd    12    14    16    14 14.00
4      4      M     Ma     9    11    11    10 10.25
5      5      F     Ma    10    13    11    11 11.25
6      6      F    Doc    15    18    15    16 16.00
7      7      F     Ma     8     9    11     9  9.25
8      8      M    Bac    12    14    14    13 13.25
9      9      M     Ma    11    11    16    13 12.75
10    10      M    Bac    15    18    16    16 16.25
11    11      M   Prim    11    13    15    13 13.00
12    12      F    Doc    14    14    12    13 13.25
```

En particulier, comme on peut le voir ci-dessus, les lignes **ne sont pas** réarrangées selon le groupage qui a été défini. Ne confondez pas l'opération de *groupage*, qui est réalisée par la fonction `group_by()`, et le *tri*, pour lequel on utilise la fonction `arrange()` !

Vous noterez cependant quelques changements lorsque vous affichez le tableau dans la Console :

- l'entête comporte une indication sur le type de vecteur pour chaque colonne
- plus important, deux lignes supplémentaires au-dessus du tableau indiquent ses dimensions et son groupage.

Deux fonctions supplémentaires permettent à tout moment de retrouver l'information sur le groupage : `groups()`, qui donne la liste des variables de groupage, et `group_size()`, qui indique l'effectif des différents groupes.

```
groups(d)
[[1]]
Sexe
group_size(d)
[1] 5 7
```

Quel est l'effet d'un groupage ? Il modifie la manière dont les fonctions de base sont réalisées : au lieu de porter sur l'ensemble des lignes du tableau, les opérations porteront sur **chacun des groupes** de lignes définis par la commande de groupage[1].

En particulier, les opérations d'agrégation, avec `summarise()` donneront maintenant les résumés par groupe et non pour le tableau entier.

```
summarise(d, M=mean(math), SD= sd(math))
```

1 Un peu bizarrement, `arrange()` semble faire exception dans la version actuelle : on s'attendrait à ce que les tris se fassent en prenant en compte la structure de groupe, mais ce n'est pas le cas, le tri n'est pas affecté par le groupage qui a été défini.

```
# A tibble: 2 × 3
    Sexe      M     SD
  <fctr>  <dbl>  <dbl>
1      F 11.000 3.3166
2      M 11.429 1.9024
```

Selon le même principe, `slice()`, `sample_n()` et `sample_frac()` porteront sur **chaque groupe** et non plus sur le tableau entier.

```
slice(d, 1:2)
Source: local data frame [4 x 8]
Groups: Sexe [2]

     Id   Sexe Niveau  math   sci  lang   gén total
  <int> <fctr>  <ord> <dbl> <dbl> <dbl> <dbl> <dbl>
1     2      F    Bac     8     9    16    12 11.25
2     5      F     Ma    10    13    11    11 11.25
3     1      M   Secd    10    10    15    12 11.75
4     3      M   Secd    12    14    16    14 14.00
```

`slice()` produit donc ici les deux premières lignes de chaque groupe, et `sample_n()` échantillonne deux observations parmi les femmes et deux parmi les hommes

```
sample_n(d, 2)
Source: local data frame [4 x 8]
Groups: Sexe [2]

     Id   Sexe Niveau  math   sci  lang   gén total
  <int> <fctr>  <ord> <dbl> <dbl> <dbl> <dbl> <dbl>
1     5      F     Ma    10    13    11    11 11.25
2    12      F    Doc    14    14    12    13 13.25
3     9      M     Ma    11    11    16    13 12.75
4    10      M    Bac    15    18    16    16 16.25
```

Toujours selon la même logique, le résultat de `mutate()` pourra être affecté par le groupage ou pas, selon que le calcul fait appel ou non à l'ensemble des valeurs. Par exemple, la commande

```
mutate(d, note = (math + sci + gén + lang) / 4,
          note.pct = note * 100/20)
```

crée une nouvelle colonne `note` qui combine les quatre notes, et transforme ensuite cette note sur 20 en pourcentage. Les deux opérations sont des opérations vectorielles simples, qui s'appliquent à **chaque élément** des vecteurs indépendamment. Elles ne peuvent donc pas être influencées par le groupage.

Par contre, si le calcul fait appel à **l'ensemble des éléments** du vecteur, le groupage modifiera forcément les résultats, comme illustré dans l'exemple ci-dessous : on voit que pour chaque ligne la valeur de la nouvelle variable `Mlang` est la moyenne du **groupe** dont la ligne fait partie, 14.71 pour les hommes, et 13.00 pour les femmes, et non la moyenne générale.

```
mutate(d, Mlang = mean(lang))
Source: local data frame [12 x 9]
Groups: Sexe [2]

     Id   Sexe Niveau  math   sci  lang   gén   Mlang
  <int> <fctr>  <ord> <dbl> <dbl> <dbl> <dbl>   <dbl>
1     1      M   Secd    10    10    15    12 14.714
2     2      F    Bac     8     9    16    12 13.000
3     3      M   Secd    12    14    16    14 14.714
4     4      M     Ma     9    11    11    10 14.714
5     5      F     Ma    10    13    11    11 13.000
6     6      F    Doc    15    18    15    16 13.000
7     7      F     Ma     8     9    11     9 13.000
8     8      M    Bac    12    14    14    13 14.714
9     9      M     Ma    11    11    16    13 14.714
10   10      M    Bac    15    18    16    16 14.714
11   11      M   Prim    11    13    15    13 14.714
12   12      F    Doc    14    14    12    13 13.000
```

De la même manière une opération comme le calcul d'une note standardisée fait appel à la moyenne et à l'écart-type de la variable, ce qui suppose de prendre en compte **l'ensemble des éléments** du vecteur, et dans ce cas, *l'ensemble considéré* sera différent selon les groupages qui ont été définis.

```
mutate(d, Total.z = (Total - mean(Total))/sd(Total) )
```

Par exemple, si les notes standardisées sont calculées sur un tableau avec groupage selon le sexe, les notes standardisées seront calculées par rapport à la moyenne et à l'écart-type de chaque groupe et non par rapport à la moyenne et à l'écart-type de l'ensemble des observations.

Il en sera de même pour les filtrages conditionnels : si la condition fait appel à une valeur qui dépend de l'ensemble des éléments, le filtrage sera influencé par la structure de groupe. Par exemple, on a sélectionné ci-dessous les observations pour lesquelles la note `lang` est inférieure à la moyenne *du groupe correspondant.*

```
filter(d, lang < mean(lang))
Source: local data frame [5 x 8]
Groups: Sexe [2]

     Id   Sexe Niveau  math   sci  lang   gén total
  <int> <fctr>  <ord> <dbl> <dbl> <dbl> <dbl> <dbl>
1     4      M     Ma     9    11    11    10 10.25
2     5      F     Ma    10    13    11    11 11.25
3     7      F     Ma     8     9    11     9  9.25
4     8      M    Bac    12    14    14    13 13.25
5    12      F    Doc    14    14    12    13 13.25
```

En résumé,

Lorsqu'un groupage est défini, les opérations de sélection, de manipulation et d'agrégation s'appliquent indépendamment à chacun des groupes de lignes définis par le groupage et non plus à l'entièreté du tableau

Trois remarques supplémentaires :

1. Les exemples qui précèdent groupent plusieurs sujets selon une caractéristique (par exemple, les hommes et les femmes). Le groupage pourrait également porter sur les sujets, si le tableau comporte une ligne par essai, comme c'est typiquement le cas avec les résultats d'expériences informatisées. L'utilisation de `group_by()` et `summarise()` permettra dans ce cas d'agréger les données pour obtenir les temps ou taux de réponse moyens par sujet et par condition.

2. `select()` n'est pas affecté par le groupage, si ce n'est que les variables utilisées pour définir les groupes seront d'office conservées.

3. Chaque commande `group_by()` efface et remplace un éventuel groupage antérieur. Ainsi, la manière la plus simple de supprimer un groupage consiste à introduire un groupage vide : `group_by(d)`.

1.2. Enchaîner des opérations

Il est évidemment fréquent qu'on veuille réaliser plusieurs opérations successives sur un tableau. Prenons un exemple : on souhaite calculer la note totale (moyenne de math, sci, lang et gén), et obtenir les statistiques descriptives en fonction du sexe. Il faudra donc d'abord calculer la note totale, définir ensuite le groupage selon le sexe, et finalement utiliser `summarise()` pour calculer les statistiques désirées.

Jusqu'ici, la seule possibilité était de réaliser plusieurs opérations successives, en modifiant chaque fois le tableau ou en créant un nouveau tableau :

```
d1 <-  mutate(d, total = (math+sci+lang+gén)/4)
d1 <- group_by(d1, Sexe)
summarise(d1, M=mean(total), SD= sd(total))
# A tibble: 2 × 3
    Sexe      M     SD
  <fctr>  <dbl>  <dbl>
1      F 12.200 2.5520
2      M 13.036 1.8621
```

`dplyr` offre une possibilité de simplification, qui consiste à enchaîner des opérations, à l'aide d'un opérateur spécial : `%>%`

L'opérateur `%>%` permet de passer directement le résultat d'une opération à la commande suivante.

Voici ce que cela donne, une commande complexe qui enchaîne en une seule fois les trois opérations :

```
d %>%
   mutate(total = (math+sci+lang+gén)/4) %>%
   group_by(Sexe) %>%
   summarise(M= mean (total), SD= sd(total))
# A tibble: 2 × 3
    Sexe      M     SD
  <fctr>  <dbl>  <dbl>
1      F 12.200 2.5520
2      M 13.036 1.8621
```

La première ligne passe le tableau `d` à `mutate()`, qui calcule la nouvelle colonne `total`. Le résultat (à savoir, le contenu du tableau `d` avec une colonne supplémentaire, `total`) est passé à `group_by()`. Le tableau structuré en deux groupes selon le sexe produit par `group_by(Sexe)` est ensuite passé à `summarise()`, qui calcule les moyennes et les écarts-types de `total` en fonction du groupage défini.

Le tableau `d` n'a **jamais été modifié**. De plus il n'est plus nécessaire de répéter à chaque étape sur quel tableau porte l'opération : elle porte d'office sur le tableau qui résulte de l'étape précédente. Ainsi dans la commande `group_by(Sexe)` on n'a pas mentionné `d` comme dans les exemples précédents.

Cette technique qui existe dans divers langages de programmation s'appelle "faire un tuyau", et le symbole d'enchaînement utilisé est souvent appelé *pipe* (*païp*, tuyau en anglais).

Figure 1.– Ceci n'est pas une pipe

Il y a plusieurs avantages à l'utilisation du pipe :

- On évite la création d'une multitude de tableaux intermédiaires, ou l'ajout de variables temporaires au tableau de données.
- On regroupe en petits blocs les opérations complexes. Cela facilite la relecture et la compréhension du code.
- On simplifie l'écriture, puisqu'il ne faut indiquer qu'une seule fois le tableau qui est à l'origine de la chaîne d'opérations.

`commande1 %>% commmande2` signifie que le tableau résultat de la commande1 est passé directement comme input à la commande2.
Il n'est donc plus nécessaire de le spécifier dans la commande2.

1.3. Les fonctions d'agrégation

On appelle *fonctions d'agrégation* les fonctions qui s'appliquent à un vecteur pour en extraire un résultat synthétique : la somme, la moyenne, le minimum, la première valeur,un certain quantile,… sont des exemples de fonctions d'agrégation.

Nous en avons déjà rencontré plusieurs, sans doute les plus importantes, qui font partie des fonctions standard de R : `min()`, `max()`, `sum()`, `prod()`, `mean()`, `median()`, `sd()`, `var()`, `quantile`.

`dplyr` en ajoute quelques-unes qui peuvent être utiles :

- `n()` indique le nombre d'observations dans les groupes. C'est une fonction un peu particulière puisqu'elle n'accepte

aucun paramètre. Elle se notera donc toujours `n()`, avec les parenthèses vides.

```
d %>%
   summarise(N=n(),M.math=mean(math),M.sci=mean(sci),
     M.lang=mean(lang),M.gén=mean(gén))
# A tibble: 2 × 6
    Sexe     N M.math M.sci M.lang M.gén
  <fctr> <int>  <dbl> <dbl>  <dbl> <dbl>
1      F     5 11.000  12.6 13.000  12.2
2      M     7 11.429  13.0 14.714  13.0
```

La colonne `N` indique donc ici qu'il y a 5 observations (cinq lignes) dans la catégorie `F` et 7 dans la catégorie `M`.

- `n_distinct()` indique le nombre de valeurs différentes d'une variable ou d'une combinaison de variables

```
d %>%
   summarise(V= n_distinct(Niveau))
# A tibble: 2 × 2
    Sexe     V
  <fctr> <int>
1      F     3
2      M     4
```

La variable `V` indique qu'il y a trois valeurs de Niveau dans le groupe `F` ("Bac", "Ma", "Doc") et quatre dans le groupe des hommes ("Prim", "Secd", "Bac", "Ma").

- `first()`, `last()`, `nth()` fournissent respectivement la première, dernière ou n^{ème} valeur d'une variable. Sans autre spécification, l'ordre pris en considération est l'ordre d'apparition dans le tableau (dans les groupes du tableau).

```
d %>%
   summarise(prem = first(math), der = last(math),
             num3 = nth(math, 3, default = NA))
# A tibble: 2 × 4
    Sexe  prem   der  num3
  <fctr> <dbl> <dbl> <dbl>
1      F     8    14    15
2      M    10    11     9
```

1.4. Les fonctions de rang et de cumul

En plus des deux types de fonctions les plus courantes, à savoir, les fonctions simples qui traitent chaque ligne/unité d'observation séparément (ex. `round()`, indépendamment des autres lignes du tableau), et les fonctions d'agrégation qui produisent une valeur unique à partir de l'ensemble des lignes du vecteur (ex. `mean()`), on peut définir une troisième catégorie qui est composée de fonctions qui s'appliquent à des vecteurs et renvoient pour chaque élément un résultat qui dépend de l'ensemble (ou d'une fenêtre) d'éléments du vecteur. Ces fonctions ne jouent pas un rôle aussi important que les précédentes, et vous ne les utiliserez probablement pas ou pas souvent, mais il est bon de savoir qu'elles existent. Vous pouvez donc considérer cette section comme optionnelle.

Un exemple est le **rang** : À chaque élément *i* du vecteur on fait correspondre son rang dans l'échelle des valeurs rencontrées dans l'ensemble du vecteur.

Ainsi, dans la commande ci-dessous, on a calculé le rang des douze observations pour la note de math de deux manières différentes selon la manière de traiter les doublons `min_rank()` attribue le rang minimum aux doublons et laisse un trou dans la séquence ensuite (1, 1, 3, 4,…), tandis que `dense_rank()` poursuit simplement la séquence.

La notation scientifique des nombres?

R affiche parfois les très grands et très petits nombres en utilisant ce qu'on appelle la *notation scientifique*. Elle permet d'éviter des écritures lourdes (et pas très lisibles) comme .000000007132 ou 3100000000000.

Le principe est de transformer le nombre en deux parties, la partie significative (la *mantisse*), qui est un nombre compris entre 0 et 10 (non inclus), et l'exposant, une puissance de 10.

Par exemple $.000000007132 = 7.132 \times 10^{-9}$ et $3100000000000 = 3.1 \times 10^{12}$.

Dans le domaine informatique une convention courante est de séparer les deux parties par la lettre E et d'omettre le 10 : vous pourriez donc lire `7.132E-9` ou `3.1E12`

Notez que le tableau ne doit pas être trié selon la variable dont on veut obtenir le rang. Le tri (cf. dernière ligne) a été effectué après le calcul, uniquement pour faciliter la lecture.

```
d %>%
   group_by() %>%
   mutate(rang1 = min_rank(math),
          rang2 = dense_rank(math)) %>%
   select (Id, math, rang1, rang2) %>%
   arrange(math)
# A tibble: 12 × 4
      Id  math rang1 rang2
   <int> <dbl> <int> <int>
1      2     8     1     1
2      7     8     1     1
3      4     9     3     2
4      1    10     4     3
5      5    10     4     3
6      9    11     6     4
7     11    11     6     4
8      3    12     8     5
9      8    12     8     5
10    12    14    10     6
11     6    15    11     7
12    10    15    11     7
```

Un autre type de fonction liée au rang permet d'associer à chaque ligne la valeur précédente (`lag()`, retard) ou suivante (`lead()`, avance). Les deux fonctions demandent de spécifier le décalage (i.e., 1 pour la valeur immédiatement précédente ou suivante). Elles sont basées sur l'ordre du tableau si aucun ordre n'est spécifié explicitement, ou sur l'ordre de la variable spécifiée par le paramètre `order_by`. Il est donc prudent de spécifier.

```
d %>%
   group_by() %>%
   mutate(prec = lag(math, 1, order_by= math),
          suiv = lead(math, 1, order_by= math)) %>%
   arrange(math) %>%
   select(math, prec, suiv)
# A tibble: 12 × 3
    math  prec  suiv
   <dbl> <dbl> <dbl>
1      8    NA     8
2      8     8     9
3      9     8    10
4     10     9    10
5     10    10    11
6     11    10    11
7     11    11    12
8     12    11    12
9     12    12    14
10    14    12    15
11    15    14    15
12    15    15    NA
```

```
d %>%
   group_by() %>%
   mutate(somme = cumsum(math),
          moy = cummean(math)) %>%
   select(math, somme:moy)
# A tibble: 12 × 3
    math somme    moy
   <dbl> <dbl>  <dbl>
1     10    10 10.000
2      8    18  9.000
3     12    30 10.000
4      9    39  9.750
5     10    49  9.800
6     15    64 10.667
7      8    72 10.286
8     12    84 10.500
9     11    95 10.556
10    15   110 11.000
11    11   121 11.000
12    14   135 11.250
```

Enfin, on peut adjoindre à cette série les fonctions cumulatives : un exemple est la **somme cumulée** : À chaque élément *i* du vecteur correspond la somme de tous les éléments jusqu'à l'élément *i*.

Testez vos connaissances !

2. Les instructions de contrôle de flux dans R

Contrôle de flux ? Keksekça ?

Deux mots d'explication : à partir du moment où on compose une série de commandes, dans un éditeur, en vue de les faire exécuter en bloc, on définit un **flux**, une séquence. Et à partir du moment où on peut créer des séquences d'opérations, il arrive assez souvent qu'on doive envisager soit de moduler une certaine séquence selon telle ou telle condition –c'est presque la même mais pas tout-à-fait pour la situation 1 et pour la situation 2– ou de répéter une séquence ou une partie de séquence plusieurs fois. Les **instructions de contrôle de flux** sont des commandes qui rendent possible de moduler et répéter des suites d'opérations.

Pour rendre les choses un peu plus concrètes, prenons un exemple. Voici une situation que vous rencontrerez certainement, et où il est à peu près impossible de s'en tirer autrement qu'en utilisant le contrôle de flux. Vous avez conçu et réalisé une étude, et les données que vous avez récoltées ont la forme de 42 fichiers, un pour chaque sujet, et chacun avec 360 lignes qui correspondent aux 360 essais de votre expérience.

Et pour chacun de ces fichiers, vous voulez effectuer les opérations suivantes :

- importer le fichier dans R
- ajouter une colonne qui permettra d'identifier le participant
- visualiser la distribution des temps de réponse
- le cas échéant, éliminer les temps de réponse que vous avez décidé de considérer comme anormaux : ceux qui sont in-

férieurs à 100 ms ou supérieurs à 5000 ms.

- vérifier s'il reste au moins 80% des observations dans chacune des conditions
- si c'est le cas, joindre les données avec celles des autres sujets retenus dans un tableau global pour les analyses suivantes

L'exemple est assez typique : il y a une séquence d'opérations à répéter (*pour chaque fichier,…*), et certaines parties de cette séquence sont conditionnelles (*si il reste au moins 80%…*, alors, joindre…)

Nous allons découvrir les différents éléments nécessaires pour réaliser ce genre de projet.

Vous retrouverez les mêmes principes et fonctions logiques dans le Coder avec PsychoPy (et ce sont d'ailleurs les mêmes que dans beaucoup d'autres langages de programmation), mais les détails d'écriture sont un peu différents, et les mots-clés sont similaires mais pas exactement les mêmes : dans R, les mots-clés importants sont `if`, `else`, `for`, `while`, `break` et aussi `repeat` et `next`.

2.1. Créer une boucle simple

Considérez la situation suivante : Vous avez planifié une étude dans laquelle les participants voient des exemplaires d'objets de six catégories différentes, A, B, C, D, E et F. Ils auront donc devant eux un boîtier avec six boutons, et ils utiliseront trois doigts (index, majeur, annulaire) de chaque main pour répondre. Vous vous intéressez au temps de décision selon les objets et les catégories.

Or, les temps de réponse peuvent être affectés par les particularités de la modalité de réponse : on est plus rapide de la main dominante, et plus rapide avec l'index qu'avec le majeur ou l'annulaire. Comme ces variations ne sont pas l'objet de votre recherche, vous cherchez une manière de les éviter, et on vous conseille de varier la relation entre les six réponses et les six positions. Problème : il y a 720 manières différentes d'associer les six réponses aux six boutons ! On voudrait donc créer une association différente, tirée au hasard parmi les 720 possibilités, pour chaque participant. Nous allons utiliser les abréviations "AG", "MG" et "IG", respectivement pour l'Annulaire, le Majeur, et l'Index Gauche, et "AD", "MD" et "ID" pour la main droite.

Nous pouvons donc créer un tableau avec une première colonne qui reprend les six boutons :

```
modrép <- data.frame(Bouton = c("AG", "MG", "IG", "ID",
"MD", "AD"))
```

Il nous reste à générer une répartition aléatoire des six catégories A à F pour chaque sujet qui passera l'expérience; Vous possédez déjà un élément de la solution, trouver une répartition aléatoire des six réponses A:F est simple, c'est le vecteur qui résulte de la commande `sample(LETTERS[1:6])`.

Il suffirait donc de répéter la commande `sample(…)` autant de fois qu'il y a de participants prévus. Mais plutôt que de reproduire la commande *x* fois, nous allons utiliser le principe de la **boucle `for`**, qui permet de faire en sorte qu'un bloc de commande soit automatiquement répété un certain nombre de fois. Commençons

par définir le nombre de sujets :

```
nb.sujets <- 5
```

Pour créer la boucle, nous avons besoin d'une variable qui fera office de *compteur* et servira à vérifier le nombre de répétitions, et il faudra spécifier les valeurs que cette variable doit prendre.

```
for (i in 1:nb.sujets){
   modrép <- cbind(modrép, X = sample(LETTERS[1:6]))
}
```

Dans le code ci-dessus, on a créé une boucle avec `for`, en spécifiant que la variable compteur `i` doit prendre les valeurs `1:nb.sujets`, soit 1, 2, 3, 4 et puis 5.

Pour chaque valeur du compteur `i`, le bloc de commande entre accolades est répété. Ici, le bloc de commande ne comporte qu'une seule commande : attribuer au tableau `d` le résultat de l'ajout d'une nouvelle colonne constituée d'un nouvel arrangement aléatoire des 6 lettres A à F.

Le résultat est donc un tableau qui comporte 6 colonnes, la colonne `Bouton` qui définit les six boutons, et cinq variables avec des arrangements différents des six catégories.

Utiliser une boucle `for` nous a donc permis de générer rapidement pour chaque sujet une association aléatoire entre les catégories de stimuli et les modalités de réponse. Ainsi, quand un objet de la catégorie A sera présenté, le sujet 1 devra appuyer avec l'index droit, le sujet 2 avec le médium gauche, etc… Pour un stimulus de la catégorie F, le premier sujet utilisera l'annulaire droit, tandis que le deuxième utilisera l'annulaire droit…

```
modrép
  Bouton X X X X X
1     AG B B E D F
2     MG E A F A B
3     IG D C B B E
4     ID F F C F D
5     MD C D D E A
6     AD A E A C C
```

Une boucle *for* comporte donc toujours au moins les trois éléments suivants :

1. le mot-clé : `for`
2. la définition du compteur et de ses valeurs : une expression de la forme (*variable* `in` *vecteur*). La variable définie sert de compteur et le vecteur indique les valeurs successives que prendra le compteur. Pour chaque valeur, le bloc de commandes qui suit est répété. Lorsque le compteur a épuisé toutes les valeurs du vecteur, on "sort de la boucle", et ce sont les commandes qui suivent l'accolade fermante qui sont ensuite exécutées.
3. le bloc de commandes : la ou les commandes à exécuter à chaque répétition. Elles doivent être entre accolades.

Bon à savoir

Il est habituel d'ouvrir le bloc de commande à la fin de la ligne `for` et d'indenter les commandes qui font partie du bloc de commandes. Ce n'est pas une obligation, mais c'est une bonne habitude qui améliore la lisibilité du code.

Il est également habituel d'utiliser les noms "i", "j", ou "k" pour les variables qui servent de compteurs.

Dans `R`, c'est donc l'accolade droite qui indique la fin du bloc de commandes.

2.2. Créer une boucle avec un index

Dans la situation précédente, le contrôle du nombre de répétitions était basé sur la variable `i` qui servait uniquement de compteur et n'avait aucun autre rôle que de compter les répétitions.

Mais rien n'interdit d'utiliser le compteur dans la boucle elle-même. En voici un exemple : dans notre tableau `d`, on voudrait transformer les différentes notes (`math`, `sci`, `lang`, `gen`) en notes standardisées [2] avant de les moyenner entre elles.

On pourrait écrire la séquence de commandes suivantes

```
d$math <- (d$math- mean(d$math))/sd(d$math)
d$sci <- (d$sci- mean(d$sci))/sd(d$sci)
d$gén <- (d$gén- mean(d$gén))/sd(d$gén)
d$lang <- (d$lang- mean(d$lang))/sd(d$lang)
```

ou encore

```
d[, 4] <- (d[, 4]- mean(d[, 4]))/sd(d[, 4])
d[, 5] <- (d[, 5]- mean(d[, 5]))/sd(d[, 5])
d[, 6] <- (d[, 6]- mean(d[, 6]))/sd(d[, 6])
d[, 7] <- (d[, 7]- mean(d[, 7]))/sd(d[, 7])
```

Mais on voit que les quatre lignes sont exactement identiques, mis à part l'index de la colonne qui varie de 4 à 7.

On pourrait donc éviter de réécrire manuellement plusieurs fois la même commande (ce qui augmente le risque d'introduire des erreurs), et construire une boucle avec quatre répétitions, et un compteur qui varie de 4 à 7 :

```
d1 <- data.frame(d)
for (i in 4:7){
   d1[, i] <- (d1[, i]- mean(d1[, i]))/sd(d1[, i])
}
```

La différence par rapport à la situation précédente est que la variable `i` est cette fois utilisée non seulement comme compteur des exécutions de la boucle, mais également comme indice de colonne dans le bloc de commande, de sorte que celui-ci s'appliquera à des colonnes différentes au fur et à mesure des répétitions.

2 Rappelez-vous… Une note standardisée est une note dont la moyenne = 0 et l'écart-type = 1. Donc pour transformer une note brute en note standardisée, il suffit de soustraire à chaque note brute la moyenne de celles-ci et de diviser le résultat par l'écart-type.

2.3. Réaliser une opération conditionnelle

A l'examen, on a constaté que trois questions de l'épreuve de maths présentaient une erreur et on veut donc attribuer trois points supplémentaires pour cette épreuve. On pourrait intégrer ce changement dans la boucle, à l'aide d'une commande conditionnelle, avant le calcul de standardisation[3].

```
d1 <- data.frame(d)
for (i in 4:7){
   if (i == 4){
       d1[, i] <- d1[, i] + 3
   }
   d1[, i] <- (d1[, i]- mean(d1[, i]))/sd(d1[, i])
}
```

= et ==

Ne confondez pas le signe == et le signe =. Le signe == est un opérateur logique de comparaison entre deux vecteurs. L'opérateur == calcule si la valeur de l'expression à gauche (pour chaque élément du vecteur) est identique à la valeur de l'expression à droite, et le résultat est TRUE si oui, et FALSE si non.

Le signe = est équivalent à <- et permet d'assigner le résultat d'une commande à un objet. A l'intérieur des parenthèses qui délimitent les arguments d'une fonction, il sert à identifier les arguments, ou à définir le nom d'un élément (par exemple, les colonnes d'un tableau)

Exemple : `echantillon <- rnorm(n = 1000, mean = 100, sd = 15)`

3 L'exemple simple utilisé ici a été construit pour la démonstration, mais notez qu'il n'a pas beaucoup d'intérêt en pratique. Il serait en effet plus simple de faire la modification de `d$math` en une fois, avant la boucle !

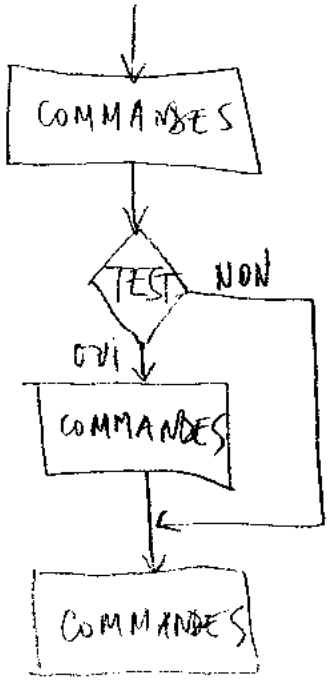

Figure 2.– La logique de commandes conditionnelles

Comme illustré ci-dessus, le principe est le suivant : **Si** une certaine **condition** est vérifiée, on exécute le **bloc de commandes conditionnelles**. Il y a trois éléments à respecter pour mettre en place un bloc de commandes conditionnelles :

1. le mot-clé : `if`
2. la condition : une expression logique qui est évaluée comme `TRUE` ou `FALSE`. Elle doit être écrite entre parenthèses.
3. le bloc de commandes : la commande, ou les commandes qui ne seront exécutées que si la condition est évaluée comme `TRUE`. Elles doivent être entourées d'accolades.

Pour réaliser une opération ou un ensemble d'opérations conditionnelles, on utilise le schéma
`if` (*condition*) {*commandes conditionnelles*}

2.4. Réaliser des opérations différentes selon qu'une condition est vérifiée ou non

Une situation un peu plus complexe qu'on rencontre parfois demande des **actions différentes** selon l'alternative qui est vérifiée. Dans ce cas on utilisera une construction de contrôle un peu différente : `if` (*condition*) {*commandes conditionnelles*} `else` {*autres commandes*}, ce qui correspond à *Si la condition est vérifiée, faire ceci,* **autrement**, *faire cela.*

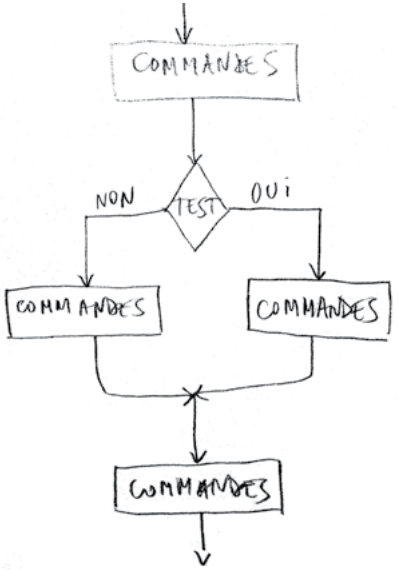

Figure 3.– commandes conditionnelles

Par exemple on veut transformer les scores des 4 épreuves en pourcentages, en tenant compte de la même correction que précédemment pour la note de maths. On considèrera donc que le maximum possible est 17 pour les maths, et 20 pour les autres notes :

```
001    d1 <- data.frame(d)
002    for (i in 4:7){
003       if (i == 4){
004           d1[, i] <- d1[, i] * 100 / 17
005       } else {
006           d1[, i] <- d1[, i] * 100 / 20
007       }
008    }
```

La condition (ligne 03) indique que si c'est la colonne pour les maths qui est considérée (`if (i==4)`), alors le pourcentage est calculé sur 17 ; **sinon** (`else`), il est calculé sur 20.

Un détail important : il est toujours préférable de placer le mot clé `else` sur la même ligne que l'accolade droite qui clôture le premier bloc.

2.5. Un exemple réaliste...

Vous avez conduit une expérience. Vous avez testé 100 sujets, et vous disposez d'un dossier de données qui contient 100 fichiers, un pour chaque participant de l'expérience, et vous voulez importer les données dans R.

Cet exemple n'est pas réalisable sans disposer des fichiers de données. Une variante sera reprise comme objet d'un exercice en séance.

Erreur fréquente

Faites attention lorsque vous importez un fichier .txt ou .csv :

Le caractère qui séparé les données n'est pas toujours le même : cela peut être une virgule, un point-virgule, le caractère de tabulation →| , etc. Il faut le spécifier avec le paramètre `sep` de `read.table()`: `sep= ","`, `sep= ";"`, `sep = "\t"`

R utilise toujours le point comme séparateur décimal. Si vos fichiers de données sont dans un format différent (par exemple avec un virgule), n'oubliez pas de le spécifier lors des opérations de lecture et écriture de fichiers avec le paramètre `dec = ","`

Si la première ligne contient le nom des variables, il faut le spécifier : `header = TRUE`.

Vous savez comment charger **un** fichier dans R, avec la fonction `read.table()`. Si le fichier correspondant au premier sujet s'appelle

"Sujet_1_script_expeMEM_2015_nov_4_0822.csv",

vous pourriez écrire dans votre script, par exemple :

```
s1 <- read.table(file =
     "Sujet_1_script_expeMEM_2015_nov_4_0822.csv",
     header = TRUE)
```

Et vous pourriez ensuite recopier 100 fois la ligne, et puis modifier les noms des tableaux de `s1` à `s100` et les noms de fichiers en conséquence. Mais cela ressemble un peu trop à une punition d'école primaire au siècle passé.

Comment faire ? Une simple boucle !

Supposons que la variable `listeFichiers` contient la liste des 100 noms de fichiers (nous verrons plus loin comment faire cela), voici un premier essai

```
for (f in listeFichiers){
   ds <- read.table(file = f, header = TRUE)
}
```

C'est un début, mais cela ne fait pas l'affaire parce que comme on assigne le résultat du chargement chaque fois au même objet `ds`, on écrase le tableau résultant de l'importation de chaque fichier par le résultat de l'importation suivante…

Comment faire pour concaténer les 100 fichiers ? On peut utiliser la fonction `rbind()`[4] pour assembler les lignes de deux tableaux :

```
001    d <- data.frame()
002    for (f in listeFichiers){
003       ds <- read.table(file = f, header = TRUE)
004       d <-  rbind(d, ds)
005    }
```

On créé une variable `d` qui est un tableau de données vide (ligne 1). Puis on assemblera successivement les données de chaque sujet avec `d`.

Au 1er passage dans la boucle, `f` est le nom du premier fichier, et (ligne 3) `ds` est le tableau des données importé à partir de ce fichier, qui est ensuite assemblé (ligne 4) avec `d` vide. A ce moment là, `d` contient donc juste les données du sujet 1. Au deuxième passage dans la boucle, les données du sujet suivant (stockées à nouveau dans `ds`) sont assemblées avec `d` qui contiendra donc maintenant les données du sujet 1 et celles du sujet 2. Et ainsi de suite. À chaque passage dans la boucle, on ajoute au tableau général `d` les données importées d'un sujet.

Dans ce troisième genre de boucle, la logique est presque la même que précédemment : on crée une variable qui va servir à contrôler les répétitions du bloc de commande, et qui prendra successivement toutes les valeurs spécifiées par un vecteur. Ici la variable est `f`, et elle prendra successivement toutes les valeurs spécifiées par le vecteur `listeFichiers`, qui contient les noms des 100 fichiers de données). Le bloc de commandes de la boucle va donc être exécuté une fois pour chaque valeur de `f`, soit, avec un fichier différent, "Sujet_1…", "Sujet_2…", etc.

Et voilà, le tour est joué ! Nous avons maintenant un tableau

4 Voir Chapitre 2, Section 3.4)

contenant les données des 100 sujets à la suite les uns des autres.

L'astuce consiste à utiliser la liste des noms de fichiers pour contrôler les répétitions de la boucle (plutôt que les nombres de 1 à 100), ce qui permet d'utiliser directement la variable compteur `f` dans la commande `read.table()`. Remarquez qu'on aurait aussi bien pu écrire

```
001    d <- data.frame()
002    for (i in 1:100){
003       f <- listeFichiers[i]
004       s <- read.table(file = f, header = TRUE)
005       d <-  rbind(d, s)
006    }
```

ce qui aurait simplement demandé une opération de plus à chaque itération.

On pourrait ajouter une variable qui permette d'identifier les participants : comme les noms des fichiers contiennent un numéro d'identification, nous allons ajouter une colonne reprenant le nom du fichier, qui servira d'identificateur. Nous verrons plus loin (*Section 3.3 page 32*) comment extraire le numéro identificateur du sujet à partir du nom de fichier complet.

```
001    d <- data.frame()
002    for (f in listeFichiers){
003       s <- read.table(file = f, header = TRUE)
004       s$Id <- f
005       d <-  rbind(d, s)
006    }
```

De plus rien n'empêcherait d'insérer dans la boucle une commande (ligne 04) produisant l'histogramme des temps. En supposant que la colonne qui contient les temps en millisecondes s'appelle *TR* :

```
001    d <- data.frame()
002    for (f in listeFichiers){
003       s <- read.table(file = f, header = TRUE)
004       hist(s$TR)
005       s$Id <- f
006       d <-  rbind(d, s)
007    }
```

On pourrait également facilement supprimer les temps inférieurs à 200 msec et supérieurs à 5 sec (lignes 06-09)

```
001    d <- data.frame()
002    for (f in listeFichiers){
003       s <- read.table(file = f, header = TRUE)
004       hist(s$TR)
005       s$Id <- f
006       # on ne retient que les lignes avec RT > 200
007       s <- s[s$TR>200, ]
008       # on ne retient que les lignes avec RT < 5000
009       s <- s[s$TR<5000, ]
010       d <-  rbind(d, s)
011    }
```

Et finalement on pourrait insérer un test, pour s'assurer qu'il reste au moins 80 % des données :

```
001    d <- data.frame()
002    for (f in listeFichiers){
003       s <- read.table(file = f, header = TRUE)
004       hist(s$TR)
005       s$Id <- f
006       nbEssais <- length(s$TR)
007       # on ne retient que les lignes avec RT > 200
008       s <- s[s$TR>200, ]
009       # on ne retient que les lignes avec RT < 5000
010       s <- s[s$TR<5000, ]
011       # on teste s'il reste 80% des essais
012       if (length(s$TR)/nbEssais >= 0.80){
013           d <-  rbind(d, s)
014       }
015    }
```

A la ligne 06, on encode le nombre total d'essais (avant élimination). Après élimination des lignes avec des temps trop courts ou trop longs, on teste (lignes 11-14) si il reste au moins 80% des essais, et on ne retient les données du sujet que si c'est le cas.

2.6. Récupérer la liste des fichiers

Pour récupérer la liste des fichiers, on utilisera la fonction `dir()`, qui renvoie la liste des noms de fichiers contenus dans le répertoire actif.

De plus, pour le cas où le répertoire actif comporte d'autres documents que les fichiers à importer (par exemple, des scripts), `dir()` peut filtrer en fonction de certaines restrictions, avec le paramètre `pattern = ….`

Il est possible de définir des restrictions très subtiles, mais nous n'entrerons pas dans les détails. Le principe de base est que la liste des fichiers retenus est limitée à ceux qui contiennent la suite de caractères spécifiées dans le paramètre `pattern =`.

Ainsi `dir(pattern = "csv")` listera tous les noms de fichiers qui contiennent la suite de lettres "csv", `dir(pattern = "Feb_12")` liste tous les noms de fichiers qui contiennent la suite "Feb_12", etdir(pattern = "script_expeMEM")` liste tous les fichiers dont le nom contient "script_expeMEM".

Ceci dit, il est prudent de toujours vérifier que la longueur de la liste correspond bien au nombre de sujets qui ont passé l'expérience ! Voici une façon de le faire :

```
listeFichiers <- dir()
length(listeFichiers)
```

Si le résultat affiché est 102 et que vous avez testé 100 participants… c'est que deux fichiers non appropriés ont été repris dans la liste ! À vous de les identifier !

Testez vos connaissances !

3. Les deux formats de tableaux

Comme on l'a vu au début du cours, d'une manière très générale, les données rassemblent différentes **informations** pour une série d'**unités d'observation**.

Il existe deux façons d'organiser des données en tableau, qu'on appelle parfois *format large* et *format long*. Les deux formats ont des avantages et des inconvénients, de sorte qu'il est nécessaire de pouvoir passer de l'un à l'autre.

Dans le format large, toutes les données concernant une unité d'observation sont sur la même ligne. Si plusieurs mesures ont été prélevées pour une même unité d'observation, elles seront définies par des variables (colonnes) différentes. Souvent chaque ligne correspondra à un sujet, avec les données pour les différentes conditions (voir *Figure 4*).

sujet	cond_A_rt	cond_B_rt	cond_C_rt
1	930	710	639
2	885	510	970
3	875	836	535
4	615	813	928

Figure 4.– Exemple de tableau large

Dans l'exemple que nous avons utilisé tout au long de ce chapitre, 12 personnes ont passé quatre épreuves (maths, sciences, langue, culture). Les quatre épreuves sont identifiées par le nom de la colonne, et les lignes correspondent aux 12 personnes. Le tableau que nous avons utilisé jusqu'ici était au format large.

Dans le format long, chaque ligne ne comporte qu'**une seule**

mesure. Si on a prélevé plusieurs mesures pour chaque unité d'observation, elles devront donc nécessairement figurer sur des lignes différentes. La *Figure 5* donne deux exemples du format long : dans les deux, les données sont les mêmes qu'à la *Figure 4*, elles sont arrangées par sujet (à gauche) ou par condition (à droite).

sujet	condition	RT
1	A	930
1	B	710
1	C	639
2	A	885
2	B	510
2	C	970
3	A	875
3	B	836
3	C	535
4	A	615
4	B	813
4	C	928

sujet	condition	RT
1	A	930
2	A	885
3	A	875
4	A	615
1	B	710
2	B	510
3	B	836
4	B	813
1	C	639
2	C	970
3	C	535
4	C	928

Figure 5.– Exemples de tableaux longs

Le format long est très pratique pour beaucoup de manipulations : notamment pour filtrer, grouper, et agréger des données, comme nous l'avons fait précédemment avec `dplyr`. De plus, la plupart du temps, c'est sous cette forme que vous recevrez et importerez les données brutes. Typiquement, les logiciels de gestion d'expérience produisent un fichier par participant dans lequel chaque essai produit une ligne. Par contre, le format long est moins pratique voire inapproprié pour d'autres opérations, par exemple, lorsque l'on veut mettre en relation plusieurs mesures correspondant aux mêmes unités d'observation. Par exemple, si on veut examiner la relation entre la note de maths et la note de science, il faudra disposer du tableau large.

Pour passer du format long au format large et inversement, nous allons utiliser une autre librairie, qui apporte un complément à `dplyr`. Elle s'appelle `tidyr`(*tidy* veut dire ordonné, rangé) et fournit deux fonctions qui nous seront très utiles : `spread()` (étaler, comme la confiture) et `gather()` (ramasser, serrer). Vous aurez peut-être deviné que `spread()` permet de passer du format long au format large, tandis que `gather()` permet de convertir un tableau large en tableau long.

3.1. Serrer

`gather()`, *serrer*, consiste à combiner plusieurs colonnes de mesures en une seule. Dans notre exemple, il faut distinguer entre les variables qui sont des *descripteurs* liés aux unités d'observation et les variables qui sont des *mesures* : nous allons considérer que l'identifiant, le sexe, et le niveau d'études sont des descripteurs, et que les quatre notes sont des mesures. Les descripteurs et les mesures sont traités différemment : on veut que les descripteurs soient répétés sur chaque ligne d'une même unité d'observation, tandis que les différentes mesures déterminent autant de lignes différentes. La *Figure 6* illustre la fonction, à partir du tableau qui nous a servi d'exemple précédemment.

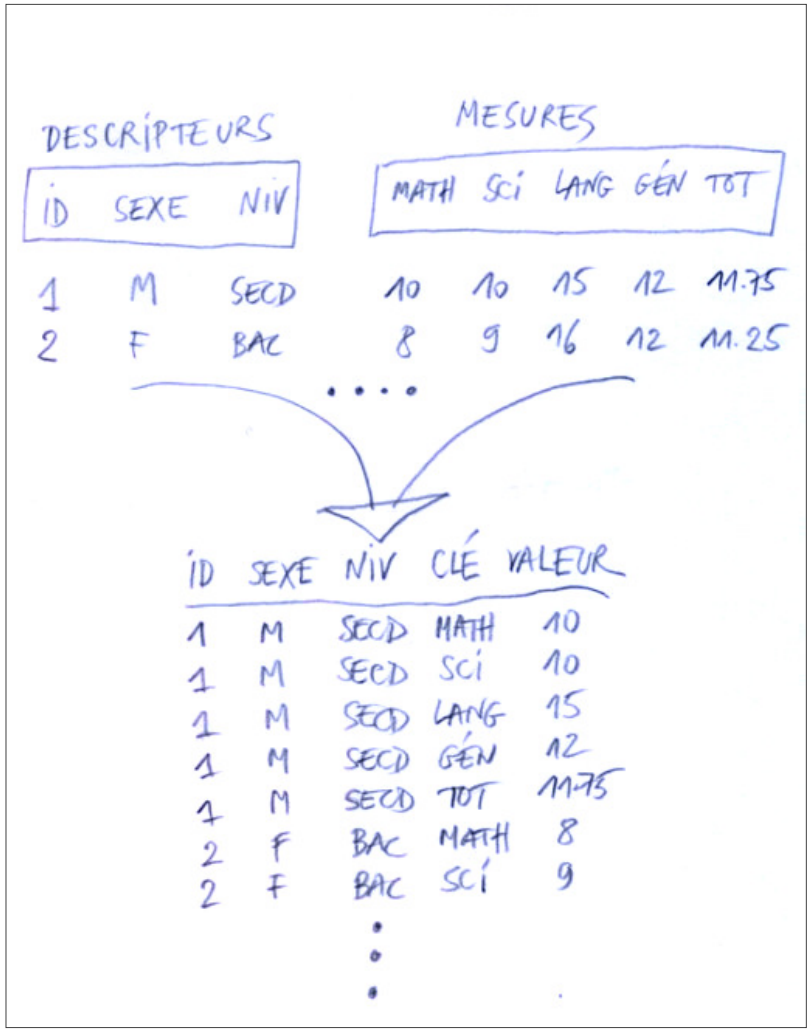

Figure 6.– Le principe de gather()

Pour faire la transformation, il faut donc trois informations :

- la liste des variables qui sont considérées comme descripteurs et non comme mesures
- le nom de la nouvelle colonne qui contiendra les clés, à savoir le nom des différentes mesures
- le nom de la nouvelle colonne qui contiendra les valeurs, à savoir les mesures correspondant aux différentes clés.

Dans l'exemple ci-dessus, nous considérons que les variables `Id`, `Sexe`, et `Niveau` sont des descripteurs, tandis que les variables `math:total` sont les mesures. On passe d'un tableau de 12 lignes et 8 colonnes à un tableau de 12 * 5 = 60 lignes (5 lignes pour chaque sujet), et 5 colonnes (trois descripteurs, la colonne "clé" et la colonne "valeur").

Pour essayer par vous-même les démonstrations qui suivent, il faudra charger le package `tidyr`. Pour rappel, vous installez le package via le bouton `Install` et vous le chargez pour la session avec la fonction `library()`.

```
install.packages("tidyr")
library(tidyr)
dlong <- gather(d, key = branche, value = note,
math:total) %>%
   arrange(Id)
dlong
```

```
Source: local data frame [60 x 5]
Groups: Sexe [2]

      Id   Sexe Niveau branche  note
   <int> <fctr>  <ord>   <chr> <dbl>
1      1      M   Secd    math 10.00
2      1      M   Secd     sci 10.00
3      1      M   Secd    lang 15.00
4      1      M   Secd     gén 12.00
5      1      M   Secd   total 11.75
6      2      F    Bac    math  8.00
7      2      F    Bac     sci  9.00
8      2      F    Bac    lang 16.00
9      2      F    Bac     gén 12.00
10     2      F    Bac   total 11.25
# ... with 50 more rows
```

3.2. Etaler

`spread()` permet de faire l'opération inverse, "étaler", c'est-à-dire rassembler sur une même ligne plusieurs mesures qui appartiennent à la même unité d'observation.

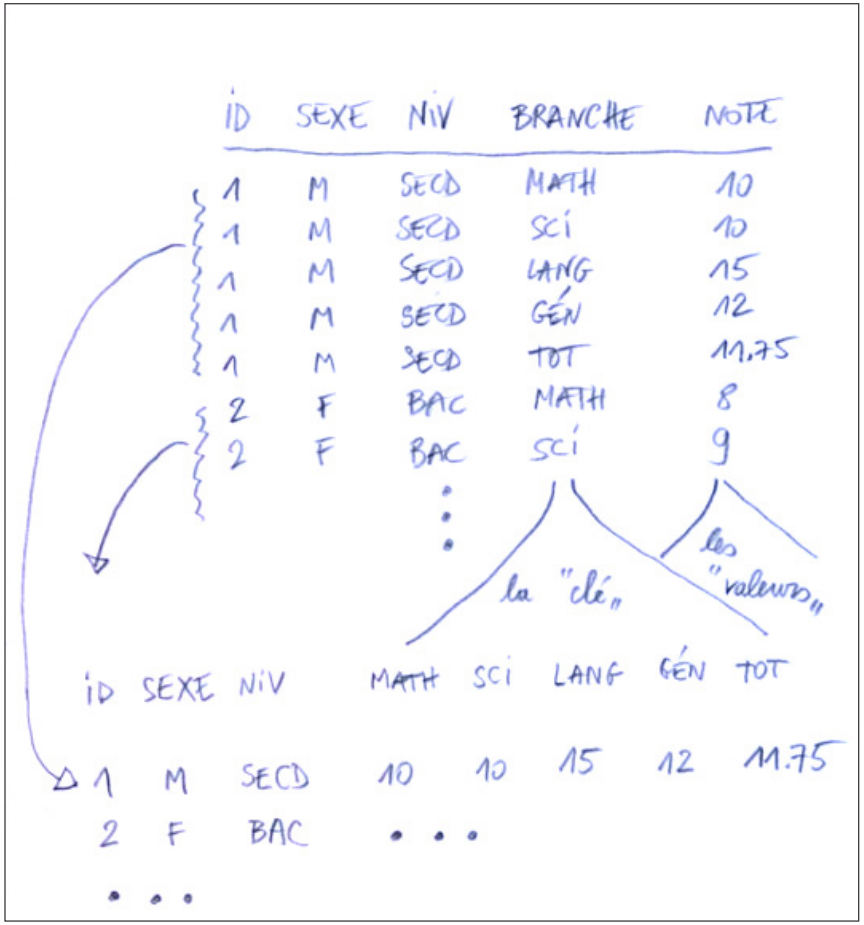

Figure 7.– Le principe de spread()

Il faut donc indiquer quelle est la variable qui contient les mesures, et quelle est la variable "clé" qui va déterminer la répartition en colonnes et les noms des variables à créer.

```
spread(dlong, key = branche, value = note)
```

```
Source: local data frame [12 x 8]
Groups: Sexe [2]

      Id   Sexe Niveau   gén  lang  math   sci total
*  <int> <fctr>  <ord> <dbl> <dbl> <dbl> <dbl> <dbl>
1      1      M   Secd    12    15    10    10 11.75
2      2      F    Bac    12    16     8     9 11.25
3      3      M   Secd    14    16    12    14 14.00
4      4      M     Ma    10    11     9    11 10.25
5      5      F     Ma    11    11    10    13 11.25
6      6      F    Doc    16    15    15    18 16.00
7      7      F     Ma     9    11     8     9  9.25
8      8      M    Bac    13    14    12    14 13.25
9      9      M     Ma    13    16    11    11 12.75
10    10      M    Bac    16    16    15    18 16.25
11    11      M   Prim    13    15    11    13 13.00
12    12      F    Doc    13    12    14    14 13.25
```

Il y a deux difficultés potentielles à bien repérer pour utiliser `spread()` sans difficulté :

- il faut partir d'un tableau long, c'est-à-dire qu'il ne peut y avoir qu'une seule mesure dans le tableau d'origine.
- une seule colonne/variable doit fournir toutes les valeurs de la "clé"

Pour illustrer ces deux points, imaginons une expérience dans laquelle on mesure les temps de réponse. Une fois de plus, les données sont simulées aléatoirement, de façon à vous permettre de créer le tableau en copiant et collant les commandes ci-dessous. Les 25 participants ont passé quatre conditions, et le tableau qui est réalisé reprend ce qui correspondrait au temps de réponse moyen et au nombre de réponses correctes pour les différentes conditions.

On a utilisé une boucle pour créer les données de chaque participant. Le vecteur `Cond` sert à donner des noms (A,B,C,D) aux quatre conditions. Dans la boucle on utilise `sample()` pour générer quatre valeurs supposées représenter les temps de réponse moyens aux quatre conditions, et quatre valeurs correspondant au nombre de réponses correctes. On assemble ensuite le numéro d'ordre du sujet, les noms des conditions, les temps et les nombre de réponses correctes dans un tableau `ds`, et on joint ce tableau aux précédents pour former le tableau général `d3`.

```
Nb.participants <- 25
Nb.Conditions <-  4
Cond <- LETTERS[1:Nb.Conditions]
d3 <- data.frame()
for (Id in 1:Nb.participants){
   TR <- sample(500:750, Nb.Conditions)
   RC <- sample(0:10, Nb.Conditions, repl=T)
   ds <- data.frame(Id, Cond, TR, RC )
   d3 <- rbind(d3, ds)
}
head(d3)
```

```
  Id Cond  TR RC
1  1    A 707  7
2  1    B 667  4
3  1    C 697  9
4  1    D 526  7
5  2    A 696  0
6  2    B 638  5
```

À partir de ce tableau, qui comporte 100 lignes (25 sujets x 4 conditions), on voudrait passer au format large, avec sur une même ligne les TR de chaque participant dans les quatre conditions. Nous avons vu que `spread()` requiert deux paramètres, le nom de la variable qui identifie les différentes mesures, et le nom de la variable qui contient les valeurs pour les différentes mesures. Dans le tableau `d3` ci-dessus, la variable `Cond` indique de quelle mesure il s'agit, et la variable TR donne le temps de réponse moyen correspondant.

Sur cette base on pourrait écrire la commande suivante

```
spread(d3, key=Cond, value=TR)
```

mais malheureusement cela ne donnera pas le résultat escompté (vous pouvez essayer !), parce que le tableau d'origine comporte une *deuxième mesure*, le nombre de réponses correctes. C'est la première condition :

Pour étaler, le tableau long ne peut contenir qu'une seule variable qui constitue la mesure.

Une manière de procéder dans cette situation serait de commencer par transformer le tableau de départ en tableau long.

```
d3long <- d3 %>%
     gather(key=mesure, value=valeur, TR:RC)
head(d3long)
  Id Cond mesure valeur
1  1    A     TR    707
2  1    B     TR    667
3  1    C     TR    697
4  1    D     TR    526
5  2    A     TR    696
6  2    B     TR    638
```

Mais du coup, il y a deux variables qui fournissent l'identité de la mesure : `Cond`, qui indique la condition, et `clé`, qui indique s'il s'agit de la mesure du temps ou des réponses correctes. Pour pouvoir étaler, il va d'abord falloir combiner ces deux variables en une seule colonne. La fonction `unite()` réalise cette opération.

`unite()` remplace deux ou plusieurs variables par une seule qui combine les valeurs avec un caractère de séparation.

```
unite(d3long, col=Cond.Mesure, c(mesure, Cond),
          sep = ".")
```

```
    Id Cond.Mesure valeur
1    1        TR.A    707
2    1        TR.B    667
3    1        TR.C    697
4    1        TR.D    526
5    2        TR.A    696
6    2        TR.B    638
7    2        TR.C    631
8    2        TR.D    695
9    3        TR.A    619
10   3        TR.B    715
# ... with 190 more rows
```

Une fois cette combinaison réalisée, on peut procéder à la transformation.

```
d3Large <- d3 %>%
   gather(key=Mesure, value = valeur, TR:RC) %>%
   unite(col = Mesure.Cond, c(Mesure, Cond)) %>%
   spread(key = Mesure.Cond, value = valeur)
head(d3Large)
  Id RC_A RC_B RC_C RC_D TR_A TR_B TR_C TR_D
1  1    7    4    9    7  707  667  697  526
2  2    0    5    8    7  696  638  631  695
3  3    0    1    3    5  619  715  609  560
4  4    5    3    7    2  666  601  727  572
5  5    3    9    3    3  620  691  520  717
6  6    8   10    4    7  619  723  715  596
```

Une autre manière de faire qui peut parfois convenir serait de séparer les deux mesures et de fabriquer deux tableaux longs distincts :

```
d3TR <- select(d3, -RC) %>% spread(key = Cond, value =
TR)
head(d3TR)
  Id   A   B   C   D
1  1 707 667 697 526
2  2 696 638 631 695
3  3 619 715 609 560
4  4 666 601 727 572
5  5 620 691 520 717
6  6 619 723 715 596
d3RC <- select(d3, -TR) %>%
     spread(key = Cond, value = RC)
head(d3RC)
  Id A  B C D
1  1 7  4 9 7
2  2 0  5 8 7
3  3 0  1 3 5
4  4 5  3 7 2
5  5 3  9 3 3
6  6 8 10 4 7
```

3.3. Unite & Separate

Nous avons utilisé la fonction `unite()` pour combiner les valeurs de plusieurs colonnes en une seule variable combinée, qui concatène les valeurs d'origine.

La fonction `unite()` reçoit comme paramètres le nom du tableau de départ, le nom de la colonne à créer, `col = ...`, le caractère séparateur choisi, `sep = ...`, et la liste des colonnes à concaténer.

```
d4 <- unite(d3long, col=Cond.Mesure, c(mesure, Cond),
sep = ".")
head(d4)
  Id Cond.Mesure valeur
1  1        TR.A    707
2  1        TR.B    667
3  1        TR.C    697
4  1        TR.D    526
5  2        TR.A    696
6  2        TR.B    638
```

L'opération inverse est parfois utile, et elle peut être réalisée avec la fonction `separate()`, qui découpe les valeurs d'une variable en plusieurs colonnes, sur base d'un séparateur.

```
separate(d4, col = Cond.Mesure, into = c("V1", "V2"))
```

```
    Id V1 V2 valeur
1    1 TR  A    707
2    1 TR  B    667
3    1 TR  C    697
4    1 TR  D    526
5    2 TR  A    696
6    2 TR  B    638
7    2 TR  C    631
8    2 TR  D    695
9    3 TR  A    619
10   3 TR  B    715
# ... with 190 more rows
```

Testez vos connaissances !

4. Prêt.e pour le TP ?

Ma Checklist

- ☐ Je sais comment définir des sous-groupes dans un tableau de données
- ☐ Je vois comment agréger les données d'un tableau en fonction d'un ou plusieurs facteurs
- ☐ J'ai compris la fonction de l'opérateur `%>%` J'ai ajouté les fonctions `n()` et `n_distinct()` à mon répertoire de fonctions d'agrégation
- ☐ J'ai compris le principe des boucles simples et des boucles avec index
- ☐ J'ai compris le principe des commandes conditionnelles et la logique de `if` et de `if…else`
- ☐ Je peux identifier les variables *descripteurs* et les variables *mesures* pour transformer un tableau large en tableau long
- ☐ Je peux identifier la variable *clé* et la variable *valeur* dans un tableau long pour le transformer en tableau large
- ☐ J'ai compris comment utiliser les fonctions `unite()` et `separate()`

5. Le package ggplot2 [section optionnelle]

Plutôt que d'offrir différents types de graphiques sous des noms différents, `ggplot2` est un système structuré de construction de graphiques en deux dimensions à partir d'un ensemble d'éléments de base. Il propose donc des règles générales qui permettent de réaliser à peu près n'importe quel type de graphique. Cela présente l'avantage d'offrir une très grande flexibilité et de permettre de contrôler finement l'aspect des graphiques (couleurs, fond, taille, caractères, etc.) en utilisant toujours les mêmes "outils de base"."

5.1. Les principes essentiels

L'idée principale est que n'importe quel graphique statistique peut être construit à partir d'un petit nombre de spécifications. Le nom du package, *ggplot2* vient de "Grammar of Graphics", qui est l'idée de définir les graphiques de manière structurée, avec des composantes élémentaires et des règles de combinaison de ceux-ci, un peu comme un langage.

Quelles sont les spécifications essentielles pour définir un graphique ? En prenant un peu de recul, et de manière abstraite, un graphique 2D est une représentation de données basée sur la disposition spatiale d'éléments géométriques.

Par exemple, dans un *diagramme en bâtons* (*barplot*), on dessine des barres, leur position horizontale indique les différentes modalités d'une variable, et leur hauteur, la valeur d'une autre variable dans chaque modalité; dans un *diagramme de dispersion* (*scatterplot*), on dessine des *points* dont les coordonnées horizontale et verticale indiquent les valeurs de deux variables pour chaque unité d'ob-

servation; le cas échéant, les points pourraient avoir des formes ou des couleurs différentes pour distinguer des sous-groupes d'unités d'observation; dans un *diagramme en lignes* (*line chart*), on dessine une ou plusieurs lignes qui joignent des points dont la position est définie par les valeurs de deux variables; etc.

Donc, pour réaliser un graphique, il faut se poser au moins trois questions :

1. Quelles sont les données, les informations que je veux représenter ?
2. Quel type d'objet géométrique vais-je choisir pour représenter ces informations ?
3. Quelles sont les règles de correspondance entre les informations à visualiser et l'espace du graphique ?

A ces éléments de base se rajoutent des spécifications optionnelles ou secondaires :

4. Les éléments de *style* et de décoration

   La gamme de couleurs, la taille des points, la présence ou pas de grilles, les légendes, les titres, etc.

5. Les transformations statistiques

   Dans certains cas, les informations qu'on veut représenter visuellement ne sont pas directement disponibles dans les données. Pensez par exemple à un histogramme. Pour réaliser un histogramme, il faut d'abord faire un *comptage*, et c'est le résultat du comptage qu'on dessine. On pourrait évidemment passer par une étape de préparation préalable séparée de l'outil graphique (en utilisant, par , la fonction `table()` pour faire le comptage), mais certaines opérations comme le calcul des effectifs, sont tellement courantes qu'il est pratique de les inclure dans l'outil graphique.

6. La démultiplication

   Les graphiques en deux dimensions permettent de bien représenter les relations entre deux ou trois variables au maximum. Quand on veut visualiser un phénomène plus complexe, une technique qu'on appelle *faceting* consiste à démultiplier les graphes en fonction d'une ou deux variables supplémentaires. On en verra des exemples par la suite, mais imaginez que vous êtes intéressé par les aspects inter-culturels du développement cognitif et que vous voulez représenter la relation entre âge (en abscisse) et empan de mémoire (en ordonnée), selon le sexe (en paramètre, donc deux lignes distinctes), mais aussi selon le type de culture (quatre types : A,B, C, D) et la région (5 continents). Une manière de procéder est de réaliser un *graphe à facettes* dont chaque élément est un graphique présentant une partie des résultats (mettons, la relation entre âge, sexe, et empan), pour un type de culture et une région).

Une des caractéristiques de `ggplot2` qui contribue beaucoup à sa puissance et à sa flexibilité est qu'on construit le graphique par

couches (layers), couches, qui viennent se superposer pour réaliser le graphique final. Cela permet de combiner facilement plusieurs types d'objets. Ainsi par exemple, si on veut afficher des points pour indiquer les moyennes dans les différentes conditions, des barres d'erreur pour indiquer la dispersion autour de chaque moyenne, et rejoindre les points appartenant à un même groupe par des lignes, cela se fera en trois opérations, et cela constituera trois couches du graphique.

5.2. Quelques exemples

Reprenons maintenant les trois éléments principaux :

1. Les données : `ggplot()`

   Normalement les données qui serviront au graphique sont dans un tableau, et la spécification des données consiste simplement à initialiser un objet graphique et à déclarer le nom du tableau de données. C'est le rôle de la fonction `ggplot()`, dont le premier argument est simplement le nom du tableau de données dans lequel figurent toutes les variables que le graphique utilise.

   Un avantage de cette spécification est qu'il n'est plus nécessaire de spécifier chaque fois le nom du tableau dans le reste de la construction du graphique. Les noms de variables seront automatiquement associés à des colonnes du tableau.

2. Les objets géométriques : les fonctions `geom_xxx()`

La nature du contenu de l'espace graphique est défini par le choix d'un *geom* (geometric object). Est-ce qu'on veut dessiner des "points", des lignes, des surfaces, des bâtons ? Notez cependant que ce qu'on appelle ici *objets géométriques* ne correspond pas strictement aux caractéristiques purement géométriques des objets dessinés, car les geoms ont également des spécificités statistiques. Par exemple, l'objet géométrique "points" rassemble toutes les formes (carré, losange, triangle, croix…) possibles. De même `ggplot2` distingue entre `geom_bar()`et `geom_histogram()`. Le premier dessinera des bâtons et servira de base pour réaliser des barcharts; le deuxième dessinera aussi des bâtons, mais servira (comme on peut s'en douter) à créer des histogrammes.

Au total, il existe plus de 30 *geoms*, et il est donc hors de question de les décrire systématiquement ici. Vous pourrez trouver plus de détails et notamment une présentation synthétique et exhaustive de tous les éléments dans la *documentation en ligne*. Nous en présenterons seulement quelques-uns, qui sont couramment utilisés : `geom_point()`, `geom_line()`, `geom_bar()`, `geom_boxplot()`, `geom_histogram()`, `geom_errorbar()`, `geom_smooth()`.

Chaque type d'objet géométrique, ou *geom*, requiert des règles de correspondance (*mappings*) avec les données : par exemple, pour afficher des points, `geom_point()` a au minimum besoin de la spécification de *x* et de *y*; pour afficher des barres d'erreur, `geom_errorbar()` a besoin de trois caractéristiques, la position de la barre selon l'abscisse,

x, et les limites inférieure (*ymin*) et supérieure (*ymax*) de la barre verticale; `geom_histogram()` ne demande qu'une spécification, qui correspond aux valeurs de la variable en abscisse, qui déterminent la position des bâtons dénotant les effectifs.

3. Les correspondances : la fonction `aes()`

   Quelles sont les règles de correspondance entre les données numériques et les caractéristiques visuelles du graphique ? C'est le rôle de la fonction `aes()` de définir les correspondances visuelles, ce que l'auteur appelle les *aesthetics* (les "esthétiques", au sens générique et étymologique de *ce qui est donné par les sens, par la vision*, et pas comme dans l'usage courant en français, le *sens du beau*).

   Parmi les *aesthetics*, il y en a une ou deux qui doivent toujours être spécifiées. Un graphique présente des objets géométriques (points, lignes, etc) dans des positions qui sont généralement définies selon les deux coordonnées, *x* et *y* (d'autres systèmes de coordonnées sont possibles mais nous ne les aborderons pas ici). Donc, la plupart du temps, il faudra spécifier ce qui détermine la position des éléments géométriques, selon l'abscisse (x) et l'ordonnée (y).

   Selon la nature du graphique et des objets géométriques à dessiner, d'autres règles de correspondance seront nécessaires. Ainsi la couleur des éléments, leur taille, leur forme, etc. peuvent dans certains graphiques être utilisés (de manière non-décorative) pour transmettre une information supplémentaire. Par exemple, la forme ou la couleur peuvent indicer différentes conditions ou différents groupes.

   D'une façon générale, donc, toutes les fonctions `geom_xxx()` auront un premier argument, `mapping= aes(...)` qui spécifie les correspondances requises. D'autres arguments sont possibles, et varient selon les particularités propres à chaque geom. Il est également possible de spécifier les correspondances dans la fonction `ggplot()`, ce qui évite de répéter la spécification dans chaque élément.

Voyons maintenant quelques exemples pour mettre ces trois fonctions en musique. Nous allons continuer à explorer les données de l'étude de Engelhart et collègues. Les données sont dans le tableau `d`. Vous pouvez récupérer sur l'université virtuelle le fichier "Engelhart.csv".

Essayons de réaliser un diagramme de dispersion, représentant la relation entre niveau d'aggressivité non-provoquée, en abscisse, (variable `aggr1`) et niveau d'aggressivité réactionnelle (variable `aggr2`), en ordonnée.

```
ggplot(d) +
   geom_point(mapping = aes(x=aggr1, y= aggr2))
```

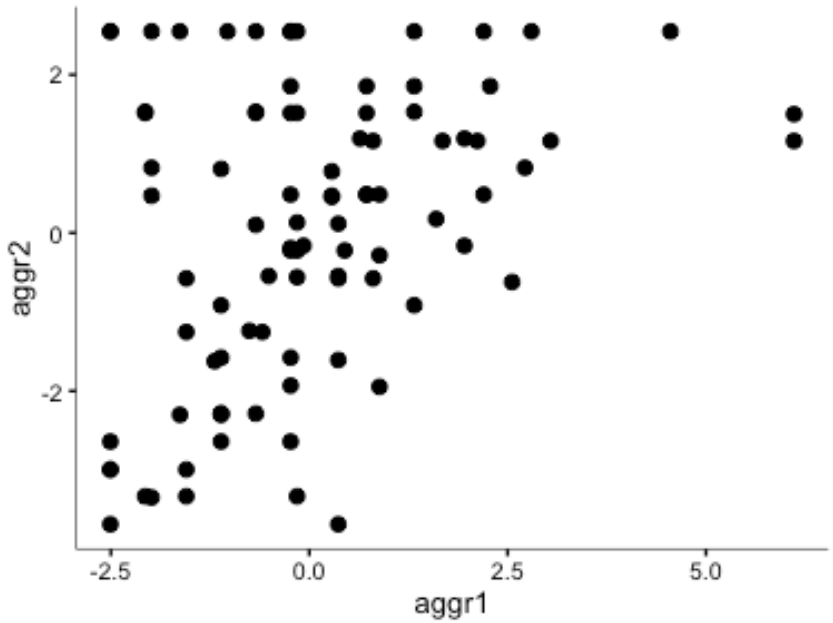

Figure 8.– Relation entre aggressivité non-provoquée (aggr1) et aggressivité réactionnelle (aggr2)

Dans ce premier exemple, on a fait deux choses : 1) référencer le tableau de données `d`, et 2) associer l'abscisse à la variable `aggr1`et l'ordonnée à la variable `aggr2`, pour dessiner des points (`geom_point()`).

Nous reprendrons cet exemple plus loin, pour en améliorer le style. Mais voyons d'abord un autre exemple : on voudrait visualiser le niveau d'agressivité selon le groupe et la condition, et y ajouter des barres d'erreur. Nous allons donc utiliser `dplyr` pour calculer le tableau des moyennes, des écart-types et des erreurs-standard par groupe et condition (rappel : l'erreur standard est l'écart-type divisé par la racine carrée du nombre d'observations).

Calculons d'abord le tableau des statistiques.

```
d %>%
   group_by (gr , cond) %>%
   summarise(m = mean(aggr1), sd = sd(aggr1), n= n(),
                 se = sd/sqrt(n))
Source: local data frame [4 x 6]
Groups: gr [?]

      gr       cond         m     sd     n      se
   <fctr>     <fctr>     <dbl>  <dbl> <int>   <dbl>
1  autism nonviolent  0.037564 1.6393    23 0.34181
2  autism    violent  0.674547 2.1060    24 0.42988
3 typical nonviolent -0.316684 1.3452    21 0.29354
4 typical    violent -0.324314 1.6052    23 0.33471
```

On peut enchaîner directement les commandes de `dplyr` avec `ggplot()`, qui utilisera alors le tableau résultant comme input pour le graphique. Nous voulons dessiner les points correspondant aux quatre moyennes, avec le groupe en paramètre et la condition en abscisse.

```
d %>%
   group_by (gr , cond) %>%
   summarise(m = mean(aggr1), sd = sd(aggr1), n= n(),
                 se = sd/sqrt(n)) %>%
   ggplot() +
     geom_point(mapping = aes(x = cond, y= m))
```

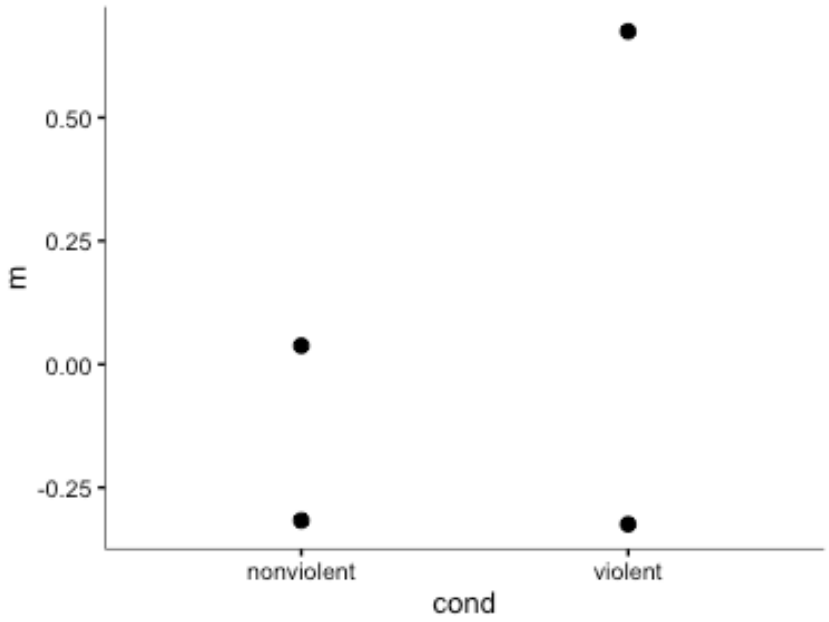

Figure 9.– Niveau d'aggressivité non-provoquée (aggr1) selon le groupe et la condition

Nous voulons ajouter deux éléments : d'abord, une différenciation entre le groupe Typique et le groupe ASD; pour cela, on va associer la couleur des points à la variable `gr` :

```
d %>%
   group_by (gr, cond) %>%
   summarise(m = mean(aggr1), sd = sd(aggr1), n= n(),
             se = sd/sqrt(n)) %>%
   ggplot() +
   geom_point(mapping = aes(x = cond, y= m,
                color = gr))
```

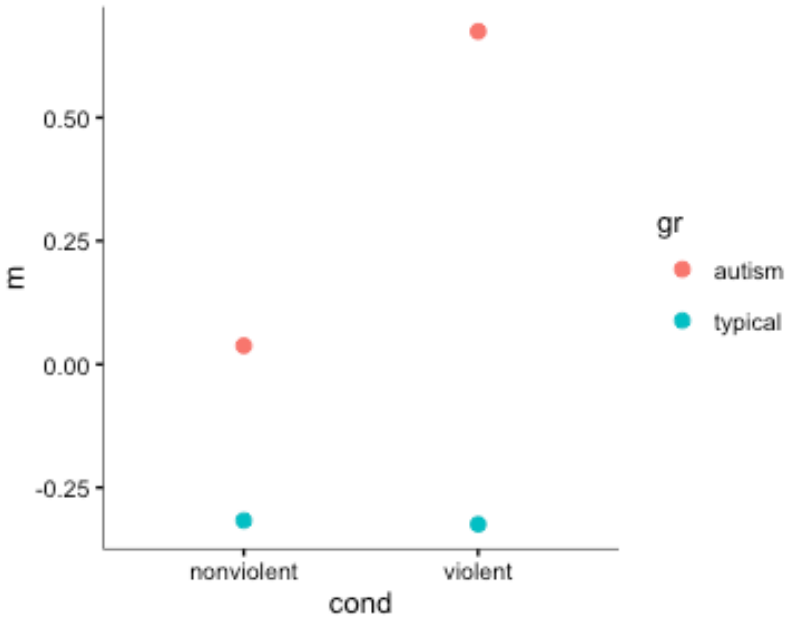

Figure 10.– Niveau d'aggressivité non-provoquée (aggr1) selon le groupe et la condition

Ensuite, on voudrait ajouter des lignes pour rendre plus visible les deux groupes distincts. Pour cela on ajoute un autre *layer*, avec `geom_line()`, et il faut spécifier une autre aesthetic, qui indique comment grouper les valeurs pour former des lignes. Ici, c'est la variable `group` qui indique quels points doivent être reliés :

```
d %>%
   group_by (gr, cond) %>%
   filter(! is.na(aggr1)) %>%
   summarise(m = mean(aggr1), sd = sd(aggr1), n= n(),
se = sd/sqrt(n)) %>%
   ggplot() +
     geom_point(mapping=aes(x =cond, y=m, color=gr)) +
     geom_line( mapping=aes(x=cond, y=m, color=gr,
group = gr))
```

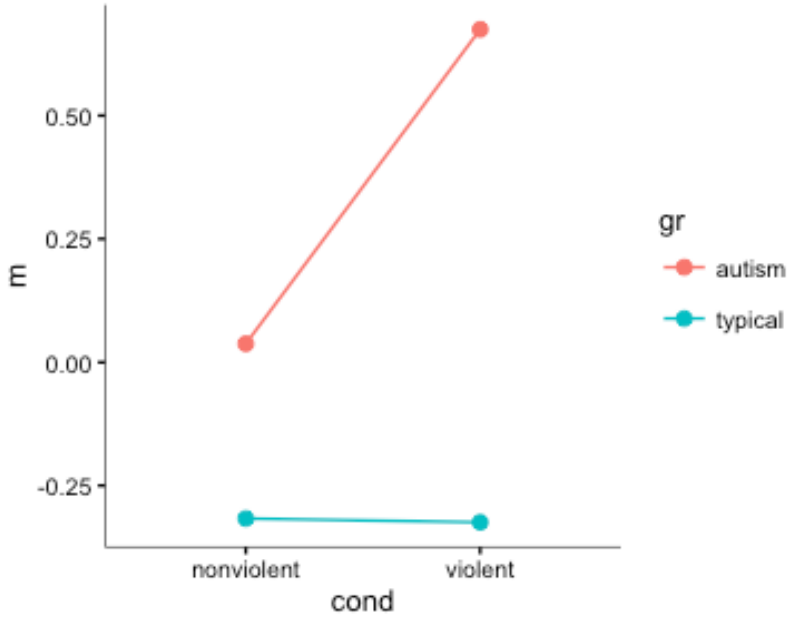

Figure 11.– Niveau d'aggressivité non-provoquée (aggr1) selon le groupe et la condition

Enfin, on voudrait ajouter des barres d'erreurs. On peut pour cela utiliser soit `geom_errorbar()`, soit `geom_linerange()`. Les deux dessinent des barres verticales. Il faut donc spécifier à quelle abscisse les barres doivent apparaître (ici, `x=cond`), ainsi que leur borne inférieure (`ymin`) et supérieure (`ymax`). Ici, nous allons afficher des barres qui reprennent 1 erreur-standard au-dessus et au-dessous de la valeur moyenne.

```
d %>%
   group_by (gr, cond) %>%
   summarise(m = mean(aggr1), sd = sd(aggr1), n= n(),
se = sd/sqrt(n)) %>%
   ggplot() +
     geom_point(mapping = aes(x = cond, y= m, color =
gr)) +
     geom_line( mapping = aes(x = cond, y= m, color =
gr, group = gr)) +
     geom_errorbar(aes(x=cond, ymin=m-se, ymax=m+se,
color = gr), width=0.1)
```

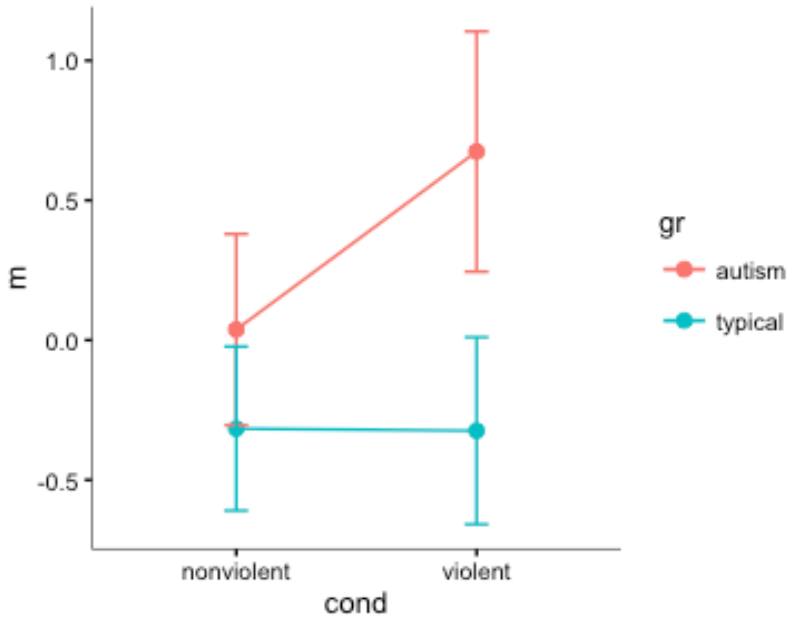

Figure 12.– Niveau d'aggressivité non-provoquée (aggr1) selon le groupe et la condition

Souvent, comme ci-dessus, les *mappings* principaux seront les mêmes pour les différents geoms. On peut alors simplifier l'écriture, en reprenant ces paramètres dans la fonction `ggplot()`, ce qui évite de les répéter à chaque fois. Les mappings spécifiés dans `ggplot(aes(...))` sont valables pour toutes les couches du graphique, sauf si les mappings sont modifiés dans la description des geoms eux-mêmes.

```
d %>%
   group_by (gr, cond) %>%
   summarise(m = mean(aggr1), sd = sd(aggr1), n= n(),
se = sd/sqrt(n)) %>%
   ggplot(mapping = aes(x = cond, y= m, color = gr,
group = gr)) +
     geom_point() +
     geom_line() +
     geom_errorbar(aes(ymin=m-se, ymax=m+se),
width=.05)
```

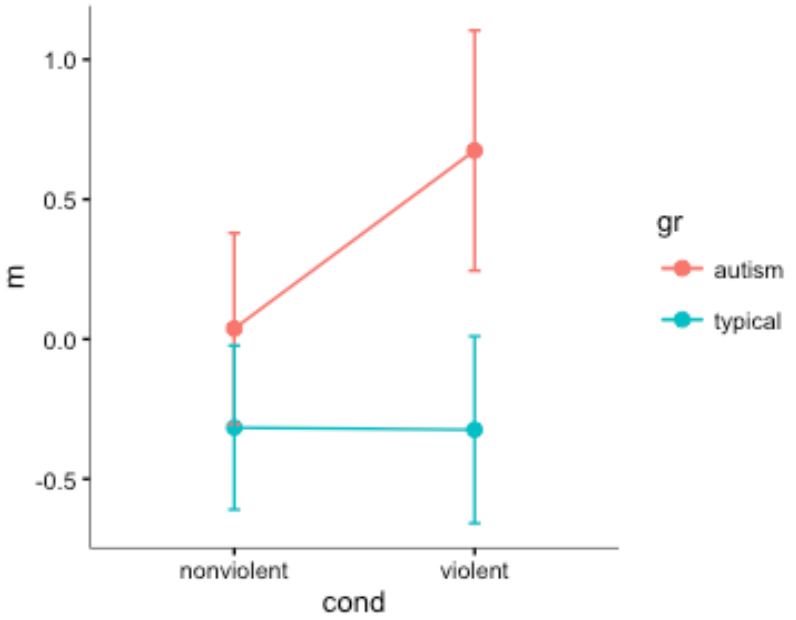

Figure 13.– Niveau d'aggressivité non-provoquée (aggr1) selon le groupe et la condition

Enfin, les graphiques produits par `ggplot2` sont des objets comme les autres, et rien n'empêche de les assigner à un nom pour les modifier. Nous allons créer deux objets avec les graphiques

précédents. Nous verrons dans les sections suivantes comment on peut les enjoliver…

```
graphe_moy <-
   d %>%
   group_by (gr, cond) %>%
   summarise(m = mean(aggr1), sd = sd(aggr1), n= n(),
se = sd/sqrt(n)) %>%
 ggplot(mapping = aes(x = cond, y= m, color = gr,
           group = gr)) +
     geom_point() +
     geom_line() +
     geom_errorbar(aes(ymin=m-se, ymax=m+se),
width=.05)

scattergram <- ggplot(d, mapping = aes(x=aggr1, y=
aggr2)) +
   geom_point()
```

5.3. Les éléments de style

Il y a deux sortes d'éléments de style : ceux qui ressortent des geoms et ceux qui relèvent de l'aspect global du graphique. Par exemple, la taille ou la forme des points, la largeur des bâtons, l'épaisseur et le style des lignes… sont des caractéristiques propres aux geoms. Par contre, la couleur du fond, la présence ou l'absence de grillage, la présentation des axes, les légendes… sont des caractéristiques globales définies dans le `theme`du graphique.

Les styles des geoms

Pour les éléments propres aux geoms, nous allons nous limiter à quelques exemples. Les caractéristiques qui peuvent être manipulées pour chaque geom sont bien décrites dans la *documentation en ligne* avec de très nombreux exemples.

Un élément qui est parfois cause de confusion est la distinction entre - établir des correspondances entre une esthétique et une variable, qui se fait via l'argument `mapping= aes(...)`, et - attribuer une valeur à une esthétique, qui se fait en assignant une valeur directement à une esthétique.

Par exemple, si je veux que tous les points du diagramme de dispersion `scattergram` que nous avons construit précédemment soient rouge, de taille moyenne , et de forme triangulaire, je pourrais ajouter les arguments suivants à `geom_point` :

```
ggplot(d, mapping = aes(x=aggr1, y= aggr2)) +
   geom_point(size=2, color="red", fill= "red",
           shape = 24)
```

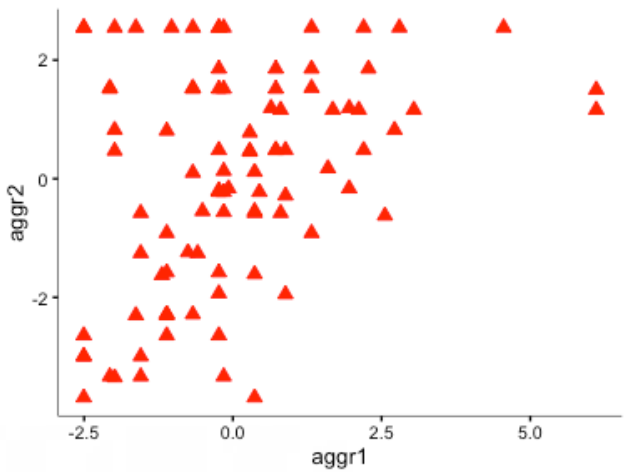

Figure 14.– relation entre niveau d'aggressivité non-provoquée et réactionnelle

Par contre, si nous voulons faire correspondre la couleur et la forme à la variable groupe, par exemple

```
ggplot(d, mapping = aes(x=aggr1, y= aggr2)) +
     geom_point(mapping= aes(color= gr, fill= gr,
          shape = gr), size=2)
```

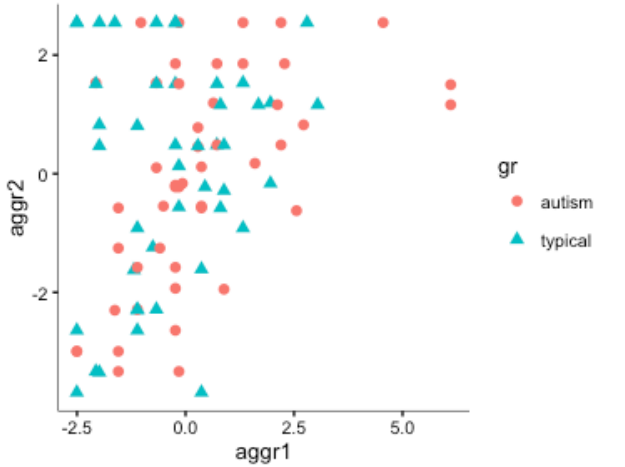

Figure 15.– relation entre niveau d'aggressivité non-provoquée et réactionnelle

Et nous pourrions même introduire une 4ème dimension par le biais de la couleur :

```
ggplot(d, mapping = aes(x=aggr1, y= aggr2)) +
     geom_point(mapping= aes(color= aggr3, fill= aggr3,
shape = gr), size=2)
```

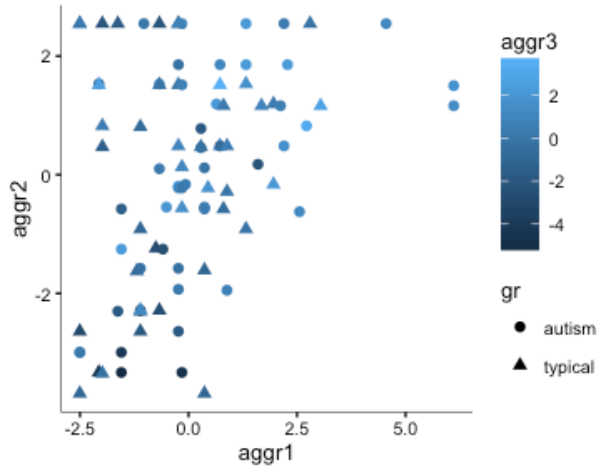

Figure 16.– relation entre niveau d'aggressivité non-provoquée et réactionnelle

Voici les caractéristiques principales, pour les quelques geoms les plus utilisés :

- `geom_point()`: la couleur du pourtour (`color=`), la couleur de remplissage (`fill=`), la taille (`size=`), la forme (`shape=`), le degré de transparence (`alpha=`)

  Pour l'attribution de **couleurs** il existe des *listes de tous les noms reconnus dans R*, avec leur code hexadecimal et correspondance RGB. Il y a aussi plusieurs palettes qui sont disponibles lorsqu'il s'agit d'établir une correspondance entre une variable discrète ou continue et des couleurs.

Le **degré de transparence**, `alpha`, est défini par une valeur numérique entre 0 et 1 ou une fraction. On utilise souvent une fraction unitaire, dont le dénominateur indique le nombre de superpositions nécessaires pour obtenir l'opacité complète : veut dire que 5 éléments superposés produisent la teinte opacifiée, signifie qu'il en faut 10, etc.

Les **formes** des symboles sont définies par des valeurs numériques de 0 à 25.

Figure 17.– Les formes de symboles (seuls les symboles 15 à 20 sont remplis)

- `geom_line()`: les mêmes que `geom_point()` + le type de ligne (`linetype=`) : `solid`, `dotted`, `dashed`,…
- `geom_bar()`: les mêmes, + la largeur des barres (`width=`). Il faut spécifier `stat="identity"` si on veut dessiner des barres dont la hauteur est définie par `y`. Sinon, la hauteur des barres est basée sur un calcul d'effectifs par intervalles (bins).
- `geom_histogram()` : les mêmes que `geom_bar()` + la largeur des intervalles (`binwidth=`).

Les éléments de style globaux

Nous avons créé précédemment un objet `graphe_moy`. Nous allons l'utiliser ici pour montrer comment on peut en modifier le style.

On peut superposer un `theme` au graphique pour modifier son aspect général (les axes, les grilles, le fond). Chaque élément du thème peut également être modifié individuellement (voir la documentation pour `theme`).

```
graphe_moy + theme_classic() + theme(legend.
position="top")
graphe_moy + theme_minimal() + theme(legend.
position="top")
graphe_moy + theme_bw() + theme(legend.position="top")
graphe_moy + theme_gray() + theme(legend.
position="top")
```

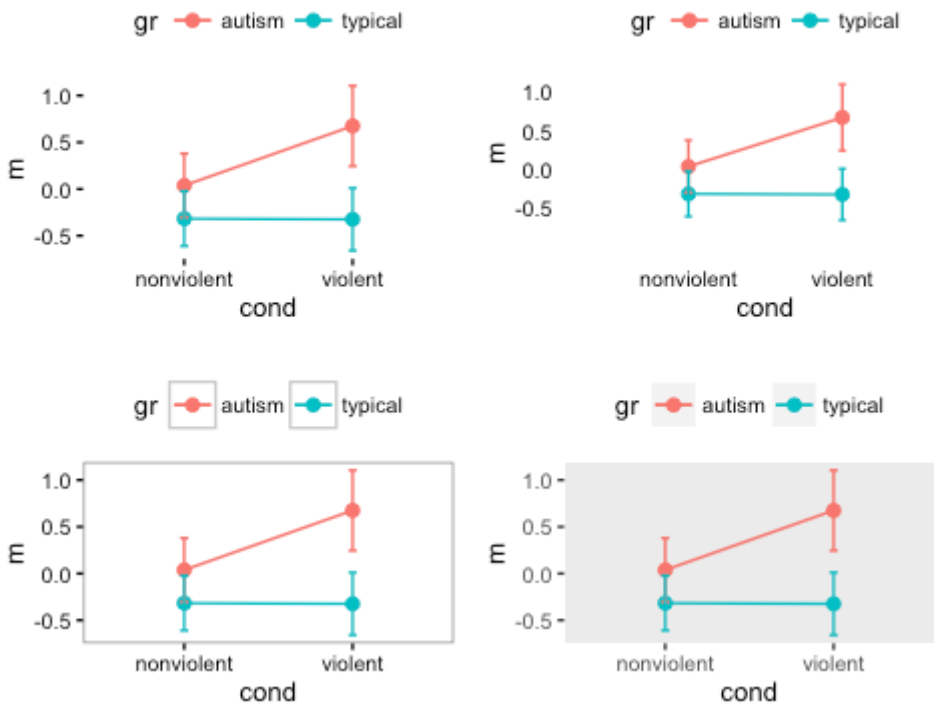

Figure 18.– Différents thèmes

On peut modifier les noms des axes et ajouter un titre général

```
graphe_moy <-  graphe_moy + labs(y="Niveau
d'aggressivité", x="Type de Jeu", color= "Groupe",
title="Aggressivité réactionnelle standardisée")
graphe_moy
```

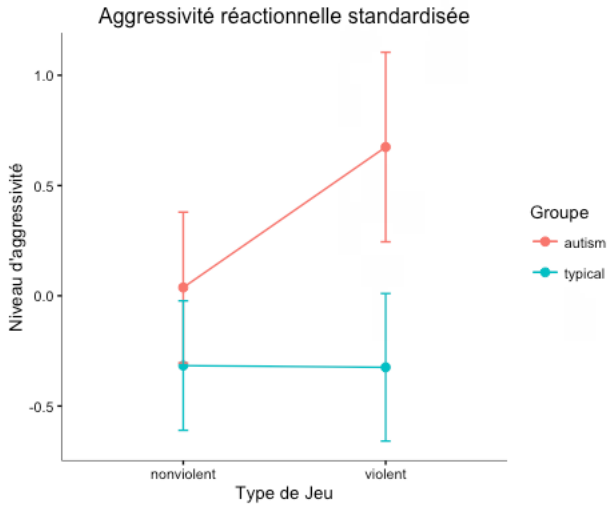

Figure 19.– Modifier les titres

On peut modifier les caractéristiques des échelles. Chaque correspondance entre une variable et une esthétique est définie par une échelle. Selon la nature de l'esthétique, on utilisera des échelles différentes. Par exemple dans le graphique ci-dessous, il y a trois échelles, `scale_x_discrete()` pour l'abscisse, `scale_y_continuous()` pour l'ordonnée, et `scale_colour_manual()` pour la couleur.

Nous utilisons la première pour modifier les étiquettes (`labels=`), la seconde pour modifier les limites et les marques, et la troisième pour modifier les couleurs en les listant explicitement. Il existe aussi beaucoup d'autres types d'échelles.

```
graphe_moy <-  graphe_moy +
   scale_colour_manual(values=c(rgb(0,191/255,255/255),
"deepskyblue4")) +
   scale_y_continuous (limits=c(-2,2), breaks=-2:2) +
   scale_x_discrete(labels=c("Non Violent", "Violent"))
graphe_moy + theme_classic() + theme(legend.position =
c(0.1,0.8))
```

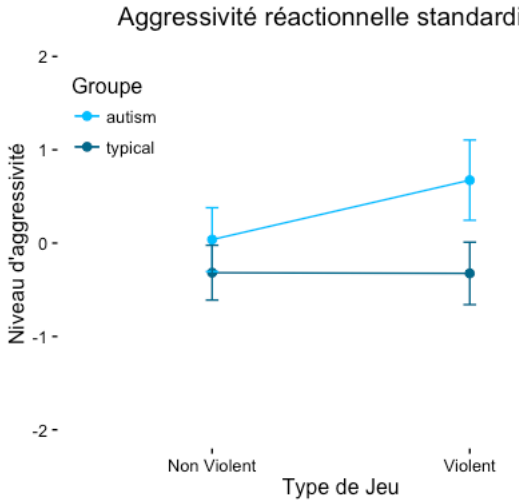

Figure 20.– Modifier les échelles

En particulier pour les couleurs, on peut également choisir parmi plusieurs palettes.

```
scatterplot <- ggplot(d, mapping = aes(x=aggr1, y=
aggr2)) +
   geom_point(mapping= aes(color= aggr3, shape = gr),
size=2)
scatterplot + scale_color_gradient(low="grey95",
high="grey10")
scatterplot + scale_color_continuous(low="red",
high="white")
```

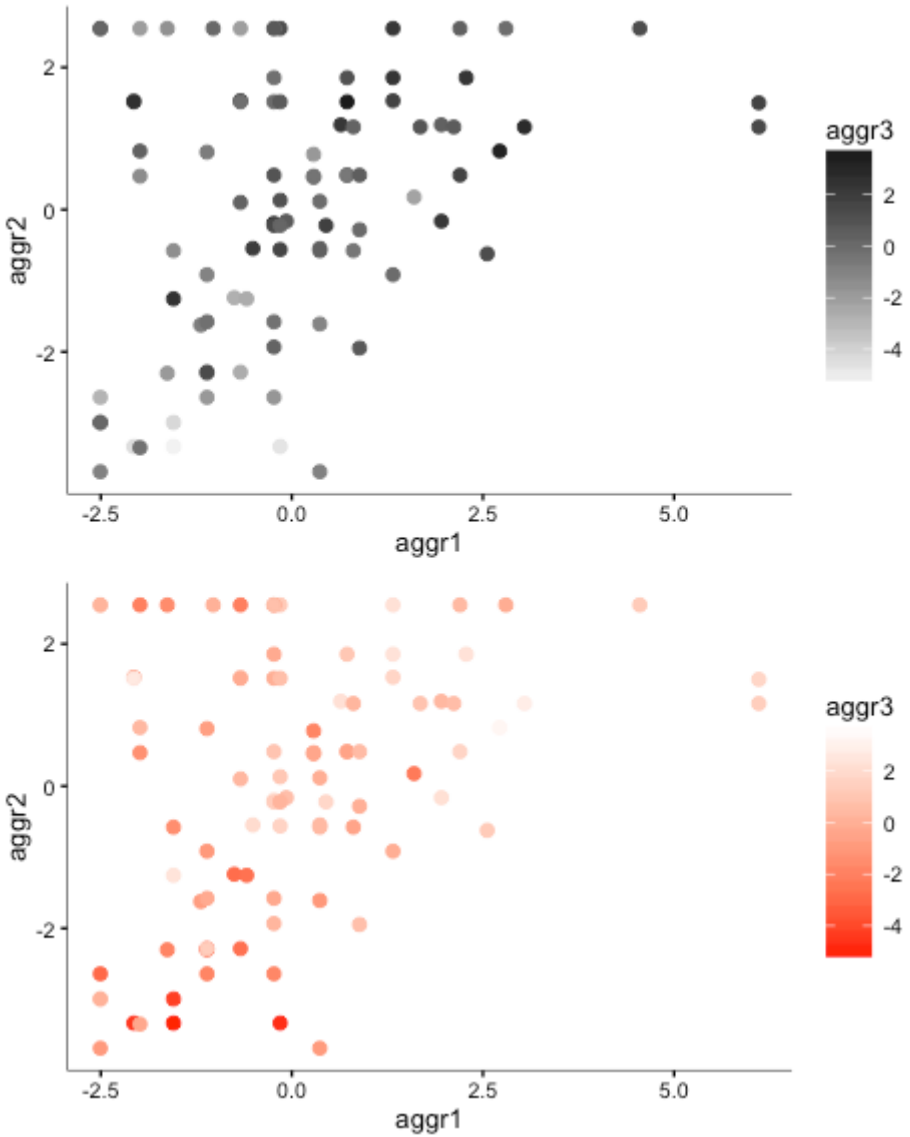

Figure 21.– Echelles continues de couleurs

5.4. La démultiplication

Dernier élément, les graphiques en plusieurs facettes. Plutôt que de rassembler les deux groupes et les deux conditions dans le même graphique, on peut très facilement décomposer le graphique en quatre *facettes*. La fonction est `facet_grid()` et son argument est une formule : les noms de variables à gauche définissent la ou les dimensions qui déterminent les lignes, et les variables à droite du signe `~` définissent les colonnes de la grille.

```
scatterplot <- ggplot(d, mapping = aes(x=aggr1, y=
aggr2)) +
   geom_point(mapping= aes(color= aggr3), size=2)
scatterplot + facet_grid(gr~cond)
```

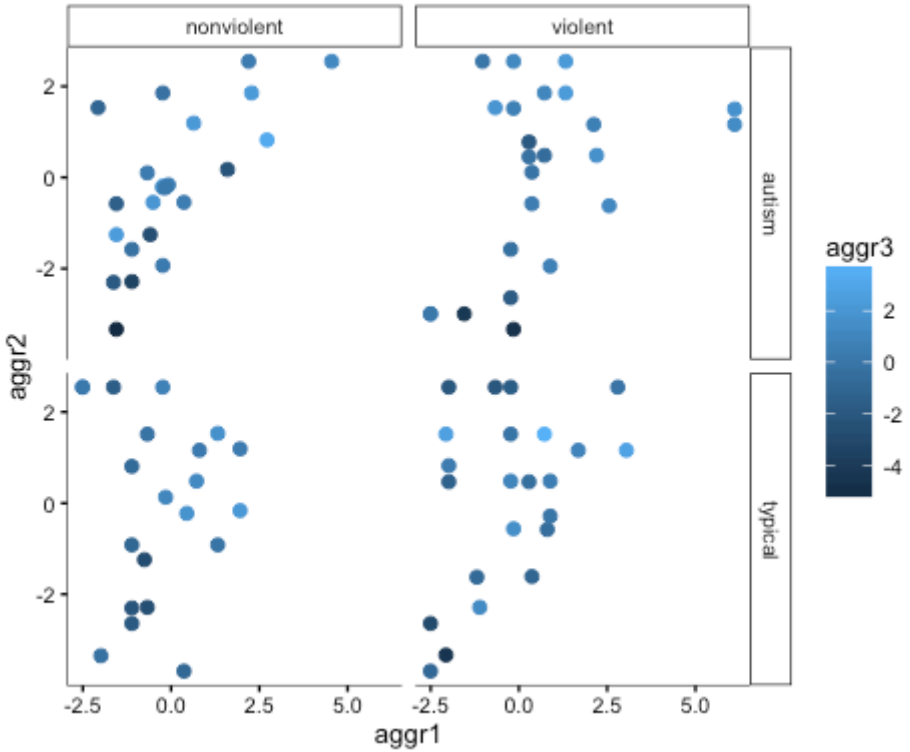

Figure 22.– Graphique à facettes, exemple 1

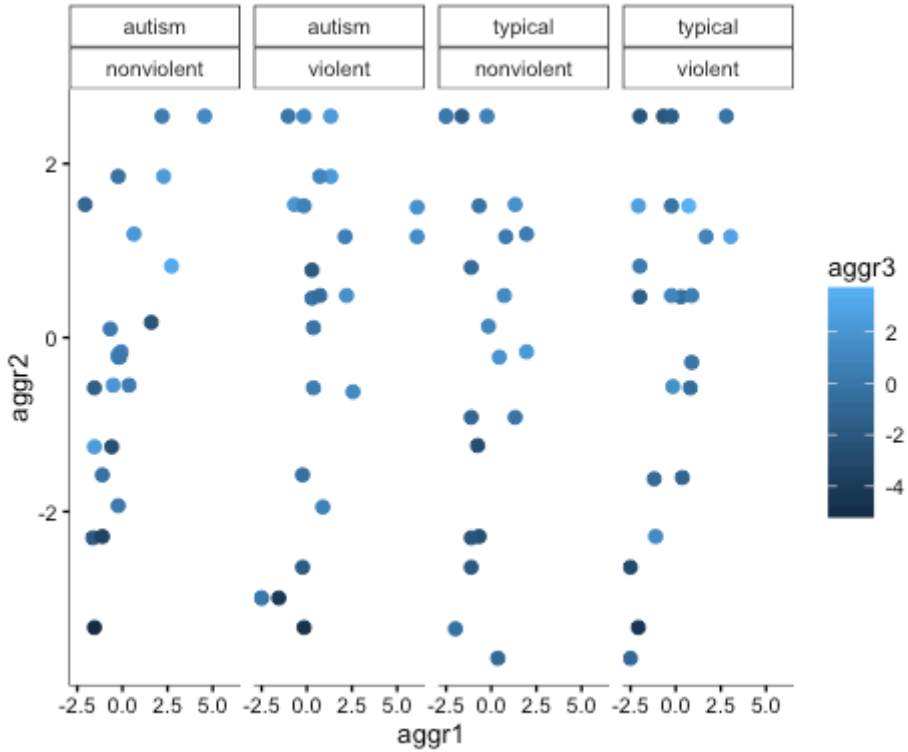

Figure 23.– Graphique à facettes, exemple 2

S'il n'y a qu'une ligne ou qu'une colonne, on le marque par un point à gauche ou à droite, respectivement.

```
scatterplot + facet_grid(.~ gr + cond)
```