Méthodes Quantitatives avec R

P.-O. Caron

2022-05-25

Contents

Bienvenue!

Bienvenue dans la version préliminaire et en ligne du livre *Méthodes Quantitatives avec **R*** de P.-O. Caron. Le livre porte sur la programmation statistique en **R** et vise démystifier les mécanismes ésotériques derrières les logiciels statistiques afin de les rendre accessibles au plus grand nombre de personnes. Son objectif est de faire le pont entre la statistique et la programmation afin que les expérimentateurs comprennent et autonomisent leur pratique. L'approche permet à la fois une meilleur compréhension des statistiques à l'aide du logiciel, mais aussi un objectif pour débuter la programmation.

Il vise un public autant intéressé à s'initier à **R** qu'à en connaître davatange sur la statistique, mais surtout un lecteur qui souhaite maîtriser les deux.

Si vous avez des commentaires ou des suggestions, n'hésitez pas à me les partager. C'est ouvrage est en construction. Il n'est pas parfait et peut contenir (contient!) des erreurs. Vous pouvez m'écrire à l'adresse suivante : pier-olivier[at]teluq.ca pour toutes suggestions d'amélioration.

Remerciements

Les étudiants et étudiantes de la classe PSY7105 d'automne 2021.

Préface

> Il faut être naïf pour croire que le code fonctionne tel que prévu.

La méthode scientifique impose aux expérimentateurs de tirer des conclusions avec scrutinie sur leur objet d'étude. Toutefois, ils font confiance à d'autres pour des tâches qui dépassent leur champs de compétence, et ce, avec raisons, car ces autres expertises peuvent être fort compliquées et nécessitées d'autres formations académiques complémentaires pour les acquérir. En sciences humaines et sociales, les statistiques font parties de ces expertises exterrieures qui se doivent d'être apprises. En comprendre les fondements et l'implantation logiciel est nécessaire pour ne pas fléchir son esprit critique, ne serait-ce qu'un instant. Trop souvent, les expérimentateurs ont une introduction aux statistiques et font confiance pour les calculs sophistiqués à des logiciels commerciaux. Bien souvent, le tout se déroule sans tracas. Une expérience plus substantielleavec ces logiciels montrera qu'ils ont eux-aussi leurs défauts, leurs bogues et leurs erreurs de calcul. Sans connaître le résultat attendu, comment distinguer le vrai du faux des sorties statistiques?

Par exemple, que vaut l'expression $-9^{(.5)}$? Reconnaître qu'il s'agit de $\sqrt{(}-9)$ indique immédiatement que le résultat n'est pas réel (c'est un nombre complexe). Si ce calcul est demandé à **R**,

```
-9^(.5)
#> [1] -3
```

celui-ci retourne la réponse -3 à cause de la priorité des opérations, $-(9)^{\cdot 5}$. Était-ce la réponse désirée?

Des idiosyncrasies computationnelles se retrouvent dans tous les logiciels. En avoir conscience est important pour le programmeur et reste trivial pour les expérimentateurs…

Bennett et al. (2010) montrent que les saumons de l'Atlantiques décédés ont des cognitions sociales. Ils placent sous imagerie par résonance magnétique fonctionnelle (IRMf) le saumon pendant une tâche de reconnaissance des émotions chez

les humains. La Figure 1 montre le résultat de leur analyse. Soit ils sont tombés sur une découverte étonnante en termes de cognition ichtyologique post-mortem, soit quelque chose cloche en ce qui concerne l'approche statistique utilisée.

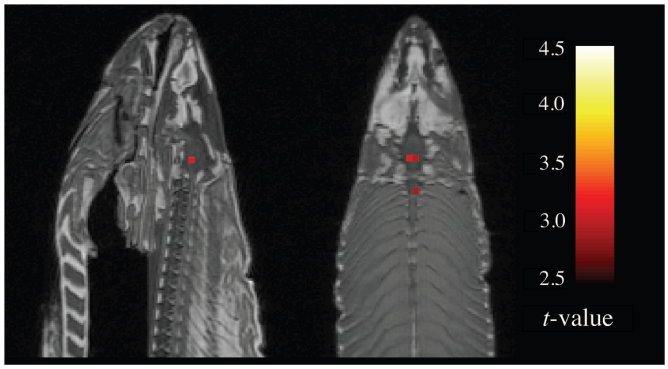

Figure 1: Cognitions sociales d'un saumon atlantique décédés. Tirés de Bennett et al. 2010, p.4

Les points rouges de la Figure 1 indiquent les congitions sociales du saumon. Plus sérieusement, ces chercheurs critiquent l'absence de contrôle de l'erreur de type I (faux positif) lors de comparaison multiple. Ce problème est d'autant plus important avec des humains sous IRMf, car ces situations comportent beaucoup de bruit statistiques, contrairement au saumon décédé (qui n'en produit pas du tout). Ils montrent bien qu'en absence de ce contrôle, l'analyse peut fournir des résultats farfelus. Il revient à l'expérimentateur de bien utiliser les analyses statistiques et de savoir ce que le logiciel fait et ne fait pas.

L'étude de Bennett et al. (2010) n'est qu'un canular. Les auteurs ont remporté le prix Ig Nobel en 2012, un prix scientifique pour de vrais chercheurs ayant réalisé une vraie étude *faisant d'abord rire, puis réfléchir.*

En pratique, ce type de problème devrait être marginal…

Eklund et al. (2016) tentent de valider les méthodes statistiques derrière l'IRMf à l'aide de données réelles. Ils utilisent des données d'IRMf provenant de 499 personnes saines et en état de repos pour effectuer 3 millions d'analyses de groupes de tâches. En utilisant ces données nulles avec différents plans expérimentaux, ils estiment l'incidence des résultats significatifs (en théorie fixé à 5% de faux positifs pour un seuil de signification de 5 %). Toutefois, ils constatent que les progiciels les plus courants pour l'analyse de l'IRMf entraînent des taux de faux positifs allant jusqu'à 70 %. Ces résultats remettent en question la validité d'un certain nombre d'études utilisant l'IRMf et ont des conséquences importantes sur l'interprétation des résultats de neuro-imagerie.

Cette étude a fait un tollé dans la littérature scientifique. Bien que certains aspects de l'étude soient attaquables, il n'en demeure pas moins qu'elle exacerbe

le danger de faire confiance aveuglément aux analyses statistiques.

L'objectif de ce court texte n'est pas de discrédité les analyses statistiques ou les chercheurs qui les emploient. Les statistiques sont très utiles et fournissent de l'information qu'il serait impossible d'obtenir autrement. Il s'agit principalemetn d'une mise en garde justifiant en partie l'approche pédagogique de ce livre.

Il est dit en partie, car le seconde, en filigrane de livre, repose sur la double approche d'apprentissage statistique et programmation. En programmant des modèles statistiques, le lecteur a une plus grande emprise sur ceux-ci. Il peut mieux les utiliser, les étudier et les modifier. Il peut s'en servir pour connaître et apprendre davantage. Certaines boîtes noires resteront irrémédiablement des boîtes noires pour le lecteur, mais plus il les démystifiera, mieux il comprendra ce qui se cache derrière les programmes statistiques.

La première section de ce livre justifie et explique le logiciel **R** et montre comment l'installer ainsi que quelques rudiments statistiques qui seront fort utiles pour le reste de la lecteur. La seconde section couvre ce qui sera probablement le plus important pour l'utilisateur : l'importation, la gestion et la visualisation des données.

TODO

Part I

Rudiments

Chapter 1

Commencer

R (R Core Team, 2022) est un logiciel de programmation statistique libre-accès et un environnement pour la computation statistique et l'affichage graphique. Il s'agit d'un projet GNU qui est similaire au langage et à l'environnement S, développés aux Laboratoires Bell (anciennement AT&T, aujourd'hui Lucent Technologies) par John Chambers. Créé par **R**oss Ihaka et **R**obert Gentleman, **R** fournit une grande variété de techniques statistiques (modélisation linéaire et non linéaire, analyses statistiques classiques, analyse de séries chronologiques, classification) et graphiques, et est hautement extensible.

R est un logiciel basé sur la syntaxe plutôt qu'une approche pointer-et-cliquer (*point-and-click*) comme les logiciels traditionnels. Il peut être plus effrayant ou apparaître trop complexe pour un nouvel utilisateur, mais une fois apprivoisée, cette bête démontre un bien meilleur potentiel que ce soit en automatisation, en personnalisation, en production de figure de haute qualité, etc. **R** a l'avantage de mettre en plein contrôle ses utilisateurs. Bref, c'est une créature qu'il vaut la peine de maîtriser.

Les logiciels traditionnelles suspendent trop souvent la réflexion critique. Ils sont dociles. L'usager clique sur les bonnes options et obtient les résultats désirés (espérons-le!). En échange d'une expérience "simple et intuitive", ils compromettent l'épanouissement de l'utilisateur et rendent l'analyse statistique comme une boîte noire, un programme dont le fonctionnement ne peut être connu. Sont évacuées toutes connaissances des analyses, seules les entrées et les sorties sont pertinentes.

En se limitant à ces logiciels, les utilisateurs sont également à la merci des compagnies qui les distribuent. Par exemple, elles maintiennent des prix exorbitants pour des licences annuelles, malgré le faible soutien technique, la désuétude ou le manque de mise à jour, la présence de bogues informatiques. Ces problèmes sont monnaie courante bien que la licence ne soit pas de la petite monnaie.

Contrairement aux logiciels traditionnels, **R** permet de réaliser les analyses, mais

aussi de les programmer soi-même, de générer des données propres à un modèle, de rester à jour sur les nouvelles tendances et les découvertes en méthodologie de la recherche, de partager aisément les connaissances et la reproduction d'analyses sophistiquées, et tout cela, gratuitement. Évidemment, cela n'est possible que par l'immense communauté derrière le logiciel.

1.1 Pourquoi R?

R est complètement gratuit. Il est le logiciel le plus utilisé parmi les scientifiques de sciences de données, statisticiens, etc. Il est l'exemple ultime d'une plateforme communautaire qui fait mieux que les compétiteurs commerciaux. De plus en plus de personnes migrent vers **R**, mais peu de ses utilisateurs quittent le logiciel vers un autre. Et plus de personnes se joignent à la communauté, plus il y a de documentation, d'aide, de soutien, que ce soit sous forme de livres, de vidéos, d'article, d'ateliers, de formations. L'expérience **R** devient de plus en plus accessible aux nouveaux immigrants. Comme le code source est ouvert, ses utilisateurs collaborent à la création de modules augmentant ses capacités qui permettent de résoudre des problèmes de plus en plus sophistiqués (et pas juste en statistiques!). Ces modules sont également gratuits et il y en a littéralement des milliers.

L'avantage de maîtriser **R**, plus spécifiquement de l'utilisation de syntaxe, pour l'expérimentateur est de rendre l'analyse statistique facilement *transmissible* entre expérimentateur (facilite grandement la collaboration), *reproductible* (vérification et collaboration), *répétable* (pour de nouvelles données ou expériences), et *ajustable* (pour de nouveaux scénarios). Quelqu'un d'autre peut jeter un oeil à l'analyse réalisée et voir exactement ce qui s'est produit, tant dans la gestion du jeu de données que l'analyse et la génération de graphiques. L'analyse peut être reproduite par les pairs, voire répéter si des données supplémentaires ont été recueillies ou si une nouvelle expérience a été réalisée.

Même si **R** est un langage de programmation et que cela peut en intimider plus d'un, il est relativement intuitif à apprendre et assez simple d'utilisation dans la mesure où les ressources appropriées pour apprendre sont accessibles à l'utilisateur. L'apprentissage de la programmation, même si ce n'est pas en **R,** permet de mieux comprendre le fonctionnement des ordinateurs en plus de reconsidérer le dogmatisme de tout-puissant "algorithmes". Il permettra aussi à l'utilisateur lorsqu'il saura suffisamment maîtriser la bête à créer lui-même la syntaxe qu'il lui permettra de résoudre les problèmes sur lesquels il s'intéresse.

Enfin, **R** possède un incroyable moteur pour la visualisation de données et de capacités graphiques. Aucun autre logiciel ne lui arrive à la cheville.

1.2 Installer R

R est compatible pour Windows, Mac et Linux. Pour télécharger le logiciel, il faut se rendre sur le site http://www.r-project.org et sélectionner les hyperliens "download R" ou "CRAN mirror". Il faut ensuite choisir un *mirror* de son pays d'origine. Par la suite, la page permet de choisir la version appropriée à son système d'exploitation. Il faut alors suivre les indications.

Il est principalement utilisé en anglais bien qu'on peut le définir en français. La plupart de l'aide sera également en anglais surtout celle pour les analyses avancées. Toutefois, il y a beaucoup de ressources accessibles en ligne en français en ce qui a attrait à l'initiation à **R**.

1.2.1 RStudio

Bien que **R** puisse être utilisé seul, son aspect rudimentaire pourra en inquiéter plus d'un (surtout un.e étudiant.e en sciences humaines et sociales). Le logiciel **R**Studio permet une utilisation plus fluide et intuitive pour les usagers ayant peu ou pas d'expériences en programmation. **R**Studio est un environnement de développement intégré (IDE pour *integrated development environment*) pour R. Il comprend une console, un éditeur de mise en évidence de la syntaxe qui prend en charge l'exécution directe du code, ainsi que des outils de traçage, d'historique, de débogage et de gestion de l'espace de travail. Le logiciel est gratuit et libre-accès pour une utilisation personnelle. Il comporte aussi une version commerciale qui nécessite un certain déboursement de fonds.

Pour télécharger le logiciel, il faut se rendre sur le site http:rstudio.com et naviguez jusqu'au téléchargement du logiciel, ou aller spécifiquement sur https://www.rstudio.com/products/rstudio/download/#download, où il sera possible de télécharger la version gratuite de **R**Studio. **R**Studio est uniquement disponible en anglais.

1.2.1.1 Les avantages de RStudio

L'avantage de **R**Studio, comparativement à l'utilisation unique de **R**, est d'offrir une gestion de la console, du script, de l'environnement des variables et des documents externes en une seule interface (et beaucoup d'autres avantages!). En plus de la console (ce que **R** fournit), **R**Studio permet l'édition de syntaxe qui pourra être commandée ligne par ligne en utilisant **CTRL + Enter** (Windows) ou **CMD + Enter** (MacOS), ce qui pourra s'avérer fort utile lors de la programmation de fonction ou d'analyses de données. **R**Studio affiche également les variables en mémoire dans le menu *Global Environment*, ce qui permet de suivre l'état de la programmation. Enfin, **R**Studio affiche dans un quatrième menu les fichiers dans le directoire des fichiers **R** ce qui permet de voir notamment les

jeux de données, mais aussi d'autres fonctions ou scripts. C'est également à cet endroit où l'aide (*help*) sera fournie et les figures (*plot*) affichées.

Tout le contenu du présent ouvrage pourra être réalisé avec **R** ou avec **R**Studio, toutefois l'usage de ce dernier sera plus agréable aux lecteurs et lectrices.

Oh! Une autre avantage de **R**studio est qu'il peut être utiliser comme un éditeur de texte. D'ailleurs, cet ouvrage est complètement rédigé avec **R**Studio (avec le package `Rmarkdown`).

1.2.2 Autres options

Il existe plusieurs interfaces utilisateurs graphiques (GUI pour *Graphical User Interface*) pour **R** comme *R Commander* (certainement l'option la plus connue en sciences humaines et sociales) ou *JASP*, mais aussi plusieurs autres. Ces deux options sont gratuites et libre-accès. Il existe aussi d'autres options payantes. Ces logiciels visent l'utilisation de **R** par une approche pointer-et-cliquer (*point-and-click*) au travers les analyses plutôt que de recourir à la syntaxe. Ces options plus intuitives pour l'utilisateur sans expérience en programmation auront parfois des effets limitatifs pour des analyses plus avancées et ont comme effet indésirable de promouvoir la boîte noire statistique.

1.3 Démarrer R ou RStudio

À l'ouverture de **R**, le logiciel présente la console, une interface très rudimentaire (et assez déstabilisante pour un logiciel pourtant si promu). Une fois le logiciel ouvert, l'application offre une invitation discrète à écrire des commandes. Le symbole `>` au bas de la console est une invite (*prompt*) indiquant où taper les commandes. C'est à cette ligne de commande que les expressions seront immédiatement évaluées.

Dans la Figure 1.1, **R** est défini en français. Cela n'a pour effet que de modifier le menu déroulant ("Fichier, Edition, etc.") au sommet du logiciel et les différentes options de ce menu. Le fonctionnement reste le même (les fonctions ne sont pas traduites, par exemple).

La console **R** n'étant pas un éditeur de texte, il faudra enregistrer la syntaxe utilisée lors d'une séance pour la conserver. Le logiciel offre une option d'écriture de script intégré, mais n'est pas lié directement à la console. Il faudra donc se résoudre à abuser du copier-coller ou à sourcer le script (tâche plus ardue pour les nouveaux utilisateurs). Plusieurs éditeurs de texte sont utiles ou même construits pour directement travailler avec **R**, le plus connu étant certainement **R**Studio. L'environnement intégré sera beaucoup plus fonctionnel.

La Figure 1.2, montre l'interface de **R**Studio, déjà un peu moins intimidante que celle de **R**. À l'ouverture de **R**Studio, quatre types de fenêtres sont disponibles

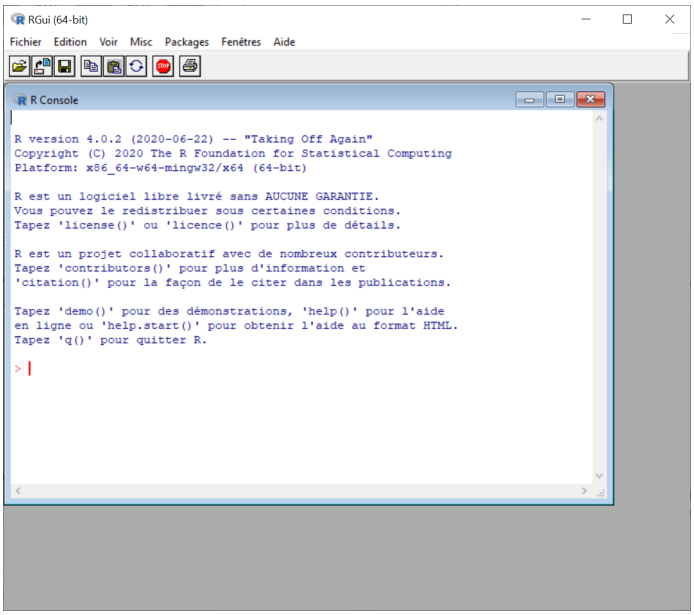

Figure 1.1: Ouverture (effrayante!) de la console R

: la console (cadran inférieur gauche), les scripts (cadran supérieur gauche), l'environnement (cadran supérieur droit) et l'affichage (cadran inférieur droit). L'emplacement de ces cadrans peut être modifié selon les désirs de l'utilisateur.

La console **R**Studio est identique à la console usuelle retrouvée avec **R**. Elle sert les mêmes fonctions. Le script est un éditeur de texte dans lequel de la syntaxe sera rédigée, sauvegardée, manipulée, et tester. S'il n'est pas ouvert ou s'il faut ouvrir un script supplémentaire, il faut procéder par le menu déroulant *File > New File > R Script*.

ou bien **CTRL + Shift + N**. Il peut y avoir plusieurs scripts ouverts simultanément. L'environnement global permet de connaître les variables et fonctions maison en mémoire vive. L'onglet *History* montre les dernières lignes de code commandées (non affichées dans la figure - il suffit de cliquer sur l'onglet à côté de *Environment*). Enfin, le cadran inférieur droit montre le fichier de travail **R** qui contiendra ordinairement les fichiers de travail actifs (scripts, jeu de données, fonctions maison, etc.) C'est très utile pour travailler par projet. Si aucun directoire n'est demandé explicitement par **R** (par exemple, si un jeu de donnée doit être téléchargé), le logiciel cherchera par défaut dans le fichier actif pour télécharger les fichiers demandés. Il est une bonne pratique que de s'assurer d'être dans le bon fichier, car cela pourra causer quelques soucis à l'occasion.

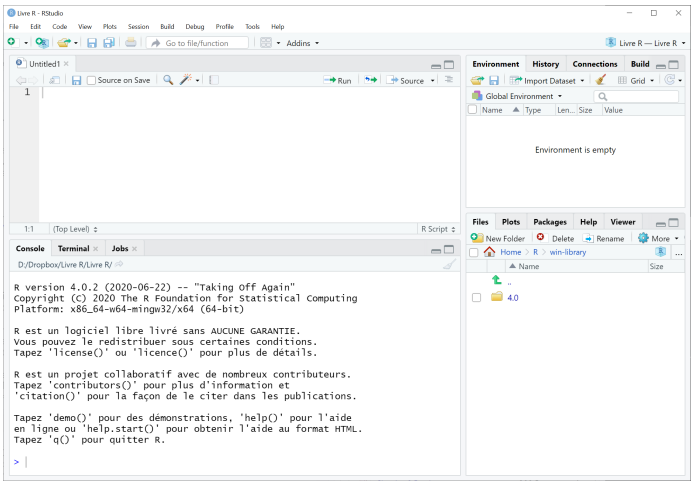

Figure 1.2: Ouverture (moins effrayante) de **R**Studio

1.4 Les scripts

Ce qu'il importe le plus avec **R**, et ce qui fait resplendir **R**Studio, est de conserver la syntaxe d'une session à l'autre. Le logiciel ne le fait pas très bien. Il faudra sauvegarder dans un script les expressions et le code utilisés. Ces fichiers ont souvent comme extension ".R" et permettront de conserver, voire partager la syntaxe. Il sera possible d'y ajouter des commentaires pour de futures utilisations. Tout éditeur de texte peut permettre la sauvegarde de syntaxe, certains seront mieux que d'autres pour l'utilisation avec **R**.

RStudio contient déjà un panneau contenant un script qu'il sera possible de sauvegarder et de rouler directement dans la console. Ce dernier est directement lié et il est possible de rouler la syntaxe ligne par ligne avec **CTRL + Enter** (Windows) ou **CMD + Enter** (MacOS).

Chapter 2

Programmer

Dans les prochaines sections, les différents éléments de programmation permettant la création et la manipulation de données seront présentés.

2.1 Les variables

Pour manipuler les données, il faut recourir à des variables. Afin de leur attribuer une valeur, il faut assigner cette valeur avec `<-` (**ALT** + `-`) ou `=`, par exemple,

```
a <- 2
a
#> [1] 2
```

où `a` est maintenant égale à `2`. La première ligne assigne la valeur à `a`. La deuxième ligne, indique à la console **R** d'imprimer le résultat pour le voir. Par la suite, `a` pourra être utilisée dans des fonctions, des calculs ou analyses plus complexes. De surcroît, `a` pourra devenir une fonction, une chaîne de caractère (*string*) ou un jeu de données.

Conventionnellement, les puristes de **R** recommanderont l'usage de `<-` plutôt que `=` pour l'assignation. Il y a quelques nuances computationnelles entre les deux, mais qui échapperont irrémédiablement aux néophytes et même aux usagers intermédiaires.

Pour nommer des variables, seuls les caractères alphanumériques peuvent être utilisés ainsi que le tiret bas `_` et le `.`. Les variables ne peuvent commencer par un nombre.

Réassigner une valeur à une variable déjà existante écrase la valeur précédente.

```
a <- 2
a <- 3
a
#> [1] 3
```

La sortie produit `3` et non plus `2`.

Cette remarque est importante, car elle signifie que l'on peut écraser des fonctions en nommant des variables. Il faut ainsi éviter de nommer des variables avec des fonctions utilisées par **R**, on évitera notamment l'utilisation des noms suivants.

```
c; q; t; C; D; I; T; F; pi; mean; var; sd; length; diff; rep;
```

Certains mots seront tout simplement interdits d'utilisation.

```
TRUE; FALSE; break; for; in; if; else; while; function; Inf; NA; NaN; NULL;
```

2.2 Les opérateurs arithmétiques

La première utilisation qu'un nouvel usager fait de **R** est généralement d'y recourir comme calculatrice. On pourra utiliser les opérateurs arithmétiques de base comme l'addition `+`, la soustraction `-`, la multiplication `*`, la division `/` , et l'exposant `^`.

```
2 + 2
#> [1] 4
1 / 3
#> [1] 0.333
2 * 3 + 2 ^ 2
#> [1] 10
```

Évidemment, ces opérateurs fonctionnent sur des variables numériques.

```
a <- 1
b <- 10
a / b
#> [1] 0.1
```

Ici, les deux premières lignes assignent des valeurs à `a` et `b`, puis imprime la division. L'absence de marqueur `<-` ou `=` indique à **R** d'imprimer la réponse dans la console. Si le résultat `a/b` devait être assigné à une variable, alors aucun résultat ne serait affiché, bien que la variable contienne la réponse.

```
resultat <- a / b
```

Il n'y a aucune réponse d'affichée. Maintenant, si `resultat` est demandé, R affiche le résultat.

```
resultat
#> [1] 0.1
```

D'autres fonctions sont aussi très utiles. Par exemple, la racine carrée `sqrt()` (qui n'est rien d'autre que `^(1/2)`) et le logarithme naturel `log()`. Il suffit d'insérer une variable ou une valeur à l'intérieur d'une de ces fonctions pour en obtenir le résultat.

```
sqrt(4)
#> [1] 2
4^(1/2)
#> [1] 2
log(4)
#> [1] 1.39
```

2.3 Les commentaires

Les scripts **R** peuvent contenir des commentaires. Ceux-ci sont désignés par le désormais célèbre `#`. Une ligne de script commençant par ce symbole sera ignorée par la console. Ces commentaires permettent aussi bien de préciser différentes étapes d'un script, que d'expliquer la nomenclature des variables ou encore d'expliquer une fonction, ses entrées, ses sorties. Les commentaires sont extrêmement utiles, car les annotations peuvent souvent sauver énormément de temps et d'effort lors d'utilisations ultérieures.

```
# La variable resultat est le quotient des variables a et b
resultat <- a / b
resultat
#> [1] 0.1
```

Dans cet exemple, la première ligne est ignorée. Autrement, la console **R** produirait une erreur, car cette ligne est pour le logiciel pur charabia!

2.4 Définir une chaîne de caractère

La plupart du temps, les variables utilisées seront numériques, c'est-à-dire qu'elles contiendront des nombres. Parfois en analyses de données, il pourra

s'agir de chaîne de caractères (*string*), autrement dit, de mots. Les chaînes de caractères sont définis par le double apostrophe `"..."`, où on remplace les trois points par les mots désirés.

```
titre <- "Bonjour tout le monde!"
titre
#> [1] "Bonjour tout le monde!"
```

2.5 Concaténer

Par défaut, **R** ne peut qu'assigner qu'une valeur à une variable. Pour grouper des éléments ensembles, c'est-à-dire, pour créer des jeux de données, des vecteurs, des matrices, des listes, il faudra utiliser des fonction de concaténation, dont voici une liste des plus utiles avec quelques exemples, de la plus stricte (vecteur) à la plus flexible (liste).

2.5.1 Création d'un vecteur

Une fonction fort utile permettra de joindre des valeurs dans une seule variable. Précédemment, l'assignation d'une valeur a des variables se limitait qu'à une chaîne de caractères ou une valeur numérique. La fonction **c**oncaténer `c()` (ou **c**ombiner, **c**réer) met plusieurs éléments (deux ou plus) dans une seule variable. Chaque élément est délimité par une virgule `,`.

```
valeurs <- c(-5, 5)
valeurs
#> [1] -5  5
```

Elle fonctionne également avec les chaînes de caractères.

```
texte <- c("Bonjour", "tout", "le", "monde")
texte
#> [1] "Bonjour" "tout"    "le"      "monde"
```

Et les deux.

```
phrase <- c(1, "Chat", 2, "Souris")
phrase
#> [1] "1"      "Chat"   "2"      "Souris"
```

La fonction `c()` est strict sur les arguments, car elles leur accord le même attribut. Par exemple, `phrase` ne contient que des chaînes de caractères. Les valeurs `1` et `2` ont perdu leur classe de numérique (elles ne sont plus utilisables comme nombre - pour l'instant).

Il faudra également faire attention à ce qui est passé comme argument à la fonction `c()`, car elle vectorise les arguments. Autrement dit, elle crée des vecteurs (une ligne en quelque sorte) avec les entrées fournies, peu importe leur structure de départ. Par exemple, un jeu de données passant par `c()` devient une seule ligne de valeurs. Les fonctions `cbind()` et `rbind()` permettront de joindre des colonnes et des lignes, respectivement.

2.5.2 Création d'une matrice

La fonction `matrix()` sera utile pour créer des matrices, commes des matrices de covariances, par exemple. La fonction utilise trois arguments, une matrice de nombre à entrer dans la matrice, un nombre de colonnes et un nombre de lignes. La fonction utilise le recyclage, ce qui pourra être utile à certaines occasions.

```
# Une matrice de 0
matrix(0, ncol = 3, nrow = 3)
#>      [,1] [,2] [,3]
#> [1,]    0    0    0
#> [2,]    0    0    0
#> [3,]    0    0    0

# Une matrice contenant les nombres 1:3 pour une matrice 3x3
matrix(1:3, ncol = 3, nrow = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    1    1
#> [2,]    2    2    2
#> [3,]    3    3    3

# Si la séquence préférée est de gauche à droite plutôt
# que de bas en haut
matrix(1:3, ncol = 3, nrow = 3, byrow = TRUE)
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    1    2    3
#> [3,]    1    2    3

# Une matrice avec un nombre d'entrées égale au nombre de cellules
matrix(1:16, ncol = 4, nrow = 4)
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    5    9   13
```

```
#> [2,]    2    6   10   14
#> [3,]    3    7   11   15
#> [4,]    4    8   12   16
```

Les matrices sont une formes de jeu données dans lequel tous les éléments partagent le même attribut (tous numériques, caractères, logiques, etc.).

2.5.3 Création d'un jeu de données

Un jeu de données (`data.frame`) est un peu comme l'extension de la matrice. La différence étant que les éléments entre les colonnes peuvent partager des attributs différents. Ainsi chaque ligne représente une unité (un participant, un objet) et chaque colonne représente une dimension (informations ou variable) différente de cette objectif. La fonction `data.frame()` permet de créer de tel objet. La fonction prend comme un argument une série de vecteurs. Si un nom

```
# Quelques variables
var1 <- c("Éloi", "Laurence")
var2 <- c(6, 3)
var3 <- c(TRUE, TRUE)

# Entrer de trois vecteurs non identifiés
jd1 <- data.frame(var1, var2, var3)

# Entrer de trois vecteurs identifiés
jd2 <- data.frame(nom = var1, age = var2, enfant = var3)

# Comparer
jd1 ; jd2
#>       var1 var2 var3
#> 1     Éloi    6 TRUE
#> 2 Laurence    3 TRUE
#>        nom age enfant
#> 1     Éloi   6   TRUE
#> 2 Laurence   3   TRUE
```

En utilisant `nom.de.variable = vecteur` à l'intérieur de `data.frame()`, les noms des colonnes deviennent `nom.de.variable`. Cela permettra une plus grande flexibilité lorsqu'il faudra [gèrer] et manipuler les données.

Comme les matrices, les jeux de données ont aussi une restriction. Alors que les jeux de données libérait l'utilisateur de la contrainte d'avoir des objets de même attributs, les jeux de données doivent être créés avec des vecteurs de même longueur. Autrement dit, chaque colonne doit avoir exactement le même nombre de lignes. Parfois, **R** procédera par recyclage pour combler les éléments.

Il faudra donc porter une attention particulière, pour vérifier si c'est l'intention ou non.

2.5.4 Création d'une liste

Une troisième option pour stocker de informations dans une seule variable est d'avoir recourt au liste. La liste libère à la fois l'utilisateur des objets de mêmes attributs et de même longueur. Ainsi, une liste, peut contenir des vecteurs, des matrices, des jeux de données et même d'autres listes.

Pour créer une liste, il faut utiliser la fonction `list()`. Comme `data.frame()`, des noms d'éléments peuvent être donner pour chaque liste pour faciliter la manipulation ultérieure de la liste.

```
#Quelques variables
var1 <- c("chat","chien")
var2 <- 1:10

# Entrer de deux vecteurs non identifiés
jd1 <- list(var1, var2)

# Entrer de deux vecteurs identifiés
jd2 <- list(animal = var1, nombre = var2)

# Comparer
jd1 ; jd2
#> [[1]]
#> [1] "chat"  "chien"
#>
#> [[2]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> $animal
#> [1] "chat"  "chien"
#>
#> $nombre
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

L'utilisation de listes est une caractéristique prédominante avec **R**. Par exemple, **R** ne peut sortir qu'une variable par fonction. Si la fonction doit retourner plusieurs éléments, ceux-ci devront se retrouver dans une liste. Ce qui sera plus nébuleux pour le lecteur, c'est que l'optimisation de **R** se fait par les listes. Cela sera noté aux moments appropriés.

2.6 Référer à des sous-éléments

Avec des variables contenant plusieurs valeurs, il peut être utile de référer à une seule valeur ou un ensemble de valeurs de la variable. Les crochets `[]` à la suite du nom d'une variable permettront d'en extraire les valeurs désirées sans tout sortir l'ensemble.

```
# Un exemple de vecteur
phrase <- c(1, "Chat", 2, "Souris")

# Extraire le premier élément de la variable phrase
phrase[1]
#> [1] "1"

# Extraire les éléments 1, 2 et 3
phrase[1:3]
#> [1] "1"    "Chat" "2"

# Extraire les éléments 2 et 4
phrase[c(2,4)]
#> [1] "Chat"   "Souris"

# Ne pas extraire l'élément 1
phrase[-1]
#> [1] "Chat"   "2"      "Souris"

# Ne pas extraire les éléments 1 et 3
phrase[-c(1, 3)]
#> [1] "Chat"   "Souris"
```

Dans le premier exemple, seul un élément est demandé. Dans le deuxième exemple, la commande `1:3` produit la série de $1, 2, 3$ et en extrait ces nombres. Dans le dernier exemple, la fonction `c()` est astucieusement utilisée pour extraire les éléments 2 et 4. Le quatrième exemple montre comment retirer un élément en utilisant des valeurs négatives et le cinquième exemple montre comment retirer des éléments.

La section Manipulation de données montrera davantage comment référer à des sous-éléments de jeux de données, de matrices et de listes.

2.7 Les fonctions

R offre une multitude de fonctions et permet également à l'usager de bâtir ses propres fonctions (fonctions maison). Elles permettent d'automatiser des calculs

(généralement, mais peut faire beaucoup plus!). Tout au long de cet ouvrage, les fonctions seront identifiées par l'ajout de parenthèse à leur fin, comme ceci : `function()`. Ces fonctions ont généralement la forme suivante.

```
nom <- function(argument1, argument2, ...) {
  # Calcul à réaliser
}
```

Ici, `nom` est le nom auquel la fonction sera référée par la suite, `function` est la fonction **R** qui permet de créer la fonction maison, `argument1` et `argument2` sont les arguments (les entrées) fournis à la fonction et à partir desquels les calculs seront réalisés, et les accolades `{}` définissent le début et la fin de la fonction dans le script.

Il sera bien utile de créer ses propres fonctions bien que **R** possède une pléthore de fonctions et de packages en contenant encore plus. Toutes les fonctions, qu'elles soient maisons ou déjà intégrées, respectent le même fonctionnement, ce pour quoi il est utile de s'y pencher. Les fonctions maison permettront d'automatiser certains calculs qui seront propres à résoudre les problèmes de l'usager et d'être réutilisé ultérieurement.

Voici un exemple trivial de fonction. Ici la somme de deux nombres.

```
addition <- function(a, b) {
  a + b
}
addition(2,3)
#> [1] 5
```

Par défaut, une fonction retourne la dernière ligne calculée si elle n'est pas assignée à une variable. Si le résultat d'une fonction est assigné, la fonction ne retourne pas le résultat dans la console, mais assigne bel et bien la variable.

```
addition2 <- function(a, b) {
  # Le résultat est assigné à une variable
  somme <- a + b
}

# Ne produit pas de sortie
addition2(100, 241)

# Comme il y a assignation, total n'est pas affichée
total <- addition2(100, 241)

# En roulant total, la sortie affiche bien la sortie de addition2()
total
#> [1] 341
```

Afin d'éviter ces problèmes ou s'il fallait retourner plusieurs arguments (ce qui sera souvent le cas!), il faudrait utiliser la fonction `return()` à la fin de la fonction.

```
addition3 <- function(a, b) {
  # Le résultat est assigné à une variable
  somme <- a + b
  return(somme)
}

# Les deux fonctions produisent une sortie
addition3(4, 6)
#> [1] 10
total <- addition3(4, 6)
total
#> [1] 10
```

2.8 Définir une boucle

Pour automatiser certains calculer, il peut être utile de recourir à une boucle (*loop*) qui pourra répéter plusieurs fois une même opération. Voici l'anatomie d'une boucle.

```
for(i in vec){
  # Calcul désiré
}
```

L'élément `for` est la fonction déclarant la boucle. Les renseignements sur les itérations se retrouvent entre les parenthèses. La variable `i` prendra successivement tous les éléments dans (`in`) le vecteur à gauche (`vec`). Tout le contenu de la boucle (ce qui sera répété) se retrouve entre les accolades `{}`, c'est ce qui sera produit à chaque boucle. Dans cet exemple, la boucle se répète k fois, soit de $1, 2, 3, ..., k$, à cause de l'expression `1:k` qui correspond à générer un vecteur de 1 à k. La variable `i` quant à elle change de valeur à chaque itération. Elle prendra tour à tour ces valeurs chaque itération $1, 2, 3, ..., k$. La variable pourra judicieusement être utilisée dans la boucle afin de profiter ce comportement, notamment pour le classement des résultats. Lorsque la boucle atteint k, elle se termine.

Il est aussi possible de rédiger la boucle en utilisant uniquement `k`. Alors, `i` prendra toutes les valeurs contenues dans `k`. La longueur du vecteur `k` définit le nombre d'itérations.

```
for(i in k){
  # Calcul désiré
}
```

2.9 Les clauses conditionnelles

Pour réaliser des opérations sous certaines conditions ou opérer des décisions automatiques, il est possible d'utiliser des arguments conditionnels avec des opérateurs logiques. Par exemple, sélectionner des unités ayant certaines caractéristiques, comme les participants ayant 18 ans et moins, les personnes ayant un trouble du spectre de l'autiste, ou encore par sexe. Il est aussi possible d'utiliser les opérateurs pour définir à quelle condition telle ou telle autre fonction doit être utilisée. Il faudra alors utiliser les arguments logiques.

Table 2.1: Symboles logiques et leur signification

Symbole	Signification
==	est égale à
!=	n'est pas égale à
<	plus petit que
>	plus grand que
<=	plus petit ou égale
>=	plus grand ou égale
&&	et
||	ou

R teste si les valeurs de la variable correspondent à l'opérateur logique en les déclarant comme vraies (`TRUE`) ou fausses (`FALSE`).

```
valeurs <- 1:6
# Toutes les valeurs plus grandes que 3.
valeurs > 3
#> [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Cela peut être utilisé pour référer à des sous-éléments comme abordés précédemment.

```
# Toutes les valeurs plus grandes que 3.
valeurs[valeurs > 3]
#> [1] 4 5 6
```

Ici, toutes les valeurs vraies de l'opérateur logique sont rapportées.

Les opérateurs logiques servent également à définir des opérations conditionnelles. La fonction `if` sera alors utilisée. Il y a trois principales formes : `if` (Si ceci, alors cela), le `if else` (Si ceci, alors cela, sinon autre chose) et les échelles `if else if else`.

```
if(x){
  # Opération désirée
}
```

L'anatomie d'une fonction `if` comporte d'abord la fonction `if`. L'argument entre parenthèses à sa plus simple expression doit être vérifié par vrai (`TRUE`) ou faux (`FALSE`). Si l'argument est vrai, alors le calcul désiré est réalisé, autrement le logiciel ignore le calcul de la fonction entre accolades `{}`.

```
x <- -2
if(x < 0){
  print("la valeur est négative")
}
#> [1] "la valeur est négative"
```

Il est possible d'élaborer cette logique avec la fonction `else` qui permet de spécifier une suite à la fonction si l'argument est faux (`FALSE`).

```
x <- 2
if(x < 0){
  print("la valeur est négative")
}else{
  print("la valeur est positive")
}
#> [1] "la valeur est positive"
```

Enfin, il est possible d'élaborer un arbre de décision avec toute une échelle de conditionnels.

```
x <- 0
if(x < 0){
  print("la valeur est négative")
}else if(x > 0){
  print("la valeur est positive")
}else{
  print("la valeur est égale à 0")
}
#> [1] "la valeur est égale à 0"
```

L'arbre de décision peut devenir aussi compliqué que l'utilisateur le désire : chacune des branches peut contenir autant de ramifications que nécessaire.

Il peut arriver pour certaines fonctions de devoir spécifier si certains paramètres sont vrais (`TRUE`) ou faux (`FALSE`) ou de définir des variables ayant ces valeurs. Lorsque c'est le cas, il est toujours recommandé d'écrire les valeurs logiques tout au long comme `TRUE` et `FALSE`, même si **R** reconnaît `T` et `F`, car ces dernières peuvent être réassignées, contrairement aux premières.

2.10 Les packages

L'utilisation de packages (souvent nommées bibliothèques, modules, paquets ou paquetage en français - ici, l'usage de *package* sera maintenu) est l'attrait principal de **R**. Pour éviter l'anglicisme, Antidote suggère *forfait*, *achat groupé* ou *progiciel* (ce dernier étant certainement le terme approprié).

Les packages sont de regroupement de fonctions. C'est certainement l'aspect qui a le plus contribué au succès et à sa dissémination de **R**. Il s'agit de la mise en commun d'un effort collaboratif afin de créer des fonctions et de les partager librement entre les usagers. Le téléchargement de base de **R** offre déjà quelques packages rudimentaires (comme `base` qui offre des fonctions comme `sum()` ou `stat` qui offre des fonctions comme `mean()` et `var()`), mais qui suffisent rarement lorsque des analyses plus avancées ou plus spécialisées sont nécessaires.

L'une des forces des packages est qu'ils sont fournis généralement avec un bon manuel d'utilisation. Plusieurs contributeurs leur sont associés (avec un responsable). Ils sont maintenus régulièrement. Le soutien des responsables est parfois aisé à obtenir et les auteurs de ces packages sont motivés à maintenir les packages opérationnels et aux bénéfices de tous. La faiblesse des packages est qu'il s'agit malheureusement de *généralement*. Il arrive que certains packages produisent des erreurs de calcul, soient laissé en désuétude par leurs auteurs, que le package ait migré sous une autre forme, que de meilleures options soient disponibles sans aucune notice à cet effet. Cela va sans dire, ce problème concerne les logiciels traditionnels également. Il s'agit toutefois d'un enjeu moindre, car les packages sont souvent recommandés par des collègues, des autorités dans leur domaine respectif ou des ouvrages de référence, ce qui aura comme tendance de promouvoir les meilleurs packages. Pas toujours. Il faut rester critique et ne pas de laisser tromper par une boîte noire.

Une dernière faiblesse : les packages agissent parfois en boîte noire, c'est-à-dire qu'ils court-circuitent la réflexion de l'utilisateur qui leur fait confiance. Il peut être parfois difficile de savoir ce que les fonctions produisent exactement. Au contraire des logiciels traditionnels, ces boîtes noires peuvent dans la plupart des cas être accessibles directement, elles sont liés en plus à des articles scientifiques ou de la documentation qui permet dans comprendre les tenants et aboutissants.

2.10.1 Installer des packages

Pour installer un package, il faut utiliser la fonction

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

où les `"..."` doivent être remplacé par le nom du package. Il est important de bien inscrire le nom du package entre guillemet anglophone. Il est aussi possible de sélectionner

> Tools; Install Packages…

puis de nommer le package sous l'onglet package. Avec **R** il faudra auparavant choisir un miroir (sélectionner un pays), ce qui ne sera pas nécessaire avec **R**Studio. Une fois téléchargé, il ne sera plus nécessaire de refaire cette étape à nouveau, à l'exception de potentielles et ultérieures mises à jour lorsqu'elles devront être réalisées.

2.10.2 Appeler un package

Ce qui n'est pas des plus intuitif avec **R**, c'est qu'une fois le package téléchargé, il n'est pas directement utilisable. Il faut d'abord l'appeler avec la fonction `library()`.

```
library("...")
```

Cette étape doit être faite à chaque ouverture de **R**. Cela permet de ne pas mettre en mémoire trop de package simultanément. Il sera ainsi important d'indiquer tous les packages utilisés en début de script sans quoi des erreurs comme l'absence de fonctions seront produites.

Une technique à laquelle l'utilisateur peut avoir recourt lorsqu'il souhaite n'utiliser qu'une fonction spécifique d'un package est l'utilisation des `::` débutant par le nom du package suivi par le nom de la fonction, comme `MASS::mvrnorm()`. La fonction s'utilise de façon usuelle. En utilisant `::`, il n'est pas nécessaire d'appeler le package avec la fonction `library()`. Il faut toute fois que le package soit bel et bien installer.

2.11 Obtenir de l'aide

En utilisant `help(nom)` ou `?nom`, où il faut remplacer `nom` par le nom d'une fonction ou d'un package, **R** offre de la documentation. Les fonctions d'aide

retournent une page de documentation contenant généralement de l'information sur les entrées et les sorties des fonctions. Certaines sont mieux détaillées que d'autres, tout dépendant de leurs créateurs et des personnes qui maintiennent ces fonctions.

```
# Obtenir de l'aide pour la fonction help()
?help
```

Il existe également la fonction `??nom` qui produit une liste de toutes fonctions **R** ayant partiellement l'inscription introduite à la place de `nom`. Aussi, `example(nom)` produit un exemple d'une fonction.

2.12 En cas de pépins

Il arrive parfois que le code utilisé ne fonctionne pas, que des erreurs se produisent ou que des fonctions fort utiles demeurent inconnues. Même après plusieurs années d'utilisation, les utilisateurs font encore quotidiennement des erreurs (au moins une!). Un excellent outil est d'utiliser un moteur de recherche dans un fureteur de prédilection, de poser une question à l'aide de quelques mots clés bien choisis, préférablement en anglais, et en y inscrivant "with R" ou "in R" ou "R". La plupart du temps, les programmeurs de packages auront une solution sur leur site ou leurs instructions de packages. Il y a aussi des plateformes publiques et en ligne, comme StackOverflow qui collectent questions et réponses sur le codage. D'autres utilisateurs peuvent avoir posé la même question et des auteurs de programmes R et d'autres usagers y auront répondu aux bénéfices de tous. Dans le cas d'une solution introuvable, ces mêmes plateformes permettent de poser de nouvelles questions. Il faudra toutefois attendre qu'un usager plus expérimenté prenne le temps d'y répondre.

Chapter 3

Calculer

Dans cette section, les fonctions essentielles couramment utilisées sont présentées en rafale. L'accent est mis sur la définition de la fonction (qu'est-ce qu'elle fait?) et son utilité (à quoi sert-elle?). Pour les fonctions essentielles de nature statistiques (moyennes, médianes, etc.), cette section développe une fonction maison (rédigée par l'utilisateur pour la mettre en pratique) et identifie la fonction déjà implantée en **R**.

3.1 La longueur

La longueur d'une variable correspond au nombre d'éléments qu'elle contient. La fonction `length()` permettra d'obtenir ce résultat. Ce sera particulièrement utile lorsqu'il faudra calculer, par exemple, le nombre de boucle à réaliser à partir des éléments d'un vecteur ou la taille d'échantillon (le nombre d'unités d'observation d'une variable), bien que `ncol()` (nombre de colonnes) et `nrow()` (nombre de lignes) soient plus intuitives pour les matrices et les jeux de données.

La somme d'une chaîne de caractères est toujours de 1, peu importe le nombre de caractères. La fonction `nchar()` produira le nombre de caractères.

Une variable qui existe, mais qui ne contient pas de valeur aura une longueur égale 0. Ce type de variable est utile lorsqu'il faut créer une variable qui aura une taille changeant d'une situation à l'autre.

```
x <- c(1, 2, 3)
length(x)
#> [1] 3

y <- "Bonjour tout le monde!"
length(y)
```

```
#> [1] 1

nchar(y)
#> [1] 22
```

3.2 La répétion

La fonction `rep()` sera utile pour répéter volontairement des valeurs. Il y deux possibilité de répétitions: l'argument `times` définit le nombre de fois que le vecteur est répété; l'argument `each` définit le nombre de fois que chaque élément est répété.

```
vec <- c(2, 4, "chat")

# Répéter vec trois fois
rep(vec, times = 3)
#> [1] "2"    "4"    "chat" "2"    "4"    "chat" "2"    "4"
#> [9] "chat"

# Répéter chaque éléments de vec trois fois
rep(vec, each = 3)
#> [1] "2"    "2"    "2"    "4"    "4"    "4"    "chat" "chat"
#> [9] "chat"
```

3.3 La séquence

La fonction `seq()` permet de générer une séquence régulière de valeurs. Les arguments sont `seq(from = , to = , by = )` traduisible par `de` , `à`, `par`. Les arguments par défaut seront très utiles pour simplifier l'écriture; La fonction commence ou termine la séquence par 1 et fera des bonds de 1 entre les valeurs. Un autre argument est la longueur de la sortie `length.out` qui spécifie le nombre d'éléments que devra comporter le vecteur de sortie.

```
# Une séquence de 1 (défaut, from = 1) à 10
seq(10)
#>  [1]  1  2  3  4  5  6  7  8  9 10

# Une séquence de 1 (défaut, from = 1) à -10
seq(-10)
#>  [1]   1   0  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10

# Une séquence de -10 (défaut, from = 1) à 1
```

```
seq(from = -10, to = 1)
#>  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1

# Une séquence de nombres paires (from = 2, to = 10, by = 2)
seq(2, 10, 2)
#> [1]  2  4  6  8 10

# Une séquence de nombres paires
seq(from = 2, by = 2, length.out = 5)
#> [1]  2  4  6  8 10
```

3.4 La somme

Il est possible de calculer des sommes de variables pour en obtenir le total. En tant qu'humain, le calcul d'une série de nombre correspond à prendre chaque nombre et de les additionner un à un. La fonction suivante reproduit assez bien ce qu'un humain ferait (avec ses quelques caprices de programmation tel que devoir déclarer l'existence de la variable de `total` et spécifier le nombre d'éléments à calculer).

```
somme <- function(x){
  n <- length(x)
  total <- 0
  for(i in 1:n){
    # Prendre le ie élément et l'additionner
    # au total des (i-1)e éléments précédents
    total <- total + x[i]
  }
  return(total)
}
x <- c(1,2,3,4,5,-6)
somme(x)
#> [1] 9
sum(x)
#> [1] 9
```

À noter que l'utilisation de la boucle est à des fins illustratives seulement. En termes de rendement computationnel, elle est bien peu efficace. Il faudra privilégier la fonction `sum()` pour calculer le total de son entrée.

Il faut prendre garde : **R** calcule le total de tous les éléments de l'entrée sans égard aux lignes et aux colonnes. Autrement dit, il vectorise les entrées. Si deux variables étaient entrées par inadvertance, alors R calculerait la somme de ces deux variables plutôt que de retourner deux totaux. À cette fin, les fonctions

`rowSums()` et `colSums()` seront utiles lorsqu'il faudra calculer des sommes sur des lignes (*row*) ou des colonnes (*col*).

3.5 La moyenne

La moyenne est une mesure de tendance centrale qui représente le centre d'équilibre d'une distribution (un centre de gravité en quelque sorte). Si le poids d'un des côtés d'une distribution de probabilité était altéré (plus lourde ou plus légère), alors la moyenne se déplacerait en conséquence.

La moyenne d'un échantillon correspond à la somme de toutes les unités d'une variable divisée par le nombre de données de cette variable ou, mathématiquement,

$$\bar{x} = \frac{\Sigma_{i=1}^{n} x}{n}$$

où x est la variable, n est le nombre d'unité et Σ_i^n représente la somme de toutes les unités de x. **R** possède déjà une fonction permettant de calculer la moyenne sans effort, `mean()` où l'argument est la variable. Il est possible de développer une fonction maison pour calculer la moyenne comme

```
x_bar <- sum(x)/length(x)
```

où `sum(x)` calculer la somme de toutes les unités de `x`, `/` permet la division et `length(x)` calcule le nombre d'unités du vecteur `x`. Par exemple, à partir d'une variable `x`, les fonctions suivantes donnent le même résultat. Par contre la fonction `mean()` est beaucoup plus robuste que cette dernière équation.

```
# Création de la variable
x <- c(0, 1, 2, 3, 4, 5)

# La moyenne
mean(x)
#> [1] 2.5

# La moyenne
sum(x)/length(x)
#> [1] 2.5
```

Comme pour `sum()`, les fonctions `rowMeans()` et `colMeans()` seront utiles lorsqu'il faudra calculer des moyennes sur des lignes (*row*) ou des colonnes (*col*).

3.6 La médiane

La médiane d'un échantillon correspond à la valeur où 50% des données se situe au-dessous et au-dessus de cette valeur. C'est la valeur au centre des autres (lorsqu'elles sont ordonnées). Quand le nombre de données est impair, le $\frac{(n+1)}{2}$e élément est la médiane. Quand le nombre est pair, la moyenne des deux valeurs au centre correspond à la médiane. Cette statistique est intéressante comme mesure de tendance centrale, car elle est plus robuste aux valeurs aberrantes (moins sensibles) que la moyenne.

Évidemment, **R** offre déjà une fonction `median()` pour réaliser le calcul. Il est toutefois possible de programmer une fonction maison. Il faut utiliser la fonction `sort()` pour ordonner les données (croissant par défaut).

```
mediane <- function(x) {
  n <- length(x)
  s <- sort(x)
  ifelse(n%%2 == 1, s[(n + 1) / 2], mean(s[n / 2 + 0:1]))
}

# Tester ensuite:
x <- c(42, 23, 53, 77, 93, 20, 37, 24, 60, 62)
median(x)
#> [1] 47.5
mediane(x)
#> [1] 47.5
```

L'expression `n%%2`, lue n mod 2, joue astucieusement le rôle de vérifier si `n` est impaire. La formule générale x mod y représente une opération binaire associant à deux entiers naturels le reste de la division du premier par le second. Par exemple, 60 mod 7, noter `60%%7` dans **R**, donne 4 soit le reste de $7 * 8 + 4 = 60$. Le logiciel le confirme.

```
60%%7
#> [1] 4
```

Il s'agit d'une technique de programmation très pratique. Dans le cas de `n%%2`, la formule donne 1 dans le cas d'un nombre impair ou 0 dans le cas d'un nombre pair, puis teste ce résultat pour déterminer s'il réalise `s[(n+1)/2]` lorsque `n%%2==1`(`TRUE`) , ce qui correspond à choisir l'élément au centre d'un vecteur de taille impair, ou bien `mean(s[n/2+0:1]` lorsque `n%%2==0`(`FALSE`) , ce qui correspond à choisir les deux éléments au centre d'un vecteur pair et d'en faire la moyenne. Il s'agit de l'une des nombreuses façons selon lesquelles il est possible de programmer la médiane.

3.7 La variance

La variance d'un échantillon est une mesure de dispersion. Elle représente la somme des écarts (distances) par rapport à la moyenne au carré divisée par la taille d'échantillon moins 1. Mathématiquement, il s'agit de l'équation (3.7).

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Il est assez aisé d'élaborer une fonction pour réaliser se calculer avec les fonctions déjà abordées.

```
variance <- function(x){
  n <- length(x)
  xbar <- mean(x)
  variance <- sum((x - xbar) ^ 2)/(n - 1)
  return(variance)
}
```

La variance peut aussi être calculée plus efficacement avec la fonction **R** `var()`.

```
x <- c(26, 6, 40, 36, 14, 3, 21, 48, 43, 2)
variance(x)
#> [1] 300
var(x)
#> [1] 300
```

3.8 L'écart type

L'écart type d'un échantillon représente la racine carrée de la variance. Elle a une interprétation plus intuitive en tant que mesure de la moyenne des écarts par rapport à la moyenne. Si le calcul avait été entrepris avec les distances par rapport à la moyenne (au lieu des écarts au carré), alors la somme serait toujours de 0, un résultat tout à fait bancal. En prenant la racine carrée des écarts au carré, ce qui constitue une mesure de distance euclidienne, l'écart type devient une mesure de l'étalement de la dispersion autour du centre d'équilibre.

$$s = \sqrt{s^2} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Avec **R**, la fonction de base est `sd()`. Il est possible de récupérer la fonction maison précédemment rédigée.

```
ecart.type <- function(x){
  et <- sqrt(variance(x))
  return(et)
}
ecart.type(x)
#> [1] 17.3
sd(x)
#> [1] 17.3
```

3.9 Les graines

Par souci de reproductibilité, il est possible de déclarer une valeur de départ aux variables pseudoaléatoires, ce que l'on nomme une graine ou *seed* en anglais. Cela permet de toujours d'obtenir les mêmes valeurs à plusieurs reprises, ce qui est très utile lors d'élaboration de simulations complexes ou lorsque des étudiants essaient de répliquer résultat tiré d'un ouvrage pédagogique.

```
set.seed("nombre")
```

Il suffit de spécifier cette commande (en remplaçant `nombre` par un nombre) en début de syntaxe pour définir la séquence de nombre. Cette fonction sera utilisée à plusieurs reprises dans le but de reproduire les mêmes sorties.

Cette fonction est présentée, car elle reviendra régulièrement dans ce livre pour qu'il soit possible de reproduire et obtenir exactement les mêmes résultats.

3.10 Les distributions

Il existe plusieurs distributions statistiques déjà programmées avec **R**. Voici les principales utilisées dans cet ouvrage.

Table 3.1: Noms des distributions, fonctions et leurs arguments

Distribution	R	Arguments
binomiale	binom	size, prob
Khi-carré	chisq	df, ncp
Fisher	f	df1, df2, ncp
normale	norm	mean, sd
student	t	df, ncp
uniform	unif	min, max

Les libellés ci-dessus ne commanderont pas de fonction. Il faut joindre en préfixe à ces distributions l'une des quatre lettres suivantes : `d`, `p`,`q`, ou `r`. La plus simple est certainement `r` (*random*) qui génère `n` valeurs aléatoires de la distribution demandée selon les paramètres spécifiés. Les fonctions `q` (quantile) prennent un argument de 0 à 1 (100%), soit un percentile et retourne la valeur de la distribution. La fonction `p` (probabilité) retourne la probabilité cumulative (du minimum jusqu'à la valeur) d'une valeur de cette distribution. Enfin, la lettre `d` (densité) permet, notamment, d'obtenir les valeurs de densité de la distribution.

Voici un exemple avec la distribution normale.

```
set.seed(9876)

# Génère 5 nombres aléatoires en fonction des paramètres
rnorm(n = 5, mean = 10, sd = .5)
#> [1] 10.51  9.42  9.90  9.95 10.01

# Retourne les valeurs associés à ces probabilités
qnorm(c(.025,.975))
#> [1] -1.96  1.96

# Retourne la probabilité d'obtenir un score de 1.645 et moins
pnorm(1.645)
#> [1] 0.95

# La valeur de la densité de la distribution
dnorm(0)
#> [1] 0.399
```

Ces quatre lettres peuvent être associées à toutes les distributions énumérées et bien d'autres. Elles respectent toutes ce cadre.

Exercices

1. Quel est le résultat de `mean <- c(1, 2, 3)`? Pourquoi?
2. Rédiger une fonction calculant l'hypoténuse d'un triangle rectangle. Rappel, le théorème de Pythagore est $c^2 = a^2 + b^2$.
3. Rédiger une fonction calculant un score-z pour une variable. Rappel, un score-z, correspond à $z = \frac{x-\mu}{\sigma}$.
4. Rédiger une fonction calculant la médiane d'une variable (ne recopier pas celle de ce livre).
5. Rédiger une fonction pour générer une séquence de Fibonacci (chaque nombre est la somme des deux précédents) jusqu'à une certaine valeur, soit $1, 1, 2, 3, 5, 8, ...$
6. Rédiger une fonction qui pivote une liste de k éléments par n. Par exemple, une liste de six ($k = 6$ comme $[1, 2, 3, 4, 5, 6]$) pivoté de deux ($n = 2$) devient ($[3, 4, 5, 6, 1, 2]$).

Part II

Jeux de données

Chapter 4

Importer

Réaliser des analyses statistiques et des représentations graphiques nécessitent un jeu de données sur lequel travailler. Après l'ouverture de **R**, l'usager remarquera aisément que le logiciel n'est pas un tableur (tableau de données), contrairement aux logiciels traditionnels. Cette caractéristique l'a peut-être même frappé à la première ouverture! Pour utiliser données, il faudra les conserver dans un fichier externe de la même façon qu'un script, à l'exception qu'il faudra importer les données pour son utilisation.

Un jeu de données porte généralement les extensions ".Rdata", lorsqu'elles proviennent de **R**, ou d'extensions ".dat" et ".txt". Évidemment, **R** permet une grande flexibilité, il est ainsi possible d'exporter et d'importer dans d'autres extensions. Les extensions ".Rdata" sont des environnements **R**, elles contiennent potentiellement plusieurs variables, comme une séance de travail complète. Elles ont aussi l'avantage que, si l'utilisateur double-clique sur un fichier d'extension ".Rdata", celui-ci s'ouvre automatiquement dans l'environnement **R**.

Il est recommandé de ne jamais manipuler les fichiers de données une fois toutes les vérifications réalisées (absence d'erreur dans les données). Il ne sera jamais nécessaire de modifier ces fichiers avec **R**. Contrairement aux logiciels traditionnels dans lesquels les modifications sont apportées directement aux valeurs ou ajoutées aux fichiers, cela n'est pas nécessaire dans la mesure où les syntaxes décrivant ces manipulations sont conservées. Il devient impossible d'endommager, de corrompre ou d'altérer le fichier de données. Les données originales restent intactes. Il suffit de les importer puis de commander la syntaxe qui lui est associée pour obtenir de nouveau la version *propre* du jeu de données.

4.1 Jeux de données provenant de R et de packages

Plusieurs packages offrent en plus des fonctions des jeux de données. Mieux encore! **R** offre des jeux de données inclus avec le logiciel. La fonction `data()` permet de voir la liste des jeux de données disponibles. Taper simplement le nom du jeu de données permet de l'utiliser, comme s'il avait été déclaré auparavant.

```
head(cars)
#>   speed dist
#> 1     4    2
#> 2     4   10
#> 3     7    4
#> 4     7   22
#> 5     8   16
#> 6     9   10
```

La fonction `head()` introduite ici donne simplement un aperçu des six premières lignes du jeu de données pour ne pas afficher le jeu de données complet (ce qui prendrait beaucoup d'espace inutilement.

Pour consulter tous les jeux de données des packages importés, il est possible d'utiliser cette ligne de code.

```
data(package = .packages((all.available = TRUE)))
```

Pour utiliser ces jeux, il faut rendre actif le package associé avec la fonction `library()`.

4.2 Création des jeux de données artificielles

TODO

Une façon rudimentaire et efficace d'obtenir des données avec **R** est de les créer à l'aide des fonctions génératrices de données pseudoaléatoires.

```
# Pour la reproductibilité
set.seed(142)
# Nombre d'unité
n <- 30
# Identifiant
id <- 1:n
```

```
# Variables
sexe <- rbinom(n, size = 1, prob = .5)
QI <-  round(rnorm(30, mean = 100, sd = 15) - 5 * sexe)
# Être "1" soustrait  5 points au QI en moyenne
# Arrondi avec round()

# Création de le jeu de donnees
donnees <-  data.frame(id = id, sexe = sexe, QI = QI)
# Enregistrement
save(donnees, file = "donnees.Rdata")
```

Et voilà un jeu de données simple et sauvegardé dans le dossier de travail auquel il sera possible de se référer.

Ici, deux nouvelles fonctions sont employées : `round()` arrondie les valeurs à l'unité et `data.frame()` crée un le jeu de données. L'utilisation des `=` permet de directement nommer les variables.

4.3 Exporter

4.3.1 Exporter de IBM SPSS

Il est possible d'importer des données de IBM SPSS vers **R**. il faudra quelques manipulations préalables. En ayant le fichier de données IBM SPSS ouvert, il faut cliquer sur "Enregistrer sous" sous le menu déroulant "Fichier". Par défaut, IBM SPSS choisira toutes les variables, mais il est possible de sélectionner seulement les variables d'intérêt en décochant les variables qu'il n'est pas nécessaire de conserver. Ensuite, sélectionner le type de fichier de sauvegarde doit être "Tabulé (*.dat)". IBM SPSS offre également la possibilité d'enregistrer les noms de variables (première option à cocher) et les libellés de valeur. Il suffit maintenant de nommer le fichier et de cliquer sur l'onglet"Enregister".

En s'assurant que nouveau fichier se trouve dans le répertoire actif de **R**, il suffit de télécharger le fichier.

```
read.table(file = "donnees.tab", header = TRUE)
```

L'option `header` devrait être `FALSE` si les noms de variables n'ont pas été conservés (première ligne du fichier) .

4.3.2 Exporter de Microsoft Excel

Fichier sous le menu déroulant. Sélectionner comme type de fichier "Texte Unicode (*.txt)". Intituler le fichier, puis cliquer sur"Enregister".

Microsoft Excel sauvegardera l'entièreté de la page active. Il est donc pertinent de créer une feuille Microsoft Excel contenant que les informations à conserver.

Par la suite, en s'assurant que nouveau fichier se trouve dans le répertoire actif de **R**, il suffit de télécharger le fichier avec `read.table()` et les arguments convenant au jeu de données.

4.4 Importer de R

Dans la plupart des situations, les analyses et les graphiques seront réalisés à partir d'un jeu de données se trouvant dans un fichier. Ce devra être importé en **R** pour être manipulé. Les jeux de données peuvent se trouver dans un fichier dans l'ordinateur, mais aussi sur le web. Ils peuvent être en différents types de format.

Pour la description de l'importation, l'ouvrage tient pour acquis que le fichier de données se retrouve dans le directoire de travail (ce qui est l'idéal en général). Un peu plus loin, la création de trajectoires pour différents emplacements sera présentée.

Les fonctions de base permettront d'importer la plupart des jeux de données, particulièrement s'ils ont été exportés dans un format compatible. Pour le cas où ces fichiers ne pourraient être exportés de cette façon, des packages pallieront ce besoin. Dans cet ouvrage, seule une présentation sommaire de ces options sera discutée, l'utilisateur est recommandé à la documentation de ces packages pour plus d'informations.

La prochaine section décrit les fonctions pour importer les bases de données "manuellement". En plus de ces méthodes, **R**Studio possède une interface permettant l'importation des données.

4.5 La fonction de base

La fonction de base `read.table()` permettra d'importer la plupart des jeux de données. C'est d'ailleurs ce qui a été présenté sommairement dans le chapitre sur les rudiments. Parfois, ceux-ci ont certaines caractéristiques qu'il faudra préciser comme argument à la fonction `read.table()` pour assurer une importation adéquate. Ces caractéristiques sont `header = FALSE`, `sep = ""`, et `fill = !blank.lines.skip` (les éléments à droite sont les options par défaut).

Parfois, certains fichiers sauvegardent le nom des variables en tête de colonne (première ligne). Par défaut, **R** assume qu'il s'agit de valeurs. L'argument `header = TRUE` ajouté à `read.table()` précisera à **R** lors de l'important que ces libellés sont des noms de colonnes.

Si un autre symbole est utilisé pour délimiter (séparer) des valeurs dans le fichier, comme `;`ou `,`, l'argument `sep = ";"` ou `sep = ","` précisera le séparateur.

Si les lignes du fichier sont de tailles inégales, **R** assumera qu'il s'agit de valeur, et ces *blancs* de texte seront ajoutés comme valeurs (`""`). Pour gérer cette situation, l'argument `fill = FALSE` devrait régler la situation.

4.6 Fichiers d'extension .txt

Un fichier d'extension `.txt` est un fichier texte délimité par des tabulations (*tab-delimited text files*) et est importé à l'aide de la fonction `read.table()`.

```
jd = read.table("fichier.txt")
```

4.7 Fichiers d'extension .dat

Un fichier d'extension `.dat` est un fichier générique de données et est importé à l'aide de la fonction `read.table()`.

```
jd = read.table("fichier.dat")
```

4.8 Fichiers d'extension .csv

Un fichier d'extension .csv use généralement de séparateur comme `";"` (lorsque le système numérique de la langue d'origine utilise la virgule - comme le français par exemple) ou `","` (pour les autres langues qui n'utilise pas la virgule) et ont généralement les noms de variables en première ligne. Ainsi, la fonction `read.table()` est utilisable pourvu que le séparateur soit précisé et la présence d'en-tête également.

```
jd = read.table("fichier.csv", sep = ";", header = TRUE)
```

Il existe aussi la fonction `read.csv()` et `read.csv2()` pour importer des fichiers d'extension `.csv`. Il s'agit exactement de `read.table()` à l'exception des arguments par défaut, mais précisant par défaut `header = TRUE` et `fill = TRUE` et détecte s'il s'agit de `";"` ou `","`.

4.9 Fichiers délimités

Pour les fichiers recourant à un autre caractère qu'une tabulation, qu'une `","`, ou un `";"` pour délimiter les valeurs, spécifier le caractère dans `read.table()` importera le fichier.

```
jd = read.table("fichier.txt", sep = "$")
```

Comme pour `read.csv()` et `read.csv2()`, les fonctions `read.delim()` et `read.delim()` pourraient être utilisées.

4.10 Fichiers d'extension .sav, .dta, .syd et .mtp

Comme le lecteur s'en doute peut-être, **R** de base ne permet pas d'importer des fichiers spécifiques d'autres logiciels. Par contre, avec les années se sont développés des packages permettant de pallier la situation. Le package `foreign` permet d'importer des fichiers issus de IBM SPSS (`.sav`), Stata (`.dta`) et Systat (`.syd`) et Minitab (`mtp`) avec, respectivement les fonctions `read.spss()`, `read.data()`, `read.systat()` et `read.mtp()`. La logique d'importation est la même pour ces quatre fonctions.

Pour `read.spss()`, deux arguments sont importants à souligner. Par défaut, la fonctionne ne retourne pas un *data frame* et utilise les libellés de valeurs (*value labels*). Dans la plupart des cas, l'utilisateur désire probablement obtenir un jeu de données de type *data.frame* et les valeurs sous-jacentes au libellés de valeurs. L'utilisateur peut alors changer ces arguments `to.data.frame = TRUE` (par défaut `FALSE`) et `use.value.labels = FALSE`(par défaut `TRUE`).

```
library(foreign)

# SPSS
jd = read.spss("fichier.sav",to.data.frame = TRUE,use.value.labels = FALSE)

# Stata
jd = read.dta("fichier.dta")

# Systat
jd = read.systat("fichier.syd")

# Minitab
jd = read.mtp("fichier.mtp")
```

Consulter la documentation du package pour plus d'informations sur les options possibles.

4.11 Fichiers d'extension .xls et xlsx

Il n'existe pas de fonction de base pour importer des fichiers Microsoft Excel (extensions `.xls` et `.xlsx`). Par contre, il existe plusieurs packages qui permettront de la faire, comme `readxl`. Le package `readlxl` permet d'utiliser la fonction `read_excel()` pour importer le fichier.

```
#Excel
library(readxl)
jd = read_excel("fichier.xls")
```

La fonction `read_excel()` possède un argument `sheet =` qui permet de préciser la feuille qu'il faut importer ou `range =` (p. ex. `range = A1:B20` qui permet d'importer un rectangle de plage de données (du coin supérieur gauche `A1` au coin inférieur droit `B20`). Consulter la documentation du package pour plus d'informations sur les options possibles.

4.12 Fichiers d'extension .html

Il n'existe pas de fonction de base pour importer des fichiers d'extension .html, (HTML, *HyperText Markup Language*). Le package `XML` fournit une solution possible avec la fonction `readHMTLTable()`.

```
#HTML
library(XML)
jd = readHMTLTable("fichier.html")
```

Consulter la documentation du package pour plus d'informations sur les options possibles.

4.13 Fichiers d'extension .json

Il n'existe pas de fonction de base pour importer des fichiers d'extension .json, (*JavaScript Object Notation*). Comme le lecteur pourra s'y attendre, il existe un package pour rectifier la situation : le package `rjson` et sa fonction `fromJSON()`.

```
#JSON
library(rjson)
jd = fromJSON("fichier.json")
```

Consulter la documentation du package pour plus d'informations sur les options possibles.

4.14 Fichiers d'extension .sas7bdat

Il n'existe pas de fonction de base pour importer des fichiers d'extension `.sas7bdat`, (*Statistical Analysis System*). Il existe le package `sas7bdat` pour importer des données de SAS vers **R** avec la fonction `read.sas7bdat()`.

```
# SAS
library(sas7bdat)
jd = read.sas7bdat("fichier.sas7dbat")
```

Consulter la documentation du package pour plus d'informations sur les options possibles.

4.15 Emplacement du jeu de données

Idéalement, le fichier contenant le jeu de données sera déjà dans le directoire de travail (ou dan le projet R en cours). Dans ce contexte, référer seulement au nom du fichier suffira.

```
jd = read.table("fichier.txt")
```

Si le jeu de données est sur le web, il peut être importé en précisant l'URL.

```
jd = read.table("https://site/ou/trouver/le/fichier.txt")
```

S'il est plutôt dans un fichier sur l'ordinateur, mais pas dans le directoire de travail, ce sera essentiellement la même méthode.

```
jd = read.table("C:/site/ou/trouver/le/fichier.txt")
```

Si l'utilisateur ne connaît pas exactement la trajectoire, il peut se résoudre à passer par l'explorateur de fichiers (Windows ou Apple) pour déterminer l'emplacement du fichier de jeu de données. Il faut alors utiliser la fonction `file.choose()` sans aucun argument à l'intérieur de la fonction d'importation.

```
jd = read.table(file.choose())
```

L'utilisateur devra alors identifier manuellement (pointer et cliquer) où se trouve le fichier. Il devra se promener de fichier en fichier jusqu'à ce qu'il arrive au bon jeu de données, un peu comme le font les logiciels traditionnels lorsque l'utilisateur souhaite sauvegarder un fichier à un certain endroit.

4.16 Importation avec RStudio

RStudio offre une interface simple pour télécharger directement un jeu de de données IBM SPSS, Microsoft Excel, SAS, STATA, et des extensions ".txt" et ".readr". Il y a même un outil de visualisation pour s'assurer que le tout est en ordre. Pour procéder, il faut faire *File > Import dataset > From "format de fichier* où"format de fichier" remplace Text, SPSS, Excel et les autres. Il suffit de suivre les instructions. En indiquant le chemin du fichier, **R** importera le fichier et fournira une syntaxe afin de reproduire l'importation pour de futurs usages.

4.17 Conseils d'importation

Parfois des valeurs s'ajoutent lors de l'exportation ou l'importation des données. Des logiciels traditionnels font parfois ce mauvais tour. Une vérification de la base de données est par conséquent impérative, surtout lors de la première utilisation du jeu de données. Deux méthodes de vérification sont suggérées. D'abord ouvrir le fichier avec un éditeur de texte de base, comme bloc-notes, pour s'assurer qu'aucun caractère ne s'est indésirable ajouté à l'insu de l'utilisateur. Ensuite, voir avec la fonction `View()` dans **R** si la base de données s'affiche correctement et que les variables, et lignes semblent correspondre à ce qui est attendu.

4.18 Voir la base de données

Il est possible de voir les données en utilisant la fonction `View()` et en y insérant le nom de la variable. Le logiciel afficher un tableur avec les données, dont il sera impossible de modifier les valeurs. Cela peut être utile pour s'assurer que le jeu de données est en ordre, bien importé, ou le consulter.

4.19 Sauvegarder un jeu de données

Si un jeu de données est directement créé avec **R**, par exemple, les jeux de données artificiels, il est possible de les sauvegarder avec la fonction `save()` qui enregistre une variable dans un fichier.

```
save("variable", file = "fichier.Rdata")
```

Il est possible à la fin d'une session de travail de sauvegarder l'environnement dans un fichier `save.image()`. Ainsi, toutes les variables et fonctions maison sont conservées pour une future utilisation.

```
save.image(file = "SessionTravail.Rdata")
```

4.20 Quelques conseils de gestion

Voici quelques conseils pour la gestion le jeu de données.

- Éviter les noms trop longs ou trop courts et dépourvus de signification. Cela augmente le risque d'erreur. L'utilisation de huit caractères ou moins est une bonne recommandation (quoique ce n'est pas une règle!). Pour conserver plus de renseignements, utiliser les commentaires de la syntaxe.
- Éviter les espaces entre les mots. Cela peut être interprété erronément comme deux éléments. À la place, collez les mots et distinguer-les avec des majuscules (MaFonction), utiliser le tiret bas (ma_fonction) ou un point (ma.fonction).
- Éviter les espaces ou les vides dans les données. Cela peut être interprété comme des données (absentes) ou non.
- Éviter les symboles suivants `?`, `$`, `%`, `^`, `&`, `*`, `(`, `)`, `-`, `#`, `?`, `,` , `<`, `>`, `/`, `|`, `\`, `[`, `]`, `{` et `}` qui peuvent erronément être interprétés comme de la syntaxe autant dans les noms de variables que dans les données.
- Vérifier que les valeurs manquantes sont identifiées `NA`.
- Si les données proviennent d'un autre logiciel, vérifier la présence de commentaires qui pourraient occasionner des lignes ou colonnes supplémentaires et ainsi corrompre le jeu de données.
- Vérifier que l'exportation et l'importation se sont bien déroulées.

Chapter 5

Entrée de données

S'il y a bien une caractéristique de **R** qui rebute les nouveaux utilisateurs, c'est certainement que le logiciel ne soit pas prévu pour la saisie de données ou du moins que cela ne soit pas mis à l'avant-plan. Lorsque le logiciel s'ouvre, que ce soit **R** ou **R**Studio, l'aspect *table de données* n'existe pas. L'utilisateur pour qui il s'agit de sa première utilisation (et habitué à des logiciels traditionnels) reste pantois : où les données sont-elles entrées?

5.0.1 Entrée de données avec `data.entry()` (R de base)

De base, **R** offre la possibilité d'entrer des données dans un tableur avec la commande `jd = data.entry()`. Si une base de données est demandée comme argument (p. ex., `data.entry(data = jd)`), alors le jeu de données est ouvert. Il est aussi possible d'ouvrir le fichier avec des variables déjà créées avec **R**. Si un tableur vierge est désiré, alors taper `data.entry(1)` dans la console ouvrira le tableur avec une seule valeur (`1`), L'utilisateur peut alors modifier les noms de colonnes et entrer les données comme il le ferait avec un logiciel traditionnel. Comme il est possible de le voir à la Figure 5.1, l'interface est bien moins attrayante que ses compétiteurs.

Lorsque l'entrée de données est terminée, l'utilisateur doit sauvegarder le jeu de données ou l'environnement de travail qui pourront être importés pour de futures utilisations ou entrées. En général, l'utilisateur qui entre manuellement ces données préférera certainement un autre tableur, mais **R** est certainement en mesure de faire ce travail.

Figure 5.1: Ouverture du tableur R

5.0.2 Entrée de données avec `data_edit` (package `DataEditR`)

Depuis tout récemment (juillet 2021), il existe un package qui permet de faire l'entrée de données en tableur avec **R**. Il s'agit du package `DataEditR` (Hammill, 2021), une interface utilisateur graphique. Il résout l'un de plus grands défis lorsqu'un utilisateur migre des tableurs traditionnels vers **R**, c'est-à-dire d'accomplir la transition d'une feuille de calcul interactive où il est possible de pointer et cliquer pour modifier, ajouter, supprimer des donnes vers un mode strict de syntaxe.

Pour démarrer, il faut d'abord installer le package, puis l'appeler. Pour commencer à entrer des données, la syntaxe `data_edit()` est suffisante. Pour ouvrir un jeu de données, il suffit de l'ajouter en argument `data_edit(jd)`.

Une fois l'interface ouvert, il est possible de manipuler le jeu de données avec les options affichés et avec le clic droit qui permettra notamment d'ajouter des lignes et des colonnes.

```
# Pour installer le package
install.packages("DataEditR")

# Pour rendre la package accessible
library("DataEditR")

# La fonction
data_edit()
```

Il est recommandé de ne laisser que les données brutes, toutes les modifications et manipulations devraient rester en syntaxe **R** dans un script associé au jeu de données. Lorsque les entrées sont terminées, il faut simplement sauvegarder la base de données, préférablement en extension .csv. Il est aussi possible de sortir le tableur en tableau de données en assignant la fonction à une variable comme `jd = data_edit()`.

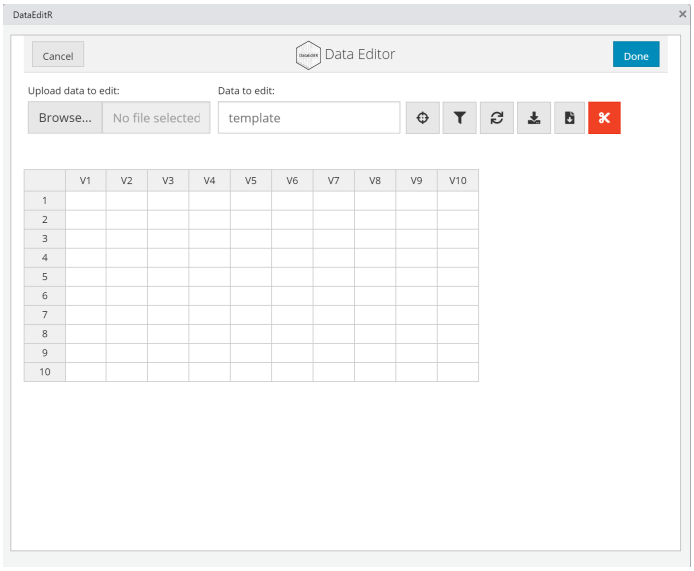

Figure 5.2: Ouverture du tableur de DataEditR

En général, l'utilisateur importera ces données dans l'environnement **R**. Il devra même le faire avec `data_edit()` à chaque ouverture d'un nouvelle séance, pour poursuivre l'entrée ou réaliser des manipulations.

Chapter 6

Manipuler

Avec **R**, il ne sera jamais nécessaire de manipuler directement le fichier contenant les données. Cette pratique est déconseillée. Préserver le fichier original intact évite de nombreuses complications, comme la compatibilité entre les versions, la reproductibilité des bases, la maintenance, etc. Toutes les manipulations seront conservées dans un script. Cela favorise le partage entre collègues, mais aussi le suivi des modifications apportées en comparant tout simplement les traces dans entre les syntaxes. Le jeu de données est importé dans la console et il ne sera plus touché par l'utilisateur. Le script conserve les manipulations réalisées qui pourront être refaites facilement en quelques cliques.

En pratique, l'expérimentateur aura le jeu de données officielles (final) avec lequel travailler. Il pourra l'importer tel que précisé dans la section précédente. Par la suite, il ne lui reste qu'à mettre en place le nettoyage et la préparation du jeu de données. Il existe plusieurs méthodes pour réalise la gestion; il ne faut pas s'étonner de voir d'autres ouvrages aborder la gestion de données d'une autre façon. Au final, la meilleure méthode est celle qui m'est l'utilisateur à son aise.

TODO

Dans cette section les manipulations rudimentaires d'un jeu de données seront expliquées.

6.1 Manipulation de données

Les tableaux ont généralement deux dimensions (lignes par colonnes). Différents éléments ou groupes d'éléments peuvent être extraits des jeux de données. Plusieurs méthodes peuvent être utiliser en fonction des besoins. Le jeu de données `cars` sera utilisé à des fins illustratives.

```
head(cars)
#>   speed dist
#> 1     4    2
#> 2     4   10
#> 3     7    4
#> 4     7   22
#> 5     8   16
#> 6     9   10
```

Elle contient 50 unités d'observation (lignes) et deux variables (colonnes), soit la vitesse (*speed*) et la distance (*dist*).

6.1.1 Référer à une variable dans un jeu de données

Il est possible de référer à une variable soit en utilisant l'emplacement de la variable par rapport aux autres en utilisant les crochets ou en utilisant le signe `$` puis le nom de la variable après le libellé de le jeu de données. L'opération est fort simple avec le symbole `$`.

```
# Avec $
head(cars$dist)
#> [1]  2 10  4 22 16 10
```

Précédemment utilisés pour extraire des valeurs dans une variable unidimensionnelle, les `[]` peuvent extraire des données sur un tableau en deux dimensions (ligne par colonne). Il faut spécifier la ou les lignes désirées, puis la ou les colonnes désirées entre crochets. Laissez une des dimensions en blanc (vide) indique au logiciel de rapporter toutes les valeurs. Par exemple, pour obtenir le même résultat, `dist` est la deuxième colonne. Il faut référer entre crochets à la colonne 2 et comme toutes les lignes sont désirées, cette dimension reste vide.

```
# Entre crochets
head(cars[,2])
#> [1]  2 10  4 22 16 10
```

Il est possible de faire la même chose avec les lignes.

```
# Entre crochets
cars[4,]
#>   speed dist
#> 4     7   22
```

Ici, toutes les variables de la 4^{e} unité sont rapportée. Remarquer bien l'absence d'argument après la virgule. La fonction `head` n'est pas nécessaire ici, car il y a peu d'informations à extraire.

Si certaines valeurs spécifiques étaient désirées, comme la valeur de la 4^{e} unité pour la 2^{e} variable.

```
# Entre crochets
cars[4, 2]
#> [1] 22
```

Enfin, à l'intérieur d'un jeu de données, les variables peuvent être commandées avec le signe de `$` placé après le nom de la variable suivi du nom de la variable ou encore en identifiant les noms de variables entre crochets.

```
# Utilisation du signe $
head(cars$speed)
#> [1] 4 4 7 7 8 9

# Nommer entre crochets
head(cars["speed"])
#>   speed
#> 1     4
#> 2     4
#> 3     7
#> 4     7
#> 5     8
#> 6     9
```

6.1.2 Référer à un sous-ensemble d'unité.

Pour référer à des unités ayant certaines caractéristiques, la fonction `subset()` peut s'avérer utile. Les arguments sont un jeu de données, le deuxième est un opérateur logique en lien avec une variable du jeu de données.

```
# Extraire les données pour toutes les unités ayant une vitesse égale à 24
subset(cars, speed == 24)
#>    speed dist
#> 46    24   70
#> 47    24   92
#> 48    24   93
#> 49    24  120
```

Cette fonction est utile s'il faut extraire les données d'un certain sexe, par exemple.

6.1.3 Nommer des variables dans un jeu de données

Il est possible d'attribuer ou de modifier des noms à des colonnes ou des lignes d'un tableau de données. Les fonctions `colnames()` et `rownames()` seront alors utilisées. Contrairement aux autres fonctions, celles-ci se retrouvent à gauche de l'équation.

```
colnames(cars) <-  c("vitesse", "distance")
head(cars)
#>   vitesse distance
#> 1       4        2
#> 2       4       10
#> 3       7        4
#> 4       7       22
#> 5       8       16
#> 6       9       10
```

Il importe de fournir autant de noms qu'il y a de colonnes (ou lignes), et ce, en chaîne de caractères.

6.1.4 Données manquantes

Les devis de recherche et les jeux de données empiriques sont rarement parfaits et peuvent souvent contenir des données manquantes. **R** reconnaît les données manquantes lorsqu'elles sont identifiées comme `NA` (*not available*). Plusieurs méthodes permettent de gérer les données manquantes. La méthode la plus simple est d'éliminer les unités ayant une donnée manquante, soit la suppression par liste (*listwise suppression*). Les fonctions natives de **R** recourront à l'argument `na.rm = TRUE`. Si cela est impossible, la fonction `na.omit()` permettra de créer des jeux de données sans les valeurs manquantes.

```
# Un vecteur
valeurs <-  c(10, 12, 14, NA, 18)

# La présence de NA empêche la moyenne d'être calculée
mean(valeurs)
#> [1] NA

# L'argument "na.rm = TRUE" gère les NA
mean(valeurs, na.rm = TRUE)
#> [1] 13.5

# na.omit omet les valeurs NA dans la nouvelle variable.
valeurs.nettoyees <-  na.omit(valeurs)
```

```
mean(valeurs.nettoyees)
#> [1] 13.5
```

6.2 Le tidyverse

Le nom *tidyverse* (Wickham et al., 2019) est une contraction de *tidy* (bien rangé) et de *universe*. Le package est fondé sur le concept de *tidy data*, développé par Hadley (Wickham, 2014). Il repose sur une philosophie d'organisation des données facilitant la gestion, la préparation et le nettoyage préalable aux analyses quantitatives. Plusieurs packages respectent cette philosophie et font partie intégrante du *tidyverse*, comme `ggplot2` (présentation graphique), `dplyr` (manipulation de données), `readr` (importation de données), `tibble` (nouvelle catégorie de *data frame*), mais bien d'autres également. Ces packages font part intégrante de l'univers *tidy* et sont téléchargés simultanément avec le package.

Pour utiliser le package `tidyverse`, il faudra d'abord l'installer puis l'appeler.

```
# Installer le package
install.packages("tidyverse")

# Rendre le package accessible
library(tidyverse)
```

6.3 Les fonctions utiles

Un des avantages et nouveautés d'utiliser le `tidyverse` est d'obtenir l'opérateur `%>%` (appelée *pipe* en anglais que l'on pourrait traduire par *tuyau*) qui provient originellement du package `magrittr` (Bache & Wickham, 2020) et est importé par `dplyr`. L'opérateur favorise la lisibilité et la productivité, car il est plus facile de suivre le flux de plusieurs fonctions à travers ces *tuyaux* que de revenir en arrière lorsque plusieurs fonctions sont imbriquées. En fait, il favorise la lecture par verbes, soit par action (fonction), dans une séquence temporelle intuitive. Si les arguments sont placés en une seule ligne, non seulement la ligne est-elle longue et complexe, voire illisible, mais, en plus, les éléments les plus à gauche (les premiers à la lecture) sont les derniers opérés. Si chacune des fonctions était en ligne, alors il faudrait écraser ou créer des variables temporaires inutiles tout simplement pour arriver à réaliser les fonctions. La philosophie `tidyverse`, par l'usage de `%>%`, évite tous ses problèmes.

L'opérateur `%>%` s'ajoute à la fin d'une ligne syntaxe. Son fonctionnement se traduit par l'argument de la ligne à gauche est introduit dans la fonction de droite, et ce, du haut vers le bas. Il peut être commandé plus rapidement avec le raccourci **Ctrl + Shift + M** sur Windows. En plus de l'opérateur `%>%`

, `dplyr` offre de nouvelles fonctions pour gérer un jeu de données. Quelques-unes des plus importantes sont décrites ici. Par la suite, une mise en situation permettra de mieux comprendre leur fonctionnement.

6.3.1 Sélectionner des variables

Pour sélectionner des données d'un très grand jeu de données, la fonction `select()` permettra de choisir les variables à conserver. Pour utiliser la fonction, il suffit d'indiquer les variables par leur nom de colonne dans la fonction. Aucun besoin de guillemets.

6.3.2 Sélectionner des participants

Pour filtrer les participants selon les caractéristiques désirées, la fonction `filter()` permettra de sélectionner les unités satisfaisant les conditions spécifiées. Pour utiliser la fonction, il faut indiquer le ou les arguments conditionnels à respecter et sur quelle variable.

Dans ce contexte la fonction `na.if()` peut être utile pour retirer une valeur aberrante.

6.3.3 Transformer et créer des variables

Pour créer ou transformer des variables, la fonction `mutate()` permettra de créer de nouvelles variables à partir des valeurs déjà dans le jeu de données. Il suffit d'indiquer dans la fonction, le calcul qui doit être opérer.

6.3.4 Sommariser les informations pertinentes

Pour obtenir des informations sur le jeu de données ainsi créées, la fonction `summarise()` permettra notamment d'obtenir des statistiques d'intérêt. En ajoutant, dans la fonction, les fonctions désirées, comme `mean()` ou `sd()`, avec les variables sur lesquelles elles devraient être opérées ou encore `n()` pour connaître la taille des groupes.

S'il y a des groupes ou des catégories, le sommaire peut être divisé avec la fonction `group_by()` où la variable nominale est spécifiée.

6.3.5 Autres fonctions

Il existe plusieurs autres fonctions possibles. Notamment, `slice()` permet de choisir les unités désirées en passant comme argument la base de données et

le ou les numéros de ligne; `sample_slice()` qui est très similaire, retourne des lignes aléatoires; `rename()`, similaire à `select()`, permet de renommer les variables; `arrange` reclasse par ordre croissant en fonction d'une variable placée en argument. Et il y en a plusieurs autres.

6.4 Mise en situation

Pour mettre en pratique la philosophie `tidyverse`, voici un exemple tiré du jeu de données `starwars`. Ce jeu de données possède de nombreuses caractéristiques (diversité de variables, de mesures, données manquantes) qui en font un jeu de données similaires à ce qu'un expérimentateur pourrait obtenir. Le jeu de données est déjà disponible avec **R**.

Avant de commencer, il faut rendre le package `tidyverse` disponible, car plusieurs fonctions seront utiles et le jeu de donnée s'y trouve.

```
# Importer le tidyverse avant de commencer
library(tidyverse)
```

Sans plus de préliminaire, la fonction `head()` donne un aperçu du jeu de données

```
starwars[,1:6]
#> # A tibble: 87 x 6
#>    name          height  mass hair_color skin_color eye_color
#>    <chr>          <int> <dbl> <chr>      <chr>      <chr>
#>  1 Luke Skywal~    172    77 blond      fair       blue
#>  2 C-3PO           167    75 <NA>       gold       yellow
#>  3 R2-D2            96    32 <NA>       white, bl~ red
#>  4 Darth Vader     202   136 none       white      yellow
#>  5 Leia Organa     150    49 brown      light      brown
#>  6 Owen Lars       178   120 brown, gr~ light      blue
#>  7 Beru Whites~    165    75 brown      light      blue
#>  8 R5-D4            97    32 <NA>       white, red red
#>  9 Biggs Darkl~    183    84 black      light      brown
#> 10 Obi-Wan Ken~    182    77 auburn, w~ fair       blue-gray
#> # ... with 77 more rows
```

Pour obtenir de l'information sur ce jeu de données.

```
?starwars
```

Voici la description du jeu de données (traduction libre),

> Les données d'origine, issues de SWAPI, l'API de Star Wars, https://swapi.dev/, ont été révisées pour tenir compte des recherches supplémentaires sur la détermination du genre et du sexe des personnages.

Peu utile comme descripteur, une inspection des données est plus informative. Pour afficher le jeu de données dans un nouvel onglet.

```
View(starwars)
```

Le fichier contient, le nom de 87 personnages mesurés sur 14 variables, soit

- le nom;
- la taille (cm);
- le poids (kg);
- la couleur des cheveux, de la peau et des yeux (trois variables);
- l'année de naissance;
- le sexe biologique (mâle, femelle, hermaphrodite ou aucun);
- le genre;
- la planète natale;
- l'espèce;
- une liste de films où le personnage apparaît;
- une liste des véhicules que le personnage a piloté;
- une liste des vaisseaux que le personnage a piloté.

L'objectif est de cette mise en situation est de comparer les hommes et les femmes humaines par rapport à leur indice de masse corporelle (IMC) ou *body mass index* (BMI). Le calcul de l'IMC consiste à diviser le poids par la taille au carré (kg/m^2).

Les étapes à considérer sont les suivantes : sélectionner les variables pertinentes, filtrer en retirant les unités d'espèces non humaines, tenir compte des données manquantes, corriger la taille des unités qui devrait être en mètre et non en centimètre (divisé par 100) et créer l'indice de masse corporelle.

```
jd <-  starwars %>%
  select(sex, mass, height, species) %>%
  filter(species == "Human") %>%
  na.omit() %>%
  mutate(height = height  / 100) %>%
  mutate(IMC = mass / height^2)
jd
#> # A tibble: 22 x 5
#>    sex     mass height species   IMC
#>    <chr>  <dbl>  <dbl> <chr>   <dbl>
#>  1 male      77   1.72 Human    26.0
#>  2 male     136   2.02 Human    33.3
#>  3 female    49   1.5  Human    21.8
#>  4 male     120   1.78 Human    37.9
#>  5 female    75   1.65 Human    27.5
#>  6 male      84   1.83 Human    25.1
#>  7 male      77   1.82 Human    23.2
#>  8 male      84   1.88 Human    23.8
#>  9 male      80   1.8  Human    24.7
#> 10 male      77   1.7  Human    26.6
#> # ... with 12 more rows
```

Les étapes de la syntaxe se lisent comme suit :

- La première ligne `starwars %>%` indique l'objet sur lequel il faut passer les fonctions subséquentes et la sortie est assignée à `jd`;
- puis, `select(sex, mass, height, species) %>%` indique les variables à conserver pour les fonctions subséquentes;
- puis, `filter(species == "Human")` filtre les unités qui sont humains et passe aux fonctions subséquentes;
- puis, `na.omit() %>%` retire les valeurs manquantes des unités dans le jeu de données et passe aux fonctions subséquentes;
- puis, `mutate(height = height  / 100) %>%`, transforme la variable `height` et passe à la dernière fonction;
- enfin, `mutate(IMC = mass / height^2)` crée la variable d'IMC.

Si une méthode plus traditionnelle avait été utilisée, la syntaxe pourrait ressembler à ceci.

```
jd <- starwars[, c("sex", "mass", "height", "species")]  # select()
jd <- jd[jd[, "species"] == "Human",]                    # filter()
```

```
jd <- na.omit(jd)                                          # na.omit()
jd[,"height"] <- jd[,"height"] / 100                       # mutate()
jd[,"IMC"] <- jd[,"mass"] / jd[,"height"]^2                # mutate()
jd
#> # A tibble: 22 x 5
#>    sex     mass height species   IMC
#>    <chr>  <dbl>  <dbl> <chr>   <dbl>
#>  1 male      77   1.72 Human    26.0
#>  2 male     136   2.02 Human    33.3
#>  3 female    49   1.5  Human    21.8
#>  4 male     120   1.78 Human    37.9
#>  5 female    75   1.65 Human    27.5
#>  6 male      84   1.83 Human    25.1
#>  7 male      77   1.82 Human    23.2
#>  8 male      84   1.88 Human    23.8
#>  9 male      80   1.8  Human    24.7
#> 10 male      77   1.7  Human    26.6
#> # ... with 12 more rows
```

Le jeu de données est créé en autant de ligne de syntaxe. Par contre, la lecture n'est pas aussi intuitive qu'avec l'utilisation de l'opérateur `%>%` et des fonctions associées `select()`, `filter()`, `mutate()`. Il ne faut pas trop penser à quoi ressemblerait ces manipulations en une seule ligne de syntaxe.

Une fois le jeu de données prêt, il est possible d'obtenir les informations sommaires. Ici, la moyenne, l'écart type, la valeur minimale et maximale ainsi que le nombre de données sont demandés en fonction du sexe. À cette étape, l'avantage d'embrasser la philosophie `tidyverse` apparaît, en quelques lignes rudimentaires, les cinq statistiques demandées sont affichées, et ce, par groupes.

```
jd %>%
  group_by(sex) %>%
  summarise(mean(IMC), sd(IMC), min(IMC), max(IMC), length(IMC))
#> # A tibble: 2 x 6
#>   sex    `mean(IMC)` `sd(IMC)` `min(IMC)` `max(IMC)`
#>   <chr>        <dbl>     <dbl>      <dbl>      <dbl>
#> 1 female        22.0      5.51       16.5       27.5
#> 2 male          26.0      4.29       21.5       37.9
#> # ... with 1 more variable: `length(IMC)` <int>
```

La base de données issues de ces opérations pourra par la suite être utilisée normalement pour réaliser des analyses statistiques. Il existe des packages pour demeurer dans le `tidyverse` comme `rstatix` où il est possible de faire des test-t avec `test_t()` ou des corrations avec `cor_test()`, par exemple. Voir la documentation complète du package pour une vue d'ensemble de ce qu'il est possible d'accomplier avec `rstatix`. Cela dit, l'utilisateur préférera probablement utiliser d'autres méthodes lorsque des analyses statistiques seront nécessaires.

```
library(rstatix)
# Test-t sur l'IMC en fonction du sexe
jd %>%
  t_test(IMC ~ sex)
#> # A tibble: 1 x 8
#>   .y.   group1 group2    n1    n2 statistic    df     p
#> * <chr> <chr>  <chr>  <int> <int>     <dbl> <dbl> <dbl>
#> 1 IMC   female male       3    19     -1.23  2.40 0.326

# Analyse de corrélations
jd %>%
  select(IMC, mass, height) %>%
  cor_test()
#> # A tibble: 9 x 8
#>   var1   var2     cor statistic         p conf.low conf.high
#>   <chr>  <chr>  <dbl>     <dbl>     <dbl>    <dbl>     <dbl>
#> 1 IMC    IMC     1      2.12e+8 5.26e-155    1.00      1
#> 2 IMC    mass    0.85   7.32e+0 4.47e-  7    0.674     0.938
#> 3 IMC    height  0.18   8.13e-1 4.26e-  1   -0.262     0.558
#> 4 mass   IMC     0.85   7.32e+0 4.47e-  7    0.674     0.938
#> 5 mass   mass    1      3.00e+8 5.13e-158    1         1
#> 6 mass   height  0.65   3.84e+0 1.02e-  3    0.317     0.842
#> 7 height IMC     0.18   8.13e-1 4.26e-  1   -0.262     0.558
#> 8 height mass    0.65   3.84e+0 1.02e-  3    0.317     0.842
#> 9 height height  1    Inf       0             1         1
#> # ... with 1 more variable: method <chr>
```

Chapter 7

Visualiser

La visualisation de données est l'un des deux objectifs fondamentaux de **R** (l'autre étant évidemment de faire des statistiques). Il existe plusieurs méthodes et packages pour produire rapidement et simplement des graphiques. Beaucoup de matériel se retrouve en ligne pour maîtriser les graphiques, mais surtout les personnaliser. L'objectif, bien modeste, de cette section n'est pas de rendre le lecteur maître de la production de figure, mais bien de lui faire faire ses premiers pas et de l'outiller pour qu'il puisse produire simplement et rapidement des graphiques de qualité.

L'exemple de cette section est basé sur celle de la section Manipuler. Voici la syntaxe pour réobtenir le jeu de données.

```
jd <- starwars %>%
  select(name, sex, mass, height, species) %>%
  filter(species == "Human") %>%
  na.omit() %>%
  mutate(height = height  / 100) %>%
  mutate(IMC = mass / height^2)
```

7.1 ggplot2

Le package `ggplot2` est une extension du `tidyverse` avec lequel il est possible de créer simplement et rapidement des graphiques. Ces graphiques sont de qualité de publications, idéale pour les articles scientifiques. Le package fournit un langage graphique pour la création intuitive de graphiques compliqués. Il permet à l'utilisateur de créer des graphiques qui représentent des données numériques et catégorielles univariées et multivariées.

La logique de `ggplot2` repose sur la grammaire des graphiques (*Grammar of Graphics*), c'est-à-dire, l'idée selon laquelle toutes les figures peuvent être construites à partir des mêmes composantes. Il s'agit de la deuxième version du package. Voilà pour l'appellation *ggplot2*.

Dans la grammaire de graphique, une figure possède huit niveaux, dont les trois principaux sont les suivants :

- *data*, les données utilisées;
- *mapping (aesthetic)*, cartographier les variables, c'est-à-dire, établir la carte des variables (abscisses, ordonnées, coleur, forme, taille, etc.);
- *geometric représentation*, la représentation géométrique ou le type de représentation graphique, par exemple, diagramme de dispersion, histogramme, boîte à moustache, etc.

Les cinq autres son, *statistics*, *facet*, *coordinate space*, *labels*, *theme* permettent de personnaliser la figure.

Les composantes les plus importantes sont les trois premières, soit les données, la cartographie et la représentation géométrique. Ce sont les éléments de base pour débuter le graphique. Les autres composantes viendront bonifier la figure tout en l'ajustant au besoin de l'utilisateur.

La fonction `ggplot()` met en place la figure. Le résultat d'utiliser la fonction `ggplot()` seule est illustrée à la Figure 7.1

```
ggplot(data = jd)
```

Il est aussi possible de *piper* (prononcé avec un fort accent anglophone) les données dans la fonction.

```
jd %>%
  ggplot()
```

Pour afficher des graphiques, il faut ajouter `+`, puis une représentation géométrique ainsi que la cartographie (*mapping*). La cartographie (`aes(mapping = )`- *aes* désigne l'esthétisme, *aesthetic*) peut se trouver dans `ggplot()` ou dans la représentation géométrique. Si elle est dans `ggplot`, elle est passée aux autres niveaux.

Voici une liste des représentations géométriques possibles.

- `geom_line()` crée une ligne qui lie toutes les valeurs, très utiles pour une série temporelle (abscisse = temps, ordonnée = variable dépendante)

Figure 7.1: La fonction ggplot() seule - Rien

- `geom_point()` crée un diagramme de dispersion ou un nuage de point, très utile pour les corrélations
- `geom_bar()` crée un diagramme à bâton, idéal pour présenter des proportions, des fréquences ou des données comptées
- `geom_histogram()` crée un histogramme des variables
- `geom_box()` crée une boîte à moustache, idéal pour identifier des valeurs aberrantes et comparer la variabilité entre des groupes.
- `geom_smooth()` crée la ligne de prédiction des données avec des intervalles de confiances, la plupart des utilisateurs voudront certainement ces arguments `geom_smooth(method = lm)` (par défaut) ou sans l'erreur standard (`se = FALSE`).
- `geom_error()` crée de

Certaines cartographies sont d'ailleurs compatibles, `geom_smooth()` et `geom_point()`, par exemple.

La figure 7.2 montre un diagramme dispersion construit à partir du jeu de données `jd` *piper* dans la fonction `ggplot()` dans laquelle la cartographie est passée `mapping = aes(x = mass, y = height)`, un second niveau est ajouté `+` et la représentation.

```
jd %>%
  ggplot(mapping = aes(x = mass, y = height)) +
  geom_point()
```

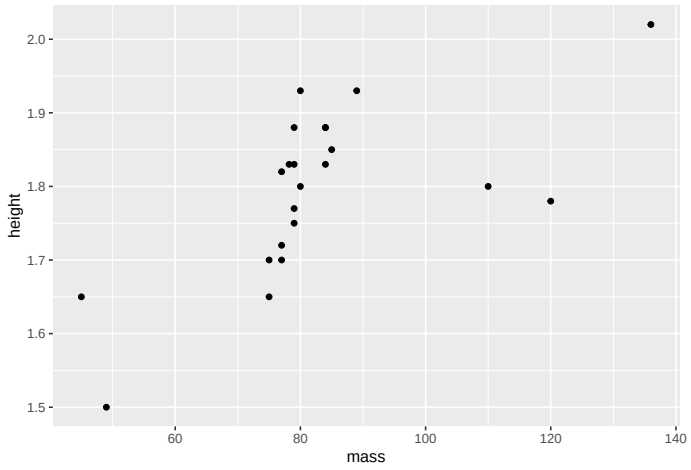

Figure 7.2: Diagramme de dispersion

Voici une liste d'exemples de différentes représentations géométriques.

7.2 Diagramme de dispersion

Pour réaliser un diagramme de dispersion, la fonction se nomme `geom_point`. La cartographie identifie la variable à l'axe des x (horizontal) et des y (vertical). Dans cet exemple, il s'agit du poids (x) et de la taille (y). La cartographie ne se limite pas aux axes par contre. Dans cet exemple, la forme `shape` est aussi un dimension manipulée. Il aurait pu s'agir de `color` et même de `size`. Dans le code ci-dessous, `size` est placé à l'extérieur de *mapping*, il s'agit alors d'une constante (elle change la taille des points), c'est-à-dire qu'elle ne varie pas avec les variables.

```
jd %>%
  ggplot() +
  geom_point(mapping = aes(x = mass, y = height, shape = sex), size = 2)
```

La figure 7.4 montre le résultat si 'size"est ajouté au *mapping* pour identifier l'IMC. Les unités avec un plus grand IMC obtiennent un plus gros pointeur.

```
jd %>%
  ggplot() +
  geom_point(mapping = aes(x = mass, y = height, shape = sex, size = IMC))
```

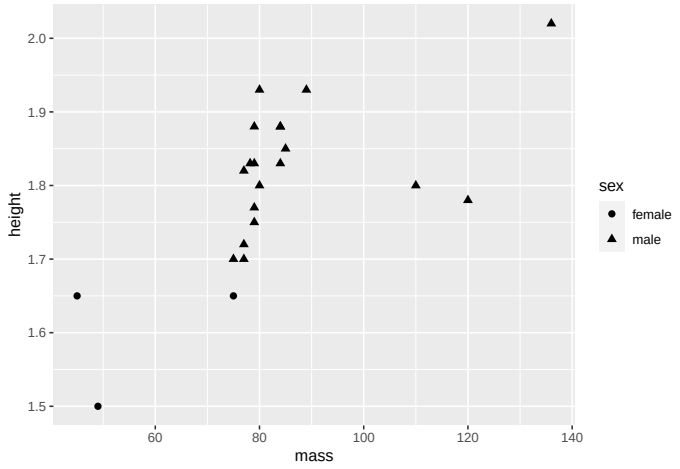

Figure 7.3: Le lien entre le poids et la taille en fonction du sexe

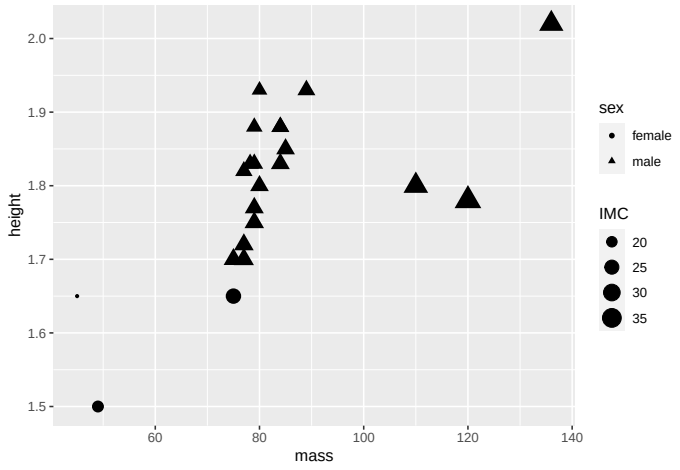

Figure 7.4: Le lien entre le poids et la taille en fonction de l'IMC et du sexe

On peut y ajouter la droite de régression, comme la Figure 7.5 le montre. Ne pas ajouter `geom_point()` ne ferait que produire la droite. Les arguments de `geom_smooth()` indique que l'utilisation du modèle linéaire et l'absence des intervalles de confiance. Dans ce code, également comme le *mapping* est ajouté à `ggplot` directement, il se généralise directement à `geom_point()` et `geom_smoooth()`

```
jd %>%
  ggplot(mapping = aes(x = mass, y = height)) +
  geom_point(size = 2) +
  geom_smooth(method = lm, se = FALSE, color = "black")
#> `geom_smooth()` using formula 'y ~ x'
```

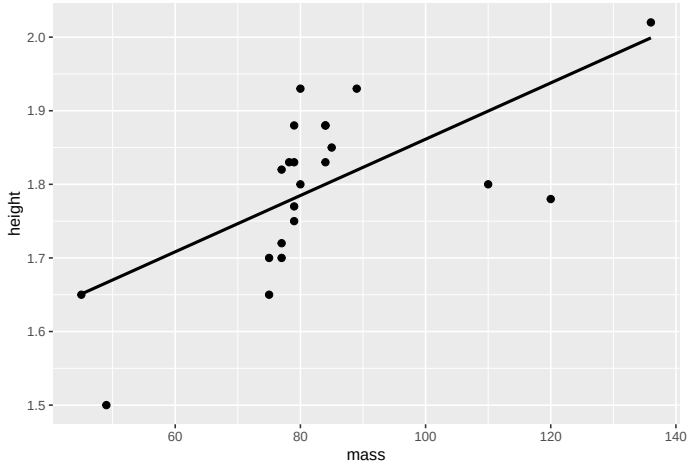

Figure 7.5: Le lien entre le poids et la taille en fonction de l'IMC

7.3 Boîte à moustache

La boîte à moustaches (*box-and-whisker plot*) est une figure permettant de voir la variabilité des données. Elle résume seulement quelques indicateurs de position soit la médiane, les quartiles, le minimum, et le maximum. Ce diagramme est utilisé principalement pour détecter des valeurs aberrantes et comparer la variabilité entre les groupes. C'est la représentation géométrique `geom_boxplot()` qui permettra de créer des boîtes à moustache. La cartographie prend en argument un variable nominale en `x` et une variable continue en `y`.

```
ggplot(data = jd) +
  geom_boxplot(mapping = aes(x = sex, y = IMC)) +
  coord_flip()
```

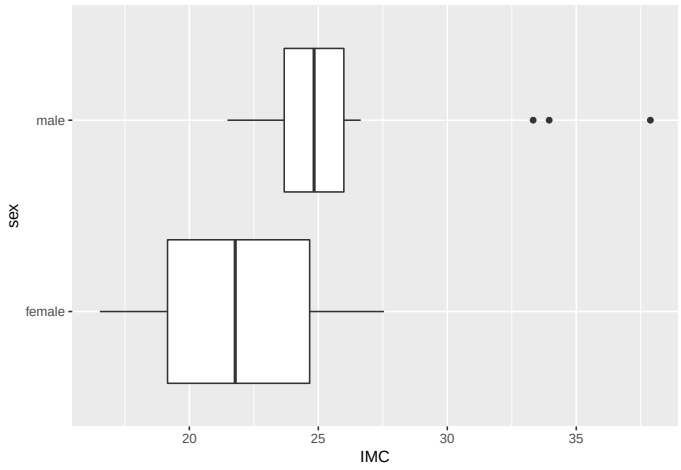

Figure 7.6: Boîte à moustache de l'IMC en fonction du sexe

Une fonction intéressante est la fonction `coord_flip()` qui tourne (*flip*) les axes, les coordonnées. L'axe x prend la place de y; y prend la place de x. Elle peut être pratique pour améliorer la qualité visuelle de certains graphiques.

7.4 Histogramme

Un histogramme permet de représenter la répartition empirique d'une variable. Il donne aperçu de la distribution sous-jacente, soit comment les données sont distribuées. Cette figure permet de voir la forme de la distribution et permet de voir si elle ne démontre pas d'anomalie. La représentation graphique `geom_histogram()` produit des histogrammes. S'il faut en produire pour différentes variables, une statégie simple est les produire en série.

```
# Trois histogrammes en trois figures
ggplot(data = jd) +
  geom_histogram(mapping = aes(x = height))

ggplot(data = jd) +
  geom_histogram(mapping = aes(x = mass))

ggplot(data = jd) +
  geom_histogram(mapping = aes(x = IMC))
```

Des techniques plus avancées permettront de créer la Figure 7.7 d'un seul coup.

```
# Trois histogrammes en une seule figure
# en optimisant avec le tidyverse
```

```
jd %>%
  keep(is.numeric) %>%
  gather() %>%
  ggplot(aes(value)) +
  facet_wrap(~ key, scales = "free") +
  geom_histogram()
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```

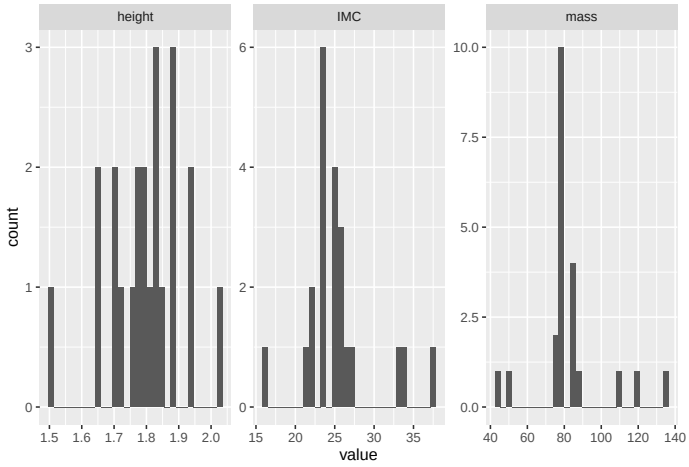

Figure 7.7: Histogrammes des variables continues

Enfin, s'il est désiré de comparer deux distributions de groupes différents, l'argument `fill` dans la cartographie indiquera à la fonction de différencier les valeurs selon le *remplissage* des histogrammes.

```
jd %>%
  ggplot(mapping = aes(x = IMC, fill = sex)) +
  geom_histogram(position = "identity", alpha = .7) +
  scale_fill_grey()
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```

Dans la figure 10.2, l'argument `position = "identity"` indique de traiter les deux groupes comme différents, autrement les colonnes s'additionneraient dans le graphique. L'argument `alpha = .7` permet une transparence entre les couleurs, autrement, les valeurs *derrière* les autres ne paraîtraient pas. La valeur de `alpha` va de 0 (transparent) à 1 (opaque) et fonctionnera dans la plupart des contextes, surtout ceux liés à `ggplot2`.

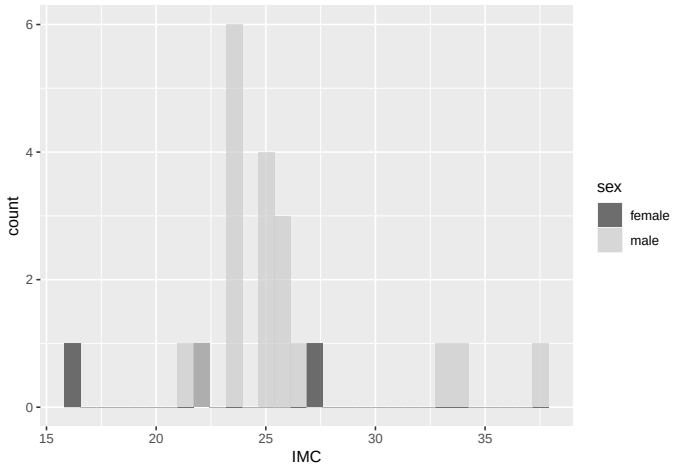

Figure 7.8: Histogrammes de l'IMC par rapport au sexe

7.5 Les barres d'erreurs

Les barres d'erreur sont une représentation géométrique à part entière. C'est une composante que l'on peut ajouter. La fonction pour les commandées est `geom_errorbar()`. Elle nécessite deux arguments, soit l'intervalle de confiance maximale et minimale autour des moyennes à afficher.

La figure 7.9 illustre les différences entre moyennes avec des barres d'erreur à partir de la base de données `ToothGrowth`, une étude de l'effet de la vitamine C (`dose`) selon leur administration (jus ou supplément `supp`) sur la longueur des dents des cochons d'inde. Il y a deux facteurs et une variable continue.

La première étape est de tirer les statistiques sommaires, moyennes, écart type, tailles des groupes. La syntaxe tire profit de `groupe_by()` pour tirer les groupes et en faire le sommaire. Le sommaire `summarise` permet d'obtenir les statistiques, notamment la moyenne, l'erreur standard (`se`) pour en calculer l'intervalle autour de la moyenne `ci`.

```
jd = ToothGrowth %>%
  group_by(dose, supp) %>%
  summarise(mlen = mean(len),
            sdlen = sd(len),
            nlen = n(),
            se = sd(len)/sqrt(n()),
            ci = qt(.975, df = n()-1) * se,
            .groups = "drop")

jd %>%
```

```
ggplot(aes(x = dose,
           y = mlen,
           shape = supp),
       size = 5) +
  geom_errorbar(aes(ymin = mlen - ci,
                    ymax = mlen + ci),
                width = .05) +
  geom_line() +
  geom_point()
```

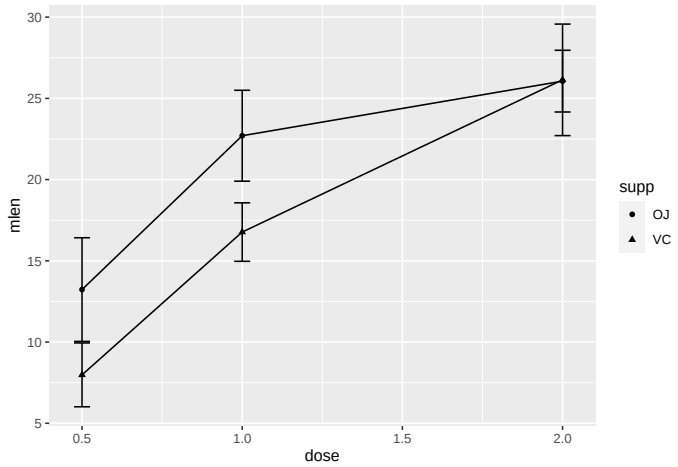

Figure 7.9: Les effets de la vitamine C sur les cochons d'inde

Une fois ces statistiques calculées et enregistrées dans le nouveau jeu de données `jd`, il est possible de créer le graphique avec les représentations géométriques désirées. Remarquer comment spécifié la cartographie dans le niveau `ggplot()` rend la syntaxe moins compliquée. Cette syntaxe produit un graphique avec `dose` à l' axe des x, `supp` comme pointeurs et les moyennes de `len` (longueur moyenne des dents). La fonction `geom_errorbar()` indique où placer les limites inférieures et supérieures des intervalles. Les arguments `size = 5` et `width = .05` sont ajoutés par pur esthétisme. L'argument `.groups = "drop"` de `summarise` permet d'éviter une avertissement expliquant qu'une variable de groupement est utilisé pour regrouper les résultats à la fin. Ajouter ou retirer cet argument ne change pas les calculs.

7.6 Pour aller plus loin

Il existe une multitudes de livres, de sites web, de tutoriels en ligne et d'atelier pour donner l'occasion au lecteur d'aller plus loin dans sa conception graphique.

Voici quelques ouvrages de références : Le *R Graphics Cookbook* (Chang) repérable à https://r-graphics.org/, *ggplot2: elegant graphics for data analysis* (Wickham) repérable à https://ggplot2-book.org/ ou *R Graphics* (Murrel) repérable à https://www.stat.auckland.ac.nz/~paul/RG2e/.

Exercices

1. Prendre le jeu de données `mtcars` et produire un diagramme de dispersion montrant la puissance brute (en chevaux) (`hp`) par rapport à consommation en km/l (basé sur `mpg`) tout en soulignant l'effet du nombre de cylindres (`cyl`). **Attention** la fonction `as_factor` permettra d'utiliser `cyl` en facteur.

2. Prendre le jeu de données `mtcars` et produire un histogramme montrant la variabilité de la consommation `mpg` par rapport à la transmission (`am`). **Attention** la fonction `as_factor` permettra d'utiliser `am` en facteur.

3. Prendre le jeu de données `msleep` et produire un diagramme à bâton pour observer la fréquence de différents type de régime (`vore`). **Attention** aux données manquantes.

4. Prendre le jeu de données `msleep` et produire une boîte à moustache pour observer le temps total de sommeil (`sleep_total`) par rapport aux régimes (`vore`). **Attention** aux données manquantes.

Part III

Statistiques

Chapter 8

Inférer

Le but principal de toute inférence statistique est de tirer des conclusions sur une population à partir d'un échantillon (un fragment beaucoup plus petit de la population). Avant d'introduire différents tests statistiques permettant de tirer ce genre de conclusions, le théorème central limite et la théorie des tests d'hypothèses seront présentés. La distribution centrale réduite (score-z) et la distribution-t serviront d'appui à la présentation.

Comme il est rarement possible de collecter des données sur l'ensemble de la population, l'expérimentateur choisi, idéalement, un échantillon représentatif tiré aléatoirement. Une fois l'échantillon recruté et mesuré, l'expérimentateur dérive des indices statistiques. Un **indice** statistique synthétise par une estimation basée sur l'échantillon de l'information sur le **paramètre** de la population. Cet indice possède un comportement, une distribution d'échantillonnage qui détermine les différentes valeurs qu'il peut prendre. En obtenant ces indices, l'expérimentateur tente de connaître le paramètre de la population. S'il s'intéresse à la relation entre l'anxiété et un cours de méthodes quantitatives, l'expérimentateur voudra savoir d'une part si cette relation n'est pas nulle, mais aussi sa force, en termes de tailles d'effet.

Cette tâche peut apparaître difficile considérant le peu d'informations sur la population, sa distribution de probabilité, les paramètres et la relative petite taille de l'échantillon par rapport à la population. Pour aider l'expérimentateur, les statisticiens ont le théorème central limite. Pour eux, il est certainement l'équivalent de la théorie de l'évolution pour le biologiste ou la théorie de la relativité générale pour le physicien. Ce théorème permet de connaître comment et sous quelles conditions se comportent les variables aléatoires.

8.1 Le théorème central limite

Les valeurs d'un échantillon sont, pour le statisticien, des variables aléatoires. Une variable aléatoire, c'est un peu comme piger dans une boîte à l'aveuglette pour obtenir une valeur. La boîte est impénétrable, personne ne sait par quel processus elle accorde telle ou telle autre valeur. Pour le statisticien, ce qui importe c'est que chaque valeur possède une chance égale aux autres d'être sélectionnée et qu'elles soient indépendantes entre elles (le fait d'en choisir une soit sans conséquence sur la probabilité des autres). Pour le non-initié aux fonctions permettant de créer des nombres pseudoaléatoires, une fonction **R** comme `rnorm()` ou `runif()` (r suivi d'un nom de distribution, voir Les distributions) joue parfaitement le rôle de cette boîte. Si l'usager demande une valeur, la fonction retourne une valeur aléatoire (imprévisible à chaque fois) sans connaître comment cette valeur est produite.

```
runif(n = 1)
#> [1] 0.843
```

Le statisticien s'intéresse à inférer comment ces valeurs sont générées. Il postule ainsi que les valeurs aléatoires suivent une distribution de probabilité. Connaître cette distribution est très important, car c'est elle qui permet de répondre à des questions comme : quelle est la probabilité d'obtenir un résultat aussi rare que x? Ou quelle sont les valeurs attendues pour 95% des tirages? Questions tout à fait pertinentes pour l'expérimentateur. Une des distributions les plus connues est certainement la distribution normale, celle qui est derrière la fonction `rnorm()` d'ailleurs. Mais, il y en a beaucoup, beaucoup d'autres.

Lorsque plus d'une variable sont issues d'une même boîte (distribution), elles sont *identiquement distribuées*. Si ces variables aléatoires sont combinées, que ce soit en termes de produit, de quotient, d'addition, de soustraction, le résultat est une nouvelle variable aléatoire qui possède sa propre distribution nommée *distribution d'échantillonnage*. Sur le plan de la syntaxe **R**, il s'agit de réaliser des opérations mathématiques avec des variables aléatoires identiquement distribuées.

```
# Pour répliquer
set.seed(1)

# Création de deux variables identiquement distribuées
a <- runif(n = 1)
b <- runif(n = 1)
a ; b
#> [1] 0.266
#> [1] 0.372
```

```
# Une nouvelle variable aléatoire
total <- a + b
```

Entre en jeu le *théorème central limite*: plus des variables aléatoires identiquement distribuées sont additionnées ensemble, plus la distribution de probabilité de cette somme se rapproche d'une distribution normale.

Par exemple, la fonction `rlnorm()` génère des variables issues d'une distribution log normale. Elle a la forme illustré à la Figure 8.1 (qui n'a rien de normal à première vue).

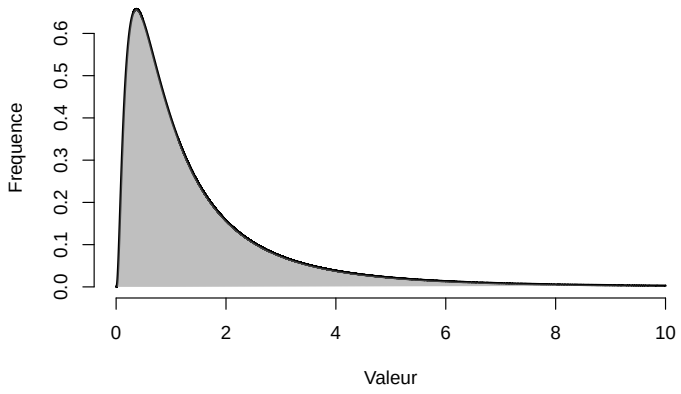

Figure 8.1: Distribution log normale

En calculant la somme de plusieurs variables aléatoires de cette distribution, pour diverses valeurs de tailles d'échantillons (nombre de variables échantillonnées), les résultats tendent de plus en plus vers une distribution normale. Le code ci-dessous présente la démarche utilisée et la figure 8.2 en fait la démonstration graphique en présentant les distributions d'échantillonnage obtenues.

```
# Cette fonction sort les nombres, mais pas les graphiques.
# Différentes tailles d'échantillons
N = seq(10, 90, by = 10)
# Nombre de tirage pour chaque élément de N
reps = 1000

# Une boucle pour tester toutes les possibilités
for(n in N){
  total = as.numeric()
  for(i in 1:reps){
```

```
    # Faire la somme de n valeurs tirés d'une distribution log normale
    total[i] = sum(rlnorm(n))
  }
  # hist(total)
}
```

La Figure 8.2 montre que la distribution d'échantillonnage de la somme des variables converge vers une distribution normale à mesure que la taille d'échantillon n augmente. Cela est vrai pour n'importe quelle distribution de probabilité de la population. Le théorème central en dit plus que simplement la forme de la distribution. Elle affirme également qu'une distribution de probabilité d'une population ayant une moyenne μ et un écart type σ échantillonnées sur n unités, générera une distribution d'échantillonnage des totaux (indicé t) ayant une espérance (la moyenne) de $n\mu_t$ et un écart type de $n\sigma_t^2$.

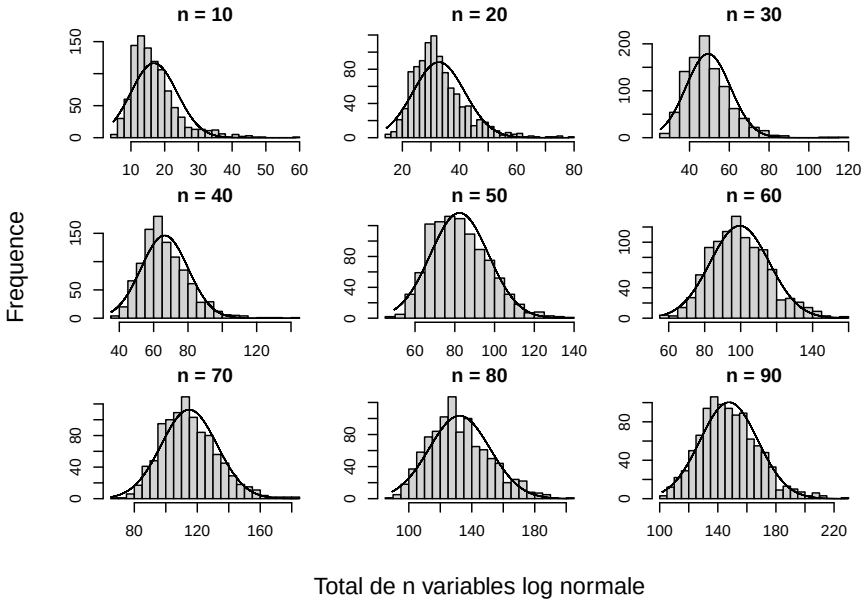

Figure 8.2: Distributions des totaux de n variables log normales

Les expérimentateurs ne connaissent pas les distributions sous-jacentes aux valeurs des unités issues de la population. Par contre, à l'aide des statisticiens et du théorème central limite, ils savent comment se comportent les sommes des variables. Les expérimentateurs s'intéressent toutefois rarement aux sommes de variable… à l'exception de la moyenne qui est une somme de variable divisée par la constante n[1]. Dans le cas de la moyenne, le théorème central limite

[1]Ou encore la variance qui est la somme des écarts au carré. Les expérimentateurs

stipule qu'une distribution de probabilité ayant une moyenne μ et un écart type σ dont l'échantillon est constitué de n unités, générera une distribution d'échantillonnage des moyennes avec une espérance de $\mu_{\bar{x}}$ et un écart type de $\sigma_{\bar{x}}/\sqrt{n}$.

Dans la mesure où l'expérimentateur connaît la distribution de la population (extrêmement rare, mais permet de mieux illustrer la théorie) ou qu'il peut recourir à une distribution d'échantillonnage connue, il pourra inférer la probabilité d'une variable aléatoire par rapport à ce qui est attendu simplement par hasard. Il pourra alors juger si cette variable est trop rare par rapport à l'hypothèse de base (nulle).

La théorie traditionnelle des tests d'hypothèses repose sur l'idée selon laquelle on compare la vraisemblance d'une variable aléatoire estimée auprès d'un échantillon par rapport à une hypothèse nulle (l'absence d'effet). En épistémologie des sciences, il n'est pas possible de montrer l'exactitude d'une hypothèse, seulement son inexactitude. Cela rappelle le principe du falsificationnisme selon lequel on ne peut prouver une hypothèse, on ne peut que la falsifier. En statistiques, c'est la rareté d'une donnée qui agira comme indice d'*inexactitude*. Si la variable aléatoire est trop rare pour l'hypothèse nulle, celle-ci est rejetée : d'autres hypothèses doivent être considérées pour expliquer ce résultat. Autrement, l'hypothèse nulle n'est pas rejetée, les preuves sont insuffisantes pour informer l'expérimentateur sur l'hypothèse nulle.

8.2 Inférence avec la distribution normale sur une unité

Un excellent exemple en sciences humaines et sociales où la distribution de probabilité de la population est connue est le quotient intellectuel (QI). Le QI d'une population occidentale est distribué normalement (établi intentionnellement par les psychométriciens) avec une moyenne de 100 ($\mu = 100$) et un écart type de $\sigma = 15$. Ces valeurs sont totalement arbitraires, il est tout aussi convenable de parler d'une moyenne de 0 et d'un écart type de 1 (la distribution pourrait être standardisée) quoiqu'il est contre-intuitif de parle d'un QI de 0. (Qui voudrait avoir une intelligence de 0?)

Dans la population, bien que la moyenne et la variance peuvent être connues, sélectionner une unité au hasard génère une variable aléatoire. Chaque individu de la population a une probabilité très faible d'être sélectionné et est indépendant des autres individus de la population. Il est très difficile de prédire le score exact d'une personne. Toutefois, il est possible d'avoir un idée de la variabilité des scores. La Figure 8.3 montre la distribution normale par rapport à la moyenne, μ pour différentes valeurs d'écart type, σ. Elle montre que 68.269% devrait se retrouvé entre plus ou moins un écart types ou encore que 95.45%

s'intéressent aux sommes de variables en fait.

devrait se retrouvé entre plus ou moins deux écarts types. Ajuster au QI, il s'agit de 85 à 115 et de 70 à 130 respectivement

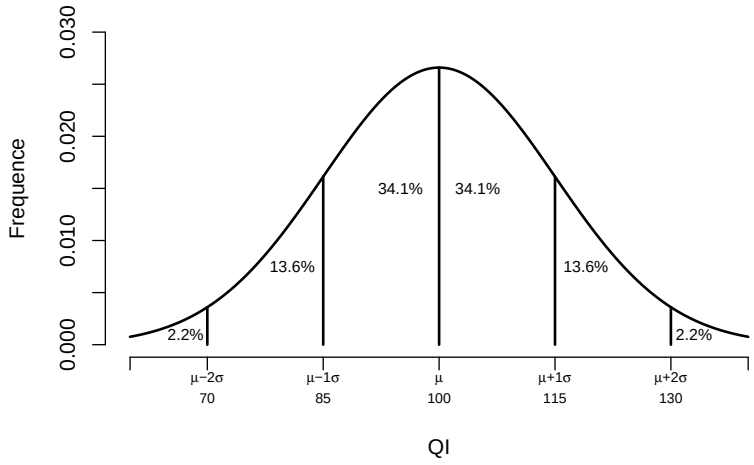

Figure 8.3: La distribution normale du QI

Une autre façon de fonctionner est de prendre une personne au hasard et de mesurer son QI. Le score obtenu est une valeur aléatoire. Comme la distribution est connue avec ses paramètres, il est possible de juger de la vraisemblance de ce score (est-il rare?) par rapport à la population.

Voici un exemple où ces informations sont pertinentes. Un groupe d'*expérimentateurs* mettent en place un outil d'évaluation qui teste si un individu donné est un humain ou un reptilien (une race d'extra-terrestre). Leur outil n'est pas si sophistiqué. En fait, il se base sur le QI, car les expérimentateurs ont remarqué que les reptiliens ont un QI beaucoup plus élevé que le QI humain.

La distribution normale du QI humain joue le rôle d'hypothèse nulle, les personnes mesurées sont admises humaines jusqu'à preuve du contraire (une présomption d'innocence en quelque sorte), un QI *trop* élevé suggérant la culpabilité.

Les expérimentateurs émettent l'hypothèse que les 5 % personnes ayant le plus haut QI sont vraisemblablement reptiliens. C'est le risque qu'ils sont prêts à prendre de sélectionner un humain et de le classer erronément comme reptilien.

Le groupe d'expérimentateurs teste leur instrument sur Fanny. Elle a un QI de 120. Comment tester si elle est reptilienne? La première étape est d'obtenir un score-z. Un score-z est une échelle standardisée des distances d'une valeur par rapport à la moyenne. Lorsqu'une échelle de mesure est transformée en score-z,

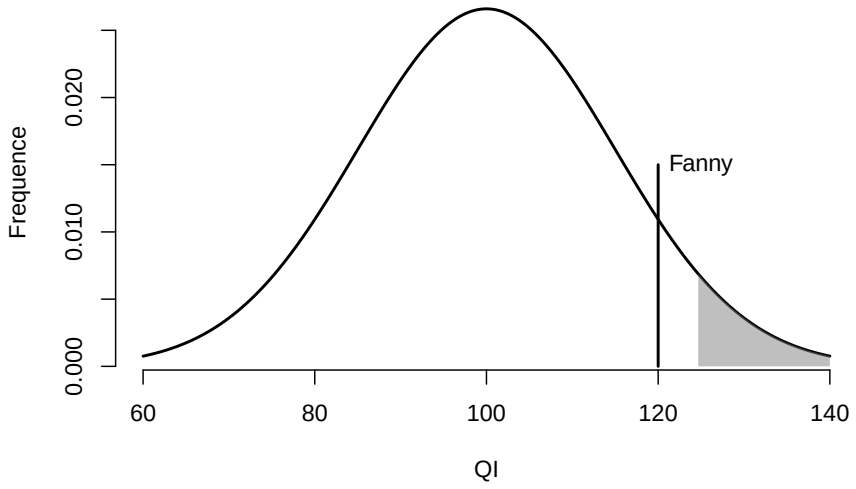

Figure 8.4: Score de Fanny sur la distribution normale

la moyenne est de 0 et l'écart type est égal à 1. Cela permet de mieux apprécier les distances et leur probabilité. La Figure 8.3 est ainsi applicable pour toutes sortes de situations où la distribution est vraisemblablement normale.

```
fanny <- 120
z.fanny <- (fanny - 100) / 15
```

Fanny a un score-z de 1.333. Maintenant, il faut traduire cette valeur en probabilité.

```
# La probabilité que Fanny ait un QI de -Inf à z.fanny
pnorm(z.fanny)
#> [1] 0.909
```

L'expectative sous l'hypothèse nulle est d'observer un score pareil ou supérieur à celui de Fanny 9.121 % du temps. Cette statistique correspond à la *valeur-p*, la probabilité de l'indice par rapport à sa distribution d'échantillonnage (hypothèse nulle). Comme elle ne dépasse pas le seuil de 5%, soit la limite selon laquelle le score est jugé invraisemblable, l'hypothèse nulle n'est pas rejetée (elle est humaine!).

Avec le critère d'identifier erronément les 5% humains les plus intelligents, il s'agit, du même coup, du **taux de faux positif** *acceptable* de l'étude. Un faible sacrifice à réaliser afin identifier des reptiliens parmi les humains. La zone de rejet, c'est-à-dire la zone dans laquelle l'hypothèse nulle (humain) est rejetée, correspond à la zone ombragée à droite de la distribution.

La logique des tests statistiques inférentiels repose sur cette série d'étapes : choisir un indice, connaître sa distribution sous-jacente, déterminer l'hypothèse nulle (généralement l'absence d'effet), calculer la probabilité de l'indice par rapport à cette hypothèse nulle.

8.3 Inférence avec la distribution normale sur un échantillon

Jusqu'à maintenant, seule une unité d'observation était traitée. L'indice et la distribution étaient également spécifiés. Dans cette section, l'exemple est étendu aux échantillons (plus d'une unité d'observation).

Fanny a un QI de 120. Si une autre personne avait été sélectionnée, cette nouvelle personne aurait inévitablement un autre score. Cette logique s'applique également aux échantillons. L'exemple ci-dessous échantillonne 10 unités d'une population de QI distribuée normalement avec les paramètres usuels.

```
# Création d'un échantillon de 10 unités
set.seed(824)

# Dix valeurs arrondies avec une moyenne de 100 et un écart type de 10
QI <- round(rnorm(n = 10, mean = 100, sd = 15))
QI
#>  [1] 110 111 102  99 109 102  99 110 132 114
```

Dans la Figure 8.5, chaque unité est présentée par un trait vertical noir. La moyenne de cet échantillon est de 108.8 et l'écart type est de 9.762. Dans cet exemple, l'indice est clairement identifié, mais quelle est la distribution d'échantillonnage des moyennes? Selon le théorème central limite, la moyenne de la distribution d'échantillonnage est $\mu = \mu_{\bar{x}} = 100$ et l'écart type est $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}} = \frac{15}{\sqrt{10}} = 4.743$. Maintenant, il est possible de calculer un score-z.

$$z = \frac{\bar{x} - \mu_{\bar{x}}}{\sigma/\sqrt{n}} = \frac{108.8 - 100}{15/\sqrt{10}} = 1.855$$

Cela donne le code suivant.

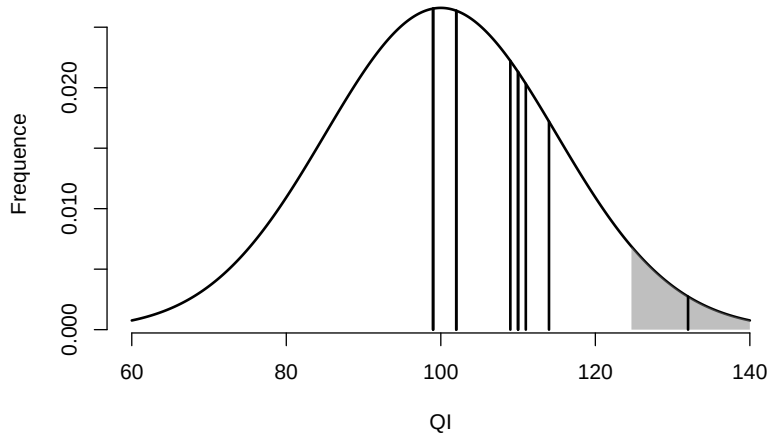

Figure 8.5: Scores des unités de l'échantillon

```
z <- (mean(QI) - 100)/(15 / sqrt(10))
z
#> [1] 1.86
```

La fonction `(1 - pnorm(z)) * 100`, retourne la probabilité (en pourcentage) d'un résultat plus rare que l'indice obtenu auprès de l'échantillon par rapport à la population. Comme pour l'exemple de Fanny, ce chiffre est une valeur-p, soit la probabilité de l'indice observé par rapport à l'hypothèse nulle. La probabilité de cet échantillon par rapport à l'hypothèse nulle est de 3.178%, juste en deçà du 5% fixé. La conclusion est par conséquent de rejeter l'hypothèse nulle, l'échantillon semble provenir d'une autre distribution (avec des paramètres différents) que celle postulée.

Qu'en est-il vraiment de ce résultat? Pour l'expérimentateur, il ne peut aller plus loin, car il ne connaît pas la boîte noire selon laquelle les valeurs de l'échantillon sont générées. Il ne peut que constater que plusieurs (9/10) unités ont un score plus élevé que 100. Par contre, comme il s'agit d'un exemple simulé, la boîte noire est connue. C'est la fonction, `round(rnorm(n = 10, mean = 100, sd = 15))`, une distribution normale ayant $\mu = 100, \sigma = 15$ qui est utilisée pour générer les valeurs. L'utilisateur sait qu'il s'agit d'un faux positif (une erreur de Type I) : l'échantillon fait partie des 5% des échantillons qui risquent de se faire rejeter accidentellement. Si l'utilisateur utilise une autre graine (`seed()`), il verra que la plupart (95%) des moyennes ne seront pas rejetées.

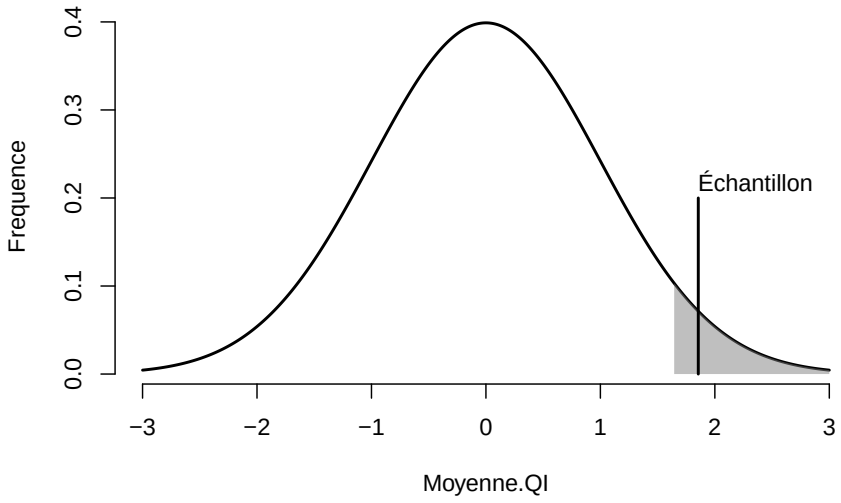

Figure 8.6: Moyenne de l'échantillon sur la distribution normale

Pour l'instant, seule une boîte noire a été examinée - celle de l'hypothèse nulle. Qu'advient-il du vrai phénomène? Par exemple, si les reptiliens existaient vraiment. Comme l'utilisateur est le maître du modèle, il peut spécifier les paramètres à sa convenance. Le QI des reptiliens pourraient être distribué comme une distribution normale ayant $\mu_r = 130, \sigma_r = 15$ où l'indice r ne fait qu'indiquer qu'il s'agit des paramètres de la population reptilienne. Les paramètres humains seront désignés par h, soit $\mu_h = 100, \sigma_h = 15$.

```
mu.h <- 100 ; sd.h <- 15
mu.r <- 130 ; sd.r <- 15
```

La Figure 8.7 présente les distributions de ces populations. Trois zones sont ajoutées pour illustrer les concepts statistiques d'erreur de type I, d'erreur de type II et de puissance statistique. Comme les populations sont connues, ces concepts statistiques sont calculables.

L'erreur de type I (présentée auparavant) représente la probabilité d'émettre un faux positif, souvent représentée par α (alpha). Elle correspond à la probabilité de rejeter l'hypothèse lorsqu'elle est vraie. Dans la Figure 8.7, il s'agit de la zone noire. Elle corresond à conclure qu'un vrai humain est un reptilien (ce qu'il n'est pas). Ce taux est fixé à l'avance par l'expérimentateur, ici, 5%. C'est

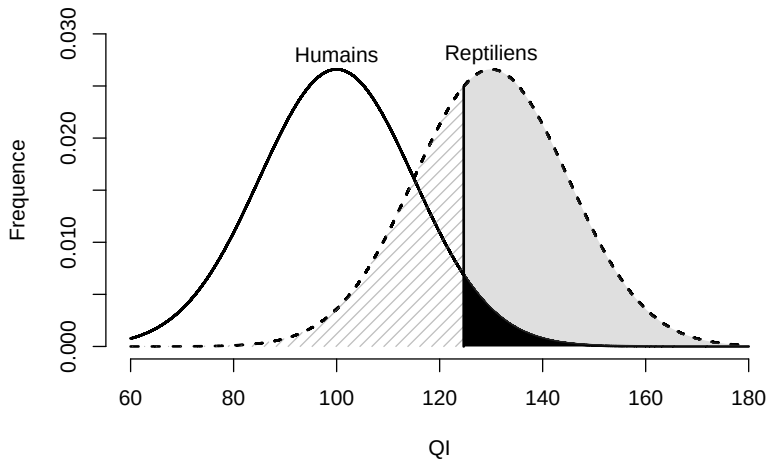

Figure 8.7: Distribution du QI des humains et des reptiliens

le risque qu'il est prêt à prendre. Ainsi, 95% des humains seront correctement identifiés comme humains.

```
# Erreur de type I (fixé à l'avance)
alpha <- .05

# Valeur critique à laquelle l'expérimentateur rejette H0
v.crit <- qnorm(1 - alpha, mean = mu.h, sd = sd.h)
v.crit
#> [1] 125

# Erreur de type II
beta <- pnorm(v.crit, mean = mu.r, sd = sd.r)
beta
#> [1] 0.361

# Puissance statistique
puissance <- 1 - beta
puissance
#> [1] 0.639
```

La zone hachurée de la Figure 8.7 correspond à l'**erreur de type II**, souvent représentée par β (beta), soit la probabilité de *ne pas rejeter* l'hypothèse nulle. Autrement dit, c'est la probabilité de ne pas trouver l'effet lorsque celui-ci est

vrai. Pour cet exemple, il s'agit d'un reptilien assez sournois (avec un QI suffisamment faible pour sa population) qu'il passe inaperçu auprès des humains, ou, en d'autres termes, de conclure qu'un vrai reptilien est un humain (ce qu'il n'est pas). Cette probabilité est estimée à 36.124%, donc 36.124% des reptiliens passeront inaperçus.

La **puissance**, la zone grise de la Figure 8.7, correspond à la probabilité de rejeter correctement l'hypothèse nulle. Il s'agit de rejeter l'hypothèse nulle lorsqu'elle est fausse. Cela signifie d'identifier un vrai reptilien correctement. C'est exactement ce que le groupe de chercheur désire réaliser. Elle est estimée à 63.876% en ne prenant qu'une mesure de QI. La puissance est l'une des statistiques qui intéressent le plus l'expérimentateur *avant* de réaliser son étude, car elle donne une approximation sur la probabilité de trouver un résultat significatif. Par contre, les expérimentateurs ne connaissent que rarement les paramètres de la vraie distribution de l'effet qu'il désire trouver. En langage statistique, il s'agit un paramètre de non-centralité (ou de décentralisation) ou `ncp` en syntaxe **R**. L'expérimentateur recourt alors à d'habiles approximations éclairées basées sur leurs connaissances du phénomène et en regard des études déjà publiées sur le sujet (ou un sujet similaire), s'il y en a. Ces estimations souffrent tout de même d'être des variables aléatoires (elles respectent elles aussi une distribution d'échantillonnage), et non les paramètres recherchés.

Avant de procéder davantage, une dernière notion est importante à présenter : la direction du test. Dans tous les exemples précédents, l'hypothèse nulle était rejetée si une valeur plus rare était obtenue. Il s'agit d'un test d'hypothèse **unilatérale** (d'un seul côté). Cela facilite grandement la présentation de certains concepts et calculs. Un test peut être unilatérale inférieure, trouver un résultat plus faible qu'une valeur critique; ou unilatérale supérieure, trouver un résultat plus élevé qu'une valeur critique; ou encore **bilatérale**, soit trouver un résultat moins élevée ou plus élevée que des valeurs critiques.

Dans le cas d'un test bilatéral, l'erreur de type I est divisée par 2, $\alpha/2$ pour couvrir l'espace des deux côtés de la distribution. L'espace total est identique même s'il est divisé sur les deux extrêmes. Cela implique des valeurs critiques plus élevées que si le test n'était que d'un côté et par conséquent une puissance statistique un peu plus faible, car l'effet est directionnel. La probabilité de rejeter l'hypothèse dans une direction a été diminuée pour tenir compte de l'autre côté. La Figure 8.8 montre un exemple de chacun de ces types de tests pour $\alpha = .05$.

Le choix entre unilatérale inférieure, supérieure ou bilatérale repose essentiellement sur la question de recherche de l'expérimentateur. Qu'attend-il ou que veut-il savoir du résultat? L'expérimentateur recourt généralement au test bilatéral afin d'identifier des résultats allant pour son hypothèse ou à l'opposé de son hypothèse. Cela permet notamment d'identifier des devis qui pourraient nuire aux participants. Par exemple, si un expérimentateur développe une intervention pour réduire la consommation de drogues chez les adolescents et n'est concerné que par la probabilité que l'intervention soit efficace, il pourrait man-

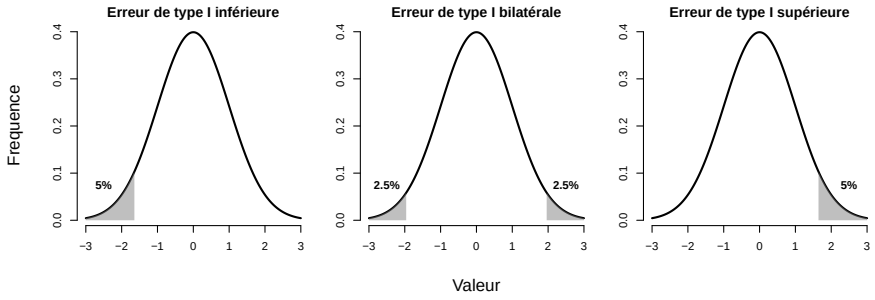

Figure 8.8: Illustration de l'erreur de type I

quer l'effet délétère d'une telle intervention, soit que l'intervention augmente la consommation de drogue. Il est certain que l'expérimentateur souhaite être rapidement mis au courant si ces résultats se dessinent.

8.4 La distribution-t

Dans la plupart des cas, l'expérimentateur ne connaît pas la variance de la population. Il recourt alors à la meilleure estimation qui lui est disponible, celle obtenue auprès de l'échantillon. De recourir à une estimation au lieu de la vraie valeur cause un réel problème: l'estimation est une variable aléatoire respectant une distribution d'échantillonnage. La distribution des variances montre une légère asymétrie positive plus flagrante pour les petites tailles d'échantillon, car les variances sont distribuées en distribution-χ^2.

Une petite simulation faite à partir d'une distribution normale standardisée par rapport à trois tailles d'échantillon et dont la variance est estimée 10^4 fois est présentée dans la Figure 8.9. Elle montre que l'estimation n'est pas biaisée (elles sont toutes les trois centrées à 1), mais relève l'asymétrie en question pour les petites tailles d'échantillon qui tend à décroître à mesure que n augmente. Autrement dit, une estimation unique (pris d'un échantillon) est plus susceptible d'être sous-estimée. Ne pas tenir compte de cet aspect augmente indûment les valeurs obtenues (à cause du dénominateur plus petit). Ainsi, il est inadéquat d'utiliser une distribution normale lorsque la variance est inconnue. Il faudra plutôt opter pour la **distribution-t** qui, elle, tient compte de la variabilité de l'estimation de la variance.

La distribution-t est symétrique, comme la distribution normale, mais a des queues plus larges pour tenir compte de la surestimation des valeurs dû à la sous-estimation de la variance. La distribution-t tend vers une distribution normale lorsque n augmente, ce qui est illustré à la Figure 8.10. La ligne pleine noire montre la distribution pour 5 unités, les deux autres lignes, traits et pointillés,

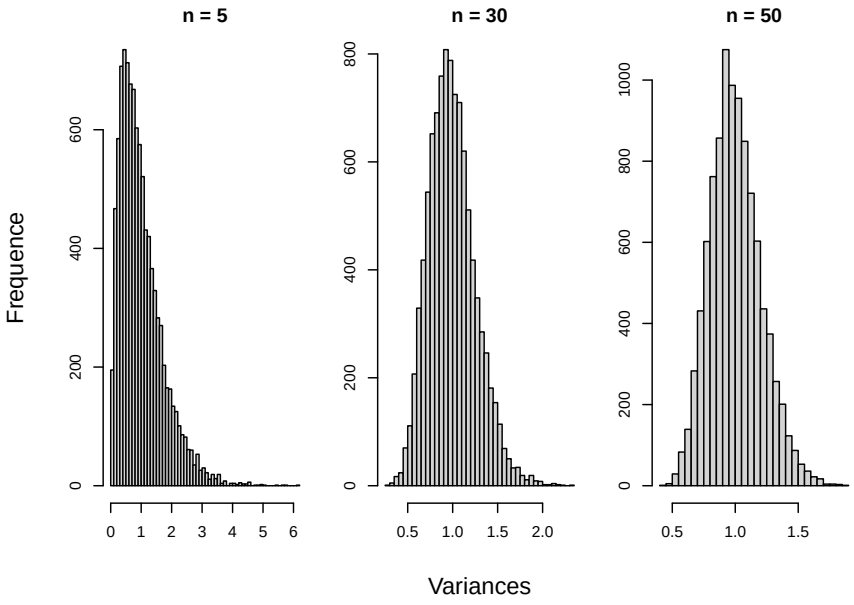

Figure 8.9: Distribution de la variance en fonction de la taille d'échantillon

pour 30 et 1000 unités respectivement. Il est difficile de distinguer ces deux dernières.

En ayant recours à des échantillons et une estimation de la variance (la variance est inconnue de l'expérimentateur, ce qui est généralement le cas), il faudra procéder avec la distribution-t. La procédure est la même que celle utilisée avec les scores-z précédemment, la différence étant que la distribution-t est utilisée au lieu de la distribution normale, car l'écart type est estimé.

8.5 Le test-t à échantillon unique

Le test-t à échantillon unique permet de tester la moyenne d'échantillon par rapport à une valeur arbitraire μ_0 (généralement $\mu_0 = 0$) qui joue le rôle d'hypothèse nulle.

$$t_{n-1} = \frac{\bar{x} - \mu_0}{(s/\sqrt{n})}$$

La distribution-t a un degré de liberté (l'indice de t dans l'équation) qui lui est associé et qui est fixé à $dl = n - 1$, la taille d'échantillon moins 1. Un

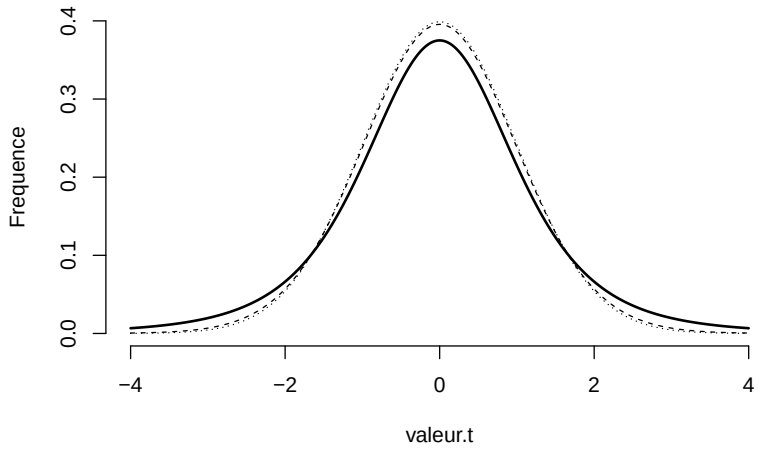

Figure 8.10: Comparaison d'une distribution normale à deux distributions-t

degré est perdu à cause de l'estimation de l'écart type de l'échantillon. Qu'en est-il de la probabilité d'obtenir cette moyenne? En recourant à la distribution intégrée de **R**, `pt()` il est possible d'obtenir la probabilité d'une valeur-t par rapport à l'hypothèse nulle. La fonction nécessite une valeur-t et le degré de liberté associé, p. ex., `pt(t = , df = n - 1)`. La valeur produite donne la probabilité d'obtenir un score plus petit jusqu'à la valeur-t. Une astuce permet de calculer aisément la probabilité lorsque la distribution d'échantillonnage est symétrique. Utiliser une valeur absolue permet de considérer les deux côtés de la distribution simultanément. Les valeurs-t négatives sont alors positives. Comme un côté est supprimé, l'espace positif est doublé, le code pour tenir compte de cet astuce est : `(1 - pt(abs(t), df = n - 1)) * 2` où `t` est la valeur-t obtenue.

```
testt <- function(x, mu = 0){
  # x est une variable continue
  # mu est une moyenne à tester comme hypothèse nulle(H0)
  xbar <- mean(x)
  sdx <- sd(x)
  n <- length(x)
  vt <- (xbar - mu) / (sdx / sqrt(n))
  dl <- n - 1
  vp <- (1 - pt(abs(vt), df = dl)) * 2
  statistique <- list(valeur.t = vt,
                      dl = dl,
```

```
                      valeur.p = vp)
  return(statistique)
}
```

Une fois la fonction créée, il est possible de la tester en la comparant avec la fonction **R** de base `test.t()`. Le code suivant crée un échantillon de 30 unités avec une moyenne de 1 et un écart type de 1.

```
# Un exemple de jeu de données
set.seed(20)
x = rnorm(n = 30, mean = 1, sd = 1)
# Vérification de la fonction maison
testt(x)
#> $valeur.t
#> [1] 3.69
#>
#> $dl
#> [1] 29
#>
#> $valeur.p
#> [1] 0.000919
# Comparer avec la fonction R
t.test(x)
#>
#>  One Sample t-test
#>
#> data:  x
#> t = 4, df = 29, p-value = 9e-04
#> alternative hypothesis: true mean is not equal to 0
#> 95 percent confidence interval:
#>  0.31 1.08
#> sample estimates:
#> mean of x
#>     0.696
```

Qu'est-ce que ces résultats signifient? La Figure 8.11 montre la distribution d'échantillonnage de l'hypothèse nulle, c'est-à-dire l'hypothèse selon laquelle la moyenne de la population est égale à 0. Les zones grises montrent les zones de rejet. Si une valeur-t se retrouve dans ces zones, elle est jugée comme *trop rare*, il faut alors rejeter l'hypothèse nulle. C'est ce qui s'est produit ici. Rien de surprenant, l'échantillon est tiré d'une distribution normale avec une moyenne de 1 et un écart type de 1.

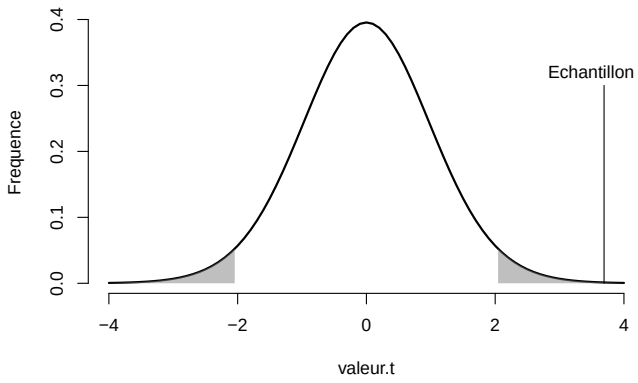

Figure 8.11: Valeur-t de l'échantillon sur la distribution-t

8.6 Critiques des tests d'hypothèses

(Cette section est en construction.)

Le présent ouvrage ne couvre que l'aspect traditionnel ou classique des tests d'hypothèse. Cette approche est remise en question depuis 1950 jusqu'à aujourd'hui. D'excellents ouvrages couvrent les failles et solutions des tests d'hypothèses classiques de façon plus approfondie qu'il ne l'est fait ici.

- L'hypothèse nulle n'est jamais susceptible d'être vraie.
- La conclusion statistique n'est pas celle désirée par l'expérimentateur.
- La significativité statistique n'implique pas la significativité pratique.
- Les expérimentateurs interprètent incorrectement la valeur-p.
- Une valeur-p faible n'est pas garantie de la reproductibilité.
- La dichotomisation de la preuve $p <= \alpha$.
- Une saine utilisation de la valeur-p n'accomplit pas grand-chose.
- L'obsession de la valeur-p cause plus de problèmes qu'elle n'en résout : p-hacking, triturage de données, abus de variables, comparaison multiple, biais pour les hypothèses originales et surprenantes.

Chapter 9

Analyser

En continuation de l'introduction des théories des tests d'hypothèses (voir Inférer) et l'aperçu donnée par le test-t à échantillon unique, cette section poursuit la présentation en introduisant des analyses statistiques de bases comme les différences de moyennes, l'association linéaire et les tests pour données nominales. Les tests-t indépendant et dépendant, l'analyse de variables, la covariance, la corrélation ainsi que le test du $\chi 2$ pour table de contingence sont présentées.

9.1 Les différences de moyennes

9.1.1 Test-*t* indépendant

En général, l'expérimentateur ne s'intéresse pas à comparer une moyenne à une valeur arbitraire, comme c'était le cas avec le test-t à échantillon unique. Cela peut lui être assez trivial. Il s'intéresse plutôt à comparer une moyenne à une autre moyenne, soit une différence entre deux groupes indépendants, par exemple, quel est la différence entre un groupe traitement et un groupe contrôle?

En se basant sur le test t à échantillon unique, la valeur-t pour deux moyennes se calcule selon l'équation (9.1)

$$t_{n-2} = \frac{\bar{x_1} - \bar{x_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{9.1}$$

où l'indice de la valeur-t est le nombre de degrés de liberté $n-2$. Comme deux variances sont estimées, deux degrés de libertés sont imputés, ce qui octroi $n-2$ degrés. En plus de considéré $\bar{x_2}$ la valeur *arbitaire* de comparaison (l'ordre de $\bar{x_2}$ et $\bar{x^1}$ est arbitraire), les deux écart types sont également considérés au dénominateur.

Une fois le calcul réalisé, la logique du test d'hypothèse est la même, à l'exception de l'hypothèse nulle qui correspond maintenant à l'absence de différence entre les deux moyennes.

Voici un exemple de programmation du test-t pour deux groupes indépendants.

```
testt.ind <- function(x1, x2){
  # x1 est une variable continue associée au groupe 1
  # x2 est une variable continue associée au groupe 2

  # Calcul des moyennes
  x1bar <- mean(x1) ; x2bar <- mean(x2)

  # Calcul des variances
  x1var <- var(x1) ; x2var <- var(x2)

  # Calcul des tailles d'échantillon
  nx1 <- length(x1) ; nx2 <- length(x2)

  # Valeur-t, degrés de liberté et valeur-p
  vt <- (x1bar - x2bar) / sqrt(x1var / nx1 + x2var / nx2)
  dl <- nx1 + nx2 - 2
  vp <- (1 - pt(abs(vt), df = dl)) * 2
  statistique <- list(valeur.t = vt, dl = dl, valeur.p = vp)
  return(statistique)
}
```

Pour générer un exemple de données, le code ci-après crée un échantillon de 15 unités réparties en deux groupes, le premier groupe (`gr0`) est tiré d'une distribution normale ayant une moyenne de 1 et un écart type de 1, le deuxième groupe (`gr1`), une moyenne de 0 et un écart type de 1. La syntaxe illustre la création de deux variables pour créer les deux groupes. Il est aussi possible d'envisager la création sur en termes d'équation linéaire,

$$y = \mu_0 + \mu_1 x_1 + \epsilon$$

où y est le score observé des unités et les autres variables construisent ce score, μ_0 correspond à la moyenne du groupe référent de population (le groupe "contrôle" en quelque sorte.), μ_1 réfère à la différence de moyenne entre les deux groupes identifiés par x_1 qui réfère à l'assignation au groupe, soit 0 pour le groupe "contrôle" et 1 pour le groupe "différent". Pour ce même exemple, $\mu_1 = -1$. Par le produit $\mu_1 x_1$, le groupe "contrôle" associé à la valeur 0 n'a pas de modification de la moyenne, $-1 * 0 = 0$ alors le groupe "différent" associé à la valeur 1, $-1 * 1 = -1$. Enfin, ϵ correspond à la variabilité entre les unités. Cette façon de programmer la création des variables illustre bien l'association linéaire qui existe même dans les différences de moyennes et sera très utile pour des modèles plus compliqués.

```
# Un exemple de jeu de données programmé de deux façons
# Méthode 1
set.seed(2021)
gr0 <- rnorm(n = 15, mean = 1, sd = 1)
gr1 <- rnorm(n = 15, mean = 0, sd = 1)

# Méthode 2
set.seed(2021)
x1 <- c(rep(0, 15), rep(1, 15)) # Appartenance au groupe (0 et 1)
e <- rnorm(n = 30, mean = 0, sd = 1) # Erreur interindividuel
mu1 <- -1 # le deuxième groupe a une différence de moyenne de -1
mu0 <- 1 # moyenne du groupe référent
y <- mu0 + mu1 * x1 + e
# Présenter quelques participants; retirer les crochets pour voir tous
(cbind(method1 = c(gr0, gr1), method2 = y))[10:20,]
#>       method1 method2
#>  [1,]  2.7300  2.7300
#>  [2,] -0.0822 -0.0822
#>  [3,]  0.7272  0.7272
#>  [4,]  1.1820  1.1820
#>  [5,]  2.5085  2.5085
#>  [6,]  2.6045  2.6045
#>  [7,] -1.8415 -1.8415
#>  [8,]  1.6233  1.6233
#>  [9,]  0.1314  0.1314
#> [10,]  1.4811  1.4811
#> [11,]  1.5133  1.5133
```

Les données sont identiques.

Une fois la fonction créée, il est possible de la tester et de la comparer avec la fonction **R** de base `test.t()`.

```
# Vérification de la fonction maison
testt.ind(gr0, gr1)
#> $valeur.t
#> [1] 3.4
#>
#> $dl
#> [1] 28
#>
#> $valeur.p
#> [1] 0.00205
# Comparer avec la fonction R
t.test(gr0, gr1, var.equal = TRUE)
#>
```

```
#>  Two Sample t-test
#>
#> data:  gr0 and gr1
#> t = 3, df = 28, p-value = 0.002
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  0.54 2.18
#> sample estimates:
#> mean of x mean of y
#>     1.332    -0.029
```

Les équations et codes précédents ne sont adéquats que si les variances sont égales entre les deux groupes. Cette notion est quelque peu trahie par la spécification dans la fonction **R** de l'argument `var.equal = TRUE` qui par défaut est `FALSE`. Pour un test-t indépendant suivant les règles de l'art, il faut appliquer une correction (approximation de Welsh) aux degrés de liberté. Les degrés de liberté deviennent moins élégants, le code devient ainsi.

$$dl = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}$$

En apportant cette correction au code initial.

```
testt.ind2 <-function(x1, x2){
  # x1 est une variable continue associée au groupe 1
  # x2 est une variable continue associée au groupe 2
  x1bar <- mean(x1) ; x2bar <- mean(x2)
  x1var <- var(x1) ; x2var <- var(x2)
  nx1 <- length(x1) ; nx2 <- length(x2)

  # erreur type
  et <- sqrt(x1var / nx1 + x2var / nx2)
  # valeur t
  vt <- (x1bar - x2bar) / et
  # correction des degrés de liberté
  dl <- et^4 /
    ((x1var^2 / (nx1^2 * (nx1 - 1))) + (x2var^2 / (nx2^2 * (nx2 - 1))))
  # valeur-p
  vp <- (1 - pt(abs(vt), df = dl)) * 2
  statistique <- list(valeur.t = vt, dl = dl, valeur.p = vp)
  return(statistique)
}
```

Le voici comparé la fonction **R** de base.

```
t.test(gr0, gr1) # Absence de l'argument var.equal = TRUE
#>
#>  Welch Two Sample t-test
#>
#> data:  gr0 and gr1
#> t = 3, df = 27, p-value = 0.002
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  0.539 2.182
#> sample estimates:
#> mean of x mean of y
#>     1.332    -0.029
testt.ind2(gr0, gr1)
#> $valeur.t
#> [1] 3.4
#>
#> $dl
#> [1] 26.9
#>
#> $valeur.p
#> [1] 0.00213
```

Les sorties sont identiques.

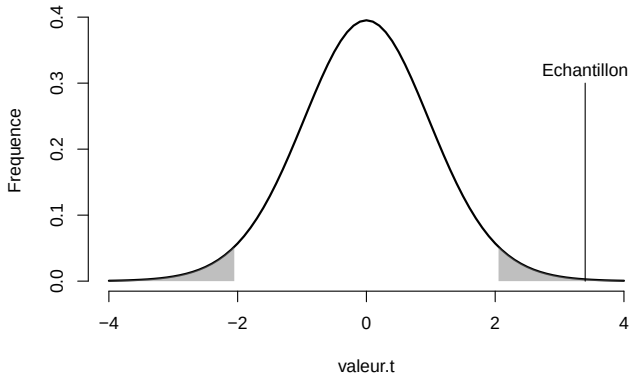

Figure 9.1: Valeur-t de la différence de moyenne sur la distribution-t

La Figure 9.1 illustre où se situe la moyenne de l'échantillon par rapport à la distribution d'échantillonnage de l'hypothèse nulle. Comme la valeur se retrouve dans la zone de rejet ou, de façon équivalente, la valeur-p est plus petite que la valeur α fixé (ici à 5%), on rejette l'hypothèse nulle, il y a vraisemblablement

une différence entre les groupes, ce qui était l'intention derrière la création des données.

9.1.2 Test-t dépendant

Un dernier test-t a présenté est le test permettant de comparer deux de temps de mesure sur les mêmes participants. L'hypothèse désirée est de constater si $\mu_1 = \mu_2$, soit la moyenne du temps 1 est égale à la moyenne du temps 2. Une habile manipulation mathématique permet de poser cette hypothèse en hypothèse nulle, $\mu_1 - \mu_2 = 0$. Il est intéressant de noter (ou de rappeler) que la différence entre deux variables est normalement distribuée si la variance est connue ou distribuée en t si la variance est inconnue.

Ce test est utile lorsqu'il faut tester si les participants se sont améliorés ou détériorés entre deux temps de mesure. Pour calculer la valeur-t,

$$t_{dl_1} = \frac{\bar{x_1} - \bar{x_2}}{\sigma_d / \sqrt{n}}$$

avec $n - 1$ degrés de liberté à cause de l'estimation de l'écart type. Les étapes subséquentes sont identiques au test-t à groupe unique.

```
testt.dep <- function(temps1, temps2){
  # temps est une variable continue mesurée
  # à deux occasion auprès des mêmes participants
  difference <- temps1 - temps2
  dbar <- mean(difference)
  dvar <- var(difference)
  n <- length(difference)
  vt <- (dbar) / sqrt(dvar / n)
  dl <- n - 1
  vp <- (1 - pt(abs(vt), df = dl)) * 2
  statistique <- list(valeur.t = vt, dl = dl, valeur.p = vp)
  return(statistique)
}
```

Pour créer le jeu de données, les étapes sont similaires au test-t indépendant pour des temps de mesure indépendants (sans corrélation) où seules les populations des temps de mesure sont définis. Il est possible de spécifier une corrélation entre les deux de mesures. Cela sera introduit ultérieurement.

Dans ce jeu de données, 25 personnes sont mesurées à deux temps de reprises. Il n'y a pas de corrélation entre les temps de mesure. La variance des temps de mesure est de 1. La différence de moyenne est de 3.

```
# Un exemple de jeu de données
set.seed(148)
temps1 <- rnorm(n = 25, mean = 0, sd = 2)
temps2 <- rnorm(n = 25, mean = 3, sd = 2)
```

La fonction base de **R** est encore `t.test()`, mais il faudra spécifier l'argument `paired = TRUE` pour commander un test apparié (une autre appellation pour un test-t dépendant).

```
t.test(temps1, temps2, paired = TRUE)
#>
#>  Paired t-test
#>
#> data:  temps1 and temps2
#> t = -3, df = 24, p-value = 0.008
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  -3.268 -0.552
#> sample estimates:
#> mean of the differences
#>                   -1.91
testt.dep(temps1, temps2)
#> $valeur.t
#> [1] -2.9
#>
#> $dl
#> [1] 24
#>
#> $valeur.p
#> [1] 0.0078
```

Les sorties sont identiques.

La Figure 9.2 montre où se situe la différence de moyenne par rapport à la distribution d'échantillonnage de l'hypothèse nulle. Comme la valeur se retrouve dans la valeur-p est plus petite que la valeur α fixée (ici à 5%), on rejette l'hypothèse nulle, il y a vraisemblablement une différence entre les temps de mesure, ce qui était l'intention derrière la création des données.

9.1.3 L'analyse de variance à un facteur

L'analyse de variance, souvent appelée ANOVA (*ANalysis Of VAriance*), permet de comparer une variable continue sur plusieurs groupes ou traitements. Il s'agit d'une extension du test-t indépendant qui compare une variable continue auprès de deux groupes. À plus de deux groupes, l'analyse de variance entre en

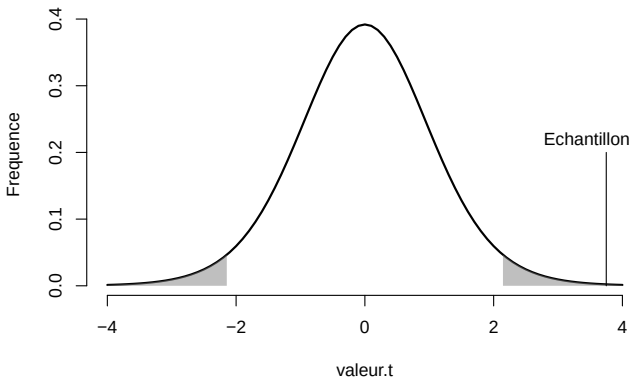

Figure 9.2: Valeur-t de moyenne de différences sur la distribution-t

jeu. **Attention!** Cette section se limite au cas où les groupes sont de tailles identiques (même nombre de participants par groupe, le symbole n_k pour référer à cette quantité) afin d'en simplifier la présentation.

9.1.3.1 La logique de l'analyse de variance

Comme il a été présenté dans la section sur le test-t indépendant, le modèle sous-jacent à l'analyse de variance à un facteur est une extension de ce modèle pour k groupes. Chaque groupe est associé à une différence de moyennes μ_i, pour $i = 1, 2, ...k$ par rapport à la moyenne groupe référent, μ_0. Les variables x_i définissent l'appartenance au groupe i par la valeur 1 et 0 pour les autres groupes (codage factice ou *dummy coding*).

$$y = \mu_0 + \mu_1 x_1 + \mu_2 x_2 + ... + \mu_k x_k + \epsilon$$

L'hypothèse nulle est la suivante.

$$\sigma_1^2 = \sigma_2^2 = ... = \sigma_k^2$$

L'analyse de variance est un test omnibus (global) qui ne teste pas où est la différence, mais bien *s'il y a au moins une différence* entre les groupes. L'hypothèse nulle peut s'avérer fausse de plusieurs façons. Il peut y avoir une ou plusieurs inégalités pour rejeter l'hypothèse nulle.

La logique sous-jacente est basée sur l'idée selon laquelle les moyennes des groupes proviennent d'une même population. Elle compare les hypothèses suivantes : l'hypothèse nulle : les données (les moyennes) proviennent d'une même

population; et l'hypothèse opposée : les données ne proviennent pas d'une même population.

Elle suggère ainsi deux types de variances (ou carré moyen, CM, dans ce contexte) : le CMI, carré moyen intergroupe, lorsque l'hypothèse nulle est vraie (la variabilité des moyennes); et le CMR, carré moyen résiduel, lorsque l'hypothèse nulle n'est ni vraie, ni fausse (la variabilité des données).

Pour calculer le CMI, la variance des moyennes des groupes est

$$s^2_{\bar{x}} = \sum_{i=1}^{k} \frac{(\bar{x}_i - \bar{\bar{x}})^2}{k-1}$$

où $\bar{\bar{x}}$ est la grande moyenne (la moyenne de toutes les unités). Cette statistique se retrouve sur la plan des distributions d'échantillonnage (comment les moyennes se distribuent). Il faut ainsi multiplier cette valeur par n_k, le nombre d'unités par groupes, pour obtenir une estimation de la variance de la population. Ainsi, $\text{CMI} = s^2_{\bar{x}} n_k$.

Il y a plusieurs méthodes pour calculer le CMR, comme le CMR est envisagé comme la moyenne des variances de groupes, il est pratique d'en référer le calcul à sa définition.

$$\text{CMR} = \bar{s^2} = \frac{1}{k} \sum_{i=k}^{k} \sigma^2_k$$

Lorsque l'hypothèse nulle est vraie, les variances des groupes sont traitées comme le résultat d'une étude sur une même population, comme si k petites études avaient été réalisées. L'estimation du CMI est de l'ordre d'une distribution d'échantillonnage connue par le théorème central limite définissant le comportement des moyennes lorsqu'elles proviennent d'une même population (c'est ce que pose comme hypothèse l'analyse de variance). Toutefois, la valeur du CMI est relative au CMR, la quantité de bruit dans les données.

L'analyse de variance pose alors la question : la variance attribuable aux différentes moyennes des groupes est-elle supérieure à la variance résiduelle? Pour tester cette question, une possibilité est de tester le ratio $\frac{\text{CMI}}{\text{CMR}}$, ce qui donne une valeur-F. Le ratio de deux variances suit une distribution-F_{dl_1, dl_2} avec deux degrés de liberté différents qui lui sont associés. Autrement dit, la distribution-F est la distribution d'échantillonnage du ratio deux variances.

$$F_{dl_1, dl_2} = \frac{n_k s^2_{\bar{x}}}{\bar{s^2}} = \frac{\text{CMI}}{\text{CMR}}$$

où $dl_1 = k - 1$, et $dl_2 = k(n_k - 1)$. L'interprétation du ratio est ainsi :

- si CMI > CMR, alors CMI/CMR > 1;

- si CMI = CMR, alors CMI/CMR = 1;
- si CMI < CMR, alors CMI/CMR < 1.

Plus les moyennes sont variables, plus la valeur du CMI est élevée. Plus les unités sont variables, plus le CMR est élevé. S'il existe au moins une différence entre les groupes, le ratio sera en faveur du CMI. En fait, plus F est élevée, toutes autres choses étant égales (les degrés de libertés, p. ex.), plus la probabilité de rejeter l'hypothèse nulle sera grande. Ainsi, lorsque, la valeur-F est obtenue, il est possible de calculer une valeur-p, et le test d'hypothèse revient au même qu'auparavant.

9.1.3.2 La création de données

Dans un jeu de données commun, une variable désignera les groupes, et une autre leur score sur une certaine mesure. La syntaxe suivante montre trois façons pour créer une variable facteur.

```
nk <- 5 # nombre d'unités par groupe
k <- 4  # nombre de groupes
# En ordre
groupe1 <- rep(1:k, each = nk)
# En alternance
groupe2 <- rep(1:k, times = nk)
# Aléatoire
set.seed(765)
groupe3 <- sample(x = 1:k, size = (k * nk), replace = TRUE)
cbind(groupe1, groupe2, groupe3)
#>       groupe1 groupe2 groupe3
#>  [1,]       1       1       4
#>  [2,]       1       2       2
#>  [3,]       1       3       4
#>  [4,]       1       4       3
#>  [5,]       1       1       3
#>  [6,]       2       2       4
#>  [7,]       2       3       3
#>  [8,]       2       4       1
#>  [9,]       2       1       4
#> [10,]       2       2       3
#> [11,]       3       3       3
#> [12,]       3       4       2
#> [13,]       3       1       3
#> [14,]       3       2       3
#> [15,]       3       3       2
#> [16,]       4       4       4
#> [17,]       4       1       4
```

```
#> [18,]       4       2       2
#> [19,]       4       3       1
#> [20,]       4       4       1
```

Il est aussi possible de remplacer les arguments, `1:k` par des chaînes de caractères (des catégories au lieu de nombres).

```
categorie <- c("char", "chat", "cheval", "chevalier", "chien")
nk <- 2
groupe <- as.factor(rep(categorie, each = nk)) # Déclarer comme facteur
groupe
#>  [1] char      char      chat      chat      cheval
#>  [6] cheval    chevalier chevalier chien     chien
#> Levels: char chat cheval chevalier chien
```

Une bonne pratique dans le contexte des comparaisons de moyenne est de déclarer les variables catégorielles comme facteur avec `as.factor()`.

Pour créer des valeurs à ces catégories, une stratégie simple est de créer des valeurs pour chacun des groupes et de les combiner.

```
set.seed(2602)
char      <- rnorm(n = nk, mean = 15, sd = 4)
chat      <- rnorm(n = nk, mean = 20, sd = 4) # Différences ici
cheval    <- rnorm(n = nk, mean = 10, sd = 4) # et ici
chevalier <- rnorm(n = nk, mean = 15, sd = 4)
chien     <- rnorm(n = nk, mean = 15, sd = 4)
# Combinés
score <- round(c(char, chat, cheval, chevalier, chien))
# Conserver toutes les informations en un jeu de données
donnees <- data.frame(groupe, score)
head(donnees)
#>   groupe score
#> 1   char    14
#> 2   char    12
#> 3   chat    17
#> 4   chat    21
#> 5 cheval    14
#> 6 cheval    10
```

9.1.3.3 Dummy coding

La plupart du temps, les variables de regroupement, les variables identifiant l'appartenance aux groupes, sont construites avec une variable de type facteur, c'est-à-dire une colonne avec différentes valeurs ou libellés. C'est d'ailleurs ce qui

a été fait dans l'exemple précédent. Cette méthode d'identification de groupement implique une programmation plus intensive, surtout pour la création de valeurs. En termes de programmation, il est plus élégant de recourir à une fonction de codage factice. Cela permettra de représenter fidèlement le modèle sous-jacent. Étrangement, il n'y a pas de fonction de base avec **R** pour du codage factice. Une fonction maison permettra d'automatiser la réassignation des groupes (en une seule variable) sur plusieurs variables désignant leur appartenance.

```
dummy.coding <- function(x){
  # Retourne un codage factice de x
  # avec les facteurs en ordre alphabétique

  tab <- table(x)    # Extraire une table
  k <- length(tab)   # Nombre de groupes
  n <- length(x)     # Nombre d'unités
  X <- matrix(0, nrow = n, ncol = k) # Création d'une nouvelle variable
  xlev <- as.factor(x)
  for (i in 1:n) {
    X[i, xlev[i]] <- 1 # Identification des groupes
  }
  colnames(X) <- names(tab)
  return(X)
}
```

La fonction maison retourne un codage factice pour les k groupes. **Attention**, le codage factice est fait en ordre alphabétique. Comme il est redondant d'avoir k groupes (identifier k groupes nécessite $k-1$ variables), un groupe référent est désigné. Ce dernier aura 0 sur tous les scores. Pour retirer un groupe, l'utilisation des crochets et une valeur négative associés à la colonne du groupe référent feront l'affaire, p. ex. `dummy.coding(x)[,-k]` déclarera le k^{e} groupe comme le groupe référent.

```
# Facteur de groupe
set.seed(2602)
k <- length(categorie)
nk <- 2
groupe <- rep(categorie, each = nk)
# Groupement
X <- dummy.coding(groupe)[,-5] # groupe 4 "chien" comme référent
# Spécifications des paramètres
mu0 <- 15
mu <- c(0, 5, -5, 0)
e <- rnorm(n = k * nk, sd = 4)
# Création des scores
```

```
score <- round(mu0 + X %*% mu + e)
cbind(donnees, score)
#>       groupe score score
#> 1       char    14    14
#> 2       char    12    12
#> 3       chat    17    17
#> 4       chat    21    21
#> 5     cheval    14    14
#> 6     cheval    10    10
#> 7  chevalier    12    12
#> 8  chevalier    17    17
#> 9      chien    21    21
#> 10     chien    18    18
```

L'expression `X %*% mu` est une multiplication d'algèbre matricielle qui multiplie la matrice $n \times p$ de codage factice X à une matrice $p \times 1$ de moyennes μ. L'opération multiple les p éléments d'une ligne de X à la colonne p correspondante de μ. En algèbre matricielle Le résultat est une matrice $n \times 1$ qui contient les différences de moyennes pour chaque unité. Dans la même ligne de syntaxe, la moyenne de la population (groupe référent), μ_0 est ajoutée et la variation individuelle, ϵ.

Les scores produits sont identiques. Dans cet exemple par contre, l'origine des différences entre groupes est plus évidente, spécialement en comparant le modèle sous-jacent à l'analyse de variance. La source d'erreur est visible ainsi que les différences de moyennes. La force de cette deuxième méthode est qu'elle pourrait facilement être automatisée pour créer des jeux de données, alors que la première serait plus compliquée. La seconde méthode nécessite six arguments (n_k, μ_0, $\mu_{1:k}$, et σ_e en plus de catégories et définir le groupe référent), la première aurait de la difficulté à spécifier tous les arrangements de différence de moyennes automatiquement, quoiqu'elles seraient plus personnalisables.

9.1.3.4 Analyse

À toute fin pratique, un jeu de données est recréé avec les catégories et paramètres précédents, mais avec $n_k = 20$ unités par groupe. La fonction **R** de base est `aov()`. Elle prend comme argument une formule, de forme `VD ~ VI` (variable dépendante prédite par variable indépendante) et un jeu de données duquel prendre les variables. Il existe également une fonction `anova()`, une fonction un peu plus complexe que `aov()`. Pour obtenir toute l'information désirée de la sortie de la fonction, il faut demander un sommaire de la sortie avec `summary()`.

```
# Anova de base
res <- aov(score ~ groupe, data = donnees)
```

```
summary(res)
#>             Df Sum Sq Mean Sq F value  Pr(>F)
#> groupe       4    834   208.4      14 5.1e-09 ***
#> Residuals   95   1414    14.9
#> ---
#> Signif. codes:
#> 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

# Fonction maison
gr <- table(donnees$groupe)
k <- length(gr)
nk <- dim(donnees)[1] / k

# Moyennes et variance des groupes
moyenne <- by(donnees$score, donnees$groupe, mean)
variance <- by(donnees$score, donnees$groupe, var)

CMI <- nk * var(moyenne)
CMR <- mean(variance)
dl1 <- k - 1
dl2 <- k * (nk - 1)
vf <- CMI / CMR
vp <- (1 - pf(vf, df1 = dl1, df2 = dl2))
# Création du tableau
resultats <- matrix(0,2,5) # Tableau vide
# Ajouter noms
colnames(resultats)  <- c("dl", "SS", "CM", "F", "p")
row.names(resultats) <- c("groupe","residu")
# Ajouter valeurs
resultats[1,] <- c(dl1, CMI * dl1, CMI, vf, vp)
resultats[2,] <- c(dl2, CMR * dl2, CMR, 0, 0)
resultats
#>        dl   SS    CM  F        p
#> groupe  4  834 208.4 14 5.15e-09
#> residu 95 1414  14.9  0 0.00e+00
```

Les résultats sont identiques, les seules différences étant dues à l'arrondissement. Comme la valeur-p est de 5.145×10^{-9}, ce qui est extrêmement plus petit que l'usuel .05 (ou un autre taux d'erreur de type I fixé à l'avance), l'hypothèse nulle est rejetée, il y a vraisemblablement une différence entre les groupes, ce qui est déjà connu. La Figure 9.3 illustre très bien la rareté d'un tel jeu de données sous l'hypothèse nulle.

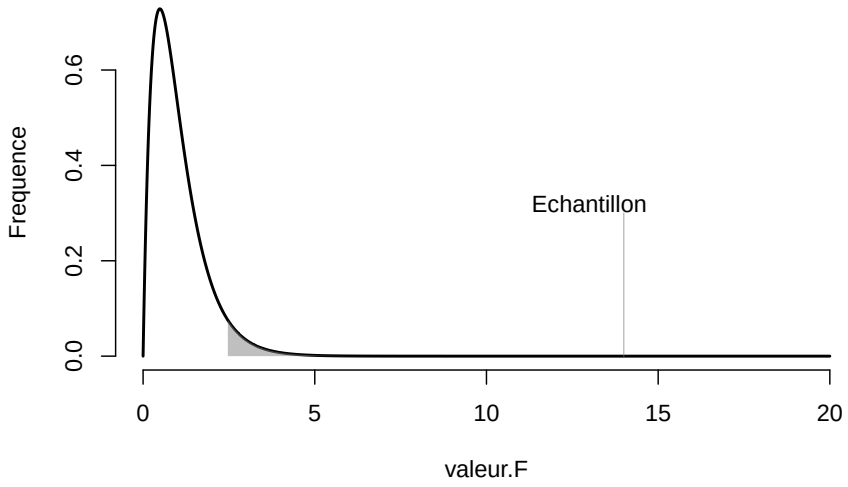

Figure 9.3: Valeur-F de la comparaison des moyennes

9.2 L'association linéaire

9.2.1 La covariance

La covariance représente une mesure du degré auquel une variable augmente lorsque l'autre augmente. Elle correspond au produit moyen entre deux variables centrées (dont les moyennes sont soustraites). On retrouve l'équation sous la forme suivante.

$$\text{cov}_{xy} = \sigma_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

La covariance est une extension multivariée de la variance. La variance est le produit d'une variable **centrée** avec elle-même (le carré de la variable),

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n} x_i x_i$$

.

En transformant x et y pour qu'elles soient **centrées**, l'aspect de produit entre

les deux variables devient évident dans l'expression de la covariance.

$$\text{cov}_{xy} = \frac{1}{n-1} \sum_{i=1}^{n} xy$$

La programmation de la covariance pourrait se traduire ainsi.

```
covariance <- function(x, y){
  # X est une data.frame ou matrice de n sujets par p variables
  n <- length(x)
  # centrer variables
  xc <- x - mean(x)
  yc <- y - mean(y)
  prod.xy <- xc * yc
  cov.xy <- sum(prod.xy) / (n-1)
  return(cov.xy)
}
```

La fonction de base `cov()` sera plus efficace (plus robuste et plus simple) que la fonction maison précédente. La fonction `cov()` peut tenir compte de plus de deux variables. La covariance indique la direction de la covariation entre les deux variables, positives ou négatives, mais n'indique pas la force de la relation. Bien qu'une covariance de 0 indique l'absence de colinéarité, une covariance de 120 n'est pas nécessairement plus forte qu'une covariance de 1. La mesure est intrinsèquement influencée par les unités l'échelle de mesure. Il serait aberrant de considérer le système métrique comme moins efficace, car ces mesures sont moins variables que les mesures impériales. (Il y a plusieurs raisons de préférer le système métrique, mais ce n'en est pas une!)

9.2.2 La corrélation

La corrélation représente le degré d'association linéaire entre deux variables. Une façon simple de considérer la corrélation est de la concevoir comme une covariance standardisée. Elle varie entre -1 et 1, ces deux dernières valeurs impliquant, respectivement, une relation parfaitement négative et positive entre les deux variables. En plus d'indiquer la direction de la relation, la corrélation suggère une force à ce lien. Lorsque la valeur de la corrélation est nulle, $r = 0$, un diagramme de dispersion ne montre qu'un nuage de point épars (sans structure), plus la corrélation augmente (en terme absolu) plus le nuage tends vers une ligne droite. La Figure 9.4 montre différentes valeurs de corrélation et un exemple de nuage de point qui lui est associé.

En statistiques, la corrélation est souvent représentée par la lettre grecque ρ (rho) ou l'usuel r lorsqu'il s'agit d'une estimation à partir d'un échantillon.

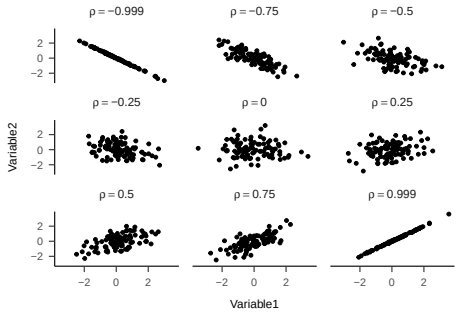

Figure 9.4: Illustrations de diagrammes de dispersion associé en fonction de corrélations

Il y a plusieurs méthodes pour calculer la corrélation. Une première est de considérer la corrélation comme le produit de variables standardisées (autrement dit, score-z). Pour la covariance, la moyenne était soustraite de la variable, donc des variables centrées. Pour la corrélation, on divise l'écart type également pour obtenir des variables standardisées.

En admettant que x et y sont déjà **standardisée**, z_x, z_y, comme c'était le cas pour la covariance, l'aspect de produit entre les deux variables est conservé.

$$\text{cor}_{xy} = r_{xy} = \frac{1}{n-1}\sum_{i=1}^{n} z_x z_y$$

Il est possible de calculer la directement la corrélation à partir de la covariance, soit en divisant par le produit des écarts types des variables.

$$r_{xy} = \frac{\text{cov}_{xy}}{\sigma_x \sigma_y}$$

Voici trois options de programmation de la corrélation. La première et la deuxième sont basées sur la covariance. Dans la première, la covariance est calculée de nouveau dans la fonction. La deuxième quant à elle profite avantageusement de l'existence d'une fonction maison qui calcule directement la covariance. La troisième recourt au produit de variables standardisées.

```
# Option 1 : Calcul complet
correlation1 <- function(x, y){
  n <- length(x)
  # centrer variables
  xc <- x - mean(x)
  yc <- y - mean(y)
  prod.xy <- xc * yc
```

```
  cov.xy <- sum(prod.xy) / (n - 1)
  cor.xy <- cov.xy / (sd(x) * sd(y))
  return(cor.xy)
}

# Option 2 : Basé sur la covariance
correlation2 <- function(x, y){
  cov.xy <- covariance(x,y)
  cor.xy <- cov.xy / (sd(x) * sd(y))
  return(cor.xy)
}

# Option 3 : Basé sur des variables standardisées
correlation3 <- function(x, y){
  n <- length(x)
  xs <- (x - mean(x)) / sd(x)
  ys <- (y - mean(y)) / sd(y)
  cor.xy <- sum(xs*ys)/(n - 1)
  return(cor.xy)
}
```

Évidemment, **R** possède une fonction de base pour le calcul d'une corrélation entre deux variables ou d'une matrice de corrélation (à partir d'un jeu de données de deux variables ou plus). Il s'agit de `cor()`. Ces trois fonctions maison en plus de la fonction **R** sont comparées ci-dessous. Pour ce faire, un jeu de données est créé possédant les caractéristiques désirées. Dans ce cas-ci, la taille d'échantillon est fixée à $n = 10000$ pour assurer une bonne précision dans l'échantillon et une corrélation $r = .7$.

Une façon simple de produire des données à partir d'une matrice de corrélation (ou de covariance) est d'utiliser la fonction `mvrnorm()` du package `MASS`. Elle évitera pour l'instant d'introduire les manipulations mathématiques nécessaires (ce sera donc pour l'instant l'une de ces boîtes noires ayant la confiance de l'utilisateur). La fonction `mvrnorm()` pour *MultiVariate Random NORMmal* nécessite trois arguments : La taille d'échantillon `n`, les moyennes `mu` des p variables et la matrice de covariance `Sigma` (la corrélation est un cas particulier de la covariance où les variables sont standardisées). Le lecteur avisé aura noté la ressemblance de nomenclature entre `mvrnorm()` (multivariée) et `rnorm()` (univariée). Enfin, la dernière étape est de produire une matrice de covariance $p \times p$. La fonction `matrix()` prend une variable de données, dans cet exemple , `c(1, r, r, 1)` qu'elle répartit ligne par colonne. Comme $p = 2$, cela crée une matrice 2×2.

```
set.seed(820)
# Taille d'échantillon importante pour réduire l'erreur d'échantillonnage
n <- 10000
```

```
# La corrélation désirée
r <- .70
# Création d'une matrice de corrélation
R <- matrix(c(1, r, r, 1), nrow = 2, ncol = 2)
# Création des données
donnees <- data.frame(MASS::mvrnorm(n = n, mu = c(0,0), Sigma = R))
colnames(donnees) <- c("x", "y")
# Voici la matrice de corrélation
R
#>      [,1] [,2]
#> [1,]  1.0  0.7
#> [2,]  0.7  1.0
# Voici une visualisation partielle des données
head(donnees)
#>        x      y
#> 1  0.635  0.914
#> 2 -0.193  0.360
#> 3 -0.602 -1.213
#> 4 -0.841 -0.986
#> 5 -0.105 -1.771
#> 6 -2.147 -1.590
# Vérifications des variables
mean(donnees$x); mean(donnees$y); sd(donnees$x); sd(donnees$y)
#> [1] -0.00578
#> [1] -0.00531
#> [1] 0.999
#> [1] 1.01
# Les résultats sont près des attentes
# Vérifications de la corrélations
cor(donnees$x, donnees$y)
#> [1] 0.699
correlation1(donnees$x, donnees$y)
#> [1] 0.699
correlation2(donnees$x, donnees$y)
#> [1] 0.699
correlation3(donnees$x, donnees$y)
#> [1] 0.699
# Les résultats sont près des attentes et identiques
```

Pour tester la probabilité d'obtenir cette valeur, la corrélation r peut se transformer monotonement en valeur-t, ce qui permet de calculer des valeurs-p par la suite.

$$t_{n-2} = \frac{r}{(\frac{\sqrt{1-r^2}}{\sqrt{n-2}})}$$

Cette formule se traduit simplement en code **R**.

```
# Création d'une matrice de corrélation,
# il pourrait également s'agir d'un vecteur ou d'un scalaire.
r <- matrix(c(1,.5,.4,.5,1,.3,.4,.3,1),3,3)
# Nombre d'unités (valeur arbitraire ici)
n <- 10
# Valeurs t
vt <- r * sqrt(n - 2) / sqrt(1 - r ^ 2)
# Valeurs-p
vp <- (1 - pt(abs(vt), df = n - 2)) * 2
vt ; vp
#>      [,1]  [,2]  [,3]
#> [1,]  Inf 1.633 1.234
#> [2,] 1.63   Inf 0.889
#> [3,] 1.23 0.889   Inf
#>       [,1]  [,2]  [,3]
#> [1,] 0.000 0.141 0.252
#> [2,] 0.141 0.000 0.400
#> [3,] 0.252 0.400 0.000
```

Dans le code ci-dessous, l'équation précédente est subtilement réarrangée pour être plus simple et élégante quoiqu'équivalente.

$$t_{n-2} = \frac{r}{(\frac{\sqrt{1-r^2}}{\sqrt{n-2}})} = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}}$$

La première a l'avantage de montrer la relation entre l'estimateur et l'erreur type alors que la seconde est moins encombrante.

9.3 Les données nominales

Jusqu'à présent, deux types d'association de données ont été présenté : une variable nominale (identifiant des groupes) avec une variable continue (différences de moyenne) et deux variables continues (association linéaire). Dans cette section, l'association entre deux variables nominales est présentée. Une façon de représenter l'association entre deux variables nominales est le tableau de contingence, soit l'illustration d'une distribution d'une variable (en ligne) pour chaque catégorie de l'autre (en colonne). En voici, un exemple.

Table 9.1: Tableau de contingence de la relation entre posséder une voiture et le climatoscepticisme

Climatosceptique	Voiture	
	non	oui
non	87	104
oui	7	52

Le tableau 9.1 présente deux variables nominales (`climatosceptique`, à la verticale, et posséder une `voiture`, à l'horizontale) ayant chacune deux catégories (`oui` et `non`). Il montre les proportions **observées**. Ces variables sont-elles associées? Pour traiter cette question, les proportions attendues jouent un rôle crucial. En fait, l'hypothèse nulle sous la table de contingence postule que toutes les proportions du tableau de contingence sont indépendantes. Mathématiquement, il s'agit de postuler que les proportions d'une variable (en ligne ou en colonne) n'influencent pas celles de l'autre variable. Pour obtenir les proportions **théoriques**, les totaux des lignes et colonnes sont calculés ainsi que le grand total. La fréquence attendue d'une cellule est obtenue en calculant, pour chaque cellule, le produit de sa colonne et de sa ligne respective divisé par le grand total.

$$t_{ij} = \frac{\text{total}_i \times \text{total}_j}{\sum \text{total}}$$

L'équation ci-dessus illustre l'idée sous-jacente à la proportion attendue, t d'une cellule de ligne, i, et colonne j.

Table 9.2: Tableau de contingence étendu avec les totaux

Climatosceptique	Voiture		Total
	non	oui	
non	87	104	191
oui	7	52	59
Total	94	156	250

9.3.1 Le χ^2 pour table de contingence

Maintenant que la question statistique et que l'hypothèse nulle sont posées, comment mesurer le degré selon lequel les données s'écartent des attentes *théoriques*.

Le χ^2 (prononcé khi-carré) permet de calculer une telle mesure. Le χ^2 correspond à la distance entre une valeur observée et théorique au carré, pondérée par la valeur théorique.

$$\chi^2_v = \sum_{i=1}^{l}\sum_{j=1}^{c}(\frac{(o_{ij} - t_{ij})^2}{t_{ij}})$$

où o correspond aux valeurs observées, t réfère aux valeurs théoriques, v représente les degrés de liberté et i et j les lignes et colonnes respectivement. Le degré de liberté est

$$v = (n_{\text{ligne}} - 1)(n_{\text{colonne}} - 1)$$

Si l'hypothèse nulle est vraie, les valeurs observées et théoriques devraient être très près. Le carré permet de calculer une distance euclidienne et le dénominateur pondère la distance. Comme l'analyse de variance, le test de χ^2 pour table de contingence est global, il n'informe pas d'où provient la dépendance, mais bien s'il y a au moins une dépendance.

```
khicarre <- function(obs){
  # Obs est une table de contingence
  # Somme colonne
  SC <- as.matrix(colSums(obs))
  # Somme ligne
  SL <- as.matrix(rowSums(obs))
  # Grand total
  TT <- sum(obs)
  # Proportion théoriques
  theo <- SL %*% t(SC) / TT
  # khi carré
  khi2 <- sum((obs - theo) ^ 2 / theo)
  dl <- (length(SL) - 1) * (length(SC) - 1)
  vp <- 1 - pchisq(khi2, dl)
  # Sortie
  statistiques <- list(khi2 = khi2, dl = dl, valeur.p = vp)
  return(statistiques)
}
```

Il y a plusieurs techniques pour créer une base de données, l'essentiel étant de lier les proportions désirées. Ici, une variable sexe est créée avec 50-50% de chance d'être l'un ou l'autre sexe. Par la suite, une proportion différente liée au sexe est utilisée pour générer la fréquence de consommation de tabac. La syntaxe sert principalement à identifier les valeurs `homme` et `femme` de la première variable pour en associer une valeur `tabac`.

```
  set.seed(54)
  n <- 100
  # Première variable
  sexe <- sample(c("homme","femme"),
                 size = n,
                 replace = TRUE,
                 prob = c(.5,.5))
  tabac <- rep(0, 100)
  # Proportions conditionnelles
  tabac[sexe == "femme"] <- sample(c("fumeur","non-fumeur"),
                                   size = sum(sexe == "femme"),
                                   replace = TRUE,
                                   prob = c(.2,.8))
  tabac[sexe == "homme"] <- sample(c("fumeur","non-fumeur"),
                                   size = sum(sexe == "homme"),
                                   replace = TRUE,
                                   prob = c(.1,.9))
  #base données
  donnees <- data.frame(sexe, tabac)
  table(donnees)
#>         tabac
#> sexe     fumeur non-fumeur
#>   femme      8         38
#>   homme      4         50
```

La fonction `table()` de **R** génère une table de contingence. La fonction maison et deux méthodes de base **R** sont présentées et produisent les mêmes résultats pour le χ^2. Un autre, le test exact de Fisher est également présenté. Celui-ci est plus robuste, mais peut requérir plus intensive des ressources de l'ordianteur pour les grosses tables de contingences.

Le test du χ^2 est sujet à certains problèmes qu'il est important de considérer sans quoi l'interprétation pourra être faussée. Si la taille d'échantillon (le nombre d'unités d'observation observées est trop faible), alors le test est biaisé. Par exemple, trois lancers de pile ou face ne sont pas suffisants pour tester une hypothèse de khi-carré, le nombre de résultats différents est de 2, ce qui ne donne pas une approximation satisfaisante de la distribution d'échantillonnage. La convention est de dire qu'une fréquence théorique est trop petite si elle est plus petite que 5. Si ce n'est pas cas, des corrections ou d'autres options doivent être considérées.

Dans le présent exemple, bien que les fréquences attendues respectent les critères usuels, il peut être utile d'envisager un test plus robuste comme le test exact de Fisher ou le χ^2 avec correction de continuité. Le premier se commande avec la fonction `fisher.test()` et le second est la fonction par défaut de `chisq.test()`. Pour montrer l'équivalent entre la fonction `chisq.test()` et la fonction maison, l'option de correction est désactivée avec l'argument `correct = FALSE`.

```
# Table de contingence
TC <- table(donnees)
# Fonction maison
khicarre(TC)
#> $khi2
#> [1] 2.34
#>
#> $dl
#> [1] 1
#>
#> $valeur.p
#> [1] 0.126

# Fonction de base
summary(TC)
#> Number of cases in table: 100
#> Number of factors: 2
#> Test for independence of all factors:
#>  Chisq = 2.3, df = 1, p-value = 0.1

# Autre fonction de base
resultat <- chisq.test(TC, correct = FALSE)
resultat
#>
#>  Pearson's Chi-squared test
#>
#> data:  TC
#> X-squared = 2, df = 1, p-value = 0.1

# Et pour plus de précision...
resultat$statistic
#> X-squared
#>      2.34
resultat$p.value
#> [1] 0.126

# Le test exact de Fisher
resultat.fisher <- fisher.test(TC)
resultat.fisher
#>
#>  Fisher's Exact Test for Count Data
#>
#> data:  TC
#> p-value = 0.2
#> alternative hypothesis: true odds ratio is not equal to 1
#> 95 percent confidence interval:
```

```
#>   0.641 12.731
#> sample estimates:
#> odds ratio
#>       2.61
resultat.fisher$p.value
#> [1] 0.216
```

Chapter 10

Simuler

Dans plusieurs contextes statistiques, les informations cruciales nécessaires pour réaliser une analyse statistique ou tirer des résultats spécifiques ne sont pas connues (par exemple le chapitre Analyser. Pire, parfois certains postulats sont violés rendant les démarches usuelles caduques. Enfin, dans certains contextes, une analyse formelle pourrait s'avérer fort complexe, voire irrésoluble, alors qu'une analyse plus empirique usant des techniques de rééchantillonnage résoudra ces problèmes simplement et immédiatement.

Jusqu'à présent, les situations connues des analyses statistiques ont été présentées. Maintenant, les techniques de rééchantillonnage, comme le bootstrap et de simulations Monte-Carlo seront présentés.

10.1 Les simulations Monte-Carlo

Les simulations Monte-Carlo sont une famille de méthode algorithmique qui permet de calculer une valeur numérique en utilisant des procédés aléatoires. Le nom provient du prestigieux casino de Monte-Carlo à Monaco sur la Côte d'Azur, où des jeux de hasard se jouent constamment, et ayant ainsi une connotation très forte avec le hasard et les nombres aléatoires.

Une simulation de Monte-Carlo prédit une étendue de résultats possibles à partir d'un ensemble de valeurs d'entrée fixes et en tenant compte de l'incertitude inhérente de certaines variables. En d'autres termes, une simulation de Monte-Carlo produit les résultats possibles (sorties) d'un modèle à partir des entrées fixes et d'autres entrées variables. Ces sorties sont calculées encore et encore (des milliers de fois, parfois plus), en utilisant à chaque fois un ensemble différent de nombres aléatoires, générant des sorties probables, mais différentes à chaque fois. Les estimations (et leur tendance) obtenues peuvent alors être interprétées.

Les simulations Monte-Carlo sont particulièrement utiles, car elles permettent, en mathématiques, de calculer des intégrales très complexes; en physique, d'estimer la forme d'un signal ou la sensibilité; en finance, simulent des conditions permettant de prévoir le marché; et en psychologie, simuler des processus comportementaux ou cognitifs.

Les simulations Monte-Carlo possèdent trois caractéristiques ou étapes :

1. Construire un modèle ayant des variables indépendantes (entrées) et dépendantes (sorties);
2. Spécifier les caractéristiques aléatoires des variables indépendantes et définir des valeurs crédibles;
3. Rouler les simulations de façon répétitive jusqu'à ce que suffisamment d'itérations soient produites et que les résultats convergent vers une solution.

Maintenant, ces étapes sont mises en contexte avec un problème.

10.1.1 Le problème de Monty-Hall

Rien de mieux pour confronter le sens commun que d'être confronté à un problème contre-intuitif. Les simulations Monte-Carlo nous permettront de vérifier les résultats de façon empirique, parallèlement à ce qu'une analyse formelle fournit.

Le problème de Monty Hall (attribuables à Selvin, 1975) présente un joueur devant un présentateur (Monty Hall). Trois portes sont offertes au choix du joueur. Derrière l'une d'entre elles se retrouve un magnifique prix : une voiture *électrique* de luxe. Derrière chacune des deux autres portes se retrouvent un prix citron : une chèvre chacune. Le joueur ne sait pas ce qu'il y derrière les portes. Le joueur peut alors choisir une porte. Évidemment, à ce moment le joueur à une chance sur trois de choisir la voiture.

Par la suite, le présentateur, qui connaît le contenu derrière les portes, ouvre une porte qui (a) ne cache pas la voiture et (b) que le participant n'a pas choisie. Le joueur peut alors choisir (a) de conserver la porte choisie ou (b) de changer de porte. Quelle option, s'il y en a une, assurera le meilleur gain (choisir la voiture)? Rester ou changer? Par exemple, suivant l'illustration de la figure 10.1, le joueur choisit la porte 1, alors le présentateur doit ouvrir la porte 2 (non choisie et ne contient pas la voiture). Le joueur doit-il changer son choix? Ici, la réponse est évidente, car la réponse est connue. Qu'en est-il alors si le joueur avait choisi la porte 3 et le présentateur ouvert l'une des deux autres porte? *Le joueur doit-il changer ou rester?* La question sous-jacente est qu'elles sont les probabilités de rester et changer respectivement. Sont-elles différentes?

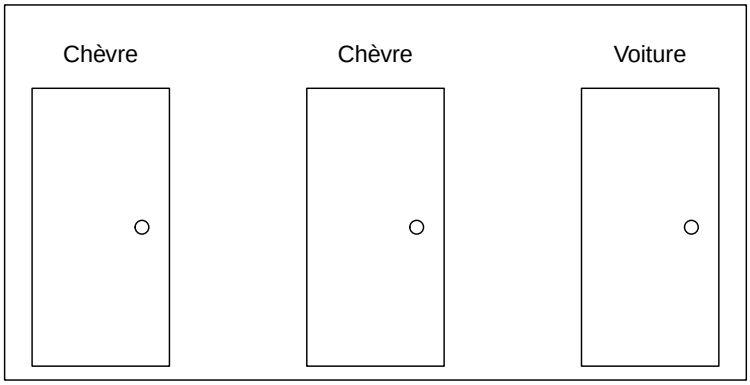

Figure 10.1: Illustration du problème de Monty Hall

Comme le résultat n'est pas des plus intuitif (et sans divulgâcher le résultat), une petite simulation s'impose (ou une analyse formelle pour les lecteurs enclins mathématiquement). La situation sera recréée un millier de fois pour vérifier l'option (rester ou changer) qui maximise de gagner la voiture.

En se référant aux trois points caractérisant une simulation Monte-Carlo susmentionné :

1. Le problème de Monty Hall tel qu'imminemment décrit.
2. Les variables indépendantes sont (a) le tirage aléatoire du contenu derrière les portes (distribution uniforme, chaque porte à la même probabilité d'avoir un prix ou non), (b) le choix du joueur (distribution uniforme, pourrait être différent), et (c) la porte ouverte par le présentateur. Deux variables indépendantes sont fixes (les deux options sont étudiées), l'action de rester ou de changer. Il y a deux variables dépendantes, le succès (choisir le prix) ou l'échec (choisir une autre porte) de rester ou de changer.
3. La simulation est reprise 1000 fois.

La simulation doit comporter une distribution aléatoire du contenu derrière les trois portes, identifier une porte gagnante et deux perdantes. Il est intéressant de

note que, sur le plan de computationnelle, il est inutile d'attribuer aléatoirement la porte gagnante, toutefois le scénario sera plus acceptable pour un lecteur scrupuleux. Par la suite, le joueur fait son premier choix. Ici, plusieurs modèles peuvent être utilisés, comme constamment choisir la même porte (ce qui simplifie la situation et ne change pas les probabilités parce que le contenu des portes, lui, est aléatoire) ou, pour rester vrai à la situation, le joueur fera un choix aléatoire. Le comportement du présentateur s'enclenche, il doit "ouvrir" une porte non choisie et qui ne contient pas le prix. Les deux stratégies sont alors étudiées, *rester* versus *changer*. La simulation enregistre alors s'il y a eu gain ou non et l'additionne au total de chacun. La simulation est reprise un nombre important de fois, ici, 1000 suffira, mais des situations compliquées peuvent demander beaucoup plus d'itérations.

```
# Simulation du problème de Monty Hall
portes <- c("Chèvre","Chèvre","Voiture")
nreps <- 1000
set.seed(7896)
# Valeur initial des gains
gain.rester <- 0
gain.changer <- 0

for(i in 1:nreps){
  # Arrangement initial des portes
  tirage = sample(portes)
  # Trouver le prix
  prix = which(tirage %in% "Voiture")
  # Choix aléatoire
  choix1 = sample(length(portes), size = 1)

  # Sélectionner la porte avec la chèvre (pas un prix) et
  # qui n'est pas celle choisie (choix1)
  option = c(choix1, prix)
  if(choix1 != prix){
    # Si une porte valide
    monty = c(1:3)[-option]
  }else{
    # Si deux portes valides, en choisir une aléatoirement.
    monty = sample(c(1:3)[-option], size = 1)
  }
  # Décision : Reste à choix1
  choix.rester = choix1

  # Décision : Changer de choix changer
  # Changer = ne pas prendre la porte initial, ni la porte ouverte
  choix.changer = c(1:3)[-c(choix1, monty)]
```

```
  # Enregistrement des gains
  gain.rester <- gain.rester + (tirage[choix.rester] == "Voiture")
  gain.changer = gain.changer + (tirage[choix.changer] == "Voiture")
}

cbind(gain.rester,gain.changer)  / nreps
#>      gain.rester gain.changer
#> [1,]       0.316        0.684
```

Une petite digression avant de poursuivre. *Il ne faut jamais prendre pour acquis que le code fonctionne comme prévu.* À cause d'un inconvénient de la fonction `sample()`, une approche conditionnelle doit être utilisée. En fait, la fonction échantillonne les éléments de `x` (premier argument) jusqu'à obtenir `size` objets. Par contre, si une seule valeur est donnée à `x` et si elle est numérique, alors la fonction utilise les éléments de `1:x` pour rééchantillonner. Ici, si le choix et le prix sont différents, il n'y a qu'une seule valeur retournée (l'autre chèvre), ce qui occasionne le problème, et par conséquent, de l'utilisation du conditionnel. En programmation, il faut toujours s'assurer que les fonctions s'accordent avec les attentes.

Quelles sont les résultats de cette simulation? Le fait de *rester* sur le premier choix devrait obtenir 33% de chance de réussir, comme le joueur a initialement une chance sur trois d'avoir la bonne réponse. Qu'en est-il pour *changer*? Est-ce qu'ouvrir une porte *chèvre* modifie les probabilités? Les résultats de la simulation montre que *rester* gagne 31.6% et que de *changer* gagne 68.4%. À long terme, il est fort avantageux de changer.

Une explication simple est de dénombrer les possibilités. Dans le cas où le joueur ne change pas d'idée, il a une chance sur trois, soit : choisir la chèvre 1, et garder la chèvre 1; choisir la chèvre 2, et garder la chèvre 2; et choisir la voiture, et garder la voiture. Si le joueur change d'idée après l'ouverture des portes, les chances sont maintenant de deux sur trois, soit choisir la chèvre 1, changer pour le prix; choisir la chèvre 2 et changer pour le prix; ou choisir le prix et changer pour une chèvre.

Une autre façon de rendre se problème plus évident est de considérer le problème avec 100 portes au lieu de 3. Le présentateur ouvre alors les 98 portes qui ne sont pas un prix. Si le joueur choisit une porte et garde, il a effectivement 1% de chance de remporter le prix. Par contre, s'il choisit une porte et que le présentateur lui ouvre toutes les autres portes chèvres, il appert que le joueur gagne à tout coup s'il change, sauf s'il a choisi le prix au premier coup. Avec quelques modifications de la syntaxe précédente, le code suivant illustre le cas à 100 portes.

```
# Simulation du problème de Monty Hall
portes <- c("Voiture", rep("Chèvre", 99))
n <- length(portes)
```

```
nreps <- 1000
set.seed(7896)

# Valeur initial des gains
gain.rester <- 0
gain.changer <- 0

for(i in 1:nreps){
  # Arrangement initial des portes
  tirage <- sample(portes)

  # Trouver le prix
  prix <- which(tirage %in% "Voiture")

  # Choix aléatoire
  choix1 <- sample(n, size = 1)

  # Sélectionner la porte avec la chèvre (pas un prix) et
  # qui n'est pas celle choisie (choix1)
  option <- c(choix1, prix)

  # Les autres portes
  monty <- sample(c(1:n)[-option], size = n - 2)

  # Décision : Reste à choix1
  choix.rester <- choix1

  # Décision : Changer de choix changer
  # Changer = ne pas prendre la porte initial, ni la porte ouverte
  choix.changer <- c(1:n)[-c(choix1, monty)]

  # Enregistrement des gains
  gain.rester <- gain.rester + (tirage[choix.rester] == "Voiture")
  gain.changer <- gain.changer + (tirage[choix.changer] == "Voiture")
}

cbind(gain.rester,gain.changer)  / nreps
#>      gain.rester gain.changer
#> [1,]       0.009        0.991
```

La simulation montre que rester gagne 0.9% et que de changer gagne 99.1%.

Des situations fort plus compliquées pourraient être apportées en simulations Monte-Carlo. Le jeu de Black Jack en est un exemple, bien qu'il soit déjà résolu pour des résultats optimaux (Millman, 1983). Il suffit d'avoir la patience de programmer le tout pour confirmer les résutats analytiques.

Sur le plan pratique pour l'expérimentateur, les simulations Monte-Carlo peuvent être utiles pour calculer des tailles d'échantillons pour des modèles complexes, comme des modèles par équations structurelles compliquées, mutliniveaux ou de classes latentes. Une sous-famille est d'une importance signficative pour l'expérimentateur, celle qui sera abordée maintenant, le bootstrap.

10.2 Le bootstrap

Le *bootstrap* (dont il n'y a pas d'excellente traduction en français) est une des techniques de rééchantillonnage astucieuses permettant des inférences statistiques (Efron & Tibshriani, 1979). Il fait partie de la famille des simulations Monte-Carlo, car il se base sur la réitération multiple d'une statistique à partir d'un jeu de données. Sans vouloir entrer trop dans les détails, il existe plusieurs types de techniques de bootstrap, qui font elles-mêmes parties d'une plus grande famille, les simulations stochastiques. Y est inclus les simulations de Monte-Carlo (dont le bootstrap fait parti) ou les méthodes numériques bayésiennes.

Cet ouvrage insiste sur le bootstrap non paramétrique, impliquant que les distributions sous-jacentes aux échantillons ne soient pas spécifiées, bien qu'il existe du bootstrap paramétrique. Ce type de bootstrap est certainement la plus utile pour l'expérimentateur.

Dans le bootstrap, l'échantillon initial est considéré comme une pseudopopulation. Il ne nécessite pas d'autre information que celle fournie par l'échantillon. Il permet d'obtenir une distribution d'échantillonnage et tous ses bénéfices. Alors que dans le chapitre précédent, il fallait préciser et connaître quelle distribution d'échantillonnage correspond à quelle statistique (p. ex., le score-z d'une unité à la distribution normale; la moyenne d'un échantillon à la distribution t si la variance est inconnue), aucune de ces connaissances n'est nécessaire. L'étendue d'application du bootstrap appert immense, surtout pour les concepts statistiques qui offriront plus de défis, ce qui sera vu dans des chapitres ultérieurs.

Obtenir la distribution d'échantillonnage permet ainsi :

- d'estimer l'indice désiré;
- d'estimer son erreur standard;
- d'estimer ses intervalles de confiance;
- de réaliser un test d'hypothèse;

tout cela en ignorant les distributions et postulats sous-jacents qui pourraient empêcher ou limiter leurs usages. Elle n'a que deux hypothèses fondamentales :l'échantillon reflète la population et chaque unité est indépendante et identiquement distribuée. Autrement dit, chaque unité provienne bel et bien d'une même

boîte (*population*, voir chapitre sur l'inférence statistique) et en sont un portrait juste en plus que le tirage d'une unité n'en influence celle d'une autre.

Le fondement du bootstrap repose sur les étapes suivantes :

1. Sélectionner avec remise les unités d'un échantillon;
2. Calculer et enregistrer l'indice statistique désiré auprès de ce nouvel échantillon;
3. Réitérer les deux premières étapes un nombre élevé de fois.

Ce processus hautement facilité par l'excellente performance des ordinateurs d'aujourd'hui, se réalise très facilement et rapidement. L'exemple suivant est basé sur le rééchantillonnage (1000 fois) de la moyenne à partir d'une variable aléatoire tirée d'une distribution uniforme avec un minimum de 5 et d'une maximum de 15.

```
set.seed(158)
# Nombre d'unités
n <- 30

# Le nombre de rééchantillonnage
nreps <- 10000

# Création de la variable
X <- runif(n = n, min = 5, max = 15)

# Création d'une variable vide pour l'enregistrement
moyenne.X <- numeric()

# L'utilisation d'une boucle permet de réitérer les étapes
for (i in 1:nreps){
  # Rééchantillonnage avec remise
  id <- sample(n, replace = TRUE)

  # Nouvel échantillon
  nouveau.X <- X[id]

  # Calculer et enregistrer la moyenne
  moyenne.X[i] <- mean(nouveau.X)
}
```

La fonction `sample(, replace = TRUE)` rééchantillonne avec remplacement les identifiants des unités. Par simplicité, il est possible d'éliminer la ligne `nouveau.X = X[id]` en utilisant `mean(X[id])` ou plus simplement

`mean(sample(X, replace = TRUE))`. Cette utilisation se limite seulement au cas d'un vecteur de données. S'il s'agit d'un jeu de données avec plus d'une variable, alors `mean(X[id, ])` sera utilisé. Noter l'usage de la `,` entre crochets. De cette façon, toutes les variables des unités identifiées par `id` sont extraites.

À partir des informations obtenues, il est possible de faire des inférences statistiques. Toutes les estimations peuvent être présentées comme un histogramme tel que l'illustre la figure 10.2. Le code se retrouve ci-dessous. Ici, quelques formules pour améliorer la présentation se retrouvent dans la syntaxe. L'utilisation simple de `hist()` pourra convenir.

```
hist(moyenne.X,              # Données
     ylab = " Frequence",    # Changer l'axe y
     main = "",              # Retirer le titre
     breaks = 50,            # Nombre de colonnes
     xlim = c(7,13)          # Définir l'étendu de l'axe x
)
```

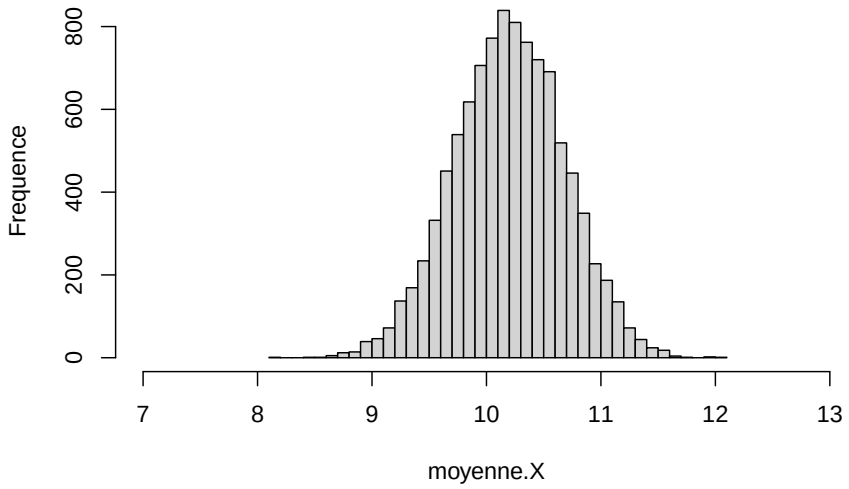

Figure 10.2: Historgamme des estimations des échantillons

La distribution de la figure 10.2 peut être désignée comme la distribution de la *population*, comme c'était le cas dans le chapitre sur les inférences. Elle permet d'avoir une estimation plus robuste de l'erreur standard, qui se trouve à être

l'écart type des indices rééchantillonnés. Elle est également utilisée pour construire des intervalles de confiance et permet ainsi de faire des tests d'hypothèse comme : l'intervalle de confiance à $(1 - \alpha) \times 100$ % contient-elle la valeur 0? Comme il est fait avec une hypothèse nulle traditionnelle.

Ici, le cas illustré est trivial au sens où, par le théorème central limite, la réponse est déjà connue. Si l'exemple était plutôt sur la médiane, il faudrait obligatoirement procéder par bootstrap pour calculer son erreur type et ses intervalles de confiance, car aucune formule exacte ne permet sa dérivation (il existe bien des approximations cela dit). Le bootstrap est alors des plus appropriés pour calculer l'erreur type et en tirer des intervalles de confiances, voire même en faire des tests d'hypothèses. Plusieurs statistiques auront recours au bootstrap pour dériver ces informations, la plus utilisée étant le coefficient de détermination (voir le chapitre sur la régression).

À partir des informations obtenues auprès du rééchantillonnage, il est possible d'obtenir les éléments désirés. Comme le cas est trivial, les indices statistiques seront très similaires. Cela confirmera au lecteur qu'aucun principe ésotérique ne s'est déroulé devant ses yeux en plus de confirmer que les statistiques attendues se produisent effectivement.

```
# Voici la moyenne et l'erreur standard originales
mean(X) ; sd(X)/sqrt(n)
#> [1] 10.2
#> [1] 0.485

# La moyenne et l'erreur standard à partir des rééchantillons
mean(moyenne.X)  ; sd(moyenne.X)
#> [1] 10.2
#> [1] 0.481
```

La moyenne et l'erreur standard sont très près de la moyenne de l'échantillon et celle de la population (qui est de 10) et de l'erreur type attendue. Il est possible de créer des intervalles de confiances avec la fonction `quantile` qui prend en argument un vecteur de données et les probabilités désirées. Dans le cas de 95% pour une erreur de type I de 5%, soit $\alpha = .05$, il s'agit de $.05/2 = .025$ et $1 - .05/2 = .975$, laissant au total 5% aux extrémités.

```
# Erreur de type I
alpha = .05

# Valeurs critiques
crit = c(alpha/2, (1-alpha/2))
tv = qt(crit, df = n - 1)

# Intervalles basés sur les indices de l'échantillon
```

```
mean(X) + tv * sd(X)/sqrt(n)
#> [1]  9.2 11.2

# Intervalles basés sur les indices du rééchantillonnage
mean(moyenne.X) + tv * sd(moyenne.X)
#> [1]  9.22 11.18

# Intervalles sur le rééchantillonnage
quantile(moyenne.X, crit)
#>  2.5% 97.5%
#>  9.26 11.13
```

Pour réaliser le test d'hypothèse nulle, il suffit de constater si l'intervalle de confiance contient ou non zéro. Dans le cas où l'intervalle contient 0, le test n'est pas significatif, il est vraisemblable que l'absence d'effet soit vraie. S'il ne contient pas 0, l'hypothèse est rejetée, le test est significatif et il est vraisemblable qu'il y ait un effet. Dans cet exemple, la moyenne est clairement différente de 0, car elle ne contient pas cette valeur.

10.2.0.1 Meilleures pratiques

R n'est pas particulièrement efficace pour réaliser des boucles. Il existe une famille de fonctions `apply` qui permettent d'appliquer une fonction sur une série de vecteurs (comme une liste, un jeu de données ou une matrice). L'appel à l'aide `?apply` donne beaucoup d'informations. La fonction principale est `apply()`. Elle nécessite le jeu de données (`X`) sur lequel appliqué la fonction (`FUN`) et un argument (`MARGIN`) pour identifier dimension selon laquelle il faut appliquer la fonction, comme `MARGIN = 1` produit l'opération par lignes; `MARGIN = 2` produit l'opération par colonnes.[1]

```
# Il faut un vecteur de données `vec`, un nombre de répétitions `nreps`
# et une fonction `theta` à calculer sur le vecteur
bootstrap <- function(vec, nreps, theta){
  btsp.vec <- matrix(sample(vec, nreps * length(vec), replace = TRUE),
                     nrow = nreps)
  btsp.res <- apply(X = btsp.vec, MARGIN = 1, FUN = theta)
  return(btsp.res)
}

# En utilisant les données de l'exemple précédent
btsp <- bootstrap(vec = X, nreps = nreps, theta = mean)
```

[1] Il y a aussi des cas plus complexes lorsque le jeu possède plus de deux dimesions, comme `MARGIN = c(1,2)` produit l'opération sur une matrice (lignes par colonnes) ou `MARGIN = n` produit l'opération sur la n^{e} dimension.

```
# Pour fin de comparaison :
# Voici la moyenne et l'erreur standard de ce bootstrap
mean(btsp)  ; sd(btsp)
#> [1] 10.2
#> [1] 0.484

# Voici la moyenne et l'erreur standard à partir du premier bootstrap
mean(moyenne.X)  ; sd(moyenne.X)
#> [1] 10.2
#> [1] 0.481

# Et voici la moyenne et l'erreur standard originales
mean(X) ; sd(X)/sqrt(n)
#> [1] 10.2
#> [1] 0.485
```

10.2.1 Les packages

Il existe plusieurs packages pour réaliser du bootstrap dans **R**. Il y a `bootstrap` (Tibshirani & Leisch, 2019) et `boot` (Canty & Ripley, 2021). Ajouter à cela que plusieurs fonctions **R** possèdent des options de bootstrap intégrées (qu'il faut commander dans les arguments). Plusieurs analyses statistiques qui recourent régulièrement aux bootstraps auront déjà des options implantées.

Exercices

TODO

1. Créer une table de valeur-t critique pour $dl = 1, 2, 3, ..., 30$ et $\alpha = .05$ unilatérale.

2. Comparer la puissance de la distribution-t par rapport à une distribution normale centrée réduite. La différence par rapport à l'hypothèse nulle est 2 et l'écart type est 1, l'$\alpha = .05$ bilatérale. Tester pour différentes valeurs de n.

3. Créer un jeu de données pour test-t dépendant avec une corrélation de .3 entre les temps de mesure, des variances de 1, une différence de moyenne de 1 et une taille d'échantillon $n = 20$. Analyser ce jeu de données : vérifier la corrélation et la différence.

4. Calculer la puissance d'une corrélation de $\rho = .30$ pour une taille d'échantillon $n = 80$ et $alpha = .05$ bilatérale.

5. Comparer l'erreur de type I du test-t indépendant avec équivalence versus non-équivalence des variances pour une différence de moyennes de 1 et des variances de .25 pour le groupe 1 et 1 pour le groupe 2 pour une taille d'échantillon de 30 séparé également entre les groupes.

Bibliography

Bache, S. M., & Wickham, H. (2020). *Magrittr: A forward-pipe operator for r* [R package version 2.0.1]. https://CRAN.R-project.org/package=magrittr

Bennett, C. M., Baird, A. A., Miller, M. B., & Wolford, G. L. (2010). Neural correlates of interspecies perspectives taking in the post-moterm atlantic salmon : An argument for proper multiple comparisons correction. *Journal of the Serendipitous and Unexpected Results*, *1*(1), 1–5.

Canty, A., & Ripley, B. D. (2021). *Boot: Bootstrap R (S-plus) functions* [R package version 1.3-28].

Efron, B., & Tibshriani, R. (1979). *An introduction to the bootstrap*. Chapman & Hall.

Eklund, A., Nichols, T. E., & Knutsson, H. (2016). Cluster failure: Why fmri inferences for spatial extent have inflated false-positive rates. *Proceedings of the National Academy of Sciences of the United States of America*, *113*(28), 7900–7905. https://doi.org/10.1073/pnas.1602413113

Hammill, D. (2021). *Dataeditr: An interactive editor for viewing, entering, filtering & editing data* [R package version 0.1.3]. https://CRAN.R-project.org/package=DataEditR

Millman, M. H. (1983). A statistical analysis of casino blackjack. *The American Mathematical Monthly*, *90*(7), 431–436.

R Core Team. (2022). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, Austria. https://www.R-project.org/

Selvin, S. (1975). A problem in probability (letter to the editor). *The American Statistician*, *29*(1), 67.

Tibshirani, R., & Leisch, F. (2019). *Bootstrap: Functions for the book "an introduction to the bootstrap"* [R package version 2019.6]. https://CRAN.R-project.org/package=bootstrap

Wickham, H. (2014). Tidy data. *Journal of Statistical Software, Articles*, *59*(10), 1–23. https://doi.org/10.18637/jss.v059.i10

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Moller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., … Yutani, H. (2019). Welcome to the tidyverse.

Journal of Open Source Software, *4*(43), 1686. https://doi.org/10.21105/joss.01686