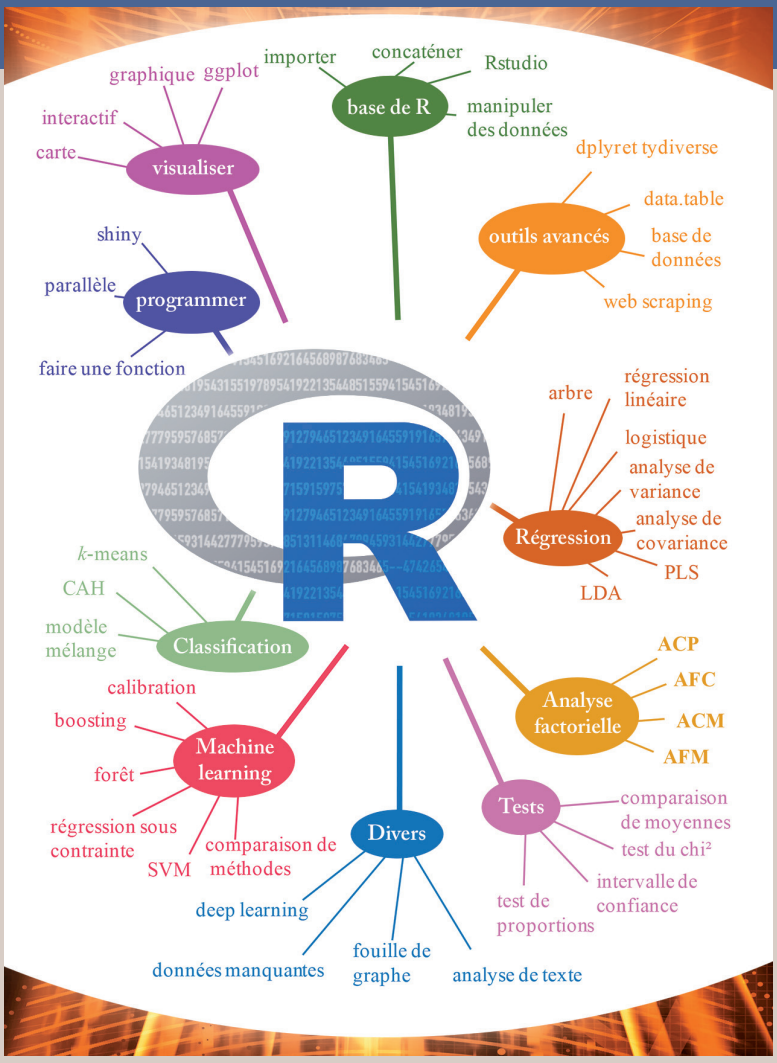
Sous la direction de
FRANÇOIS HUSSON

PRATIQUE DE LA STATISTIQUE

R pour la statistique et la science des données



PIERRE-ANDRÉ CORNILLON
ARNAUD GUYADER
FRANÇOIS HUSSON
NICOLAS JÉGOU
JULIE JOSSE
NICOLAS KLUTCHNIKOFF
ERWAN LE PENNEC
ERIC MATZNER-LØBER
LAURENT ROUVIÈRE
BENOÎT THIEURMEL

SFdS SOCIETE FRANÇAISE DE STATISTIQUE
PUR

R pour la statistique et la science des données

Collection « Pratique de la statistique »

Dirigée par François Husson

Mónica Bécue-Bertaut,
Analyse textuelle avec R, 2018, 190 p.

François Husson, Sébastien Lê et Jérôme Pagès,
Analyse de données avec R, 2e éd. revue et augmentée, 2016, 240 p.

François Husson et Jérôme Pagès,
Statistiques générales pour utilisateurs, 2. Exercices et corrigés, 2e éd. augmentée, 2013, 364 p.

Pierre-André Cornillon, Arnaud Guyader, François Husson, Nicolas Jégou, Julie Josse, Maela Kloareg, Éric Matzner-Løber et Laurent Rouvière
Statistiques avec R, 3e éd. revue et augmentée, 2012, 300 p.

Mohammed Bennani Dosse,
Statistique bivariée avec R, 2011, 296 p.

Jérôme Pagès,
Statistiques générales pour utilisateurs, 1. Méthodologie, 2e éd. augmentée, 2010, 280 p.

Argentine Vidal,
Statistique descriptive et inférentielle avec Excel. Approche par l'exemple, 2e éd., 2010, 292 p.

Agnès Hamon et Nicolas Jégou,
Statistique descriptive. Cours et exercices corrigés, 2008, 210 p.

Jean-Pierre Georgin et Michel Gouet,
Statistiques avec Excel. Descriptives, tests paramétriques et non paramétriques à partir de la version Excel 2000, 2005, 344 p.

Laurent Ferrara et Dominique Guégan,
Analyser les séries chronologiques avec S-Plus. Une approche paramétrique, 2002, 160 p.

Jean-Pierre Georgin,
Analyse interactive des données avec Excel 2000, 2002, 188 p.

Jean-Jacques Daudin, Stéphane Robin et Colette Vuillet,
Statistique inférentielle. Idées, démarches, exemples, 1999, 186 p.

Brigitte Escofier et Jérôme Pagès,
Initiation aux traitements statistiques. Méthodes, méthodologie, 1997, 264 p.

Thierry Foucart,
L'analyse des données. Mode d'emploi, 1997, 200 p.

Sous la direction de
François Husson

Avec la contribution de
Pierre-André Cornillon, Arnaud Guyader, François Husson,
Nicolas Jégou, Julie Josse, Nicolas Klutchnikoff, Erwan Le Pennec,
Eric Matzner-Løber, Laurent Rouvière et Benoît Thieurmel

R pour la statistique et la science des données

Pratique de la statistique
Presses universitaires de Rennes
2018

Saic Édition – Université Rennes 2
2 avenue Gaston-Berger – Bâtiment Germaine-Tillion
35043 Rennes Cedex

www.pur-editions.fr

Dépôt légal : 2e semestre 2018
ISBN : 978-2-7535-7573-8
ISSN : 1295-1765

Avant-propos

Ce livre s'adresse à toute personne ayant à traiter des jeux de données, indépendamment du domaine d'application. Cette pratique impliquant typiquement de grandes quantités d'informations, l'aspect numérique est bien entendu primordial. De fait, il existe aujourd'hui de nombreux outils répondant à ces besoins. Nous avons opté ici pour le logiciel R dont le triple intérêt est d'être gratuit, très complet et en essor permanent. Néanmoins, aucune connaissance sur celui-ci n'est prérequise. Le livre se divise en effet en deux grandes parties : la première est centrée sur le logiciel lui-même, la seconde sur la mise en œuvre de méthodes statistiques classiques avec R.

Nous présentons les concepts de base du logiciel dans le premier chapitre. Le deuxième traite de la manipulation des données, c'est-à-dire des opérations courantes en statistique. Le bilan d'une étude passant par une visualisation claire des résultats, nous décrivons alors, en chapitre 3, certaines possibilités offertes par R dans ce domaine. Nous y présentons aussi bien la construction de graphiques simples que certaines variantes plus avancées. Les bases de la programmation sont quant à elles présentées au chapitre 4 : nous expliquons comment construire ses propres fonctions mais exposons aussi quelques-unes des procédures prédéfinies pour automatiser et paralléliser des analyses répétitives. Témoin du développement continu de R, le chapitre 5 se veut une introduction à quelques outils récents dédiés à des données en constante évolution, tant dans leurs formes que dans leur volume. Focalisée sur le logiciel R, cette première partie permet de comprendre les commandes apparaissant dans les méthodes exposées par la suite.

La seconde partie du livre propose de balayer un large spectre de techniques aussi bien classiques que récentes en traitement des données : intervalles de confiance et tests, procédures d'analyse factorielle, classification non supervisée, méthodes usuelles de régression, machine learning, gestion de données manquantes, analyse de texte, fouille de graphe, etc. Chaque méthode est illustrée sur un exemple et traitée de façon autonome dans une fiche spécifique. Après une brève présentation du contexte, les lignes de commandes R sont détaillées et les résultats commentés.

On pourra télécharger les jeux de données et retrouver tous les résultats décrits ainsi que les solutions des exercices proposés en première partie sur le site du livre : `https://r-stat-sc-donnees.github.io/`

Sommaire

I R et son fonctionnement **1**

1 Concepts **3**
1.1 Installation de R et RStudio . . . 3
1.2 Environnement de travail . . . 5
1.3 Introduction à RMarkdown . . . 6
1.3.1 Création d'un document RMarkdown . . . 6
1.3.2 Bases du langage . . . 6
1.3.3 Insertion du code R . . . 7
1.4 Les différentes aides . . . 8
1.5 Les objets R . . . 8
1.5.1 Création, affichage, suppression . . . 9
1.5.2 Le mode d'un objet . . . 10
1.5.3 La valeur manquante . . . 11
1.5.4 Les vecteurs . . . 11
1.5.5 Les matrices . . . 17
1.5.6 Les facteurs . . . 22
1.5.7 Les listes . . . 23
1.5.8 Les data-frames . . . 25
1.5.9 La classe d'un objet . . . 26
1.6 Les fonctions . . . 27
1.7 Les packages . . . 28
1.7.1 Installation d'un package . . . 29
1.7.2 Utilisation d'un package . . . 29
1.7.3 Mise à jour des packages et de R . . . 30
1.8 Exercices . . . 31

2 Manipuler les données **35**
2.1 Importer des données . . . 35
2.1.1 Importation d'un fichier texte . . . 35
2.1.2 Importation de données textes volumineuses . . . 37

2.1.3 Importation d'autres formats de données . . . 38
2.1.4 Importation via le menu de RStudio . . . 38
2.2 Exporter des résultats . . . 38
2.3 Manipuler les variables . . . 39
2.3.1 Changer de type . . . 39
2.3.2 Découpage en classes . . . 41
2.3.3 Travail sur le niveau des facteurs . . . 42
2.4 Manipuler les individus . . . 45
2.4.1 Repérer les données manquantes . . . 45
2.4.2 Repérer les individus aberrants univariés . . . 47
2.4.3 Repérer et/ou éliminer des doublons . . . 48
2.5 Concaténer des tableaux de données . . . 49
2.6 Tableau croisé . . . 52
2.7 Exercices . . . 54

3 Visualiser les données **57**
3.1 Les fonctions graphiques conventionnelles . . . 57
3.1.1 La fonction plot . . . 58
3.1.2 Représentation d'une distribution . . . 62
3.1.3 Ajouts aux graphiques . . . 64
3.1.4 Graphiques en plusieurs dimensions . . . 66
3.1.5 Exportation de graphiques . . . 68
3.1.6 Plusieurs graphiques . . . 69
3.1.7 Amélioration et personnalisation des graphiques . . . 71
3.2 Les fonctions graphiques avec ggplot2 . . . 74
3.2.1 Premiers graphes avec ggplot2 . . . 75
3.2.2 La grammaire ggplot . . . 76
3.2.3 Group et facets . . . 83
3.2.4 Compléments . . . 86
3.3 Les graphiques interactifs . . . 88
3.4 Construire des cartes . . . 90
3.4.1 Carte statique dans R . . . 90
3.4.2 Carte dans un navigateur . . . 93
3.4.3 Carte avec contours : le format shapefile . . . 97
3.5 Exercices . . . 101

4 Programmer **109**
4.1 Structures de contrôle . . . 109
4.1.1 Commandes groupées . . . 109
4.1.2 Les boucles (for ou while) . . . 109
4.1.3 Les conditions (if, else) . . . 111
4.2 Construire une fonction . . . 112
4.3 La famille apply, des fonctions d'itération prédéfinies . . . 114
4.4 Calcul parallèle . . . 121

4.4.1 Introduction . . . 121
4.4.2 Le package parallel . . . 122
4.4.3 Le package foreach . . . 124
4.4.4 Exemple avancé . . . 125
4.5 Faire une application shiny . . . 128
4.6 Exercices . . . 137

5 Outils avancés pour la préparation des données 139
5.1 Le package data.table . . . 139
5.1.1 Importation avec fread . . . 140
5.1.2 Syntaxe . . . 142
5.1.3 Sélection . . . 142
5.1.4 Manipulation . . . 145
5.1.5 Pour aller plus loin . . . 149
5.2 Le package dplyr et le tidyverse . . . 151
5.2.1 Le package dplyr . . . 151
5.2.2 Manipulation de tables . . . 155
5.2.3 Pour aller plus loin . . . 158
5.3 Bases de données . . . 162
5.3.1 SQL : Structured Query Language . . . 162
5.3.2 JSON : JavaScript Object Notation . . . 171
5.4 Web scraping . . . 180
5.4.1 Introduction . . . 180
5.4.2 Approche naïve . . . 181
5.4.3 Le package rvest . . . 182
5.4.4 Pour aller plus loin . . . 187
5.5 Exercices . . . 189

II Fiches thématiques 193

6 Intervalles de confiance et tests d'hypothèses 195
6.1 Intervalle de confiance d'une moyenne . . . 196
6.2 Test du χ^2 d'indépendance . . . 199
6.3 Comparaison de deux moyennes . . . 204
6.4 Tests sur les proportions . . . 210

7 Analyses factorielles 213
7.1 Analyse en Composantes Principales . . . 214
7.2 Analyse Factorielle des Correspondances . . . 223
7.3 Analyse des Correspondances Multiples . . . 228
7.4 Analyse Factorielle Multiple . . . 237

8 Classification non supervisée **243**
8.1 Classification Ascendante Hiérarchique 244
8.2 Méthode des K-means 252
8.3 Modèles de mélange 256

9 Méthodes usuelles de régression **263**
9.1 Régression simple 264
9.2 Régression multiple 270
9.3 Analyse de la variance 276
9.4 Analyse de la covariance 282
9.5 Régression logistique 287
9.6 Analyse discriminante linéaire 294
9.7 Arbres 302
9.8 Régression Partial Least Square (PLS) 312

10 Machine learning **321**
10.1 Calibration d'un algorithme avec caret 323
10.2 Forêts aléatoires 331
10.3 Régression sous contraintes 340
10.4 Gradient boosting 348
10.5 SVM 354
10.6 Réseaux de neurones et deep learning 362
10.7 Comparaison de méthodes 369

11 Divers **373**
11.1 Gestion de données manquantes 374
11.2 Analyse de texte 383
11.3 Fouille de graphe 393

Annexes **401**
A.1 Écriture d'une formule pour les modèles 401
A.2 Environnement RStudio 402
A.3 Le package Rcmdr 403

Bibliographie **405**

Index des fonctions **407**

Index **410**

Première partie

R et son fonctionnement

Chapitre 1

Concepts

Ce chapitre propose une présentation générale du logiciel R. On décrit tout d'abord brièvement l'environnement RStudio et l'outil RMarkdown. Sont ensuite expliqués les objets manipulés par R (vecteurs, matrices, etc.) ainsi que quelques notions sur les fonctions. On conclut par une présentation des packages, ou bibliothèques de programmes externes, qui seront d'utilité constante dans la suite de l'ouvrage.

1.1 Installation de R et RStudio

R est un logiciel de statistique distribué librement par le CRAN (Compréhensive R Archive Network) à l'adresse suivante `http://cran.r-project.org`. L'installation de R varie selon le système d'exploitation (Windows, Mac OS X ou Linux) mais les fonctionnalités sont exactement les mêmes et les programmes sont portables d'un système à l'autre. L'installation de R est très simple, il suffit de suivre les instructions.

L'utilisation du logiciel R est facilitée par l'utilisation d'un environnement de développement intégré (IDE). Nous préconisons et utiliserons dans ce livre l'environnement RStudio. Son installation suppose d'avoir préalablement installé R mais se fait sans difficulté (`https://www.rstudio.com`).

L'environnement RStudio se présente sous la forme d'une fenêtre globale (voir Fig. 1.1) scindée en 4 sous-fenêtres distinctes :
- la fenêtre de scripts (en haut à gauche),
- la console (en bas à gauche),
- la fenêtre d'environnement et d'historique (en haut à droite),
- la fenêtre des fichiers, graphes, packages et d'aide (en bas à droite).

Pour exécuter du code R écrit dans la fenêtre de scripts, on appuie sur le bouton `Run`. Le code s'affiche dans la console R avec un prompt de la forme : `>`. Par exemple, si on exécute `2+3.2`, on obtient dans la console :

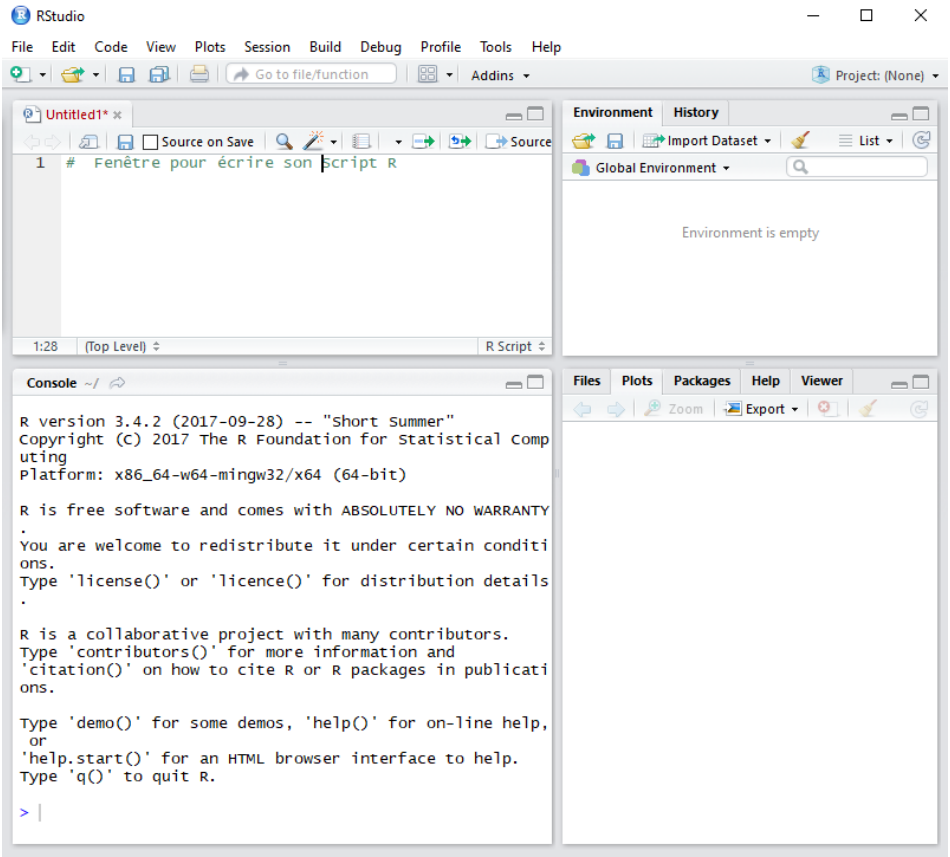


FIGURE 1.1 – Environnement RStudio.

```
> 2+3.2
[1] 5.2
```

Le `[1]` signifie que la première (et unique) coordonnée du vecteur de résultat vaut `5.2`. Quand R attend la suite d'une instruction, l'invite de commande devient : `+`. Par exemple, si on exécute `1-`, alors s'affiche dans la console :

```
> 1-
+
```

R attend la deuxième partie de la soustraction et l'invite de commande est `+`. Pour soustraire 2 à 1, on tape `2` et en exécutant on obtient :

```
+ 2
[1] -1
```

Généralement, il s'agit d'un oubli de) ou ". Il suffit alors de proposer une (ou plusieurs) parenthèse(s) ou double guillemet pour terminer la commande, ou bien taper « Escap » pour récupérer la main.

Remarques

Soulignons que R peut être utilisé sans RStudio. Par ailleurs, des interfaces graphiques facilitent également l'utilisation de R : citons par exemple Rcmdr (prononcer « pkgR commandeur ») dont l'utilisation est décrite en annexe A.3 page 403. Pour plus de détails sur RStudio, voir l'annexe A.2 page 402.

Il est conseillé de mettre à jour sa version de R et de RStudio régulièrement (au moins une fois par an). Pour R, voir la section 1.7.3 et pour RStudio faire `Help` puis `Check for updates`.

1.2 Environnement de travail

Répertoire de travail : pour chaque projet, nous recommandons d'utiliser un répertoire de travail permettant de sauvegarder tout ce qui est lié au projet : jeu(x) de données, script(s), graphe(s), environnement, etc.

Lors du démarrage d'une session R, le répertoire de travail peut être connu par la commande (get working directory) :

```
> getwd()
```

Pour modifier, si besoin, le répertoire de travail on utilise la fonction **setwd**() :

```
> setwd("C://MonNouveauRepertoire/")
```

On peut de façon équivalente cliquer dans la fenêtre RStudio en bas à droite sur l'onglet `Files`, puis sélectionner son répertoire de travail, appuyer sur le bouton `More` et cliquer sur `Set As Working Directory`.

Script : quand on écrit un script, on le sauvegarde régulièrement dans son répertoire de travail en cliquant sur `File` puis `Save`, ou sur la disquette (en haut à gauche dans la fenêtre du script). Le script est sauvegardé dans un fichier dont l'extension est `.R`.

Environnement : par défaut, R conserve en mémoire les objets (variables, tableaux de résultats, etc). On retrouve tous ces objets dans la fenêtre environnement (en haut à droite). Ceux-ci peuvent être sauvegardés dans une image de la session nommée **.RData**, grâce à la commande **save.image**() ou en quittant. Les objets sauvegardés seront alors disponibles pour une session future grâce à la fonction

load ou en allant dans la fenêtre en bas à droite, dans l'onglet `Files` et en cliquant sur le fichier `.RData`.

Une alternative à l'ensemble de ces étapes (spécification du répertoire de travail, sauvegarde du script `.R`, de l'image `.RData` et des résultats) consiste à créer un projet RStudio en cliquant sur `File` puis `New Project...`, `New Directory` et enfin `Project`. Un fichier avec l'extension `.Rproj` est créé et en cliquant dessus, on réouvrira tout le projet.

1.3 Introduction à RMarkdown

Pour travailler avec R, on peut simplement écrire son code dans un script. Cependant, nous recommandons l'utilisation de l'outil RMarkdown qui permet d'assurer un travail reproductible où le code, les résultats de l'exécution du code (comme des sorties numériques ou graphiques) et les commentaires sont dans un document unique.

1.3.1 Création d'un document RMarkdown

Dans RStudio, à partir de l'icône `File`, sélectionner `New File` puis `R Markdown`. Il est alors demandé de préciser le type de document de sortie (document texte, diaporama, ...), le titre et l'auteur du document ainsi que le format de sortie, HTML, PDF (si Latex est installé sur son ordinateur) ou Word. Si le type de document de sortie est du texte, nous pouvons préférer faire un Notebook en faisant `File`, sélectionner `New File` puis `R Markdown` qui permet d'actualiser la sortie html à chaque sauvegarde de façon instantanée.

Ceci fait, une nouvelle fenêtre de script s'ouvre dans RStudio : elle propose par défaut un exemple de document RMarkdown qui doit permettre à l'utilisateur de comprendre quelques éléments basiques de syntaxe avant de se lancer dans ses propres réalisations. Ensuite on enregistre son fichier (soit en faisant `File` puis `Save`, soit en faisant `Knit`) au format `.Rmd`. Enfin, via le bouton `Knit`, on visualise la sortie correspondant au format précédemment choisi.

1.3.2 Bases du langage

`Markdown` est un langage de balisage qui offre une syntaxe facile à utiliser. Il permet de structurer le document en sections et sous-sections en gérant la profondeur des titres, de créer des listes numérotées ou non, de faire apparaître du texte en gras, en italique, d'insérer des liens, des images, etc. Un aide-mémoire des fonctions principales est disponible dans RStudio via `Help` → `Cheatsheets` → `R Markdown Cheat Sheet`. En voici quelques exemples :

```
# Titre de niveau 1
## Titre de niveau 2
```

```
### Titre de niveau 3

*Ecrire en italique*
**Ecrire en gras**

Insérer un lien : <https://cran.r-project.org/>

Listes  :
* item 1
* item 2
1. item numero 1
2. item numero 2

<span style="color: blue"> Ecrire en couleur </span>
```

1.3.3 Insertion du code R

Le code R est dans des « chunks », par exemple dans le document ouvert par défaut, le premier chunk est :

```
```{r cars}
summary(cars)
```
```

On insère un nouveau « chunk » grâce au bouton `Insert` ou au raccourci clavier `ctrl+alt+i`. On écrit le code R à l'intérieur du « chunk ».

```
```{r}
2+2
```
```

Pour exécuter le chunk, on appuie sur le bouton `Run` puis `Run current chunk` (ou en appuyant sur la flèche verte à coté du chunk). Il est souvent nécessaire d'exécuter les chunks précédents, ce qui peut être fait avec `Run` puis `Run all chunks above`.

Si maintenant on appuie sur le bouton `Knit`, le code ainsi que son résultat apparaissent dans la sortie au format choisi. En spécifiant l'option `echo=FALSE`, il est possible d'afficher les sorties sans faire apparaître le code correspondant :

```
```{r, echo=FALSE}
2+2
```
```

D'autres options permettent par exemple :
– d'afficher uniquement le code sans les résultats (`results = "hide"`) ;
– de ne pas afficher les graphes (`fig.show = "hide"`) ;

– de préciser la taille des graphiques réalisés dans la fenêtre chunk (`fig.height=4, fig.width=6`) ;
– de masquer le code et la sortie, le code étant cependant pris en compte pour les chunks suivants (`include=FALSE`).

Pour plus d'options, faire `Help` → `Cheatsheets` → `R Markdown Cheat Sheet`.

1.4 Les différentes aides

Pour obtenir l'aide d'une fonction, par exemple la fonction **mean**, il suffit d'exécuter le code R :

```
> help(mean)
```

ou

```
> ?mean
```

Dans RStudio, l'aide est également accessible par l'onglet `Help` dans la fenêtre des fichiers, graphes, packages et d'aide (en bas à droite) en écrivant le nom de la fonction dans le moteur de recherche.

Dans l'aide d'une fonction, on trouve la liste des arguments à renseigner avec éventuellement leur valeur par défaut, la liste de résultats retournée par la fonction et le nom de la personne qui l'a programmée. En fin d'aide, un ou plusieurs exemples d'utilisation de la fonction sont souvent donnés et peuvent être copiés-collés dans R afin de comprendre son utilisation.

D'autres ressources permettent d'obtenir de l'aide :
– sur le site du CRAN `https://cran.r-project.org` avec des manuels, des mailing lists, des foires aux questions, des tasks view qui recensent des packages par domaines ce qui facilite l'exploration thématique des fonctionnalités de R ;
– une recherche directe sur le web à l'aide d'un moteur de recherche avec les mots-clés R et `CRAN` ;
– le site `https://www.r-bloggers.com`, étoffé et souvent pertinent ;
– des livres et des collections spécifiques (*Pratique R*, *use R* de *Springer*) ou revues spécifiques (*Journal of Statistical Software* et le *R journal*) ;
– des vidéos sur Internet de tutoriels ou de conférences sur R comme les *Rencontres R* en France et la conférence *useR !* à l'international.

1.5 Les objets R

R utilise des fonctions ou des opérateurs qui agissent sur des objets (vecteurs, matrices, etc.). Toutes les fonctions de cet ouvrage sont notées en caractères gras. Cette partie est consacrée à la présentation et à la manipulation des différents objets R.

1.5.1 Création, affichage, suppression

La création d'un objet peut se faire par affectation avec un des trois opérateurs « `<-` », « `->` », « `=` » en donnant un nom à cet objet :

```
> b <- 41.3  # crée l'objet b en lui donnant la valeur 41.3
> x <- b     # x reçoit la valeur b
> x = b      # x reçoit la valeur b
> b -> x     # x reçoit la valeur b
```

Nous utilisons dans la suite de cet ouvrage l'opérateur d'affectation « `<-` ». Le symbole « `#` » indique que tout le reste de la ligne ne sera pas interprété par R : cela permet d'ajouter des commentaires dans un programme.
Si un objet n'existe pas, l'affectation le crée. Sinon l'affectation écrase la valeur précédente sans message d'avertissement. On affiche la valeur d'un objet `x` via la commande :

```
> print(x)
```

ou plus simplement :

```
> x
```

Par défaut R conserve en mémoire tous les objets créés dans la session. Il est donc recommandé de supprimer régulièrement des objets. Pour connaître les objets de la session, on utilise les fonctions **objects**`()` ou **ls**`()`. Dans RStudio, ils sont consultables via l'onglet `Environment` de la fenêtre d'environnement et d'historique (en haut à droite). Pour supprimer l'objet `x`, on tape :

```
> rm(x)
```

et pour en supprimer plusieurs :

```
> rm(objet1,objet2,objet3)
```

Enfin, pour supprimer une liste d'objets qui possèdent une partie de leur nom en commun, par exemple la lettre `a`, on utilise :

```
> rm(list=ls(pattern=".*a.*"))
```

Dans RStudio, on peut supprimer tous les objets de sa session en appuyant sur le balai dans la fenêtre en haut à droite d'environnement.

1.5.2 Le mode d'un objet

Avant d'aborder les différents objets de R, il faut connaître les principaux modes de ces objets. Ces derniers sont :
- null (objet vide, que nous notons nul) : `NULL` ;
- logical (booléen, que nous notons logique) : `TRUE`, `FALSE` ou `T`, `F` ;
- numeric (réel, entier, que nous notons numérique) : `1`, `2.3222`, `pi`, `1e-10` ;
- complex (nombre complexe, que nous notons complexe) : `2+0i`, `2i` ;
- character (chaîne de caractères, que nous notons caractère) : `'hello'`, `"K"`.

Pour connaître le mode d'un objet `x` de R, il suffit d'exécuter la commande :

```
> mode(x)
```

Il est aussi possible de tester l'appartenance d'un objet `x` à un mode particulier. Le résultat est un booléen qui prend les valeurs `TRUE` ou `FALSE` :

```
> is.null(x)
> is.logical(x)
> is.numeric(x)
> is.complex(x)
> is.character(x)
```

Il est possible de convertir un objet `x` d'un mode à un autre de façon explicite grâce aux commandes :

```
> as.logical(x)
> as.numeric(x)
> as.complex(x)
> as.character(x)
```

Il faut cependant être prudent quant à la signification de ces conversions. R retourne toujours un résultat à une instruction de conversion même si cette dernière n'a pas de sens. À titre indicatif, nous avons le tableau de conversion suivant :

De	en	Fonction	Conversions
logique	numérique	**as.numeric**	`FALSE` → 0 `TRUE` → 1
logique	caractère	**as.character**	`FALSE` → `"FALSE"` `TRUE` → `"TRUE"`
caractère	numérique	**as.numeric**	`"1"`, `"2"`, ... → 1, 2, ... `"A"`, ... → `NA`

caractère	logique	**as.logical**	`"FALSE"`, `"F"` → `FALSE`
			`"TRUE"`, `"T"` → `TRUE`
			autres caractères → `NA`

numérique	logique	**as.logical**	0 → `FALSE` autres nombres → `TRUE`
numérique	caractère	**as.character**	1, 2, ... → `"1"`, `"2"`, ...

Un objet a deux attributs intrinsèques : son mode **mode** et sa longueur **length**. Il existe aussi des attributs spécifiques qui diffèrent selon les objets : **dim**, **dimnames**, **class**, **names**. On demande la liste de ces attributs en exécutant la commande :

```
> attributes(objet)
```

1.5.3 La valeur manquante

Pour différentes raisons, il se peut que certaines données ne soient pas collectées lors d'une expérience. On parle alors de données manquantes. Elles ne sont donc pas disponibles pour l'utilisateur et R les note `NA` pour « Not Available ». Ce n'est pas un véritable mode et il possède ses propres règles de calcul. Par exemple :

```
> x <- NA
> print(x+1)
> NA
```

Pour savoir où se trouvent les valeurs manquantes d'un objet `x`, on écrit :

```
> is.na(x)
```

Ceci retourne un booléen de même longueur que `x`. La question est posée élément par élément. Dans le cas d'un vecteur, cela renvoie un vecteur logique de même longueur que `x` avec `TRUE` si l'élément correspondant de `x` est `NA` et `FALSE` sinon.

Mentionnons aussi les valeurs spéciales `Inf` pour l'infini et `NaN` pour « Not a Number », valeurs résultant de problèmes calculatoires, par exemple **exp**`(`**exp**`(10))` ou **log**`(-2)` respectivement.

1.5.4 Les vecteurs

Le vecteur est un objet atomique, c'est-à-dire d'un mode unique (null, logique, numérique, etc.), composé d'un ensemble de valeurs appelées composantes, coordonnées ou éléments. L'attribut longueur, obtenu par la fonction **length**, donne le nombre d'éléments du vecteur.

Les vecteurs numériques

Pour construire un vecteur, différentes solutions sont possibles. En voici quelques-unes :

– Construction par la fonction collecteur **c** ; les éléments sont séparés par des virgules :

```
> x <- c(5.6,-2,78,42.3) # vecteur de numériques à 4 éléments
> x
[1]  5.6 -2.0 78.0 42.3
```

```
> x <- c(x,3,c(12,8))    # vecteur à 7 éléments
> x
[1]  5.6 -2.0 78.0 42.3  3.0 12.0  8.0
> x <- 2 # vecteur de longueur 1
> x
[1] 2
```

– Construction par l'opérateur séquence « : » :

```
> 1:6
[1] 1 2 3 4 5 6
```

– Construction par la fonction **seq** :

```
> seq(1,6,by=0.5)
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0
> seq(1,6,length=5)
[1] 1.00 2.25 3.50 4.75 6.00
```

– Construction par la fonction **rep** :

```
> rep(1,4)
[1] 1 1 1 1
> rep(c(1,2),each=3)
[1] 1 1 1 2 2 2
```

– Construction par la fonction **scan**. R demande alors d'entrer les éléments au fur et à mesure. Avec l'argument `n=3`, on précise que trois éléments seront collectés. Si on ne précise pas `n`, on arrête de collecter en mettant une valeur vide.

```
> note <- scan(n=3)
1: 12
2: 18
3: 15
Read 3 items
```

Remarque

Noter que le mode `numeric` peut être de deux types : `double` ou `integer`. Le type `integer` utilisant près de 2 fois moins de mémoire que le type `double`, il est préférable de l'utiliser pour stocker et travailler sur des données de grande taille, quand celles-ci prennent des valeurs entières bien sûr. Voici un petit exemple qui illustre ce phénomène :

```
> A = rep(1,1000)     # création d'un vecteur de 1
> typeof(A)
[1] "double"          # type de stockage réel
> object.size(A)
8048 bytes            # taille du vecteur stocké en réels
> object.size(as.integer(A))
4048 bytes            # taille du vecteur converti en entiers
```

Toute opération effectuée sur un vecteur d'entiers risque de retourner un résultat stocké comme réel. Par exemple, en ajoutant 1 à un vecteur d'entiers, le résultat est stocké en réel, sauf si l'on précise explicitement que la valeur 1 ajoutée est un entier grâce au codage 1L. La lettre « L » majuscule qui suit directement l'entier indique explicitement que l'on ajoute un entier, et que le type du résultat (`double` ou `integer`) doit rester le même que celui du vecteur :

```
> typeof(as.integer(A)+1)
[1] "double"          # type de stockage réel
> object.size(as.integer(A)+1)
8048 bytes            # taille importante de l'objet

> typeof(as.integer(A)+1L)
[1] "integer"         # type de stockage entier
> object.size(as.integer(A)+1L)
4048 bytes            # taille plus petite
```

Ainsi, pour initialiser le vecteur `A`, on préférera utiliser le codage entier en utilisant le « L » :

```
> A = rep(1L,1000)    # vecteur de 1 stocké en entiers
> object.size(A)
4048 bytes
```

Les vecteurs de caractères

Il est possible de créer des vecteurs de caractères de la même façon, en utilisant les fonctions **c** ou **rep**. Par exemple :

```
> x <- c("A","BB","C1")
> x
[1] "A"  "BB" "C1"
> x <- rep('A',5)
> x
[1] "A" "A" "A" "A" "A"
```

Même si R interprète indifféremment `"` et `'`, nous utilisons dorénavant `"`. Il est aussi possible de créer des vecteurs de caractères grâce au vecteur `letters` qui contient les lettres de l'alphabet. La fonction **format** permet la mise en forme de données numériques en chaîne de caractères de même longueur (voir aussi la fonction **toString**).

On peut aussi manipuler différents objets R et les concaténer ou en extraire une partie. Pour la concaténation, on utilise la fonction **paste** :

```
> paste("X",1:5,sep="-")
[1] "X-1" "X-2" "X-3" "X-4" "X-5"
> paste(c("X","Y"),1:5,"txt",sep=".")
[1] "X.1.txt"  "Y.2.txt"  "X.3.txt"  "Y.4.txt"  "X.5.txt"
> paste(c("X","Y"),1:5,sep=".",collapse="+")
[1] "X.1+Y.2+X.3+Y.4+X.5"
```

L'argument `collapse` rassemble tous les éléments dans un vecteur de longueur 1. Pour l'extraction on utilise la fonction **substr** :

```
> substr("livre",2,5)
[1] "ivre"
```

Ceci extrait de `livre` les caractères de rangs 2 à 5, ce qui donne `"ivre"`.

La fonction **grep** cherche si un motif (une suite de caractères) est contenue dans chaque élément d'un vecteur de caractères. La fonction **gsub** permet de remplacer les motifs :

```
> txtvec <- c("arm","foot","lefroo", "bafoobar")
> grep("foo", txtvec)
[1] 2 4
> gsub("foo", txtvec, replacement = "DON")
[1] "arm"      "DONt"     "lefroo"   "baDONbar"
```

Les vecteurs logiques

Les vecteurs de booléens sont en général générés grâce à des opérateurs logiques : « `>` », « `>=` », « `<` », « `<=` », « `==` », « `!=` », etc. Ils peuvent aussi être générés par les fonctions **seq**, **rep**, **c**. Ils permettent des sélections complexes (voir § 1.5.4, p. 15) ou des opérations de conditions (voir § 4.1, p. 109). Prenons l'exemple suivant :

```
> 1>0
[1] TRUE
```

Cette commande retourne un vecteur logique de longueur 1 qui est `TRUE` puisque 1 est plus grand que 0. La commande

```
> x>13
```

retourne un vecteur logique de même longueur que `x`. Ses éléments ont la valeur `TRUE` quand l'élément correspondant satisfait la condition (ici strictement supérieur à 13) et la valeur `FALSE` s'il ne la satisfait pas. Lors d'opérations arithmétiques, les logiques sont transformés en numériques avec la convention que les `FALSE` sont transformés en 0 et les `TRUE` en 1. Voyons cela sur un exemple. Nous créons un objet `test` qui est le vecteur de logiques (`FALSE,FALSE,TRUE`) puis nous calculons le produit terme à terme de 2 vecteurs :

```
> x <- c(-1,0,2)
> test <- x>1
> (1+x^2)*test
[1] 0 0 5
```

On peut aussi utiliser les fonctions **all** ou **any**. La fonction **all** renvoie `TRUE` si tous les éléments satisfont la condition et `FALSE` sinon. La fonction **any** renvoie `TRUE` dès que l'un des éléments satisfait la condition, `FALSE` sinon :

```
> all(x>1)
[1] FALSE
> any(x>1)
[1] TRUE
```

Sélection d'une partie d'un vecteur

Elle s'opère avec l'opérateur de sélection `[ ]` et un vecteur de sélection :

```
> x[vecteurdeselection]
```

Le vecteur de sélection peut être un vecteur d'entiers positifs, d'entiers négatifs ou de logiques. La sélection la plus naturelle est la sélection par des vecteurs d'entiers positifs. Les entiers sont les indices des éléments à sélectionner et doivent être compris entre 1 et **length**`(x)`. La longueur du vecteur d'indices peut être quelconque :

```
> v <- 1:100
> v[6] # donne le sixième élément de v
> v[6:8] # donne les 6ème, 7ème et 8ème éléments de v
> v[c(6,6,1:2)] # donne les 6ème, 6ème, 1er et 2ème éléments de v
> v[10:1] # donne les 10ème, 9ème, ..., 1er éléments de v
```

Une autre méthode consiste à enlever les éléments du vecteur que l'on ne souhaite pas conserver : c'est la sélection par des vecteurs d'entiers négatifs. Le vecteur d'indices indique les indices des éléments à exclure du résultat :

```
> v[-(1:5)] # donne v sans ses 5 premiers éléments
> v[-c(1,5)] # donne v sans le premier et le cinquième élément
```

Une autre façon de procéder consiste à sélectionner des éléments du vecteur en fonction de leur valeur ou d'autres éléments provenant d'autres objets R. Cela conduit à la sélection par des vecteurs de logiques. Cette sélection permet l'extraction d'éléments particuliers que l'on sait caractériser par une périphrase ou par une condition logique : « l'élément de sexe `F` » ou « l'élément qui possède une valeur inférieure à 5 et (ou) supérieure ou égale à 12 » :

```
> v <- 1:15
> print(v)
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> v[(v<5)]
[1] 1 2 3 4
> v[(v<5)&(v>=12)] # & signifie "et"
integer(0)
```

Aucun élément n'est à la fois inférieur à 5 et supérieur à 12, la fonction retourne alors l'ensemble vide, spécifié par `integer(0)`.

```
> v[(v<5)|(v>=12)]  # | signifie "ou"
[1]  1  2  3  4 12 13 14 15
```

On peut aussi sélectionner les valeurs d'un vecteur à partir des valeurs d'un autre vecteur de même longueur :

```
> T <- c(23, 28, 24, 32)
> O3 <- c(80, 102, 87, 124)
> O3[T>25]
[1] 102 124
```

Cas pratiques de sélection

Nous pouvons substituer à un sous-ensemble sélectionné des valeurs nouvelles :

```
> x[is.na(x)] <- 0 # les éléments NA de x reçoivent la valeur 0
> y[y<0]<- -y[y<0]
```

Cette dernière manœuvre revient à prendre la valeur absolue :

```
> y <- abs(y)
```

Une opération courante consiste à trouver le maximum ou le minimum d'un vecteur. Voici deux possibilités pour le minimum :

```
> which.min(x)
> which(x==min(x))
```

L'opérateur logique « == » permet de tester l'égalité et renvoie un booléen, ce qui peut être utile lorsque le minimum est atteint plusieurs fois. A contrario, dans ce cas, la fonction **which.min** ne renvoie que le plus petit indice où ce minimum est atteint.

1.5.5 Les matrices

Les matrices sont des objets atomiques, c'est-à-dire de même mode ou type pour toutes les valeurs. Chaque valeur de la matrice peut être repérée par son numéro de ligne et son numéro de colonne. Les deux attributs intrinsèques d'un objet R sont la longueur **length**, qui correspond ici au nombre total d'éléments de la matrice, et le mode **mode**, qui correspond ici au mode des éléments de cette matrice. Les matrices possèdent également l'attribut de dimension **dim**, qui retourne le nombre de lignes et le nombre de colonnes. Elles peuvent aussi posséder un attribut optionnel **dimnames** (voir § 1.5.7, p. 25). Voici les principales façons de créer une matrice. La plus utilisée est la fonction **matrix** qui prend en arguments le vecteur d'éléments et le nombre de lignes ou de colonnes de la matrice :

```
> m <- matrix(c(1,17,12,3,6,0),ncol=2)
> m
     [,1] [,2]
[1,]    1    3
[2,]   17    6
[3,]   12    0
> m <- matrix(1:8,nrow=2,byrow=TRUE)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

Notons que par défaut R range les valeurs dans une matrice par colonne. Pour ranger les éléments par ligne, on utilise l'argument `byrow`.

Lorsque la longueur du vecteur est différente du nombre d'éléments de la matrice, R remplit toute la matrice. Si le vecteur est trop grand, il prend les premiers éléments, si le vecteur est trop petit, R le répète (et envoie un avertissement) :

```
> m <- matrix(1:4,nrow=3,ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    4    3
[2,]    2    1    4
[3,]    3    2    1
```

Il est possible de remplir une matrice d'un élément unique sans avoir à créer le vecteur des éléments :

```
> un <- matrix(1,nrow=2,ncol=4)
> un
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    1    1    1
```

Un vecteur n'est pas considéré par R comme une matrice. Il est cependant possible de transformer un vecteur en une matrice unicolonne avec la fonction **as.matrix** :

```
> x <- seq(1,10,by=2)
> x
[1] 1 3 5 7 9
> as.matrix(x)
     [,1]
[1,]    1
[2,]    3
[3,]    5
[4,]    7
[5,]    9
```

Sélection d'éléments ou d'une partie d'une matrice

L'emplacement d'un élément dans une matrice est en général donné par le numéro de sa ligne et le numéro de sa colonne. Ainsi, pour sélectionner l'élément (i, j) de la matrice `m`, il faut écrire :

```
> m[i,j]
```

Il est rare qu'on ait besoin de ne sélectionner qu'un élément dans une matrice. Usuellement, on sélectionne une ou plusieurs lignes et/ou une ou plusieurs colonnes. Voyons les différents cas rencontrés en pratique :

- Sélection par des entiers positifs :

  ```
  > m[i,]
  ```

  Attention, cela retourne la ligne i sous la forme d'un vecteur.

  ```
  > m[i,,drop=FALSE]
  ```

  donne la ligne i sous la forme d'une matrice uniligne et non plus d'un vecteur, ce qui permet le cas échéant de conserver le nom de la ligne.

  ```
  > m[,c(2,2,1)]
  ```

  donne la 2^{e}, la 2^{e} et la 1re colonnes : c'est donc une matrice à trois colonnes.
- Sélection par des entiers négatifs :

  ```
  > m[-1,]     # matrice m sans sa première ligne
  > m[1:2,-1]  # 2 premières lignes de m privée de sa 1ère colonne
  ```

- Sélection par des logiques :

  ```
  > m <- matrix(1:8,ncol=4,byrow=TRUE)
  > m
       [,1] [,2] [,3] [,4]
  [1,]    1    2    3    4
  [2,]    5    6    7    8
  ```

L'instruction suivante retourne uniquement les colonnes de `m` pour lesquelles la valeur sur la première ligne est strictement supérieure à 2 :

```
> m[,m[1,]>2]
     [,1] [,2]
[1,]    3    4
[2,]    7    8
```

C'est donc une matrice, tandis que l'instruction suivante retourne un vecteur contenant les valeurs de `m` supérieures à 2 :

```
> m[m>2]
[1] 5 6 3 7 4 8
```

L'instruction suivante remplace les valeurs de `m` supérieures à 2 par des `NA` :

```
> m[m>2] <- NA
> m
     [,1] [,2] [,3] [,4]
[1,]    1    2   NA   NA
[2,]   NA   NA   NA   NA
```

Calculs sur les matrices

```
> m <- matrix(1:4,ncol=2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> n <- matrix(3:6,ncol=2,byrow=T)
> n
     [,1] [,2]
[1,]    3    4
[2,]    5    6
> m+n
     [,1] [,2]
[1,]    4    7
[2,]    7   10
> m*n    # produit élément par élément
     [,1] [,2]
[1,]    3   12
[2,]   10   24
> sin(m)  # sinus élément par élément
> exp(m) # exponentielle élément par élément
> sqrt(m) # racine carrée élément par élément
> m^4    # puissance quatrième élément par élément
```

Le tableau suivant donne les principales fonctions utiles en algèbre linéaire.

Fonction	Description
`X%*%Y`	produit de matrices
t`(X)`	transposition d'une matrice
diag`(5)`	matrice identité d'ordre 5
diag`(vec)`	matrice diagonale avec les valeurs du vecteur `vec` dans la diagonale
crossprod`(X,Y)`	produit croisé (**t**`(X)%*%Y`)
det`(X)`	déterminant de la matrice `X`
svd`(X)`	décomposition en valeurs singulières
eigen`(X)`	diagonalisation d'une matrice
solve`(X)`	inversion de matrice
solve`(A,b)`	résolution de système linéaire
chol`(Y)`	décomposition de Cholesky
qr`(Y)`	décomposition QR

Voici un exemple d'utilisation des fonctions **eigen** et **solve** :

```
> A <- matrix(1:4,ncol=2)
> B <- matrix(c(5,7,6,8),ncol=2)
> D <- A%*%t(B)
> D
     [,1] [,2]
[1,]   23   31
[2,]   34   46
> eig <- eigen(D)
> eig
$values
[1] 68.9419802  0.0580198

$vectors
           [,1]       [,2]
[1,] -0.5593385 -0.8038173
[2,] -0.8289393  0.5948762
```

La fonction **eigen** retourne les valeurs propres dans l'objet `values` et les vecteurs propres dans l'objet `vectors`. Si on veut extraire le premier vecteur propre, il suffit d'écrire :

```
> eig$vectors[,1]
```

Par ailleurs, pour résoudre le système d'équations suivant :

$$\begin{cases} 23\ x + 31\ y = 1 \\ 34\ x + 46\ y = 2 \end{cases}$$

qui revient à écrire `Dz=V`, avec `V=c(1,2)`, on utilise la fonction **solve** :

```
> V <- c(1,2)
> solve(D,V)
[1] -4  3
```

La solution est donc $x = -4$ et $y = 3$.

Opérations sur les lignes et colonnes

Nous présentons dans ce paragraphe quelques fonctions utiles :

- Dimensions : **dim**(X), **nrow**(X), **ncol**(X) donnent respectivement la dimension, le nombre de lignes et de colonnes de X.

```
> X <- matrix(1:6,ncol=3)
> X
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(X)
[1] 2 3
> nrow(X)
[1] 2
> ncol(X)
[1] 3
```

  Ces fonctions renvoient NULL si X est un vecteur.
- Concaténation : par colonne avec la fonction **cbind**, par ligne avec la fonction **rbind** (pour plus de détails, voir § 2.5, p. 49) :

```
> cbind(c(1,2),c(3,4))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- La fonction **apply** permet d'appliquer une fonction **f** aux lignes (MARGIN=1) ou aux colonnes (MARGIN=2) de la matrice (pour plus de détails, voir § 4.3, p. 114). Par exemple :

```
> apply(X,MARGIN=2,sum)  # sommes par colonne
[1]  3  7 11
> apply(X,1,mean) # moyennes par ligne
[1] 3 4
```

- **sweep**(X,2,ecartype,FUN="/") permet de diviser chacune des colonnes (car MARGIN=2) de X par un élément du vecteur ecartype (pour plus de détails, voir § 4.3, p. 118). Cet exemple permet de réduire les variables d'une matrice.

Remarque

On peut construire des cubes de données ou des tableaux de dimensions supérieures grâce à la fonction **array**. La manipulation de ces tableaux est similaire à celle des matrices.

1.5.6 Les facteurs

Les facteurs sont des vecteurs permettant la manipulation de données qualitatives. La longueur est donnée par la fonction **length**, le mode par **mode** et les modalités du facteur par **levels**. Ils forment une classe d'objets et bénéficient de traitements particuliers pour certaines fonctions, telle la fonction **plot** pour les graphiques. Les facteurs peuvent être non ordonnés (féminin, masculin) ou ordonnés (niveaux de ski). Trois fonctions permettent de créer les facteurs :

– la fonction **factor**

```
> sexe <- factor(c("M","M","F","M","F","M","M","M"))
> sexe
[1] M M F M F M M M
Levels: F M
```

On peut également nommer chaque niveau lors de la construction du facteur :

```
> sexe <- factor(c(2,2,1,2,1,2,1),labels=c("femme","homme"))
> sexe
[1] homme homme femme homme femme homme homme homme
Levels: femme homme
```

– la fonction **ordered**

```
> niveau <- ordered(c("débutant","débutant","champion","champion",
    "moyen","moyen","moyen","champion"),
    levels=c("débutant","moyen","champion"))
> niveau
[1] débutant débutant champion champion moyen moyen moyen champion
Levels: débutant < moyen < champion
```

– la fonction **as.factor**

```
> salto <- c(1:5,5:1)
> salto
 [1] 1 2 3 4 5 5 4 3 2 1
> salto.f <- as.factor(salto)
> salto.f
 [1] 1 2 3 4 5 5 4 3 2 1
Levels: 1 2 3 4 5
```

Pour retrouver les niveaux, le nombre de niveaux et l'effectif par niveau de `salto.f`, nous utilisons :

```
> levels(salto.f)
[1] "1" "2" "3" "4" "5"
> nlevels(salto.f)
[1] 5
> table(salto.f)
salto.f
1 2 3 4 5
2 2 2 2 2
```

La fonction **table** permet également de construire des tableaux croisés (§ 2.6, p. 52).

Pour convertir les facteurs en numériques, nous utilisons les ordres suivants :

```
> x <- factor(c(10,11,13))
> as.numeric(x)
[1] 1 2 3
> as.numeric(as.character(x))
[1] 10 11 13
```

Le détail de cette conversion est donné § 2.3.1 (p. 41).

1.5.7 Les listes

La liste est un objet hétérogène. C'est un ensemble ordonné d'objets qui n'ont pas toujours même mode ou même longueur. Les objets sont appelés composantes de la liste. Ces composantes peuvent avoir un nom. Les listes ont les deux attributs des vecteurs (**length** et **mode**) et l'attribut supplémentaire **names**. Les listes sont des objets importants car toutes les fonctions (cf. § 1.6) qui retournent plusieurs objets le font sous la forme d'une liste.

Création

```
> vecteur <- seq(2,10,by=3)
> matrice <- matrix(1:8,ncol=2)
> facteur <- factor(c("M","M","F","M","F","M","M","M"))
> ordonne <- ordered(c("débutant","débutant","champion",
  "champion","moyen","moyen","moyen","champion"),
  levels=c("débutant","moyen","champion"))
> maliste <- list(vecteur,matrice,facteur,ordonne)
> length(maliste)
[1] 4
> mode(maliste)
[1] "list"
```

Les composantes de la liste ainsi créé n'ont pas de noms. Nous pouvons les nmmer par :

```
> names(maliste)
NULL
> names(maliste) <- c("vec","mat","sexe","ski")
> names(maliste)
[1] "vec"  "mat"  "sexe" "ski"
```

Extraction

Pour extraire une composante de la liste, on peut toujours le faire en indiquant la position de l'élément que l'on souhaite extraire. Les `[[ ]]` permettent de retourner l'élément de la liste :

```
> maliste[[3]]
[1] M M F M F M M M
Levels: F M
> maliste[[1]]
[1] 2 5 8
```

On peut aussi utiliser le nom de l'élément s'il existe, ce que l'on peut écrire de deux façons :

```
> maliste$sexe
[1] M M F M F M M M
Levels: F M
> maliste[["sexe"]]
[1] M M F M F M M M
Levels: F M
```

Il est possible d'extraire plusieurs éléments d'une même liste, ce qui crée une sous-liste. Noter qu'ici on utilise `[ ]` et non `[[ ]]` :

```
> maliste[c(1,3)]
[[1]]
[1] 2 5 8

[[2]]
[1] M M F M F M M M
Levels: F M
```

Quelques fonctions sur les listes

- **lapply** applique une fonction (comme la moyenne, la variance, etc.) successivement à chacune des composantes.
- **unlist**`(maliste)` créer un seul vecteur contenant tous les éléments de la liste. Les éléments d'un vecteur étant nécessairement de même mode, il faut faire attention à la conversion automatique pratiquée par R.
- **c**`(liste1,liste2)` concatène deux listes.
- La fonction **write.infile**`(maliste,file="monchemin/monfichier")` du package FactoMineR écrit tous les objets de la liste `maliste` dans le fichier (au format csv) `monchemin/monfichier`.

La liste dimnames

C'est un attribut optionnel d'une matrice qui contient dans une liste à deux composantes les noms des lignes et des colonnes. Par exemple :

```
> X <- matrix(1:12,nrow=4,ncol=3)
> nomligne <- c("ligne1","ligne2","ligne3","ligne4")
> nomcol <- c("col1","col2","col3")
> dimnames(X) <- list(nomligne,nomcol)
> X
       col1 col2 col3
ligne1    1    5    9
ligne2    2    6   10
ligne3    3    7   11
ligne4    4    8   12
```

Il est possible de sélectionner des lignes et/ou des colonnes grâce aux dimnames :

```
> X[c("ligne4","ligne1"),c("col3","col2")]
       col3 col2
ligne4   12    8
ligne1    9    5
```

Il est aussi possible de modifier (ou supprimer) les noms des lignes et/ou des colonnes via :

```
> dimnames(X) <- list(NULL,dimnames(X)[[2]])
> X
     col1 col2 col3
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

1.5.8 Les data-frames

Les data-frames, ou jeux de données, sont des listes particulières dont les composantes sont de même longueur, mais dont les modes peuvent différer. Les tableaux de données usuellement utilisés en statistique sont souvent considérés comme des data-frames. En effet, un tableau de données est constitué de variables quantitatives et/ou qualitatives mesurées sur les mêmes individus.

Les principales manières de créer un data-frame consistent à utiliser les fonctions suivantes : **data.frame** qui permet de concaténer des vecteurs de même taille et éventuellement de modes différents ; **read.table** (cf. § 2.1) qui permet d'importer un tableau de données ; **as.data.frame** pour la conversion explicite. Par exemple, si on concatène le vecteur numérique `vec1` et le vecteur alphabétique `vec2` :

```
> vec1 <- 1:5
> vec2 <- c("a","b","c","c","b")
> df <- data.frame(nom.var1 = vec1, nom.var2 = vec2)
  nom.var1 nom.var2
1        1        a
2        2        b
3        3        c
4        4        c
5        5        b
```

Pour extraire des composantes du data-frame, on peut utiliser les méthodes présentées pour les listes ou pour les matrices.

Remarques

- Pour transformer une matrice en data-frame, on utilise la fonction **as.data.frame**.
- Pour transformer un data-frame en matrice, on utilise la fonction **data.matrix**.

1.5.9 La classe d'un objet

Comme nous l'avons évoqué plus haut, certains objets possèdent une *classe*. Cette notion (tout comme celle de méthode) est généralement consubstantielle au concept de *programmation orientée objets*. Sans entrer dans des détails théoriques, expliquons l'utilité des classes dans le fonctionnement de R. Certaines fonctions se comportent de façon différente suivant l'objet qu'elles reçoivent en paramètre. Prenons l'exemple de la fonction **print** pour illustrer notre propos. Appliquée à un data-frame, cette fonction affiche dans la console un résultat bien formaté sous forme tabulaire :

```
> df <- data.frame(x = 1:2, y = c(3, 5))
> print(df)
  x y
1 1 3
2 2 5
```

Ce comportement est dû au fait que **print** est une fonction *générique* qui, en fonction de la *classe* de l'objet qu'elle reçoit en paramètre, appelle la *méthode* adaptée. Ici il s'agit de la méthode **print.data.frame**. On peut forcer l'utilisation de la méthode par défaut (celle qui est appelée à défaut de méthode spécifique définie) :

```
> print.default(df)
$x
[1] 1 2

$y
[1] 3 5
```

```
attr(,"class")
[1] "data.frame"
```

Nous constatons que l'affichage est radicalement différent ! Pour terminer, notons qu'un objet peut posséder plusieurs classes. C'est le cas par exemple des data-tables sur lesquels nous reviendrons en détail (voir § 5.1). Donnons un exemple :

```
> library(data.table)
> dt <- data.table::data.table(df)
> dt
   x y
1: 1 3
2: 2 5
> class(dt)
"data.table" "data.frame"
```

L'affichage est subtilement modifié [1] par rapport à l'affichage du `data.frame` correspondant mais c'est bien la méthode **print.data.table** qui est appelée dans cette situation.

L'intérêt de posséder plusieurs classes est le suivant : lors de l'appel à une fonction générique, la méthode spécifique à la première classe est cherchée (c'est-à-dire ici **print.data.table**). Si cette méthode n'était pas trouvée, la méthode correspondant à la deuxième classe serait utilisée (ici **print.data.frame**)... sauf si elle n'existait pas ! En dernier recours, la méthode par défaut serait utilisée : **print.default**.

Il est ainsi possible pour des programmeurs d'étendre ou de modifier assez facilement des classes existantes.

1.6 Les fonctions

Une fonction est un objet R. Un grand nombre de fonctions sont prédéfinies dans R, par exemple **mean** qui calcule la moyenne, **min** qui calcule le minimum, etc. Il est également possible de créer ses propres fonctions (voir § 4.2). Une fonction admet des arguments en entrée et retourne un résultat en sortie.

Les arguments d'une fonction

Les arguments d'une fonction sont soit obligatoires soit optionnels. Dans ce dernier cas, ils possèdent une valeur par défaut.

Prenons l'exemple de la fonction **rnorm** qui génère des nombres aléatoires suivant une loi normale. Cette fonction admet trois arguments : `n` le nombre de valeurs, `mean` la moyenne et `sd` l'écart-type de la loi. Ces deux derniers sont fixés par défaut

1. Nous vous invitions à constater que la différence est encore plus visible si l'on construit au départ un data-frame plus grand avec la commande `df <- data.frame(x = 1:100, y = 1:100)`.

à 0 et 1. Afin de pouvoir retrouver les mêmes résultats d'une simulation à l'autre, il faut « fixer la graine » du générateur de nombres aléatoires à l'aide de la fonction `set.seed` à une valeur quelconque, ici 13.

```
> set.seed(13) # fixe la graine du générateur
> rnorm(n=3)
[1]  0.5543269 -0.2802719  1.7751634
```

Ces trois nombres ont donc été tirés selon une loi normale $\mathcal{N}(0,1)$. Si on souhaite tirer quatre nombres selon une loi normale de moyenne 5 et d'écart-type 0.5, nous utilisons :

```
> set.seed(13) # fixe la graine du générateur
> rnorm(n=4,mean=5,sd=0.5)
[1] 5.277163 4.859864 5.887582 5.093660
```

Il n'est pas indispensable de préciser le nom des arguments à condition de respecter leur ordre. Cet ordre est défini lors de la création de la fonction. La fonction **args** renvoie ces arguments (on peut aussi consulter l'aide de cette fonction) :

```
> args(rnorm)
function (n, mean = 0, sd = 1)
NULL
```

Certaines fonctions admettent des arguments optionnels qui ne sont pas nommés. Ils sont représentés par `...`, par exemple pour la fonction **plot** :

```
> args(plot)
function (x, y, ...)
NULL
```

Cette fonction trace `y` en fonction de `x`, les `...` permettant de contrôler l'aspect du graphique (voir § 3.1.1, p. 58).

Les sorties

La plupart des fonctions retournent plusieurs résultats et l'ensemble de ces résultats est contenu dans une liste. Pour visualiser l'ensemble des sorties, il est nécessaire de connaître l'ensemble des éléments de la liste en utilisant la fonction **names**. On utilise `[[j]]` pour extraire l'élément j de la liste ou encore $ si les éléments de la liste ont des noms (cf. § 1.5.7, p. 23).

1.7 Les packages

Un package, paquet ou bibliothèque de programmes externes, est simplement un ensemble de programmes qui complète et permet d'augmenter les fonctionnalités

de R. Un package est généralement dévolu à une méthode particulière ou à un domaine d'application spécifique.

Il existait environ 1 300 packages en 2008 et dix fois plus en 2018. Un certain nombre d'entre eux sont considérés comme indispensables (MASS, rpart, etc.) et sont fournis avec R. Les autres constituent des avancées plus ou moins récentes en statistique et science des données et peuvent être téléchargés librement sur le réseau CRAN. Par ailleurs, il existe de très nombreux packages disponibles dans des projets, par exemple BioConductor qui propose des packages dédiés au traitement des données génomiques.

1.7.1 Installation d'un package

L'installation est à faire une seule fois. Elle permet de mettre à disposition le package et ses fonctions sur l'ordinateur. Les packages sont disponibles sur le réseau du CRAN (`http://cran.r-project.org/`). Il en existe de nombreux miroirs, c'est-à-dire des copies exactes du site du CRAN. Dans RStudio, on peut installer un package à l'aide de l'onglet `Tools -> Install Packages` du menu déroulant, ou encore dans la fenêtre en bas à droite via l'onglet `Packages` puis `Install`. Ceci peut aussi se faire grâce à la fonction **install.packages** :

```
> install.packages("rpart",dependencies=TRUE)
```

Ensuite, choisir le miroir le plus proche de vous et sélectionner le package à installer, ici rpart. Les fonctions du package sont alors disponibles pour être utilisées.

Remarque
On peut aussi configurer l'utilisation d'un proxy (**help**`(download.file)`) ou configurer les emplacements de l'arborescence où seront installés les packages.

1.7.2 Utilisation d'un package

Une fois le package installé, il est possible d'utiliser ses fonctions de deux façons. Nous allons illustrer ces deux méthodes en simulant trois vecteurs de $\mathbb{R}^2$ selon une loi gaussienne de moyenne $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ et de variance $\begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$. Ceci est possible grâce à la fonction **mvrnorm** du package MASS. La simulation peut alors se faire :

– soit en chargeant le package, grâce aux fonctions **library** ou **require** qui rendent disponibles toutes les fonctions du package qui peuvent alors être appelées simplement par leur nom :

```
> set.seed(45)   # fixe la graine du générateur aléatoire
> require(MASS)  # ou library(MASS)
> mvrnorm(3,mu=c(0,1),Sigma=matrix(c(1,0.5,0.5,1),2,2))
```

– soit en appelant directement la fonction, sans charger le package, mais en précisant bien de quel package provient la fonction avec la structure générique `NomPackage::NomFonction` :

```
> set.seed(45)   # fixe la graine du générateur aléatoire
> MASS::mvrnorm(3,mu=c(0,1),Sigma=matrix(c(1,0.5,0.5,1),2,2))
```

La première façon de faire a l'avantage de charger le package une fois pour toutes, et toutes les fonctions du package sont alors disponibles. L'inconvénient est qu'avec l'augmentation très importante du nombre de packages, il est possible que plusieurs fonctions de packages différents aient le même nom. Seule la fonction du dernier package chargé sera disponible, les fonctions du même nom des autres packages étant alors masquées et indisponibles. Pour éviter ce problème, nous conseillons donc d'appeler les fonctions avec la seconde solution, nonobstant sa lourdeur syntaxique. Bien entendu, les deux codes donnent les mêmes résultats :

```
            [,1]      [,2]
[1,] -0.07788253 1.6681649
[2,] -1.05816423 0.8399431
[3,] -0.49608637 0.8387077
```

Remarque
Les fonctions **library** et **require** sont presque équivalentes. Cette dernière a l'avantage de retourner `FALSE` et non une erreur si le package n'a pas été préalablement installé sur l'ordinateur. Il est alors intéressant d'écrire les lignes de code suivantes dans votre programme pour le partager avec d'autres personnes. Ainsi, si la personne n'a pas préalablement le package installé sur son ordinateur, l'installation sera faite avant que le package soit chargé. Voici un exemple avec le package rpart :

```
> if (!require(rpart)) install.packages("rpart") # installe rpart si besoin
> require(rpart)                                 # ou library(rpart)
```

Pour avoir de l'aide sur un package, on peut taper :

```
> help(package="rpart")
```

Certains packages proposent des vignettes qui présentent en détail les fonctionnalités du package, avec éventuellement des tutoriaux par exemple.

1.7.3 Mise à jour des packages et de R

Certains packages sont en constante évolution, avec de nouvelles versions disponibles. Il est donc judicieux de les mettre à jour régulièrement. Pour cela, ils ne doivent pas être en cours d'utilisation. Le mieux est de mettre à jour les packages à l'ouverture d'une session R, en utilisant :

```
> update.packages(ask=FALSE)
```

de choisir un miroir proche de vous et de répondre aux éventuelles questions. Sous RStudio, ceci peut aussi se faire à partir de l'onglet `Tools` puis `Check for Package Updates`, ou dans la fenêtre en bas à droite via l'onglet `Packages` puis `Update`.

Attention, la mise à jour des packages est fonction de la version de R utilisée. Certains packages ne sont pas compatibles avec d'anciennes versions ou ne sont pas compilés pour celles-ci. Il est donc recommandé de mettre également à jour sa version de R de temps en temps.

Pour mettre à jour R, nous conseillons d'utiliser le package installr si vous êtes sous Windows. Vous pourrez ainsi réinstaller R puis mettre à jour tous vos packages déjà installés. Sous Mac et linux, vous devrez aller sur le site du CRAN pour réinstaller R puis écrire les lignes de code suivantes :

```
> packs = as.data.frame(installed.packages(.libPaths()[1]),
    stringsAsFactors = F)
> install.packages(packs$Package)   ## réinstallation des packages
```

1.8 Exercices

Exercice 1.1 (Creation de vecteurs)

1. Créer les 3 vecteurs ci-dessous à l'aide de la fonction **rep** :

```
vec1 = 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
vec2 = 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
vec3 = 1 1 2 2 2 3 3 3 3 4 4 4 4 4
```

2. Créer le vecteur ci-dessous avec la fonction **paste** :

```
vec4 = "A0)" "A1)" "A2)" "A3)" "A4)" "A5)" "A6)" "A7)"
       "A8)" "A9)" "A10)"
```

3. Le vecteur `letters` contient les 26 lettres de l'alphabet. À l'aide de la commande `==`, trouver la position de la lettre `q` dans l'alphabet et créer le vecteur de coordonnées `a1`, `b2`, et ainsi de suite jusqu'à `q??`, où `??` désigne la position de `q`.

Exercice 1.2 (Travailler avec les valeurs manquantes `NA`)

1. Fixer la graine du générateur de nombres aléatoires (**set.seed**) à la valeur 007 et créer un vecteur `vec1` de 100 nombres uniformément tirés entre 0 et 7 (**runif**). Calculer la moyenne et la variance des valeurs contenues dans `vec1`.

2. Créer un vecteur `vec2` égal à `vec1`. Fixer la graine du générateur de nombres aléatoires à la valeur 008, sélectionner 10 coordonnées au hasard (**sample**) et remplacer leur valeur par `NA` (valeur manquante). Trouver les coordonnées des valeurs manquantes grâce aux fonction **is.na** et **which**.

3. Calculer la moyenne et la variance de `vec2` : directement (sans gestion des `NA`) puis en utilisant l'argument `na.rm=TRUE` (en gérant les `NA`).

4. Créer un vecteur `vec3` en supprimant les coordonnées de `vec2` qui contiennent une valeur manquante et calculer la moyenne et la variance de ce vecteur `vec3`.

5. Créer un vecteur `vec4` en remplaçant les coordonnées de `vec2` qui contiennent une valeur manquante par la moyenne de `vec3`. Que valent la moyenne et la variance de `vec4` ?

6. Créer un vecteur `vec5` en remplaçant les coordonnées de `vec2` qui contiennent une valeur manquante par un tirage aléatoire selon une loi normale de moyenne et d'écart-type égaux à la moyenne et l'écart-type de `vec3` (**rnorm**). Que deviennent la moyenne et la variance de `vec5` ?

7. Même question que la précédente avec un tirage selon une loi uniforme entre le minimum et le maximum de `vec3` (**runif**) afin de former un vecteur `vec6`.

8. Créer `vec7` à partir de `vec2` en remplaçant les `NA` par un tirage aléatoire *avec remise* parmi les valeurs non manquantes de `vec2` (**sample**).

Exercice 1.3 (Création et inversion d'une matrice)

1. Créer la matrice `mat` suivante (avec les noms de lignes et de colonnes) :

```
        colonne 1 colonne 2 colonne 3 colonne 4
ligne-1         1         5         5         0
ligne-2         0         5         6         1
ligne-3         3         0         3         3
ligne-4         4         4         4         2
```

2. Créer le vecteur contenant les éléments diagonaux de `mat` (**diag**).

3. Créer une matrice contenant uniquement les deux premières lignes de `mat`.

4. Créer une matrice contenant uniquement les deux dernières colonnes de `mat`.

5. Créer une matrice contenant toutes les colonnes de `mat` sauf la troisième.

6. Calculer le déterminant puis inverser la matrice en utilisant les fonctions appropriées.

Exercice 1.4 (Sélection et tri dans un data-frame)

1. À partir du jeu de données `iris` disponible sous R (utiliser **data**`(iris)` pour le charger puis `iris[1:5,]` pour visualiser ses cinq premières lignes), créer un sous-jeu de données comportant uniquement les données de la modalité `versicolor` de la variable `species` (appeler ce nouveau jeu de données `iris2`).

2. Trier par ordre décroissant les données de `iris2` en fonction de la variable `Sepal.Length` (vous pourrez utiliser la fonction **order**).

Exercice 1.5 (Utilisation de la fonction apply)

1. Calculer les statistiques de base (moyenne, min, max, etc.) des trois variables du jeu de données `ethanol` disponible sous R dans le package lattice.

2. Calculer les quartiles de chacune des trois variables. Pour cela, vous pouvez utiliser la fonction **apply** avec la fonction **quantile**.

3. Toujours avec la fonction **apply**, calculer tous les déciles de chacune des trois variables en utilisant l'argument `probs` de la fonction **quantile**.

Exercice 1.6 (Sélection dans une matrice avec la fonction apply)

1. À partir de la matrice `mat` de l'exercice 1.3, créer une nouvelle matrice qui ne contient que les colonnes de `mat` dont tous les éléments sont strictement plus petits que 6 (utiliser **apply** et **all** pour sélectionner ces colonnes).

2. De même, créer une nouvelle matrice qui ne contient que les lignes de `mat` qui ne présentent aucun 0.

Exercice 1.7 (Utilisation de la fonction lapply)

1. Charger le jeu de données `Aids2` du package MASS et le résumer (**summary**).

2. Utiliser les fonctions **is.numeric** et **lapply** pour créer un vecteur de booléens de longueur le nombre de colonnes de `Aids2` et dont chaque coordonnée est `TRUE` quand la colonne correspondante de `Aids2` n'est pas numérique, et `FALSE` quand la colonne correspondante de `Aids2` est numérique.

3. Sélectionner les variables qualitatives de `Aids2` et les affecter dans un nouveau data-frame nommé `Aids2.qual`.

4. Donner les modalités de chaque variable qualitative de `Aids2.qual` (**levels**).

Exercice 1.8 (Modalités des variables qualitatives et sélection)

1. Charger le jeu de données `Aids2` du package MASS.

2. Selectionner les lignes de `Aids2` dont la variable `sex` vaut `M` et dont la variable `state` ne vaut pas `Other`. Affecter le résultat de cette sélection à l'objet `res`.

3. Résumer `res` et vérifier que a) la variable `sex` n'a plus d'individu de sexe `F`, et b) la modalité `F` est toujours présente.

4. Imprimer les attributs de la variable `sex` (fonction **attributes**). Vérifier que a) il existe un attribut `levels` avec `F` et `M`, et b) la classe de l'objet est bien `factor`.

5. En utilisant **as.character**, transformer la variable `sex` de `res` en un objet `sexc` de type `character`. Vérifier que `sexc` n'a plus d'attributs.

6. En utilisant **as.factor**, transformer `sexc` en un objet nommé `sexf` de type facteur. Vérifier que `sexf` possède a) un attribut `level`, b) une classe de type `factor`, et c) aucune modalité `F`. Conclusion : transformer un facteur en caractère puis le retransformer en facteur permet de réinitialiser les niveaux.

7. En utilisant l'exercice 1.7, créer un vecteur de booléens de longueur égale au nombre de colonnes de `res` et dont chaque coordonnée vaut `TRUE` si la variable correspondante est une variable qualitative et `FALSE` si la variable correspondante est quantitative.

8. Transformer toutes les variables qualitatives de `res` en type `character` en utilisant la fonction **lapply**.

9. Les retransformer en variables qualitatives en utilisant **lapply**.

Chapitre 2

Manipuler les données

En statistique, les données constituent le point de départ de toute analyse. De fait, il est primordial de savoir maîtriser des opérations telles que l'importation, l'exportation, les changements de types, repérer les valeurs manquantes, concaténer les niveaux d'un facteur, etc. La plupart de ces notions sont exposées dans ce chapitre, lequel se veut à la fois synthétique et suffisant en pratique.

2.1 Importer des données

Les données sont initialement collectées, voire prétraitées avec un logiciel, que ce soit un tableur, un logiciel de statistique, etc. Chaque logiciel possédant son propre format de stockage, il existe plusieurs façons d'importer les données.

2.1.1 Importation d'un fichier texte

Le format texte (extension `.txt` ou `.csv`) est très commun. En général, les données sont dans un fichier avec les individus en lignes et les variables en colonnes. Le format texte est constitué de données qui sont toutes séparées par un séparateur de colonnes. Ainsi le tableau ci-dessous regroupe la mesure de quatre variables (taille, poids, pointure et sexe) pour trois individus et le séparateur de colonnes utilisé est le `";"`. La première colonne représente l'identifiant des individus, ici leur prénom : tous les prénoms doivent donc être différents. Cet identifiant est en général une chaîne de caractères ou le numéro de l'individu. Dans certains cas, les individus n'ont pas d'identifiant et/ou les variables n'ont pas de nom.

```
individu;taille;poids;pointure;sexe
roger;184;80;44;M
théodule;175,5;78;43;M
nicolas;158;72;42;M
```

Supposons que ces données soient contenues dans un fichier `donnees.csv` qui est dans un répertoire `DONNEES`. Si les données ont été saisies manuellement dans le fichier texte, veillez à bien retourner à la ligne en fin de dernière ligne ; cette opération est effectuée automatiquement par les tableurs. Les données sont importées dans l'objet `tablo` de la façon suivante :

```
> tablo <- read.table("DONNEES/donnees.csv",sep=";",header=TRUE,
          dec=",",row.names=1)
```

Le nom du fichier est entre guillemets (" ou ' mais pas les guillemets de traitements de texte comme Word). L'option `sep` indique que le caractère qui sépare les colonnes est ici `";"`. Pour un espace on utilise `" "` et pour une tabulation `"\t"`. L'argument `header` indique si la première ligne contient ou non les intitulés des variables. L'argument `dec` indique que le séparateur décimal est le caractère `","` (ici pour le nombre 175,5). Enfin, l'argument `row.names` indique que la colonne 1 n'est pas une variable mais l'identifiant des lignes, ici les prénoms des individus.

Attention, les utilisateurs de Windows sont habitués à spécifier les chemins à l'aide de « \ ». Pour R, même sous Windows, on utilise « / ». Ainsi, si le jeu de données est situé dans le lecteur logique `C:` dans le répertoire `Temp`, l'importation devient :

```
> tablo <- read.table("C:/Temp/donnees.csv",sep=";",header=TRUE,
          dec=",",row.names=1)
```

Le chemin peut aussi être une URL, il suffit alors d'écrire :

```
> decath <- read.table("https://r-stat-sc-donnees.github.io/decathlon.csv",
    sep=";",dec=".",header=T,row.names=1)
```

Quelquefois, un caractère spécial indique qu'il manque une valeur. Ce caractère s'appelle `"NA"` (Not Available) sous R. Il est important de spécifier le caractère pour les données manquantes. Considérons le fichier `donnees2.csv` suivant :

```
taille poids pointure sexe
184 80 44 "M"
175.5 78 43 "M"
. 72 42 "M"
178 . 40 "F"
```

et importons-le en modifiant la valeur par défaut de `na.strings` qui est `"NA"` :

```
> tablo <- read.table("DONNEES/donnees2.csv",sep=" ",header=TRUE,
          na.strings = ".")
```

L'option `quote` (non utilisée dans notre exemple) permet de préciser ce qui entoure une chaîne de caractères. Par défaut, il s'agit du double ou du simple guillemet.

L'argument `encoding` (non utilisée dans l'exemple) permet quant à lui de préciser l'encodage du fichier de données : ceci est important lorsque certains caractères sont accentués. Pour d'autres options permettant de contrôler finement l'importation, on se reportera à l'aide de la fonction **read.table**.

Remarquons que le résultat de l'importation est toujours un objet de type data-frame. Les types de variables sont « devinés » par la fonction d'importation. Celle-ci peut se tromper et il est donc nécessaire de vérifier le type de chacune des variables grâce à la fonction **summary**. Pour le tableau `tablo` issu de l'importation précédente, nous avons :

```
> summary(tablo)
     taille          poids          pointure     sexe
 Min.   :175.5   Min.   :72.00   Min.   :40.00   F:1
 1st Qu.:176.8   1st Qu.:75.00   1st Qu.:41.50   M:3
 Median :178.0   Median :78.00   Median :42.50
 Mean   :179.2   Mean   :76.67   Mean   :42.25
 3rd Qu.:181.0   3rd Qu.:79.00   3rd Qu.:43.25
 Max.   :184.0   Max.   :80.00   Max.   :44.00
 NA's   :  1.0   NA's   : 1.00
```

Ici, les types des variables sont corrects : quantitatif pour la taille, le poids et la pointure, qualitatif pour le sexe avec les deux modalités `F` et `M`. Le cas échéant, il est nécessaire de changer le type de la variable. Parfois, lorsque l'importation ne fonctionne pas, l'erreur peut provenir du fichier texte lui-même. Mentionnons quelques chausse-trappes classiques :

- séparateur de colonnes mal spécifié ;
- séparateur décimal mal spécifié (les variables risquent d'être considérées comme qualitatives) ;
- tabulation qui remplace un blanc ;
- nom de ligne ou de colonne comprenant une apostrophe ;
- encodage du fichier de donné lorsqu'il y a de caractères accentués ;
- problème dans un guillemet non fermé.

Dans le dernier cas, il est souvent difficile de trouver l'erreur. Une solution « rustique » est proposée en exercice 2.1.

Remarque

De nombreux jeux de données sont disponibles dans R ou dans des packages. Pour les charger il suffit d'écrire :

```
> data(iris)
```

2.1.2 Importation de données textes volumineuses

Pour importer des jeux de données volumineux très rapidement, il est conseillé d'utiliser la fonction **fread** du package data.table (voir § 5.1.1). Par défaut, cette

fonction devine automatiquement le séparateur de colonnes et si la première ligne correspond au nom des variables. Attention, cette fonction crée un objet de classe data-table, laquelle est différente de la classe data-frame.

```
> library(data.table)
> fread("DONNEES/donnees.csv")
```

2.1.3 Importation d'autres formats de données

En R, il est possible d'importer de nombreux formats de données comme SAS, SPSS, Excel, JSON, bases de données relationnelles, etc. Pour ce faire, il existe de nombreux packages spécifiques, par exemple :
- le package readxl pour importer les formats depuis Excel (.xls, .xlsx) ;
- le package sas7bdat pour importer depuis SAS ;
- le package foreign pour les formats SPSS (**read.spss**), STATA (**read.dta**) ;
- le package DBI pour importer des bases de données (voir § 5.3) ;
- le package jsonlite pour les fichiers JSON (voir § 5.3.2) ;
- le package rvest pour faire du web scraping (voir § 5.4).

D'autres packages permettent d'importer des données depuis Twitter (package twitteR), depuis facebook (package Rfacebook), linkedIn (package Rlinkedin), etc.

2.1.4 Importation via le menu de RStudio

RStudio permet d'importer directement les fichiers via le menu (`File` puis `Import Dataset`) ou le bouton raccourci (dans l'onglet `Environment`, généralement en haut à droite). Pour retrouver l'importation vue au chapitre précédent, il suffit d'utiliser l'importation `From Text (base)`. L'objet importé sera alors un data-frame.

Une autre possibilité pour importer du texte consiste à utiliser le package readr via le menu `From Text (readr)`. La présentation est légèrement différente mais les questions restent globalement les mêmes. Notons que cette méthode permet de contrôler et de changer colonne par colonne les types (character, factor, double, integer) de chacune des colonnes sous leur intitulé respectif, mais l'objet importé sera un `tibble` de classe `tbl_df` (voir § 5.2.1 p. 151). Il sera peut-être utile pour certaines méthodes (qui ne peuvent pas utiliser de `tibble`) de changer de classe et de revenir à un data-frame avec la fonction **as.data.frame**.

Ce menu propose aussi des importations directes de formats propriétaires classiques comme Excel (xls ou xlsx), SAS ou SPSS. Là encore, l'objet résultant de l'importation sera un `tibble` de classe `tbl_df`.

2.2 Exporter des résultats

Une fois les analyses effectuées et les résultats obtenus, on doit pouvoir communiquer ces résultats à d'autres personnes ou à d'autres logiciels. Un résultat se

présente la plupart du temps sous la forme d'un tableau et nous allons donc exporter celui-ci. L'exportation d'un tableau au fichier texte est très simple :

```
> write.table(tablo,"monfichier.csv",sep=";",row.names=FALSE)
```

L'objet `tablo` est exporté dans le fichier `monfichier.csv` à l'endroit où R travaille. On peut bien sûr spécifier un chemin particulier. L'exportation ci-dessus permet de contrôler le séparateur de colonnes fixé à `";"` et le nom du fichier `monfichier.csv`. Il est possible de contrôler d'autres options comme :

- le fichier de résultats contient-il le nom des colonnes (`col.names`) ? Par défaut `col.names= TRUE` ;
- le fichier de résultats contient-il le nom des lignes (`row.names`) ? Par défaut `row.names= TRUE` ;
- les chaînes de caractères sont-elles délimitées par des guillemets ? Par défaut `quote= TRUE` ;
- le séparateur décimal (`dec`) qui est par défaut `"."` et la chaîne de caractères pour les valeurs manquantes (`na`) qui est par défaut `"NA"`.

Ainsi, pour exporter sans nom de lignes ou colonnes ni guillemets et avec un séparateur tabulation, nous utilisons :

```
> write.table(tablo,"monfichier.txt",quote=FALSE,row.names=FALSE,
    col.names=FALSE,na=" ",sep="\t")
```

Ce type de fichier qui mélange tabulation (séparateur) et espace (valeur manquante) est à proscrire car il est difficile de retrouver l'origine d'un éventuel problème. Selon notre expérience, le séparateur `";"` semble le choix le plus pertinent.

Remarques

- La fonction **write.infile** du package FactoMineR permet d'écrire tous les objets d'une liste dans un même fichier, sans avoir à préciser chacun des objets contenus dans la liste.
- La fonction **fwrite** du package data.table est une alternative rapide à la fonction **write.table**.
- La fonction **write_feather** du package feather exporte au format de fichier binaire pour faciliter le partage entre de nombreux langages comme Python (il existe bien sûr une fonction **read_feather** pour relire ces données).
- Comme pour l'importation, de nombreux packages sont disponibles pour exporter dans d'autres formats spécifiques.

2.3 Manipuler les variables

2.3.1 Changer de type

Il est très souvent nécessaire de changer le type d'une variable. Par exemple, à l'issue de l'importation, une variable qualitative dont les modalités sont codées à

l'aide de chiffres est comprise par R comme une variable quantitative. Dans ce cas, il faut passer de quantitatif (type `numeric`) à qualitatif (type `factor`).

Le passage de variable quantitative à qualitative est très simple : il s'agit de créer un facteur grâce à la fonction **factor**. Prenons l'exemple d'une variable `X` comportant des numériques :

```
> X <- c(rep(10,3),rep(12,2),rep(13,4))
> X
[1] 10 10 10 12 12 13 13 13 13
```

Il existe deux méthodes classiques pour savoir si un objet de type vecteur est une variable quantitative ou une variable qualitative sans avoir à afficher la totalité du vecteur. La première consiste à interroger R sur le type :

```
> is.factor(X)
[1] FALSE
> is.numeric(X)
[1] TRUE
```

La seconde consiste à effectuer un résumé de la variable (**summary**). Quand il s'agit d'une variable quantitative, le minimum, le maximum, les quartiles et la moyenne sont affichés. Par contre, pour un facteur, le nombre d'observations pour les six premiers niveaux de la variable qualitative est donné :

```
> summary(X)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  10.00   10.00   12.00   11.78   13.00   13.00
```

De manière évidente, il s'agit bien ici d'une variable quantitative. Cette instruction est la plus pratique pour des tableaux de données (data-frame) issus d'une étape d'importation. En effet, le résumé est proposé variable par variable et il permet de repérer rapidement les erreurs de type pour les variables.

Le passage en facteur se fait simplement en utilisant la fonction **factor** :

```
> Xqual <- factor(X)
> Xqual
[1] 10 10 10 12 12 13 13 13 13
Levels: 10 12 13
> summary(Xqual)
10 12 13
 3  2  4
```

L'affichage d'un facteur permet clairement de le distinguer d'un numérique par la présence des niveaux (`levels`) en fin d'affichage. Il en va de même pour le résumé délivré par **summary**.

Le passage de facteur en numérique se fait en deux étapes. On transforme le facteur en vecteur de type caractère puis on transforme ce dernier en numérique. Si l'on transforme directement le facteur en numérique, les niveaux sont recodés dans l'ordre (le premier niveau sera 1, le deuxième 2, etc.) :

```
## conversion avec recodage des modalités
> as.numeric(Xqual)
[1] 1 1 1 2 2 3 3 3 3
## conversion sans recodage des modalités : 2 étapes
> provisoire <- as.character(Xqual)
> provisoire
[1] "10" "10" "10" "12" "12" "13" "13" "13" "13"
> as.numeric(provisoire)
[1] 10 10 10 12 12 13 13 13 13
```

2.3.2 Découpage en classes

Le passage d'une variable quantitative à une variable qualitative, ou découpage en classes, est d'un usage fréquent en statistique. Le découpage d'une variable continue `X` peut être réalisé de deux façons :

- un découpage réalisé selon des seuils choisis par l'utilisateur ;
- un découpage automatique proposant des effectifs comparables dans chacune des classes.

Commençons par le premier choix. Affectons à `X` un vecteur de 15 nombres aléatoires selon une loi normale centrée réduite :

```
> set.seed(654) ## fixe la graine du générateur pseudo-aléatoire
> X <- rnorm(15,mean=0,sd=1)
> X
 [1] -0.76031762 -0.38970450  1.68962523 -0.09423560  0.095301
 [7]  1.06576755  0.93984563  0.74121222 -0.43531214 -0.107260
[13] -0.98260589 -0.82037099 -0.87143256
```

Découpons cette variable en 3 niveaux : entre le minimum et -0.2, entre -0.2 et 0.2 puis entre 0.2 et le maximum. La fonction **cut** permet de réaliser cette étape automatiquement. Les classes sont de la forme $]a_i; a_{i+1}]$.

```
> Xqual <- cut(X,breaks=c(min(X),-0.2,0.2,max(X)),include.lowest=TRUE)
> Xqual
 [1] [-0.983,-0.2] [-0.983,-0.2] (0.2,1.69]    (-0.2,0.2]
 [5] (-0.2,0.2]    (0.2,1.69]    (0.2,1.69]    (0.2,1.69]
 [9] (0.2,1.69]    [-0.983,-0.2] (-0.2,0.2]    [-0.983,-0.2]
[13] [-0.983,-0.2] [-0.983,-0.2] [-0.983,-0.2]
Levels: [-0.983,-0.2] (-0.2,0.2] (0.2,1.69]
> table(Xqual)
> Xqual
```

```
[-0.983,-0.2]    (-0.2,0.2]    (0.2,1.69]
            7             3             5
```

Afin d'inclure le minimum de `X` dans la première classe, il est nécessaire d'utiliser l'argument `include.lowest=TRUE`. Le résultat du découpage est un facteur. Quand on dénombre les observations par modalité (par la fonction **table**) nous obtenons des effectifs déséquilibrés. Si nous souhaitons des effectifs équilibrés dans chacune des trois modalités, nous utilisons la fonction **quantile**. Cette fonction permet de calculer les quantiles empiriques et donne donc les seuils souhaités. Les seuils sont les suivants :

- le minimum, qui est le quantile empirique associé à la probabilité 0 : $\mathbb{P}_n(X < \min(X)) = 0$;
- le premier seuil, qui est le quantile empirique associé à la probabilité 1/3 : $\mathbb{P}_n(X \leq \text{seuil}_1) = 1/3$;
- le second seuil, qui est le quantile empirique associé à la probabilité 2/3 : $\mathbb{P}_n(X \leq \text{seuil}_2) = 2/3$;
- le maximum, qui est le quantile empirique associé à la probabilité 1 : $\mathbb{P}_n(X \leq \max(X)) = 1$.

```
> decoupe <- quantile(X,probs=seq(0,1,length=4))
> Xqual <- cut(X,breaks=decoupe,include.lowest=TRUE)
> table(Xqual)
Xqual
[-0.983,-0.544]  (-0.544,0.311]    (0.311,1.69]
              5               5               5
```

2.3.3 Travail sur le niveau des facteurs

Ce paragraphe permet de répondre à quelques problèmes classiques de mise en forme des données :

- comment fusionner deux ou plusieurs niveaux d'un facteur, qui ont par exemple des effectifs très faibles ?
- comment renommer les niveaux d'un facteur ?

La réponse à ces deux questions est identique sous R. Reprenons la variable `Xqual` du paragraphe précédent. Nous souhaitons changer les intitulés des niveaux. Pour cela, il suffit d'affecter une nouvelle valeur aux niveaux de `Xqual` appelés grâce à la fonction **levels** :

```
> levels(Xqual) <- c("niv1","niv2","niv3")
> Xqual
 [1] niv1 niv2 niv3 niv2 niv2 niv3 niv3 niv3 niv3 niv2 niv2
[12] niv1 niv1 niv1 niv1
Levels: niv1 niv2 niv3
```

Si nous souhaitons fusionner deux niveaux, par exemple le niveau 1 et le niveau 3, il suffit de renommer le niveau 1 et le niveau 3 avec le même intitulé (donc le répéter deux fois) :

```
> levels(Xqual) <- c("niv1+3","niv2","niv1+3")
> Xqual
 [1] niv1+3 niv2   niv1+3 niv2   niv2   niv1+3 niv1+3 niv1+3 niv1+3
[12] niv2   niv2   niv1+3 niv1+3 niv1+3 niv1+3
Levels: niv1+3 niv2
```

L'ordre d'apparition des niveaux d'un facteur n'est pas anodin. Dans le cas où le facteur est un facteur ordonné, il suffit de mentionner l'ordre des niveaux dès la création du facteur (§ 1.5.6, p. 22). Pour des facteurs dont les modalités ne sont pas ordonnées, a priori cet ordre n'est pas important. Cependant, dans certaines méthodes comme l'analyse de variance (voir fiche 9.3, p. 276), le premier niveau est utilisé comme niveau de référence et les autres niveaux sont comparés à celui-ci. Proposer le premier facteur de son choix est alors primordial.

Soit une variable qualitative codée `1`, `2` ou `3` selon que l'individu a reçu le traitement `classique`, le traitement `nouveau` ou le `placebo`. Les données sont les suivantes :

```
> X <- c(1,1,2,2,2,3)
```

et nous les transformons en facteur comme suit :

```
> Xqual <- factor(X,label=c("classique","nouveau","placebo"))
> Xqual
[1] classique classique nouveau   nouveau   nouveau   placebo
Levels: classique nouveau placebo
```

Si nous souhaitons que le niveau de référence soit `placebo`, alors nous utilisons la fonction **relevel** :

```
> Xqual2 <- relevel(Xqual,ref="placebo")
> Xqual2
[1] classique classique nouveau   nouveau   nouveau   placebo
Levels: placebo classique nouveau
```

Pour contrôler l'ordre des niveaux, il suffit de recréer un facteur à partir du facteur existant en spécifiant l'ordre d'apparition des niveaux. Ainsi, pour choisir l'ordre d'apparition des niveaux comme `placebo` puis `nouveau` et enfin `classique`, nous utilisons :

```
> Xqual3 <- factor(Xqual,levels=c("placebo","nouveau","classique"))
> Xqual3
[1] classique classique nouveau nouveau nouveau placebo
Levels: placebo nouveau classique
```

Quelquefois, nous sommes amenés à supprimer des individus aberrants [1] ou à travailler sur un sous-ensemble d'individus. Ceci peut conduire à une variable qualitative dont un niveau n'a plus d'individus. Construisons un exemple illustrant ce problème :

```
> facteur <- factor(c(rep("A",3),"B",rep("C",4)))
> facteur
[1] A A A B C C C C
Levels: A B C
```

Si nous sommes amenés à retirer le quatrième individu, le facteur est alors :

```
> facteur2 <- facteur[-4]
> facteur2
[1] A A A C C C C
Levels: A B C
```

Nous constatons que la modalité `B` n'a plus d'individu. Il faut donc enlever cette modalité. Pour cela, il est possible de réaffecter cette modalité à une autre, par exemple la modalité `A`. Une autre méthode consiste à transformer le facteur en vecteur de caractères puis à le retransformer en facteur. La première transformation enlève la notion de niveau et la seconde la remet en place à partir du vecteur de données. Comme il n'existe plus de modalité `B`, celle-ci disparaît :

```
> facteur2 <- as.character(facteur2)
> facteur2 <- factor(facteur2)
> facteur2
[1] A A A C C C C
Levels: A C
```

La dernière méthode proposée par R est d'utiliser l'option `drop=TRUE` au moment de la sélection (cette méthode simple ne marche pas pour les data-frame) :

```
> facteur3 <- facteur[-4,drop=TRUE]
> facteur3
[1] A A A C C C C
Levels: A C
```

Parfois, lorsque l'effectif d'une modalité est trop faible, on choisit de ventiler ses individus dans d'autres modalités. Ce procédé, appelé ventilation, est traité dans l'exercice 2.6.

1. "The last mass trials were a great success. There are going to be fewer but better Russians." Greta Garbo dans *Ninotchka* (1939), de Ernst Lubitsch.

2.4 Manipuler les individus

2.4.1 Repérer les données manquantes

Les données manquantes sont représentées sous R par `NA` (Not Available). Pour les retrouver, il suffit d'utiliser la fonction **is.na** qui renvoie `TRUE` si la valeur vaut `NA` et `FALSE` sinon. Construisons une variable mesurée sur 10 individus en effectuant un tirage selon une loi normale $\mathcal{N}(0,1)$. Ensuite affectons-lui des données manquantes, par exemple pour les individus 3, 4 et 6 :

```
> set.seed(23)
> variable <- rnorm(10,mean=0,sd=1)
> variable[c(3,4,6)] <- NA
```

Maintenant, plaçons-nous dans le cas où l'analyste, le statisticien ou le data scientist doit savoir repérer les individus possédant une valeur manquante. Sa première tâche est de connaître l'identifiant, ou le numéro, de ceux-ci. Pour cela, il suffit d'utiliser la fonction **is.na** :

```
> select <- is.na(variable)
> select
 [1] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
```

Les éléments 3, 4 et 6 prennent la valeur `TRUE`. Ils prennent donc la valeur `NA` dans le vecteur `variable`. La fonction **which** donne directement les indices des `TRUE` :

```
> which(select)
[1] 3 4 6
```

Si on souhaite éliminer ces individus, il faut garder tous les non-manquants, c'est-à-dire tous les `FALSE` du vecteur `select`. Pour cela, on conserve tous les `TRUE` de la négation de `select`, c'est-à-dire `!select` :

```
> variable2 <- variable[!select]
> variable2
[1]  0.19321233 -0.43468211  0.99660511 -0.27808628  1.01920549
[6]  0.04543718  1.57577959
```

Une autre méthode consiste à éliminer les coordonnées grâce à la sélection par entiers négatifs :

```
> variable3 <- variable[-which(select)]
> all.equal(variable2,variable3)
[1] TRUE
```

Généralisons à un tableau de données admettant plusieurs variables. Créons une seconde variable et constituons le tableau `don` :

```
> varqual <- factor(c(rep("M",3),NA,NA,rep("F",5)))
> don <- cbind.data.frame(variable,varqual)
> don
      variable varqual
1   0.19321233       M
2  -0.43468211       M
3           NA       M
4           NA    <NA>
5   0.99660511    <NA>
6           NA       F
7  -0.27808628       F
8   1.01920549       F
9   0.04543718       F
10  1.57577959       F
```

Le résumé statistique du jeu de données est :

```
> summary(don)
    variable       varqual
 Min.   :-0.4347   F   :5
 1st Qu.:-0.1163   M   :3
 Median : 0.1932   NA's:2
 Mean   : 0.4454
 3rd Qu.: 1.0079
 Max.   : 1.5758
 NA's   : 3.0000
```

Le décompte des individus manquants est spécifié variable par variable. Si nous souhaitons éliminer les individus qui possèdent au moins une valeur manquante, il faut d'abord repérer les valeurs manquantes :

```
> select <- is.na(don)
> select
      variable varqual
 [1,]    FALSE   FALSE
 [2,]    FALSE   FALSE
 [3,]     TRUE   FALSE
 [4,]     TRUE    TRUE
 [5,]    FALSE    TRUE
 [6,]     TRUE   FALSE
 [7,]    FALSE   FALSE
 [8,]    FALSE   FALSE
 [9,]    FALSE   FALSE
[10,]    FALSE   FALSE
```

Ensuite, il faut éliminer les lignes de `don` qui correspondent aux lignes de `select` qui ont au moins un `TRUE` : lignes 3, 4, 5 et 6. Pour cela, à chaque ligne, nous voulons

savoir s'il existe au moins un `TRUE`. La fonction **any** appliquée à un vecteur renvoie `TRUE` s'il existe au moins un `TRUE` dans le vecteur. Afin d'appliquer cette fonction à chaque ligne du tableau, nous utilisons **apply** sur les lignes (`MARGIN=1`) :

```
> aeliminer <- apply(select,MARGIN=1,FUN=any)
> aeliminer
 [1] FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE
```

Le dernier vecteur contient donc des `TRUE` à chaque ligne qui contient au moins un `NA`. Les lignes à retenir sont donc celles qui correspondent à tous les `TRUE` de la négation de `aeliminer`. Les données épurées sont donc :

```
> don2 <- don[!aeliminer,]
> don2
      variable varqual
1   0.19321233       M
2  -0.43468211       M
7  -0.27808628       F
8   1.01920549       F
9   0.04543718       F
10  1.57577959       F
```

Enfin, si nous souhaitons connaître les couples ligne × colonne des individus présentant une valeur manquante, nous pouvons utiliser l'option `arr.ind` (« array indices ») de la fonction **which** :

```
> which(is.na(don),arr.ind=TRUE)
     row col
[1,]   3   1
[2,]   4   1
[3,]   6   1
[4,]   4   2
[5,]   5   2
```

La première donnée manquante correspond à la ligne 3 et la colonne 1, ..., la cinquième donnée manquante correspond à la ligne 5 et la colonne 2.

2.4.2 Repérer les individus aberrants univariés

Afin d'illustrer ce problème, utilisons le jeu de données `kyphosis` du package `rpart`. Il faut charger le package puis le jeu de données afin de pouvoir l'utiliser :

```
> library(rpart)
> data(kyphosis)
```

Traçons une boîte à moustaches de la variable `Number` de ce jeu de données :

```
> boxplot(kyphosis[,"Number"])
```

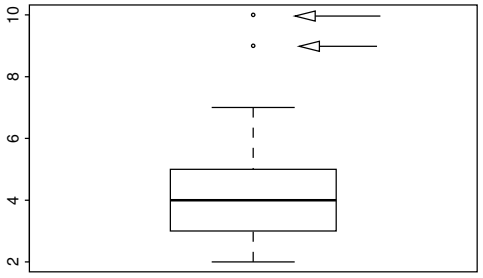


FIGURE 2.1 – Boîte à moustaches (boxplot) d'une variable et individus aberrants (figurés par des flèches).

Les deux individus au-dessus de la moustache supérieure (Fig. 2.1) sont souvent considérés comme des individus aberrants. Comment connaître leur numéro (ou identifiant) ? Une première solution consiste à retourner le résultat de la fonction **boxplot**. La composante `out` de la liste sortie nous donne les valeurs aberrantes.

```
> resultat <- boxplot(kyphosis[,"Number"])
> valaberrante <- resultat$out
> valaberrante
[1]  9 10
```

Les valeurs aberrantes de la variable `Number` sont donc 9 et 10. Il reste à connaître les identifiants des individus en question. Pour cela, nous cherchons les individus dont la valeur correspond à l'une des valeurs contenues dans `valaberrante`. L'opérateur `%in%` renvoie `TRUE` ou `FALSE` si la valeur est dans un ensemble (ici `valaberrante`) ou non. Ensuite, pour avoir le numéro des observations qui sont dans cet ensemble, nous utilisons **which** :

```
> which(kyphosis[,"Number"]%in%valaberrante)
[1] 43 53
```

Les individus 43 et 53 sont donc des individus aberrants.

Certains fonctions de packages graphiques permettent de construire des graphiques interactifs qui facilitent la détection des individus aberrants et des valeurs qu'ils prennent (voir § 3.3).

2.4.3 Repérer et/ou éliminer des doublons

Lors de la création de tableaux de données, il arrive très souvent d'avoir des lignes en double, des doublons. Il peut être important de savoir les repérer, voire de les éliminer. Constituons un data-frame miniature avec deux lignes identiques.

```
> X <- data.frame(C1=c("a","b","b","a","a"),C2=c(1,2,2,3,1))
> X
  C1 C2
1  a  1
2  b  2
3  b  2
4  a  3
5  a  1
```

La fonction **unique** permet de garder les individus (lignes) différents et d'éliminer les doublons (en ne gardant que la première occurence des lignes identiques) :

```
> unique(X)
  C1 C2
1  a  1
2  b  2
4  a  3
```

La fonction **duplicated** permet quant à elle de repérer les doublons en renvoyant un vecteur de booléen :

```
> duplicated(X)
[1] FALSE FALSE  TRUE FALSE  TRUE
```

permettant l'extraction des doublons :

```
> X[duplicated(X),]
  C1 C2
3  b  2
5  a  1
```

2.5 Concaténer des tableaux de données

Nous sommes en présence de deux tableaux et nous souhaitons les regrouper. Ce regroupement peut être vu de deux manières : mettre les tableaux l'un en dessous de l'autre (juxtaposition de lignes, voir Fig. 2.2) ou les mettre l'un à coté de l'autre (juxtaposition de colonnes, voir Fig. 2.3).

Pour effectuer une concaténation selon les lignes de `X` et `Y`, nous utilisons :

```
> Z <- rbind(X,Y)
```

Il faut donc que `X` et `Y` aient le même nombre de colonnes. Cependant, la logique est un peu différente selon que l'on concatène deux matrices ou deux data-frames. Illustrons ceci sur un exemple. Construisons d'abord une matrice avec ses identifiants de ligne et les noms des variables `X1` et `X2` :

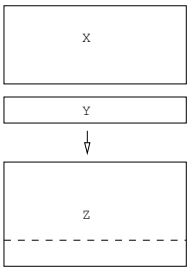


FIGURE 2.2 – Concaténation par ligne : **rbind**(X,Y).

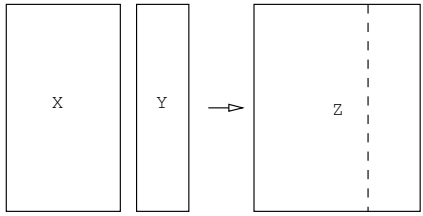


FIGURE 2.3 – Concaténation par colonne : **cbind**(X,Y).

```
> X <- matrix(c(1,2,3,4),2,2)
> rownames(X) <- paste("ligne",1:2,sep="")
> colnames(X) <- paste("X",1:2,sep="")
> X
       X1 X2
ligne1  1  3
ligne2  2  4
```

De même pour la matrice `Y` :

```
> Y <- matrix(11:16,3,2)
> colnames(Y) <- paste("Y",1:2,sep="")
> Y
     Y1 Y2
[1,] 11 14
[2,] 12 15
[3,] 13 16
```

Nous concaténons `X` et `Y` en une seule matrice :

```
> Z <- rbind(X,Y)
> Z
       X1 X2
ligne1  1  3
ligne2  2  4
       11 14
       12 15
       13 16
```

Sur cet exemple, R concatène les matrices en ne retenant que le nom des variables de la première matrice.

Si l'on utilise deux data-frames de structure identique, un message d'erreur apparaît dès que les noms des variables sont différents :

```
> Z <- rbind(data.frame(X),data.frame(Y))
Erreur dans match.names(clabs, names(xi)) :
  les noms ne correspondent pas aux noms précédents
```

Il est donc nécessaire de changer le nom des variables de l'un des data-frames :

```
> Xd <- data.frame(X)
> Yd <- data.frame(Y)
> colnames(Yd) <- c("X2","X1")
> rbind(Xd,Yd)
       X1 X2
ligne1  1  3
ligne2  2  4
3      14 11
4      15 12
5      16 13
```

Nous remarquons au passage (3^{e} ligne) que l'ordre des variables n'est pas le même. Dans ce cas, les variables sont réordonnées avant la concaténation.

La concaténation par colonnes se fait à l'identique pour les matrices ou les data-frames. La fonction **cbind** ne vérifie pas les noms des lignes. Les noms du premier tableau sont conservés :

```
> X <- matrix(c(1,2,3,4),2,2)
> rownames(X) <- paste("ligne",1:2,sep="")
> Y <- matrix(11:16,2,3)
> cbind(data.frame(X),Y)
       X1 X2  1  2  3
ligne1  1  3 11 13 15
ligne2  2  4 12 14 16
```

Ce type de concaténation, en lignes ou en colonnes, n'est cependant pas le seul envisageable. Il est possible de fusionner deux tableaux selon une clef, grâce à l'ordre classique **merge** (voir Fig. 2.4).

Constituons deux tableaux de données, l'un possédant plus de lignes que l'autre. Le premier tableau regroupe (concaténation par colonne) une variable continue (`age`) et deux variables qualitatives dans un data-frame. La fonction **cbind.data.frame** permet d'imposer que le résultat soit un data-frame :

```
> age <- c(7,38,32)
> prenom <- c("arnaud","nicolas","laurent")
> ville <- factor(c("rennes","rennes","marseille"))
> indiv <- cbind.data.frame(age,prenom,ville)
> indiv
  age  prenom     ville
```

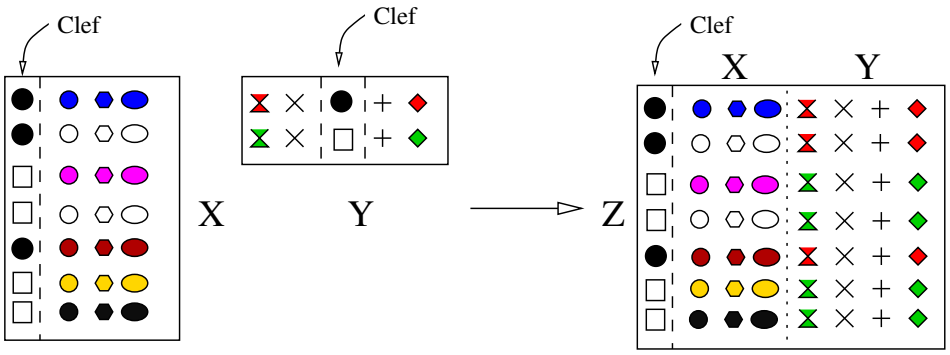


FIGURE 2.4 – Fusion par clef : **merge**(X,Y,by="clef").

```
1   7  arnaud    rennes
2  38 nicolas    rennes
3  32 laurent marseille
```

Un second tableau regroupe les caractéristiques des villes :

```
> nb.hab <- c(200,500,800)
> caractvilles <- cbind.data.frame(c("rennes","lyon","marseille"),nb.hab)
> names(caractvilles) <- c("ville","pop")
> caractvilles
      ville  pop
1    rennes  200
2     lyon   500
3 marseille  800
```

Si nous souhaitons fusionner ces deux tableaux en un seul où seront répétées les caractéristiques des villes à chaque ligne du tableau, nous utilisons une fusion par la clef `ville` :

```
> merge(indiv,caractvilles,by="ville")
      ville age  prenom  pop
1 marseille  32 laurent  800
2    rennes   7  arnaud  200
3    rennes  38 nicolas  200
```

La clef `ville` est alors la première colonne du jeu de données.

2.6 Tableau croisé

Lorsque deux variables qualitatives sont traitées, nous pouvons présenter les données sous deux formes : le tableau classique où un individu (une ligne) est décrit

par les variables (deux variables), ou bien le tableau croisé (appelé aussi tableau de contingence) qui donne l'effectif de chaque croisement de modalités. Bien évidemment, cela se généralise à plus de deux variables qualitatives.

Afin d'envisager clairement les deux types de tableau, construisons d'abord deux variables qualitatives (`laine` et `tension`) mesurées sur 10 individus. La variable `laine` correspond aux trois types de laine suivants : Angora, Merinos, Texel. La variable `Tension` indique la valeur `faible` ou `forte` à la résistance en traction.

```
> tension <- factor(c(rep("Faible",5),rep("Forte",5)))
> tension
 [1] Faible Faible Faible Faible Faible Forte Forte Forte Forte Forte
Levels: Faible Forte
> laine <- factor(c(rep("Mer",3),rep("Ang",3),rep("Tex",4)))
> laine
 [1] Mer Mer Mer Ang Ang Ang Tex Tex Tex Tex
Levels: Ang Mer Tex
```

Fusionnons ces deux variables dans le tableau `don` :

```
> don <- cbind.data.frame(tension,laine)
> don
   tension laine
1   Faible   Mer
2   Faible   Mer
3   Faible   Mer
4   Faible   Ang
5   Faible   Ang
6    Forte   Ang
7    Forte   Tex
8    Forte   Tex
9    Forte   Tex
10   Forte   Tex
```

Le tableau croisé est alors obtenu directement sur les objets par :

```
> table(don$tension,don$laine)
         Ang Mer Tex
  Faible   2   3   0
  Forte    1   0   4
```

Une autre méthode consiste à utiliser le nom des colonnes du data-frame et à faire le croisement par une formule :

```
> tabcroise <- xtabs(~tension+laine,data=don)
> tabcroise
        laine
```

```
tension  Ang Mer Tex
  Faible   2   3   0
  Forte    1   0   4
```

De nombreuses fonctions de R étant bâties sur l'hypothèse d'un tableau individus × variables, il faut pouvoir effectuer l'opération inverse lorsque les données sont proposées dans un tableau croisé. Cependant la construction du tableau individus × variables n'est pas immédiate :

```
> tabframe <- as.data.frame(tabcroise)
> tabframe
  tension laine Freq
1  Faible   Ang    2
2   Forte   Ang    1
3  Faible   Mer    3
4   Forte   Mer    0
5  Faible   Tex    0
6   Forte   Tex    4
```

Nous obtenons les fréquences de chaque combinaison et non pas une ligne pour chaque individu. Pour reconstruire le tableau initial, voir l'exercice 2.8.

2.7 Exercices

Exercice 2.1 (Importation robuste)
Dans certains cas, l'importation échoue sans qu'il soit possible de comprendre ce qui se passe. Dans ces cas-là, voici une procédure plus robuste qui permet de pointer le problème.

1. Importer le fichier `donnees.csv` dans un vecteur de caractères (le type le plus basique) grâce à la fonction **scan**. Dans une importation délicate, cette étape permet de verifier le séparateur de colonne.

2. Récupérer le nom des variables dans un objet `nomcol`.

3. Changer le séparateur décimal de `","` en `"."` (fonction **gsub**). Remarquons que cette étape peut être réalisée plus simplement avec un tableur ou un éditeur de texte.

4. Constituer une matrice de caractères de 4 lignes et 5 colonnes contenant les données, les noms des individus et des variables.

5. Récupérer l'intitulé des individus dans un objet `nomlign` et affecter le reste des données dans une matrice 4 lignes et 4 colonnes (`donnees`). Dans une importation délicate, afficher colonne par colonne les données afin de contrôler que les colonnes ne sont pas « mélangées ».

6. Transformer la matrice en data-frame et vérifier que le type des variables est correct. Sinon, transformer les facteurs en numériques, voir § 2.3.1, p. 41.

Exercice 2.2 (Importation)
Importer les fichiers `test1.csv`, `test1.prn`, `test2.csv` et `test3.csv`.

Exercice 2.3 (Importation et format date)
Le jeu de données `test4.csv` contient un échantillon où sur chaque personne sont notés : l'age, le sexe (codé 0 pour masculin, 1 pour féminin) et la date du premier jour de ski dans la vie.

1. Importer « normalement » le fichier `test4.csv` (utiliser l'argument `skip` pour importer à partir de la troisième ligne).

2. Importer de nouveau le fichier `test4.csv` en utilisant l'argument `colClasses` afin de spécifier que le sexe est de type `factor` et que la date est au format `Date`.

Exercice 2.4 (Importation et fusion)
1. Importer les fichiers `etat1.csv`, `etat2.csv` et `etat3.csv`.

2. Fusionner les trois data-frame en un seul en utilisant les colonnes communes afin de répéter les lignes de manière adéquate.

Exercice 2.5 (Fusion et sélection)
1. Importer les fichiers `fusion1.xls` et `fusion2.xls`.

2. Conserver uniquement les variables `yhat1`, `yhat3`, `Rhamnos`, `Arabinos`. Avec ces 4 variables créer un data-frame unique.

3. Ajouter à ce data-frame les variables `yres1` et `yres2` qui retranchent, individu par individu, les valeurs de `yhat1` à celles de `Rhamnos` et les valeurs de `yhat3` à celles de `Arabinos`.

Exercice 2.6 (Ventilation)
Nous sommes en présence de la variable qualitative suivante :

```
> Xqual <- factor(c(rep("A",60),rep("B",20),rep("C",17),rep("D",3)))
```

1. Calculer les fréquences de chaque modalité (ou niveau du facteur).

2. Donner les modalités dont l'effectif est inférieur à 5 % de l'effectif total.

3. Calculer les fréquences de chaque modalité sans la(les) modalité(s) de la question précédente. Le résultat sera mis dans un vecteur `proba`.

4. Sélectionner les individus prenant la(les) modalité(s) de la question 2. Leur donner une valeur parmi les modalités restantes, selon un tirage dont les probabilités sont calculées en question 3 (utiliser la fonction **sample**). Ce procédé est appelé ventilation [2].

2. « Non mais t'as déjà vu ça ? En pleine paix, y chante et pis crac, un bourre-pif, mais il est complètement fou ce mec ! Mais moi les dingues j'les soigne, j'm'en vais lui faire une ordonnance, et une sévère, j'vais lui montrer qui c'est Raoul. Aux 4 coins d'Paris qu'on va l'retrouver éparpillé par petits bouts façon puzzle... Moi quand on m'en fait trop j'correctionne plus, j'dynamite... j'disperse... et j'ventile... » Bernard Blier dans *Les tontons flingueurs*, de Georges Lautner (1963), dialogues de Michel Audiard.

Exercice 2.7 (Ventilation sur facteur ordonné)
Nous sommes en présence de la variable qualitative dont les modalités sont ordonnées :

```
> Xqual <- factor(c(rep("0-10",1),rep("11-20",3),rep("21-30",5),
    rep("31-40",20),rep("41-50",2),rep("51-60",2),rep("61-70",1),
    rep("71-80",31),rep("+ de 80",20)))
```

1. Calculer les fréquences de chaque modalité (ou niveau du facteur).

2. Donner les modalités dont l'effectif est inférieur à 5 % de l'effectif total.

3. Commencer par la première modalité (selon l'ordre des modalités) de la question 2. Fusionner cette modalité avec la modalité immédiatement « supérieure ». Si l'effectif de ces deux modalités fusionnées est supérieur à 5 % de l'effectif total, passer à la modalité suivante de la question 2 (si elle existe). Sinon, fusionner encore avec la modalité immédiatement « supérieure » (parmi toutes les modalités). Procéder de même jusqu'à épuisement des modalités de la question 2.

Exercice 2.8 (Tableau croisé → tableau de données)
1. Créer le tableau de contingence croisant les variables qualitatives type de laine (`laine`) et tension (`tension`) :

```
         Ang Mer Tex
  Faible   2   3   0
  Forte    1   0   4
```

2. À partir de ce tableau, créer une matrice de caractère `tabmat` qui contient 3 colonnes et autant de lignes que de croisement de modalités (ou cellules dans le tableau de contingence). Cette matrice de caractère sera remplie à chaque ligne par la tension (ligne du tableau précédent), le type de laine (colonne du tableau précédent) et l'effectif pour le croisement des modalités. Pour cela on utilisera les fonctions **matrix** et **rep**.

3. Transformer la matrice de caractères de la question précédente en data-frame, appelé par exemple `tabframe`, et contrôler le type des variables. Grâce à ce data-frame, affecter à `n` le nombre total d'individus (**sum**) et affecter à `nbefac` le nombre de variables qualitatives (**ncol**).

4. Créer un compteur `iter` initialisé à 1 et une matrice `tabcomplet` de caractères, par exemple le caractère `""`, de la taille du jeu de données final.

5. Faire une boucle sur le nombre de lignes de `tabmat` (voir § 4.1.2, p. 109). Chaque ligne i correspond à un croisement de modalités (ou strates). Si le nombre d'individus qui prennent ce croisement de modalités i n'est pas nul, alors répéter, sur autant de lignes `tabcomplet` qu'il faut, le croisement de modalité i (en répartissant les modalités dans la colonne qui lui correspond). On pourra utiliser une boucle, le compteur `iter`, la matrice `tabmat` et le data-frame `tabframe`.
Le résultat `tabcomplet` sera identique au tableau `don` initial.

Chapitre 3

Visualiser les données

Les graphiques constituent souvent le point de départ d'une analyse statistique. Un des avantages de R est la facilité avec laquelle l'utilisateur peut produire des graphiques de toute nature. Nous commençons ce chapitre par l'étude des graphiques conventionnels, graphiques qui sont à la base de nombreux représentations proposées dans divers packages. Des représentations plus sophistiquées sont ensuite proposées. Pour commencer, il peut être intéressant de regarder quelques exemples de représentations graphiques pouvant être construites avec R. Le site `https://www.r-graph-gallery.com` est particulièrement bien fait : il propose de nombreuses représentations et donne les lignes de code pour les réaliser.

3.1 Les fonctions graphiques conventionnelles

Nous allons donner des exemples de représentations pour des variables quantitatives et qualitatives. Nous utilisons le fichier de données `ozone.txt` que l'on importe grâce à :

```
> ozone <- read.table("ozone.txt",header=T)
```

Nous disposons de variables climatiques et d'une variable de pollution à l'ozone mesurées durant l'été 2001 à Rennes. Les variables considérées sont `maxO3` (maximum d'ozone journalier), `T12` (température à midi), `vent` (direction), `pluie` et `Vx12` (projection du vecteur vitesse du vent sur l'axe Est-Ouest). Les variables `T12`, `maxO3` et `Vx12` sont des variables quantitatives continues (numériques) alors que `vent` et `pluie` sont des facteurs :

```
> ozone <- ozone[,c("T12","maxO3","vent","pluie","Vx12")]
> summary(ozone)
      T12            maxO3          vent     pluie         Vx12
 Min.   :14.00  Min.   : 42.00  Est  :10  Pluie:43  Min.   :-7.878
```

```
 1st Qu.:18.60  1st Qu.: 70.75  Nord :31  Sec  :69  1st Qu.:-3.565
 Median :20.55  Median : 81.50  Ouest:50           Median :-1.879
 Mean   :21.53  Mean   : 90.30  Sud  :21           Mean   :-1.611
 3rd Qu.:23.55  3rd Qu.:106.00                     3rd Qu.: 0.000
 Max.   :33.50  Max.   :166.00                     Max.   : 6.578
```

3.1.1 La fonction plot

La fonction **plot** est une fonction générique de R, i.e. une fonction qui est appelée par le même ordre `plot` mais qui donne des résultats différents suivant la classe de l'objet sur lequel elle est appliquée (voir § 1.5.9). Elle permet de représenter tous les types de données. L'utilisation classique de la fonction **plot** consiste à représenter un nuage de points d'une variable `y` en fonction d'une variable `x`.

De façon classique en statistique, la fonction **plot** s'écrit avec des formules du type `y~x`. Prenons l'exemple de l'ozone et représentons différents couples de variables. Commençons par représenter deux variables quantitatives, le maximum d'ozone `maxO3` en fonction de la température `T12` (Fig. 3.1) :

```
> plot(maxO3~T12,data=ozone)
```

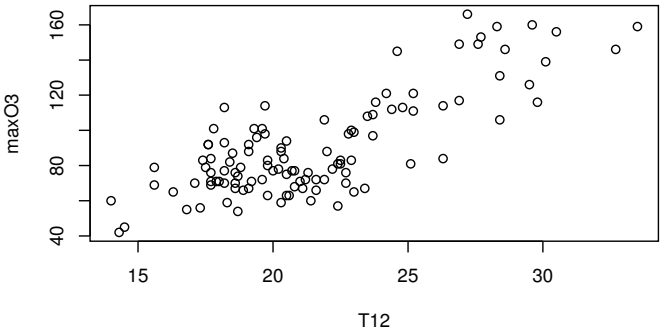


FIGURE 3.1 – Nuage de points (`T12`, `maxO3`) avec libellés des axes explicites.

Pour représenter une variable quantitative (`maxO3`) en fonction d'une variable qualitative (`vent`), nous écrivons de la même façon :

```
> plot(maxO3~vent,data=ozone,xlab="Secteur du vent",ylab="Pic d'ozone")
```

La fonction **plot** retourne dans ce cas une boîte à moustaches par modalité de la variable `vent` (Fig. 3.2).

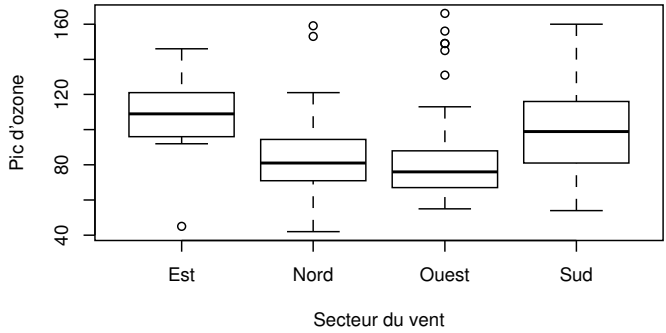


FIGURE 3.2 – Graphique avec **plot** des vecteurs `vent` et `maxO3`.

Ce graphique permet de voir qu'il existe un effet de la variable `vent` sur l'ozone. Ici, le vent d'Est semble associé à de plus fortes concentrations en ozone. Ce graphique peut aussi être obtenu grâce à la fonction **boxplot** :

```
> boxplot(maxO3~vent,data=ozone)
```

Compte tenu de la nature des variables, la fonction **plot** a choisi d'utiliser la fonction **boxplot** car cela est considéré comme plus pertinent.

On peut également représenter deux variables qualitatives par un diagramme en bandes. Par exemple (voir Fig. 3.3),

```
> plot(pluie~vent,data=ozone)
```

permet d'obtenir pour chaque modalité du facteur explicatif (ici `vent`), les fréquences relatives de chaque modalité du facteur à expliquer (ici `pluie`). La largeur de la bande est proportionnelle à la fréquence de la modalité du facteur explicatif (ici `vent`). Ce graphique peut aussi être obtenu grâce à la fonction **spineplot**.

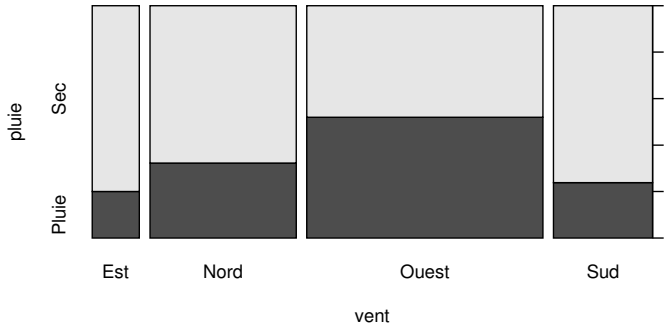


FIGURE 3.3 – Diagramme en bandes de `pluie` en fonction de `vent`.

Nous retrouvons en figure 3.3 que le temps de type « pluie » est proportionnellement plus important quand le vent souffle d'ouest (ce qui semble assez naturel à Rennes, d'où proviennent les mesures). De plus, puisque c'est la bande la plus large, c'est aussi le vent le plus fréquent à Rennes.

On peut enfin représenter une variable qualitative en fonction d'une variable quantitative, ici `vent` en fonction de `T12` (voir Fig. 3.4) :

```
> plot(ozone[,"T12"],ozone[,"vent"],pch=3,yaxt="n",xlab="T12",ylab="vent")
> axis(side=2,at=1:4,labels=levels(ozone[,"vent"]))
```

Notons qu'il est nécessaire de retravailler le graphique : l'argument `yaxt="n"` supprime les libellés des graduations de l'axe `y` qui prennent ici par défaut les différentes modalités de la variable qualitative converties en numériques : 1 à 4. La fonction **axis** permet alors de renommer les libellés des graduations.

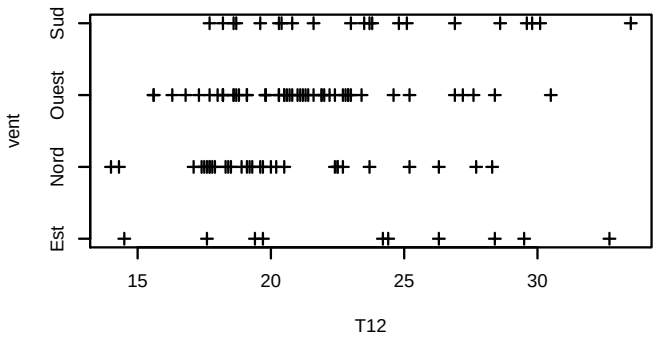


FIGURE 3.4 – Graphique d'une variable quantitative en abscisse (`T12`) et d'une variable qualitative en ordonnée (`vent`).

Ce graphique est intéressant quand il y a peu de mesures par modalité. Dans le cas contraire, on représente les données à l'aide de boîtes à moustaches (voir Fig. 3.2).

Même si, intrinsèquement, la fonction **plot** attend en entrée les deux arguments `x` et `y` associés aux abscisses et ordonnées, il est possible de ne préciser qu'un seul argument. Par exemple, pour une variable quantitative, R dessine séquentiellement cette variable avec en abscisse le numéro de l'observation (voir Fig. 3.5) :

```
> plot(ozone[,"maxO3"], xlab="num.", ylab="maxO3", cex=.5, pch=16)
```

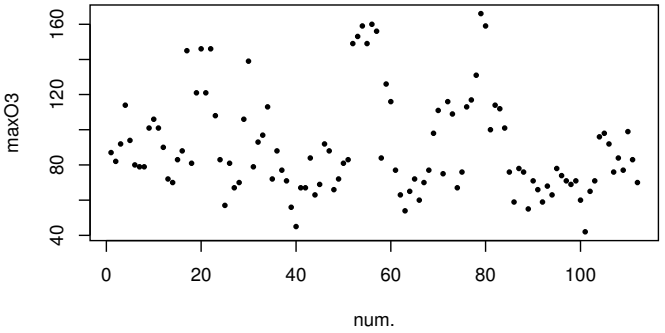


FIGURE 3.5 – Nuage de points des pics d'ozone (`maxO3`) en fonction de l'index.

Cette commande retourne un dessin où, par défaut, l'abscisse est la suite ordonnée appelée « num. » des observations de 1 à n avec n le nombre d'observations.

La taille du symbole peut être contrôlée via l'argument `cex` qui permet de gérer le facteur d'augmentation (ou de diminution) de taille (par défaut `cex=1`). L'argument `pch` permet de préciser la forme des points. Cet argument peut prendre des valeurs numériques entre 0 et 25 (voir Fig. 3.6) ou un caractère directement retranscrit à l'écran.

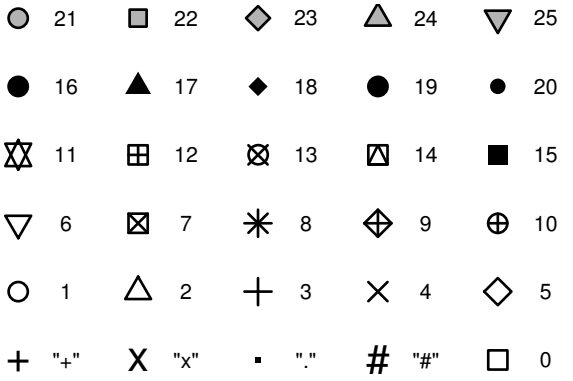


FIGURE 3.6 – Symbole obtenu pour la valeur de l'argument `pch`.

Il est également possible de modifier le type de tracé via l'argument `type` : `"p"` pour tracer des points (option par défaut), `"l"` (line) pour les relier, `"b"` ou `"o"` pour faire les deux. L'argument `type` permet aussi de tracer des bâtons verticaux (`"h"`, high-density) ou des escaliers après les points (`"s"`, step) ou avant (`"S"`, step). Une illustration graphique de ces arguments est donnée en figure 3.7.

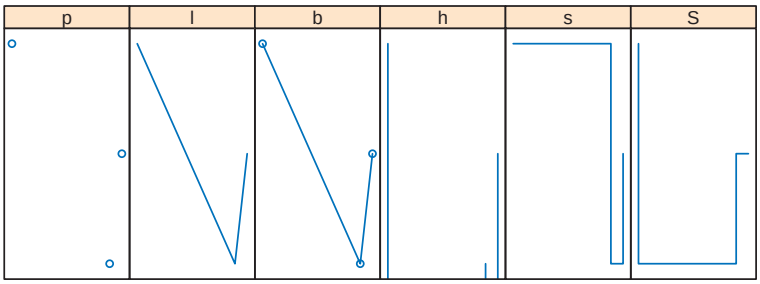


FIGURE 3.7 – Argument `type` pour les valeurs `"p"`, `"l"`, `"b"`, `"h"`, `"s"`, `"S"`.

L'évolution du maximum journalier de l'ozone durant l'été 2001 est obtenu avec l'argument `type="l"`. Ce graphique est très classique pour représenter des données temporelles (voir Fig. 3.8) :

```
> plot(ozone[,"maxO3"], xlab="num.", ylab="maxO3", type="l")
```

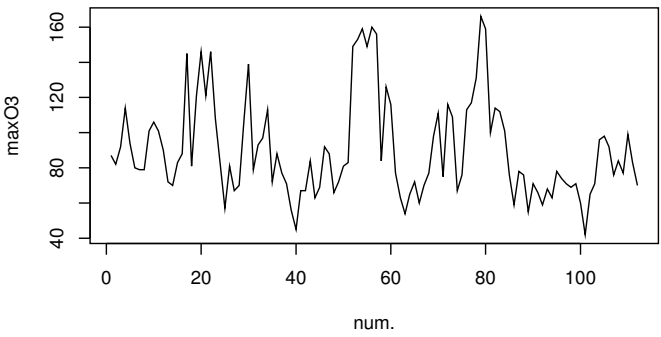


FIGURE 3.8 – Évolution des pics d'ozone (`maxO3`) durant l'été 2001.

En conclusion, la fonction **plot** peut s'adapter à chaque type de données.

3.1.2 Représentation d'une distribution

Pour représenter la distribution d'une variable continue, par exemple le vecteur de numériques `ozone[,"maxO3"]`, il existe des solutions classiques déjà programmées. L'histogramme de la variable `maxO3` (Fig. 3.9 à gauche) est un estimateur de la densité si l'on spécifie l'argument `prob=TRUE` dans la fonction **hist** :

```
> hist(ozone[,"maxO3"], main="Histogramme", prob=TRUE, xlab="Ozone",
    col="lightblue")
```

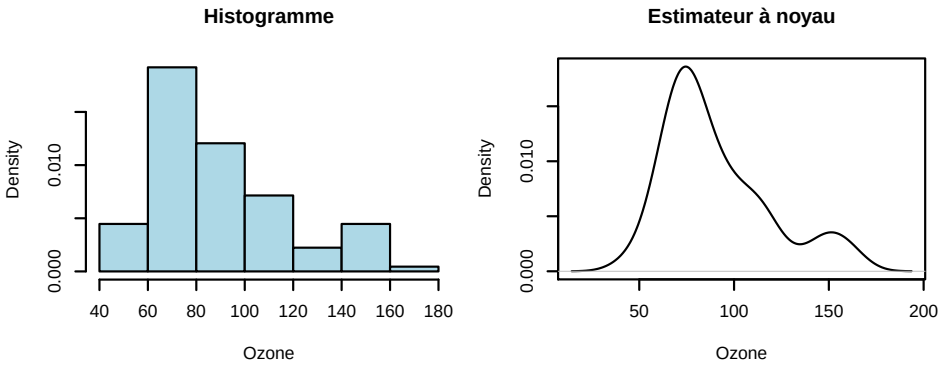


FIGURE 3.9 – Histogramme (gauche) et estimateur à noyau de la densité (droite) de l'ozone.

Un estimateur à noyau de la densité de la variable `maxO3` peut être obtenu grâce à la fonction **density**. Sa représentation graphique (Fig. 3.9 à droite) s'obtient comme suit :

```
> plot(density(ozone[,"maxO3"]), main="Estimateur à noyau", xlab="Ozone")
```

Pour représenter la distribution d'une variable qualitative, on utilise un diagramme en barres, qui s'obtient simplement en appliquant la fonction **plot** sur le vecteur des modalités. Pour le facteur `vent` :

```
> plot(ozone[,"vent"])
```

R retourne un diagramme en barres (Fig. 3.10) avec en abscisse les valeurs du facteur et en ordonnée le nombre d'éléments de chaque modalité considérée.

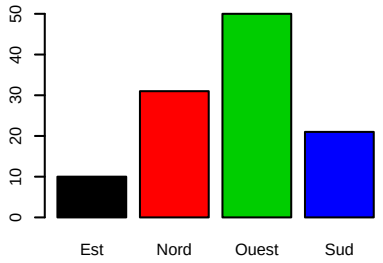


FIGURE 3.10 – Diagramme en barres d'un variable qualitative.

Le diagramme en barres peut aussi être dessiné via la fonction **barplot** appliquée à un vecteur donnant les occurrences de chaque niveau. On calcule ici les occurrences par la fonction **table** avant de dessiner le diagramme :

```
> barplot(table(ozone[,"vent"]))
```

Une autre méthode, peu recommandée, est la représentation circulaire, ou en camembert, que l'on peut obtenir avec la fonction **pie**.

3.1.3 Ajouts aux graphiques

Une fois le graphique tracé, il est possible de le compléter par d'autres informations : ajouts de lignes (**lines**), points (**points**), textes (**text**), symboles (cercles, carrés, étoiles... **symbols**), flèches (**arrows**), segments (**segments**) ou polygones (**polygon**). Nous allons présenter uniquement les ajouts de textes, lignes et points.

Nous reprenons la représentation graphique du maximum d'ozone `maxO3` en fonction de la température `T12`. Pour faire apparaître la date, il est nécessaire d'ajouter du texte (**text**) au nuage de points obtenu par **plot**. Nous écrivons uniquement le mois et le jour (i.e. caractères 5 à 8 des noms de lignes) avec une taille de fonte diminuée. Le texte est ajouté au-dessus (`pos=3`) avec un décalage de 0.3 (`offset`) :

```
> plot(maxO3~T12,data=ozone, pch=20)
> text(ozone[,"T12"],ozone[,"maxO3"], substr(rownames(ozone),5,8), cex=.75,
      pos=3, offset=.3)
```

Si de plus nous voulons ajouter un symbole de couleur pour les dates correspondant aux jours de pluie, nous allons sélectionner leurs coordonnées puis dessiner un point couleur rouge au niveau de leurs coordonnées (Fig. 3.11). Nous ajoutons une ligne vertical pour matérialiser, avec la fonction **abline**, qu'il ne pleut pas lorsque la température est supérieure à 27 degrés :

```
> abline(v=27,lty=2)
```

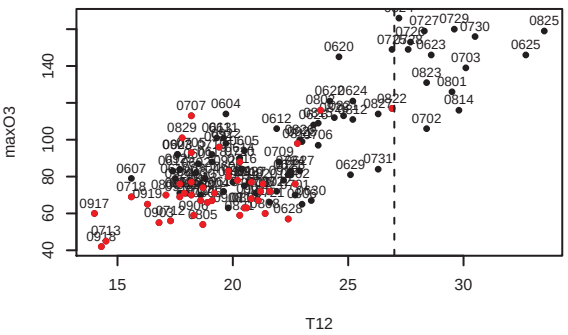


FIGURE 3.11 – Utilisation de la fonction **abline**.

On peut aussi ajouter une ligne horizontale avec **abline**(h=) ou encore une droite en précisant l'ordonnée à l'origine et la pente avec **abline**(**c**(ordonnee,pente)). Le paramètre `lty` (line type) permet de contrôler le type de ligne (1 : trait plein, 2 : tirets, 3 : pointillés, etc.).

Comme son nom l'indique, la fonction **abline** ajoute une ligne sur le graphique existant. Si nous souhaitons représenter deux courbes pour comparer, par exemple, l'évolution de l'ozone sur deux semaines différentes, il faut utiliser la fonction **lines**. Comparons l'évolution de l'ozone pendant les deux premières semaines :

```
> plot(ozone[1:7,"maxO3"],type="l")
> lines(ozone[8:14,"maxO3"],col="red") # ajout de la 2ème semaine
```

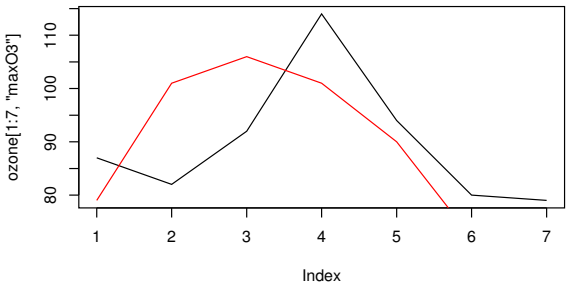


FIGURE 3.12 – Ajout d'une ligne en rouge par **lines**.

Le graphique (Fig. 3.12) n'indique pas les sixième et septième observations de cette deuxième semaine. En effet, un tracé de graphique nécessite de connaître les minima et maxima pour les abscisses et les ordonnées afin de préparer les axes, les échelles, etc. Ici, le minimum de la seconde ligne est plus faible que celui de la première. La mise à l'échelle du graphique est faite par la fonction **plot**. Lors de l'appel, cette fonction n'a connaissance que des informations concernant l'instruction **plot**. La mise à l'échelle n'est pas automatique après un ordre **lines** (ou **points**), il faut donc proposer dès l'ordre **plot** le minimum et le maximum des deux semaines. Il suffit de donner à l'argument `ylim` le résultat de la fonction **range**. Cette dernière donne le minimum et le maximum. Cet argument indique à la fonction **plot** les ordonnées entre lesquelles se situera le graphique (Fig. 3.13).

```
> ecarty <- range(ozone[1:7,"maxO3"],ozone[8:14,"maxO3"])
> plot(ozone[1:7,"maxO3"],type="l",ylim=ecarty,lty=1)
> lines(ozone[8:14,"maxO3"],col="red",lty=1)
```

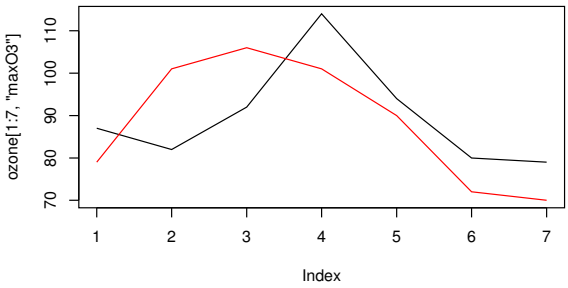


FIGURE 3.13 – Ajout d'une ligne en rouge par **lines** et mise à l'échelle commune des ordonnées via l'argument `ylim`.

Bien sûr, si les abscisses sont les mêmes, il est inutile de prévoir le minimum et le maximum sur les abscisses.

3.1.4 Graphiques en plusieurs dimensions

Les fonctions classiques de représentation 3D sont les fonctions **persp** (3D avec effet de perspective), **contour** (lignes de niveau) et **filled.contour** ou **image** (lignes de niveau avec effet de couleur).

Le classique « chapeau mexicain », proposé dans l'aide de **persp**, est le graphe de la fonction $(x, y) \mapsto z = f(x, y) = 10\ \sin\left(\sqrt{x^2 + y^2}\right)/\sqrt{x^2 + y^2}$. Pour programmer cette fonction, nous choisissons une grille régulière carrée comprenant 30 valeurs différentes comprises entre -10 et 10 (sur `x` et sur `y`) :

```
> f <- function(x,y){10*sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)}
> y <- x <- seq(-10,10,length=30)
```

Évaluons alors la fonction **f** en chaque point de la grille : pour chaque couple (`x[i]`, `y[j]`), nous calculons **f**`(x[i],y[j])`. Afin d'éviter une (double) boucle, nous utilisons la fonction **outer** qui permet ce type d'évaluation directement :

```
> z <- outer(x,y,f)
```

Nous pouvons maintenant tracer cette fonction en 3D comme suit :

```
> persp(x,y,z,theta=30,phi=30,expand=0.5)
```

Si nous souhaitons colorer chaque facette (elles sont au nombre de 29×29) nous calculons la moyenne des quatres extrémités de la facette et nous découpons cette moyenne en facteur à 100 niveaux, un niveau pour chaque couleur. Le vecteur des 100 couleurs possibles est obtenu avec la fonction **heat.colors** :

```
> zfacette <- (z[-1,-1]+z[-1,-30]+z[-30,-1]+z[-30,-30])/4
> niveaucouleur <- cut(zfacette,100)
> couleurs <- heat.colors(100)[niveaucouleur]
> persp(x,y,z,theta=30,phi=30,expand=0.5,col=couleurs)
```

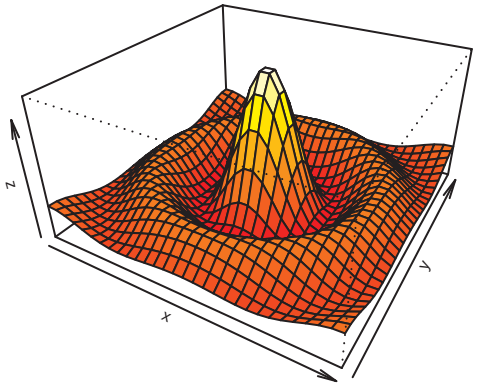


FIGURE 3.14 – Exemple d'utilisation de la fonction **persp**.

Les graphiques avec les courbes de niveaux peuvent être obtenus avec les commandes **contour**(x,y,z), **filled.contour**(x,y,z) ou **image**(x,y,z).
Nous pouvons aussi utiliser le package rgl pour construire la surface de réponse du graphique précédent :

```
> library(rgl)
> rgl.surface(x,y,z)
```

Ce package est à conseiller pour tous les problèmes de visualisation 3D. En effet, il permet, par exemple, d'opérer des rotations des graphiques avec la souris ainsi que d'utiliser des symboles avec effet d'éclairage.

Si nous souhaitons tracer le nuage de points représentant n individus caractérisés par la mesure de trois variables quantitatives, il n'est pas possible d'utiliser les graphiques « conventionnels ». Il faut alors utiliser un package comme rgl ou lattice. Commençons par le package lattice qui fait partie des packages distribués par défaut avec R et représentons les individus pour les trois variables quantitatives maxO3, T12 et Vx12. Comme nous souhaitons expliquer la variable maxO3, il est naturel de la représenter sur le troisième axe :

```
> library(lattice)
> cloud(maxO3~T12+Vx12,type=c("p","h"),data=ozone)
```

Ici, le package lattice permet d'utiliser deux arguments pour type, ce qui n'est pas possible dans les graphiques « conventionnels ».

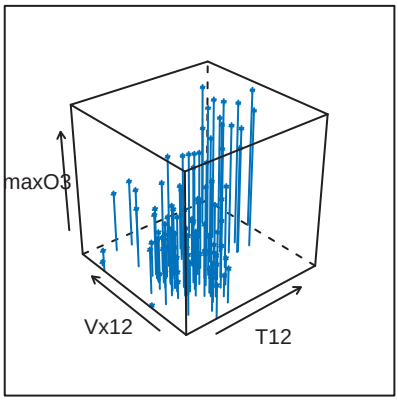


FIGURE 3.15 – Graphe 3D des points pour les variables `Vx12`, `T12` et `maxO3`.

Pour le package rgl, il suffit de taper :

```
> plot3d(ozone[,"T12"],ozone[,"Vx12"],ozone[,"maxO3"],radius=2,xlab="T12",
        ylab="Vx12",zlab="maxO3",type="s")
```

Notons que le type de points `type="s"`, spécifique à rgl, permet d'avoir des sphères dont le rayon est fixé à 2 grâce à l'argument `radius`.

3.1.5 Exportation de graphiques

Une fois le graphique construit, il est possible de l'exporter dans une multitude de formats d'échange, comme les formats jpeg, png (bitmap compressés), les formats pdf et postscript (ps) ou des formats de graphiques vectoriels comme emf (sous Windows), svg ou xfig.

Le plus simple est de sauvegarder un graphique à partir de la fenêtre graphique. Avec RStudio, le graphique apparaît dans la fenêtre en bas à droite : pour l'exporter, il suffit de cliquer sur l'onglet `Export` puis de choisir le format d'exportation et en sélectionnant `PDF Size` à `(Device Size)`.

Si on n'utilise pas RStudio, sous Windows, on fait `Fichier` → `Sauver sous` et on choisit le format adapté, de préférence vectoriel (métafichier « emf »). Il est également possible de copier le graphique et de le coller dans un document en cliquant avec le bouton droit de la souris sur le graphique, puis en cliquant sur `Copier comme vectoriel`.

Sinon, l'exportation d'un graphique suit toujours le même schéma. Par exemple, pour sauver le graphique 3.13 au format pdf, on écrit :

```
> pdf("graphik.pdf")
> ecarty <- range(ozone[1:7,"max03"],ozone[8:14,"max03"])
> plot(ozone[1:7,"max03"],type="l",ylim=ecarty,lty=1)
> lines(ozone[8:14,"max03"],col="red",lty=1)
> dev.off()
```

La fonction **dev.off** permet de finaliser le fichier `graphik.pdf` qui se trouvera à l'endroit de l'arborescence où travaille R. Il est bien sûr possible de spécifier le chemin par **pdf**`("monchemin/graphik.pdf")`.

Si l'on souhaite retoucher un graphique (déplacer une légende, un texte, agrandir une portion, etc.) il faut utiliser un logiciel de dessin vectoriel. Il faut donc exporter son graphique (format svg, emf ou xfig par exemple) puis retoucher le graphique en utilisant un logiciel adapté.

3.1.6 Plusieurs graphiques

Si l'on souhaite faire figurer plusieurs graphiques dans la même fenêtre, on mentionne le nombre de graphiques souhaité. Classiquement, on utilise la fonction **par**. L'instruction **par**`(mfrow=`**c**`(n,p))` organise `np` graphiques en `n` lignes et `p` colonnes. Pour deux graphiques sur une même ligne (Fig. 3.16), nous écrivons :

```
> par(mfrow=c(1,2))
> plot(1:10,10:1,pch=0)
> plot(rep(1,4),type="l")
```

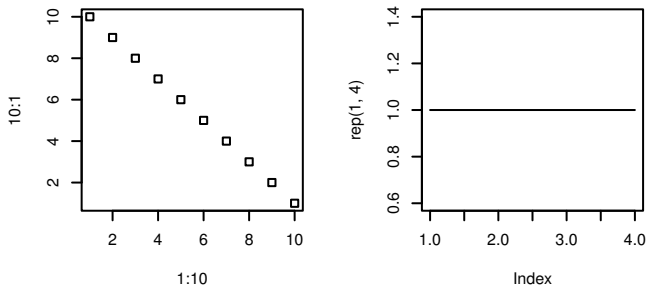


FIGURE 3.16 – Graphiques arrangés en 1 ligne × 2 colonnes grâce à **par**`(mfrow=`**c**`(1,2))`.

Pour revenir à un graphique par fenêtre, on ferme la fenêtre graphique ou on utilise la commande **par**`(mfrow=`**c**`(1,1))`.

Quelquefois, il est nécessaire d'avoir des graphiques de tailles différentes. Nous utilisons alors la fonction **layout** dont le principe est le suivant : cette fonction

découpe la fenêtre graphique en carreaux unitaires avant de les regrouper. Elle admet ainsi comme argument une matrice de taille `nrow` × `ncol`, ce qui crée `nrow` × `ncol` nombre de carreaux unitaires. Les valeurs de la matrice correspondent alors aux numéros des graphiques qui doivent être dessinés dans chaque carreau. Par exemple, pour disposer trois graphiques sur deux lignes, comme dans la figure 3.17, nous créons dans un premier temps la matrice :

$$\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix}$$

```
> mat <- matrix(c(1,1,2,3),nrow=2,ncol=2,byrow=TRUE)
```

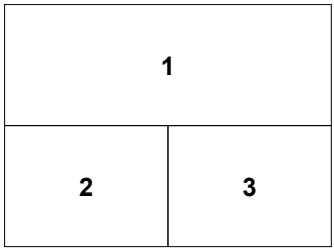


FIGURE 3.17 – Positionnement des trois graphiques sur deux lignes (grâce à **layout**).

Ensuite nous dessinons les trois graphes :

```
> layout(mat)
> plot(1:10,10:1,pch=0)
> plot(rep(1,4),type="l")
> plot(c(2,3,-1,0),type="b")
```

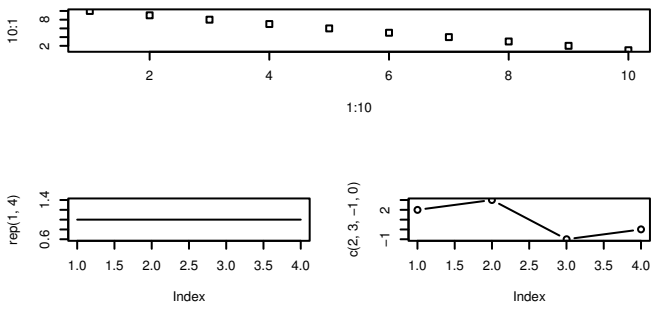


FIGURE 3.18 – Trois graphiques sur deux lignes grâce à **layout**.

Il est aussi possible de contrôler la hauteur et la largeur des colonnes (voir l'aide de **layout** et l'exercice 3.8). Remarquons que les graphiques sont très (trop) espacés mais il est possible de remédier à cela facilement (voir l'exercice 3.8).

Remarque
On peut avoir besoin d'ouvrir plusieurs fenêtres graphiques. Pour créer une fenêtre graphique, il suffit d'ouvrir le device qui correspond au système d'exploitation : `X11` sous Unix/Linux ou Windows et `quartz` sous macOS. Ainsi, si nous souhaitons réaliser deux graphiques dans deux fenêtres séparées, sous Unix/Linux ou Windows il suffit d'utiliser :

```
> plot(1:10,10:1,pch=0)
> X11()
> plot(rep(1,4),type="l")
```

Le premier graphique est réalisé. Si aucune fenêtre graphique n'est ouverte avant le premier graphique, elle est créée automatiquement. Une seconde fenêtre est ensuite ouverte, rendant inactive la première. Le second graphique est réalisé dans cette nouvelle fenêtre. Les graphiques suivants sont réalisés dans la fenêtre active.

3.1.7 Amélioration et personnalisation des graphiques

Afin d'améliorer le graphique, il est possible :

- d'utiliser des couleurs ou des nuances de gris :

  ```
  > plot(x,y,type="p",pch=o,col="grey50")
  ```

  Ces couleurs (ou nuances de gris) peuvent être appelées avec un chiffre (en général, 1 = noir, 2 = rouge, 3 = vert, etc.), avec un nom en anglais dont la liste est donnée par le vecteur **colors**(), ou avec un code RGB avec une transparence (voir **rgb**). La correspondance entre le numéro de couleur et la couleur est assurée par la palette. Il est possible de changer les couleurs de fond et de tracé du device grâce à la fonction **par** qui gère tous les paramètres graphiques d'un device :

  ```
  > par(fg="blue",bg="#f2a1c2")
  ```

- de modifier la palette qui fait la correspondance entre le numéro et la couleur :

  ```
  > palette(gray(seq(0,0.9,len=10))) # palette de gris
  > plot(1:10,rep(1,10),type="p",pch=25,bg=1:10)
  > palette("default")
  ```

- de mettre un titre :

  ```
  > plot(x,y,type="p",pch=o,main="Mon Titre")
  ```

- de contrôler l'aspect des axes en les éliminant (ainsi que leur légende) :

```
> plot(c(10,11),c(12,11),type="p",axes=FALSE,xlab="",ylab="")
```

pour enfin les « reconstruire à la main » :

```
> axis(1,at=c(10,11),label=c("coord 1","coord 2"))
```

– de construire des axes orthonormés (argument `asp=1`) :

```
> plot(x,y,type="p",asp=1)
```

– d'ajouter une légende :

```
> ecarty <- range(ozone[1:7,"maxO3"],ozone[8:14,"maxO3"])
> plot(ozone[1:7,"maxO3"],type="l")
> lines(ozone[8:14,"maxO3"],ylim=ecarty,col="grey50"))
> legend("topleft",legend=c("semaine 1","semaine 2"),
    col=c("black","grey50"),lty=1)
```

– d'insérer des symboles ou des formules mathématiques (`help(plotmath)`), par exemple pour une légende d'axe :

```
> plot(1,1,xlab=expression(bar(x) == sum(frac(x[i], n), i==1, n)))
```

Certains paramètres sont modifiables directement dans l'ordre graphique, d'autres sont accessibles par la fonction **par**.

Argument	Description
`adj`	contrôle la justification du texte par rapport au bord gauche du texte : 0 à gauche, 0.5 centré, 1 à droite ; `c(x,y)` justifie le texte horizontalement et verticalement
`asp`	précise le ratio entre `y` et `x` : `asp=1` pour un graphe orthonormé
`axes`	par défaut TRUE, les axes et le cadre sont tracés
`bg`	spécifie la couleur de l'arrière-plan : 1, 2, ou une couleur proposée par `colors()`
`bty`	contrôle le tracé du cadre, valeurs permises : "o", "l", "7", "c", "u" ou "]" (le cadre ressemblant au caractère correspondant) ; `bty="n"` supprime le cadre
`cex`	contrôle la taille des caractères et des symboles par rapport à la valeur par défaut qui vaut 1
`cex.axis`	contrôle la taille des caractères pour l'échelle des axes
`cex.lab`	contrôle la taille des caractères pour les libellés des axes
`cex.main`	contrôle la taille des caractères du titre
`cex.sub`	contrôle la taille des caractères du sous-titre
`col`	précise la couleur du graphe, valeurs possibles 1, 2, ou une couleur proposée par `colors()`
`col.axis`	précise la couleur des axes
`col.main`	précise la couleur du titre
`font`	contrôle le style du texte (1 : normal, 2 : italique, 3 : gras, 4 : gras-italique)
`font.axis`	contrôle le style pour l'échelle des axes

Argument	Description
`font.lab`	contrôle le style pour les libellés des axes
`font.main`	contrôle le style du titre
`font.sub`	contrôle le style du sous-titre
`las`	contrôle la disposition des annotations sur les axes (0, valeur par défaut : parallèles aux axes, 1 : horizontales, 2 : perpendiculaires aux axes, 3 : verticales)
`lty`	contrôle le type de ligne tracée, (1 : continue, 2 : tirets, 3 : points, 4 : points et tirets alternés, 5 : tirets longs, 6 : tirets courts et longs alternés), ou bien écrire en toutes lettres "solid", "dashed", "dotted", "dotdash", "longdash", "twodash" ou "blank" (pour ne rien écrire)
`lwd`	contrôle l'épaisseur des traits
`main`	précise le titre du graphe, par exemple `main="titre"`
`mfrow`	vecteur `c(nr,nc)` qui divise la fenêtre graphique en `nr` lignes et `nc` colonnes ; les graphes sont ensuite dessinés en ligne
`offset`	précise le décalage du texte par rapport au point (fonction **text**)
`pch`	entier (entre 0 et 25) qui contrôle le type de symbole (ou éventuellement n'importe quel caractère entre guillemets)
`pos`	précise la position du texte ; valeurs permises 1, 2, 3 et 4 (fonction **text**)
`ps`	entier qui contrôle la taille en points du texte et des symboles
`sub`	précise le sous-titre du graphe, par exemple `sub="sstitre"`
`tck`, `tcl`	précise la longueur des graduations sur les axes
`type`	précise le type de graphe dessiné, valeurs permises `"n","p","l","b,"h","s","S"`
`xlim, ylim`	précise les limites des axes, par exemple `xlim=c(0,10)`
`xlab, ylab`	précise les annotations des axes, par exemple `xlab="Abscisse"`

Nous donnons maintenant une liste de fonctions graphiques « conventionnelles ».

Fonction	Description
barplot`(x)`	trace un diagramme en barres des valeurs de `x`
boxplot`(x)`	trace le graphe en boîte à moustaches de `x`
contour`(x,y,z)`	trace des courbes de niveau, voir aussi la fonction **filled.contour**`(x,y,z)`
coplot`(x~y|f1)`	trace le graphe bivarié de `x` et `y` pour chaque valeur de `f1` (ou un petit intervalle de valeurs de `f1`)
filled.contour`(x, y,z)`	trace des courbes de niveau mais les aires entre les contours sont colorées, voir aussi **image**`(x,y,z)`
hist`(x,prob=TRUE)`	trace un histogramme de `x`
image`(x,y,z)`	trace des rectangles aux coordonnées `x`, `y` colorés selon `z`, voir aussi **contour**`(x,y,z)`

Fonction	Description
interaction.plot(f1, f2,x,fun=mean)	trace le graphe des moyennes de x en fonction des valeurs des facteurs f1 (sur l'axe des abscisses) et f2
matplot(x,y)	trace le graphe bivarié de la 1re colonne de x contre la 1re colonne de y, la 2e colonne de x contre la 2e colonne de y, etc.
pairs(x)	si x est une matrice ou un data-frame, trace tous les graphes bivariés entre les colonnes de x
persp(x,y,z)	trace une surface de réponse en 3D, voir **demo**(persp)
pie(x)	trace un graphe en camembert
plot(objet)	trace le graphique correspondant à la classe de objet
plot(x,y)	trace le nuage de points de coordonnées x et y
qqnorm(x)	trace les quantiles de x en fonction de ceux attendus d'une loi normale
qqplot(x,y)	trace les quantiles de y en fonction de ceux de x
spineplot(f1,f2)	trace le diagramme en bandes correspondant à f1 et f2
stripplot(x)	trace le graphe des valeurs de x sur une ligne
sunflowerplot(x,y)	idem mais les points superposés sont dessinés sous forme de fleurs dont le nombre de pétales correspond au nombre de points
symbols(x,y,...)	dessine aux coordonnées x et y des symboles (étoiles, cercles, boxplots, etc.)

3.2 Les fonctions graphiques avec ggplot2

Le package ggplot2 est un outil puissant et de plus en plus utilisé permettant de visualiser des données. Les graphes proposés par ggplot2 sont généralement plus élaborés et mieux finalisés que les graphes effectués avec les fonctions classiques de R. Ce package permet également d'obtenir des représentations graphiques par sous-groupes d'individus avec très peu de lignes de code. La disposition des graphes et la gestion des titres et légendes sont le plus souvent gérées automatiquement.

La syntaxe de ggplot2 est spécifique et totalement différente de celle des graphiques conventionnels de R. Elle est de type « Grammar of Graphics » et se structure comme un ensemble de commandes indépendantes. Cette section n'est qu'une introduction. Pour plus de détails, le lecteur pourra se référer à Wickham (2009), l'auteur de ce package. Il pourra également consulter (et garder à portée de mains) l'antisèche, la « Cheat Sheet », ggplot2 : il suffit de taper « Cheat Sheet ggplot2 » dans google, ou d'aller dans l'onglet Help de Rstudio où toutes antisèches sont disponibles.

3.2.1 Premiers graphes avec ggplot2

On considère le jeu de données `diamonds` qui contient le prix ainsi que certaines caractéristiques de 54 000 diamants. Afin d'alléger les représentations graphiques, on se restreint à un sous-échantillon de taille 5000 du jeu de données `diamonds` :

```
> library(ggplot2)
> set.seed(1234)
> diamonds2 <- diamonds[sample(nrow(diamonds),5000),]
```

Pour obtenir la représentation graphique souhaitée, l'approche ggplot2 consiste à segmenter les instructions. Prenons l'exemple d'un diagramme en barres pour la variable `cut` qui représente la qualité de la coupe du diamant. Pour ce type de représentation, il est nécessaire de spécifier :
– le jeu de données : `diamonds2` ;
– la variable que l'on souhaite représenter : `cut` ;
– le type de représentation voulu : `diagramme en barres`.
Le package ggplot2 utilise cette décomposition pour construire le diagramme en barres (voir graphe de gauche de la figure 3.19) :

```
> ggplot(diamonds2)+aes(x=cut)+geom_bar()
```

Le jeu de données est renseigné par la fonction **ggplot**, la variable à représenter par la fonction **aes**, le type de représentation souhaité est ensuite spécifié par la fonction **geom_bar()**. Les trois instructions sont séparées par un « + ». C'est le schéma classique de construction de graphes avec ggplot2. Avec un protocole identique, on obtient l'histogramme de la variable continue `price` (graphe de droite Fig. 3.19) :

```
> ggplot(diamonds2)+aes(x=price)+geom_histogram()
```

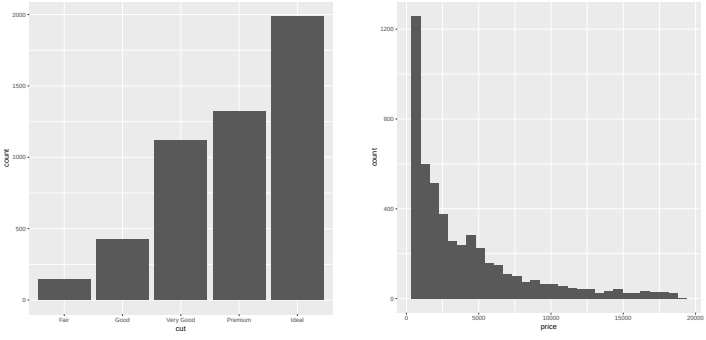


FIGURE 3.19 – Diagramme en barres pour la variable `cut` (gauche) et histogramme des effectifs pour la variable `price` (droite).

Les deux représentations précédentes nécessitent la spécification d'une seule variable : `cut` pour le diagrame en barres et `price` pour l'histogramme. Cependant, de nombreuses représentations nécessitent plus d'une variable. Pour un nuage de points par exemple, deux variables sont à spécifier. Il suffit dans ce cas d'indiquer ces deux variables dans la fonction **aes**. Pour représenter le prix d'un diamant en fonction de son nombre de carats (voir graphe de gauche 3.20), on utilise :

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point()
```

3.2.2 La grammaire ggplot

Ainsi, une représentation graphique `ggplot` se construit à partir d'un ensemble d'éléments indépendants qui constituent la grammaire de la syntaxe. Les principaux éléments de cette grammaire sont :

- **Data** (**ggplot**) : le jeu de données contenant les variables utilisées ;
- **Aesthetics** (**aes**) : les variables à représenter, on peut également inclure des couleurs ou tailles si ces dernières sont associées à des variables ;
- **Geometrics** (**geom_...**) : le type de représentation graphique souhaitée ;
- **Statistics** (**stat_...**) : les éventuelles transformations des données pour la représentation souhaitée ;
- **Scales** (**scale_...**) : permet de contrôler le lien entre les données et les aesthetics (modification de couleurs, gestion des axes...).

Nous présentons dans la suite quelques notions élémentaires sur ces éléments.

Data et aesthetics

Ces deux éléments de la grammaire spécifient le jeu de données et les variables que l'on souhaite représenter. Le jeu de données est un data-frame (ou un data-table, cf. § 5.1, ou un tibble, cf. § 5.2) que l'on renseigne dans la fonction **ggplot**. Les variables sont spécifiées dans la fonction **aes**. Cette fonction admet également des arguments tels que `color`, `size`, `fill`. Ces arguments sont à utiliser lorsqu'une couleur ou une taille est définie à partir d'une variable du jeu de données. Si on veut par exemple visualiser le prix d'un diamant en fonction de son nombre de carats avec une couleur différente selon les modalités de la variable `cut`, on utilise :

```
> ggplot(diamonds2)+aes(x=carat,y=price,color=cut)
```

Geometrics

La commande précédente ne renvoie pas de graphique ! Elle spécifie uniquement les variables du jeu de données que l'on souhaite utiliser dans la représentation. Les éléments **geom_...** précisent le type de représentation souhaitée. Pour obtenir le nuage `carat`×`price` avec une couleur différente selon `cut` (graphe de droite de la figure 3.20), on exécute donc :

```
> ggplot(diamonds2)+aes(x=carat,y=price,color=cut)+geom_point()
```

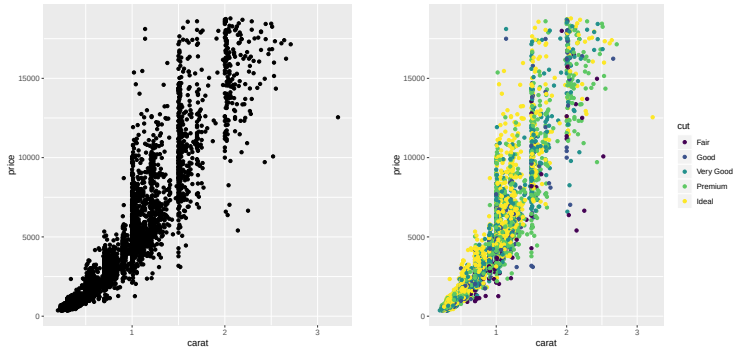


FIGURE 3.20 – Nuage de points représentant le prix d'un diamant en fonction de son nombre de carats (gauche). Même chose avec un code couleur défini par la variable `cut` (droite).

Nous remarquons qu'une légende précisant le code couleur de la variable `cut` est ajoutée directement. Le tableau 3.3 (source Murell (2011)) donne quelques exemples de geometrics (accompagnés des aesthetics) permettant de faire les représentations graphiques classiques.

Geom	**Description**	**Aesthetics**
geom_point()	Nuage de points	x, y shape, fill
geom_line()	Ligne (ordonnées selon x)	x, y, linetype
geom_abline()	Droite	slope, intercept
geom_path()	Ligne (ordre original)	x, y, linetype
geom_text()	Texte	x, y, label, hjust, vjust
geom_rect()	Rectangle	xmin, xmax, ymin, ymax fill, linetype
geom_polygon()	Polygone	x, y, fill, linetype
geom_segment()	Segment	x, y, fill, linetype
geom_bar()	Diagramme en barres	x, fill, linetype, weight
geom_histogram()	Histogramme	x, fill, linetype, weight
geom_boxplot()	Boîtes à moustaches	x, y, fill, weight
geom_density()	Densité	x, y, fill, linetype
geom_contour()	Lignes de contour	x, y, fill, linetype
geom_smooth()	Lissage	x, y, fill, linetype
Tous		color, size, group

TABLE 3.3 – Exemples de geom et aesthetics associés.

Chaque fonction **geom_...** admet également des arguments particuliers permettant de modifier le graphe (couleur, taille de points, épaisseur de traits, etc.). Si on souhaite par exemple représenter un nuage de points rouges, c'est dans le fonction **geom_point**, non dans la fonction **aes**, qu'il faut utiliser l'argument `color`. De même, pour obtenir un diagramme en barres bleu, on utilise `fill="blue"` dans **geom_bar** (voir Fig. 3.21) :

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point(color="red")
> ggplot(diamonds2)+aes(x=cut)+geom_bar(fill="blue")
```

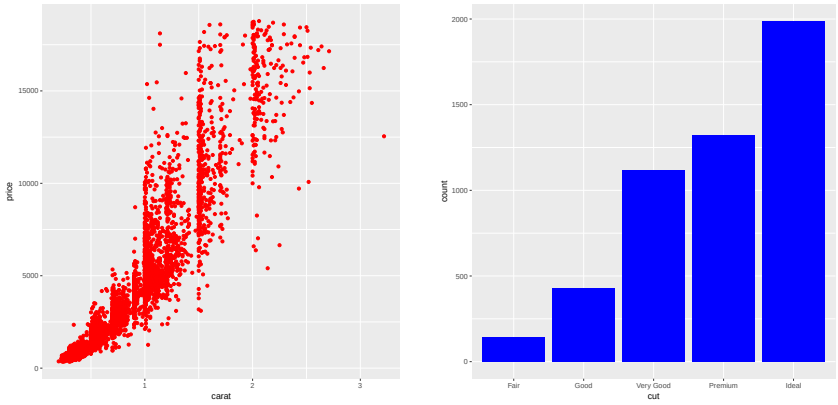


FIGURE 3.21 – Arguments dans les `geometrics`.

Statistics

De nombreuses représentations graphiques ne se déduisent pas directement des données, elles nécessitent de transformer des variables ou de calculer des indicateurs à partir des variables. Lorsque la transformation est fonctionnelle, il suffit de spécifier cette transformation dans la fonction **aes**. On peut par exemple représenter la fonction sinus sur $[-2\pi, 2\pi]$ comme suit :

```
> D <- data.frame(X=seq(-2*pi,2*pi,by=0.01))
> ggplot(D)+aes(x=X,y=sin(X))+geom_line()
```

Les valeurs du sinus ne sont pas dans le jeu de données `D`, on précise que l'on souhaite représenter en ordonnée la fonction sinus via `y=sin(X)` dans **aes**.

De nombreuses représentations nécessitent des transformations plus complexes que l'application d'une fonction usuelle. C'est par exemple le cas de l'histogramme où il faut calculer le nombre d'individus dans chaque classe pour déduire les hauteurs de l'histogramme. Les statistics permettent de gérer ces éléments intermédiaires dans

les graphes. Ils sont renseignés dans l'argument `stat` des fonctions **geom_...**. Pour **geom_histogram**, la valeur par défaut est par exemple `stat="bin"`. Cet argument permet de calculer 4 indicateurs :

- `count` : nombre de points dans chaque classe ;
- `density` : densité pour chaque classe ;
- `ncount` : équivalent de `count` modifié de manière à ce que la valeur maximale soit de 1 ;
- `ndensity` : équivalent de `density` modifié de manière à ce que la valeur maximale soit de 1.

Par défaut, la fonction **geom_histogram** représente en ordonnée le premier de ces quatre indicateurs, en l'occurrence `count`. Ainsi, la commande

```
> ggplot(diamonds2)+aes(x=price)+geom_histogram(bins=40)
```

renvoie l'histogramme où figure en ordonnée le nombre d'individus par classe, non la densité, et donne le même résultat que

```
> ggplot(diamonds2)+aes(x=price,y=..count..)+geom_histogram(bins=40)
```

Si on souhaite visualiser un autre indicateur sur l'axe des ordonnées, il faut préciser son nom dans la fonction **aes** en utilisant le format `..nom...` Pour représenter l'histogramme de la densité (voir Fig. 3.22), on utilise donc

```
> ggplot(diamonds2)+aes(x=price,y=..density..)+geom_histogram(bins=40)
```

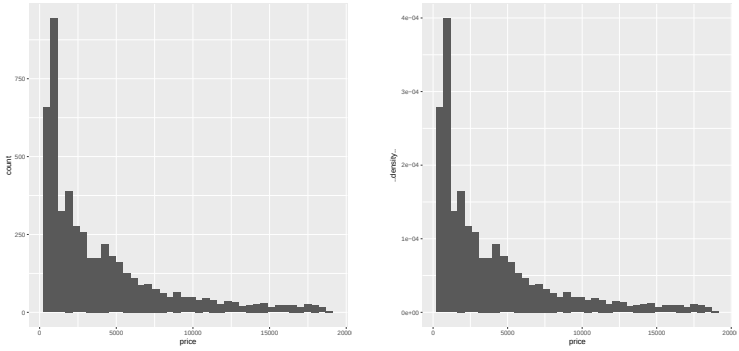


FIGURE 3.22 – Histogramme des fréquences (gauche) et de la densité (droite).

Le package ggplot2 propose une alternative pour obtenir des représentations faisant intervenir des transformations des données, il s'agit des fonctions **stat_...**. On pourra par exemple obtenir les deux histogrammes de la figure 3.22 avec

```
> ggplot(diamonds2)+aes(x=price)+stat_bin(bins=40)
> ggplot(diamonds2)+aes(x=price,y=..density..)+stat_bin(bins=40)
```

Nous avons vu que les fonctions **geom_...** possèdent un argument `stat`. De la même façon, les fonctions **stat_...** possèdent un argument `geom`. En modifiant cet argument, on change la représentation graphique. Par exemple, la fonction **stat_smooth**, qui permet de faire le lissage d'un nuage de points, admet par défaut l'argument `geom="smooth"`. Elle permet de représenter le lissage accompagné d'une bande de confiance. Si on utilise `geom="line"`, la bande de confiance n'est plus représentée, avec `geom="point"` le lissage est représenté en pointillé (Fig. 3.23) :

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point(size=0.5)+
    stat_smooth(method="loess",size=2)
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point(size=0.5)+
    stat_smooth(method="loess",geom="line",color="blue",size=2)
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point(size=0.5)+
    stat_smooth(method="loess",geom="point",color="blue",size=2)
```

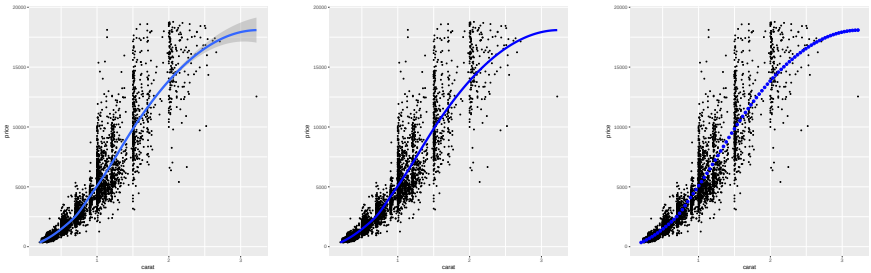


FIGURE 3.23 – Lissage avec **stat_smooth** pour l'argument `geom="smooth"` (gauche), `"line"` (centre) et `"point"` (droite).

De nombreuses représentations graphiques peuvent être obtenues en utilisant indifféremment les fonctions de type **geom_...** ou **stat_...**. On pourra par exemple obtenir les lissages de la figure 3.23 avec la fonction **geom_smooth**. C'est à l'utilisateur de choisir la syntaxe qu'il préfère. Le tableau 3.4 présente quelques exemples de fonctions **stat_...**.

Stat	**Description**	**Paramètres**
stat_identity()	Aucune transformation	
stat_bin()	Comptage	binwidth, origin
stat_density()	Densité	adjust, kernel
stat_smooth()	Lissage	method, se
stat_boxplot()	Boxplot	coef

TABLE 3.4 – Exemples de statistics.

Scales

Les scales contiennent tous les paramètres qui font le lien entre les données (data) et les aesthetics. Ils permettent généralement d'affiner le graphe en modifiant par exemple des palettes de couleurs ou en créant des dégradés de couleurs, ou encore en gérant les axes du graphe. Les fonctions scales suivent le schéma suivant :
– elles commencent par **scale_** ;
– suivi du nom de l'aesthetics que l'on souhaite modifier (**color_**, **fill_**, **x_**...) ;
– et se terminent par le nom du scale : **manual**, **identity**...
On modifie par exemple la couleur des points du graphe 3.20 avec la commande suivante (voir Fig. 3.24) :

```
> ggplot(diamonds2)+aes(x=carat,y=price,color=cut)+geom_point()+
    scale_color_manual(values=c("Fair"="black","Good"="yellow",
    "Very Good"="blue","Premium"="red","Ideal"="green"))
```

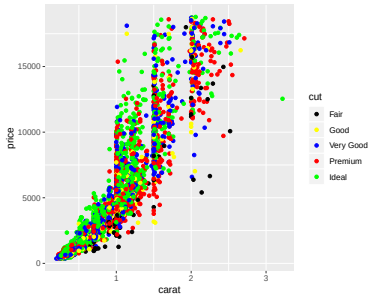


FIGURE 3.24 – Changement de couleurs à l'aide de **scale_color_manual**.

Le tableau 3.5 présente les principaux types de scales.

Position `x` (ou `y`)	Taille	Forme	Couleur `colour` (ou `fill`)
Variables continues			
_x_continuous	_size_continuous		_colour_gradient
_x_date	_area		_colour_gradient2
_x_datetime			_colour_gradientn
Variables discrètes			
_x_discrete	_size_discrete	_shape_discrete	_colour_hue
	_size_identity	_shape_identity	_colour_brewer
	_size_manual	_shape_manual	_colour_grey
			_colour_identity
			_colour_manual

TABLE 3.5 – Exemples de scales en fonction des types de variables.

Voici quelques exemples concrets d'utilisation des scales :

– **Couleurs d'un diagramme en barre** : on construit le diagramme en barre de la variable `cut` en utilisant une couleur différente pour chaque barre (Fig. 3.25 à gauche) :

```
> p1 <- ggplot(diamonds2)+aes(x=cut)+geom_bar(aes(fill=cut))
> p1
```

On change la couleur des barres en utilisant la palette de couleurs prédéfinie `Reds` (Fig. 3.25 à droite) :

```
> p1 + scale_fill_brewer(palette="Reds")
```

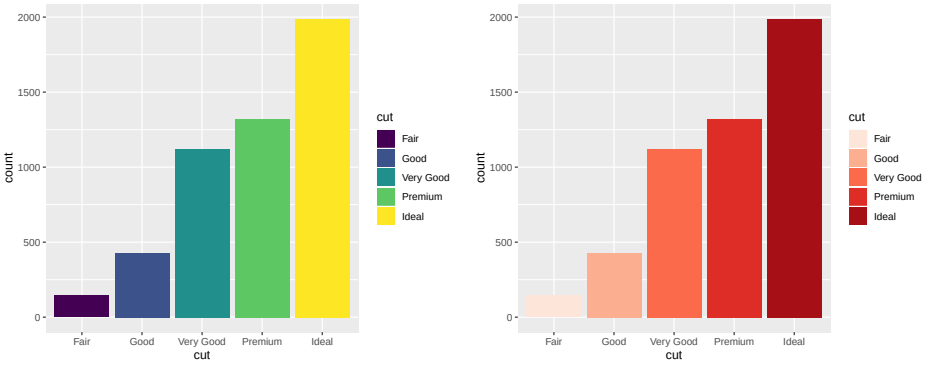


FIGURE 3.25 – Changement de couleur d'un diagramme en barres à l'aide de **scale_fill_brewer**.

– **Dégradé de couleurs pour un nuage de points** : on représente le nuage de points `carat`×`price` avec une échelle de couleur définie par la variable continue `depth` :

```
> p2 <- ggplot(diamonds2)+aes(x=carat,y=price)+
   geom_point(aes(color=depth))
> p2
```

On modifie le dégradé de couleurs proposé par défaut en définissant un nouveau dégradé allant du jaune au rouge à l'aide de **scale_color_gradient** (voir Fig. 3.26) :

```
> p2 + scale_color_gradient(low="red",high="yellow")
```

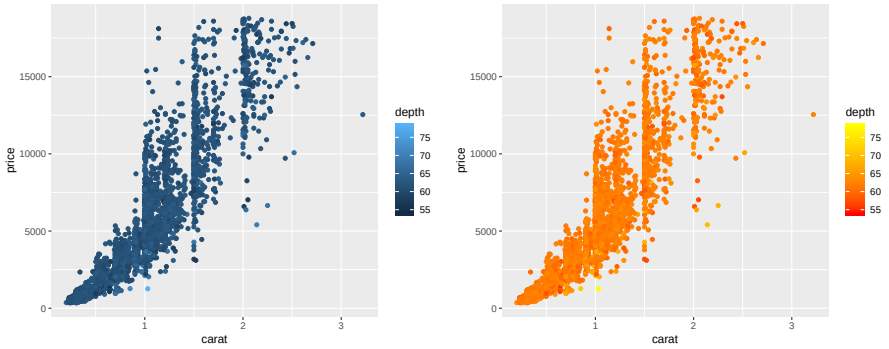


FIGURE 3.26 – Changement de couleur d'un dégradé de couleurs à l'aide de **scale_color_gradient**.

– **Gestion des axes et de la légende** : on peut modifier la graduation de l'axe des abscisses, le nom de l'axe des ordonnées et celui de la variable de la légende du graphe `p2` avec (voir Fig. 3.27)

```
> p2+scale_x_continuous(breaks=seq(0.5,3,by=0.5))+
    scale_y_continuous(name="prix")+
    scale_color_gradient("Profondeur")
```

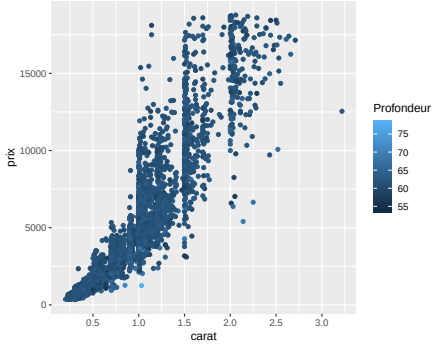


FIGURE 3.27 – Gestion des axes.

3.2.3 Group et facets

Le package ggplot2 permet aussi de faire des représentations pour des sous-groupes d'individus caractérisés par une ou plusieurs variables. Il y a essentiellement deux manières d'agir :

– on souhaite représenter les sous-groupes d'individus sur un *même* graphe : on utilise l'argument `group` dans la fonction **aes** ;

– on souhaite représenter les sous-groupes dans des graphes différents : on utilise les fonctions **facet_grid** et **facet_wrap**.

Considérons l'exemple d'un lissage du nuage de points `carat`×`price` avec la fonction **geom_smooth** (graphe de gauche sur la figure 3.28) :

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_smooth()
```

Pour obtenir un lissage pour chaque valeur de la variable `cut` (on regroupe les individus pour chaque modalité de `cut` et on fait un lissage par groupe d'individus), il suffit d'ajouter l'argument `group` dans **aes** :

```
> ggplot(diamonds2)+aes(x=carat,y=price,group=cut)+geom_smooth()
```

On visualise bien les cinq lissages sur le graphe du milieu de la figure 3.28. Ils ne sont cependant pas distingués et on ne sait pas à quelle valeur de `cut` chaque courbe correspond. Pour pallier ce problème, il suffit d'ajouter `color=cut` dans **aes** (voir le graphe de droite de la figure 3.28) :

```
> ggplot(diamonds2)+aes(x=carat,y=price,group=cut,color=cut)+
      geom_smooth()
```

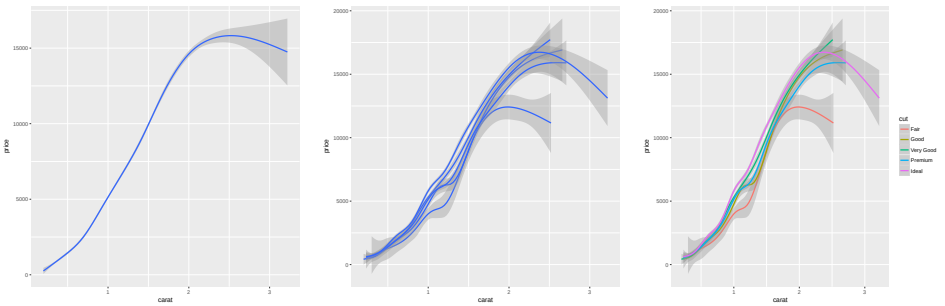


FIGURE 3.28 – Lissage en fonction de la variable `cut`.

Si l'on souhaite obtenir les lissages sur des graphes séparés, on utilise les fonctions **facet_grid** ou **facet_wrap**. Ces fonctions admettent comme argument une formule de la forme `var1~var2`. Elles renvoient en sortie la représentation souhaitée calculée sur les individus appartenant à chaque croisement des modalités de `var1` et `var2`. La principale différence entre ces deux fonctions se situe dans la disposition des graphiques. **facet_grid** produit un tableau de graphes de dimension $n_1 \times n_2$ où n_1 et n_2 représentent les nombres de modalités de `var1` et `var2`. Chaque graphe correspond à la représentation des individus possédant une modalité de `var1` en ligne et une modalité de `var2` en colonne. **facet_wrap** calcule tous les croisements et représente les graphes associés les uns à la suite des autres. On représente par

exemple le nuage de points `carats`×`price` ainsi que la droite de régression associée pour chaque croisement des variables `color` et `cut` (voir Fig. 3.29) grâce aux lignes de code :

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point()+
    geom_smooth(method="lm")+facet_grid(color~cut)
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point()+
    geom_smooth(method="lm")+facet_wrap(color~cut)
```

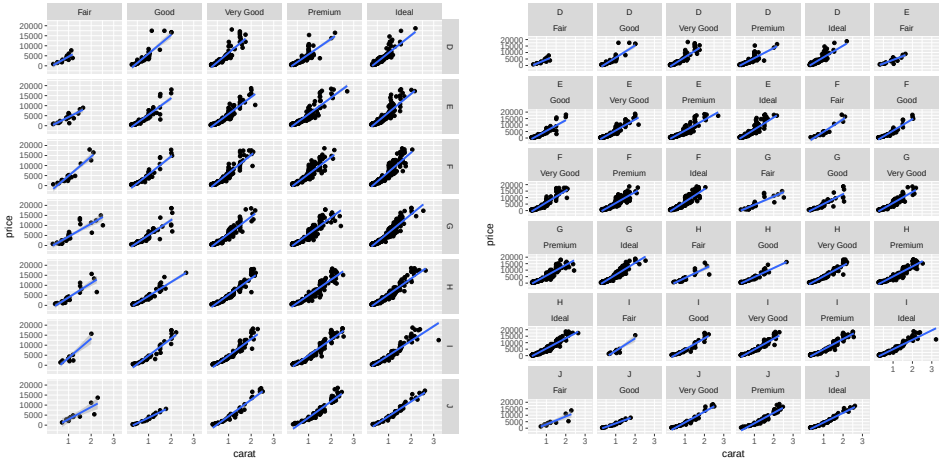


FIGURE 3.29 – Nuages de points `price vs carat` et droite des moindres carrés en fonction des variables `cut` et `color` (avec **facet_grid** à gauche et **facet_wrap** à droite).

facet_grid propose une présentation plus cohérente lorsqu'on est en présence de deux variables avec peu de modalités. Lorsqu'on est en présence d'une seule variable, ou lorsqu'une variable admet beaucoup de modalités, il est préférable d'utiliser **facet_wrap**. On peut par exemple obtenir les lissages du graphe de droite de la figure 3.28 sur des figures séparées (voir Fig. 3.30) en utilisant

```
> ggplot(diamonds2)+aes(x=carat,y=price,color=cut)+geom_smooth()+
    facet_wrap(~cut,nrow=2)
```

Remarque

Pour dessiner plusieurs graphiques de la même forme dans une même fenêtre, il est toujours préférable de créer une variable `group` ou bien d'utiliser les fonctions **facet_grid** ou **facet_wrap**. Cependant il est aussi possible de faire figurer des graphiques différents dans une même fenêtre. Pour cela, on définit dans un premier

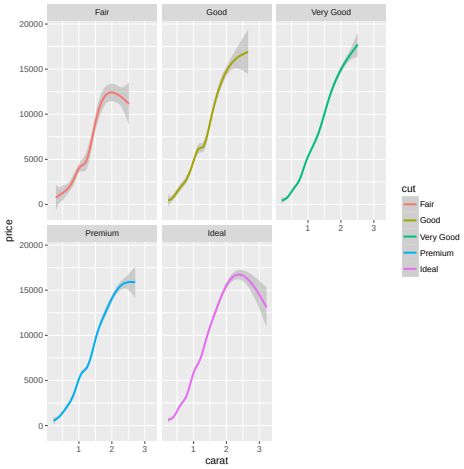


FIGURE 3.30 – Lissage en fonction de la variable `cut` avec **facet_wrap**.

temps les graphiques puis on les organise avec la fonction **grid.arrange** du package gridExtra. Par exemple, pour reproduire la figure 3.19, on écrit :

```
> gr1 <-  ggplot(diamonds2)+aes(x=cut)+geom_bar()          #  1er graphe
> gr2 <- ggplot(diamonds2)+aes(x=price)+geom_histogram()  #  2e graphe
> library(gridExtra)
> grid.arrange(gr1, gr2, ncol=2, nrow=1)          # organisation des graphes
```

3.2.4 Compléments

Nous avons présenté la structure ggplot2 selon

```
ggplot(...)+aes(...)+geom_...(...)+...
```

La syntaxe est en réalité plus flexible. On peut par exemple spécifier les `aes` dans **ggplot** ou dans les **geom_**. Les trois instructions suivantes produiront par exemple le même graphe

```
> ggplot(diamonds2)+aes(x=carat,y=price)+geom_point()
> ggplot(diamonds2,aes(x=carat,y=price))+geom_point()
> ggplot(diamonds2)+geom_point(aes(x=carat,y=price))
```

Il est également possible de produire des représentations graphiques à partir de plusieurs jeux de données. On considère par exemple les jeux de données `donnees1` et `donnees2` qui contiennent respectivement les valeurs de cosinus et sinus sur l'intervalle $[-2\pi, 2\pi]$:

```
> X <- seq(-2*pi,2*pi,by=0.001)
> Y1 <- cos(X)
> Y2 <- sin(X)
> donnees1 <- data.frame(X,Y1)
> donnees2 <- data.frame(X,Y2)
```

On représente les deux fonctions sur le même graphe en utilisant

```
> ggplot(donnees1)+geom_line(aes(x=X,y=Y1))+
    geom_line(data=donnees2,aes(x=X,y=Y2),color="red")
```

Lorsqu'on souhaite utiliser une variable qui commence par un chiffre ou un caractère spécial dans la fonction **aes**, il faut mettre cette variable entre deux accents graves (backquotes) : `` `nomvariable` ``. Par exemple, on obtiendra le nuage de points `100m`×`1500m` du jeu de données `decathlon` avec

```
> library(FactoMineR) #Pour obtenir le jeu de donnees
> data("decathlon")
> ggplot(decathlon)+aes(x=`100m`,y=`1500m`)+geom_point()
```

Il existe bien entendu un grand nombre d'autres fonctions proposées par ggplot2. On pourra citer notamment **ggtitle** qui permet d'ajouter un titre au graphique. Par ailleurs, l'environnement par défaut du graphe est défini par la fonction **theme_grey**, mais il est facile de changer ce thème (voir Fig. 3.31) :

```
> p <- ggplot(diamonds2)+aes(x=carat,y=price,color=cut)+geom_point()
> p + theme_bw()
> p + theme_classic()
> p + theme_grey()
> p + theme_minimal()
```

D'autres thèmes sont disponibles dans le package ggthemes.

Enfin la fonction **qplot** est en quelque sorte l'analogue pour ggplot2 de la fonction **plot** pour les graphiques classiques. Elle est très simple à utiliser : on obtiendra par exemple un graphe proche de la figure 3.29 avec la commande

```
> qplot(data=diamonds2,x=carat,y=price,geom=c("point","smooth"),
    facets=color~cut)
```

Exportation

Pour exporter un graphique ggplot2, on peut utiliser la fonction **ggsave** :

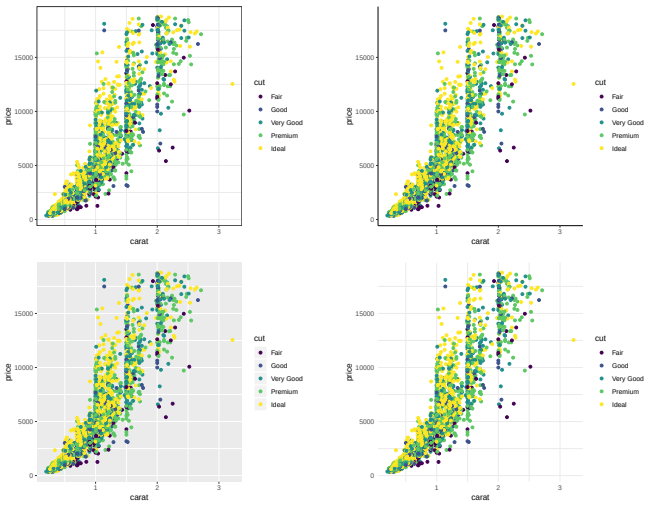


FIGURE 3.31 – Exemple de `themes`.

```
> monplot <- qplot(data=diamonds2,x=carat,y=price,geom=c("point","smooth"),
    facets=color~cut)
> ggsave("mon_graphique.pdf", plot = monplot, width = 11, height = 8)
```

3.3 Les graphiques interactifs

De nombreux packages permettent la construction de graphiques interactifs où le survol du graphe avec la souris affiche des informations complémentaires, où il est possible de zoomer pour sélectionner une partie du graphique, etc. En effet, avec l'arrivée du package htmlwidgets, de plus en plus de fonctionnalités de librairies javascript sont accessibles sous R. La page `http://www.htmlwidgets.org/` liste la plupart des packages dans l'onglet `showcase` et présente des exemples de graphiques dans l'onglet `Gallery`. Les graphiques sont visibles dans une page internet ou dans le Viewer de RStudio, et s'intègrent facilement dans des documents RMarkdown ou dans des applications shiny. La majorité de ces packages permettent ensuite l'exportation des graphiques retravaillés en divers formats (*.html*, *.png*, etc.).

Présentons par exemple le package rAmCharts qui reprend les principales fonctions graphiques de base de R. Le nom de la fonction est « am » suivi du nom de la fonction R avec la première lettre en majuscule. Par exemple **amBarplot**, **amHist**, etc. Voici un exemple pour construire des boîtes à moustaches (Fig. 3.32) :

```
> library(rAmCharts)
> amBoxplot(maxO3 ~ vent, data = ozone, export = TRUE)
```

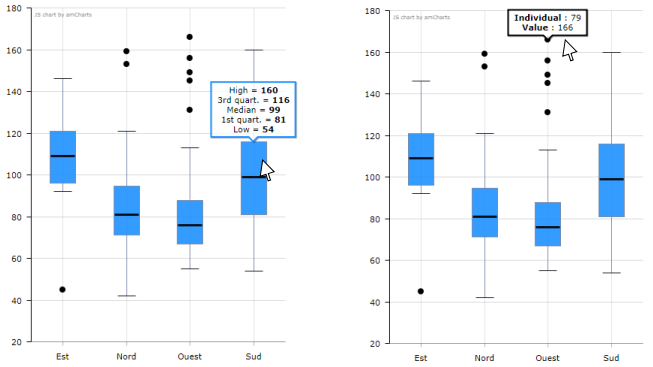


FIGURE 3.32 – Saisies d'écran de boîtes à moustaches avec information sur la boîte `sud` (à gauche) et sur un point aberrant (à droite).

En survolant le graphique avec la souris, on obtiendra différentes informations, par exemple les points aberrants, les quantiles d'une boîte à moustaches, les effectifs dans une classe d'un diagramme en barres, etc.

Il est également possible de représenter des données temporelles, de modifier la plage de temps, et de voir quelles valeurs sont prises à une date donnée par plusieurs variables en pointant sur un jour donné (voir Fig. 3.33) :

```
> data(data_stock_2)
> amTimeSeries(data_stock_2, "date", c("ts1", "ts2"))
```

FIGURE 3.33 – Représentation de deux séries temporelles.

Si on construit des graphiques avec ggplot2, on peut réaliser le même type de représentation interactive grâce au package plotly. L'utilisation est très simple : une fois le graphe **ggplot** construit, il suffit de lancer **ggplotly()** :

```
> ggplot(ozone) + aes(x=T12,y=maxO3,color=vent) + geom_point()
> library(plotly)
> ggplotly()
```

Des boutons au-dessus du graphe (Fig. 3.34) permettent de zoomer le graphique, de sélectionner un ensemble de points pour l'agrandir, etc.

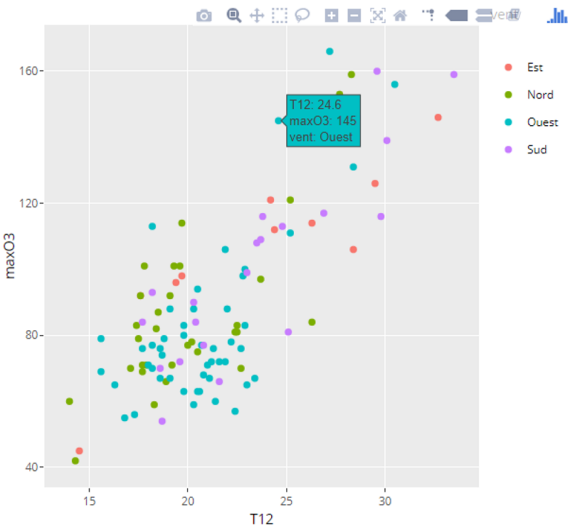


FIGURE 3.34 – Saisie d'écran d'un nuage de points.

Il est également possible de construire des graphiques animés avec plotly ou encore le package gganimate. La visualisation est une discipline en plein essor et les outils évoluent très vite.

3.4 Construire des cartes

3.4.1 Carte statique dans R

Un beau graphique vaut mieux qu'un long discours. Aussi, dans cette partie, nous allons montrer comment visualiser des informations sur une carte géographique obtenue facilement grâce au package ggmap pour des graphiques type « ggplot », ou au package RgoogleMaps pour des graphiques classiques.

Utilisons le jeu de données `quakes` proposé dans R. Il s'agit de la mesure, sur 1000 séismes le long de la faille des Tonga, de la magnitude `magn`, de la longitude `long`, de la latitude `lat` et de la profondeur `depth`. La distribution spatiale des séismes apporte de l'information sur la localisation de cette faille. L'objectif ici est de représenter les profondeurs des séismes sur une carte géographique.

Le fond de cartes, i.e. la partie de la surface terrestre qui nous intéresse, est défini par son centre, qui est dans cet exemple le point de latitude -24.66 degrés et de longitude 176.9 degrés [1], et un zoom (compris entre 0 et 19 pour GoogleMap) qui donne la dimension de la carte autour du centre :

```
> lat <- -24.66
> lon <- 176.9
> zoom <- 4
```

Nous devons ensuite télécharger ce fond de carte depuis un serveur de fonds de cartes (par exemple Google ou OpenStreetMap) grâce à la fonction **get_map** du package `ggmap`, le résultat étant stocké dans l'objet `Macarte` de classe `ggmap` (voir l'exercice 3.12 pour plus de détails) :

```
> library(ggmap)
> MaCarte <- get_map(location=c(lon,lat),zoom=zoom)
```

Enfin, nous pouvons afficher le fond de carte dans une fenêtre graphique grâce à :

```
> ggmap(MaCarte)
```

Afin d'ajouter à ce fond de carte les séismes, figurés par des points dont la couleur varie en fonction de la profondeur, nous commençons par récupérer les données (disponibles dans R) :

```
> data(quakes)
```

Nous représentons alors chaque séisme sur la carte grâce à sa latitude et sa longitude et colorions le point en fonction de sa profondeur (Fig. 3.35) en modifiant l'échelle de couleur via **scale_color_gradient** pour la faire varier du jaune (peu profond) au rouge foncé (très profond) :

```
> ggmap(MaCarte) +
    geom_point(data=quakes,aes(x=long,y=lat,colour=depth)) +
    scale_color_gradient(low="yellow2",high="red4") + theme_void()
```

Le thème « vide » (**theme_void**) permet ici d'éliminer les axes.

1. Latitude géodésique mesurée avec l'ellipsoïde de révolution WGS84, système de coordonnées code EPSG 4326.

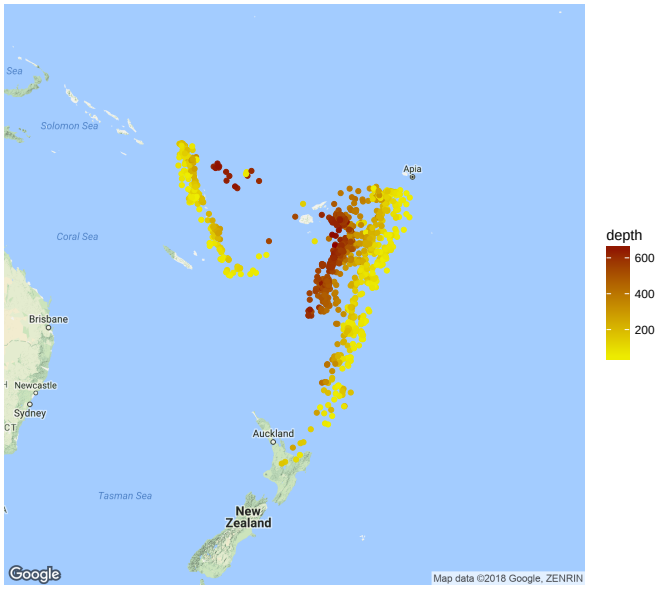


FIGURE 3.35 – Séismes autour des îles Tonga.

Le fond de carte peut aussi être défini par les coins de la carte (« bounding box ») plutôt que par un couple centre + zoom. Le package ggmap permet de déterminer ces coins dans une série de couples longitude/latitude grâce à la fonction **make_bbox**. Il suffit d'indiquer à cette fonction l'objet contenant la série de longitudes, celui contenant la série de latitudes et éventuellement le nom du data-frame les contenant :

```
> bbox <- make_bbox(long,lat,data=quakes)
> bbox
    left   bottom    right      top
164.5470 -39.9835 189.2530  -9.3265
```

Ces coins peuvent être utilisés directement pour télécharger le fond de carte optimal, lequel peut ensuite être représenté avec **ggmap** :

```
> MaCarteopt <- get_map(bbox)
> ggmap(MaCarteopt)
```

Une autre façon de déterminer un fond de carte est sa localisation en utilisant son « adresse ». Ainsi, pour obtenir la carte correspondant par exemple au chalet du pic de Rochebrune (situé près de Megève, France), nous utilisons la fonction **geocode** qui renvoie les longitude/latitude du chalet (attention, il est préférable de ne pas mettre les accents pour éviter les problèmes d'encodage) :

```
> centre <- geocode("Pic de Rochebrune, Megeve, France")
> centre
       lon      lat
1 6.615982 45.85161
```

Ce vecteur `centre` peut être utilisé pour télécharger le fond de carte à un zoom donné (ici notre choix est de 13), qui est ensuite tracé (carte non fournie) :

```
> CarteRochebrune <- get_map(centre, zoom=13)
> ggmap(CarteRochebrune)
```

3.4.2 Carte dans un navigateur

Comme en section précédente, nous allons chercher à afficher un fond de carte et y tracer éventuellement d'autres informations. Cependant, l'affichage sera ici déporté dans un navigateur (Firefox, Chrome ou autre), ce qui permettra d'ajouter de l'interactivité, par exemple pour agrandir ou diminuer le zoom.

Le package utilisé ici, leaflet, va servir d'interface entre R et la bibliothèque de programmes javascript leaflet. Le principe est de construire la carte grâce à des couches (« layers ») qui se superposent. Commençons par l'affichage de la carte du monde (voir Fig. 3.36a) :

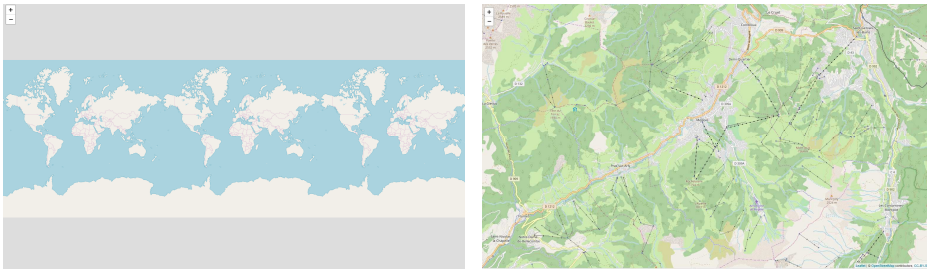
```
> library(leaflet)
> m <- leaflet()
> m <- addTiles(m)
> print(m)
```

Après le chargement du package, la fonction **leaflet** permet de créer un widget de type « map » stocké dans l'objet `m`. Ce widget contiendra toutes les couches à afficher. Les « tuiles » formant le fond de carte sont ajoutées à l'objet `m` avec la fonction **addTiles** et forment ainsi une couche. Par défaut, les tuiles sont téléchargées depuis un serveur de tuiles OpenStreetMap. Enfin, l'affichage permet de générer la page html complète (avec le code javascript leaflet) qui est ensuite affichée dans le navigateur par défaut (voir **getOption**`("browser")`). Une syntaxe plus lisible utilisant le pipe `%>%` du package dplyr (voir section 5.2.1) permet d'obtenir la même carte :

```
> m <- leaflet() %>% addTiles()
> print(m)
```

Afin d'initialiser la carte autour de la région d'intérêt, nous pouvons utiliser la fonction **setView** en utilisant les longitude et latitude de Rochebrune et un zoom de 13 (voir Fig. 3.36b) :

```
> m <- leaflet()  %>%
      setView(lng=6.62,lat=45.85,zoom=13) %>%
      addTiles()
> print(m)
```



(a) Carte non centrée. (b) Carte centrée sur Rochebrune.

FIGURE 3.36 – Cartes leaflet.

Ensuite, nous pouvons ajouter des couches d'annotations sur ces fonds de cartes. Ces annotations peuvent être de différentes formes : des cercles, des marqueurs (avec du texte apparaissant quand la souris les pointe), etc.

Reprenons l'exemple des séismes des Tonga de la section 3.4.1 et ajoutons des cercles avec une couleur fonction de la profondeur. Pour cela, nous créons donc le fond de carte selon les étapes classiques : création du widget, centrage dans la zone, spécification du zoom d'intérêt et ajout des tuiles du fond de carte :

```
> m <- leaflet(data=quakes) %>%
       setView(lng=176.9, lat=-24.66, zoom=4) %>%
       addTiles()
```

À ce fond de carte à une couche, nous ajoutons les annotations sous forme de cercles (fonction **addCircles**) positionnés avec les variables `long` et `lat` de `quakes` pour la longitude et la latitude respectivement :

```
> m %>% addCircles(~long, ~lat)
```

Si nous souhaitons colorier les cercles en fonction de la profondeur (variable `depth` de `quakes`), il est nécessaire de créer un vecteur de couleurs en RGB. La fonction **colorNumeric** du package leaflet permet de créer facilement un tel vecteur avec une syntaxe proche de celle utilisée dans ggplot2 :

```
> pal <- colorNumeric(palette=c(low="yellow2",high="red4"),
        domain = quakes$depth)
```

Cette fonction **colorNumeric** renvoie comme résultat une fonction (que nous avons nommée ici **pal**) qui prend comme argument une profondeur et qui renvoie un code couleur linéairement réparti entre le minimum et le maximum de la profondeur observée. Ainsi, cette fonction résultat appliquée aux trois premières profondeurs renvoie bien les couleurs en code RGB :

```
> pal(quakes$depth[1:3])
[1] "#A03F01" "#911901" "#EEED00"
```

La carte finale (voir Fig. 3.37) est obtenue en donnant l'argument `color` aux marqueurs de type cercle :

```
> m %>% addTiles() %>%
        addCircles(~long,~lat,popup=~mag,color=~pal(depth))
```

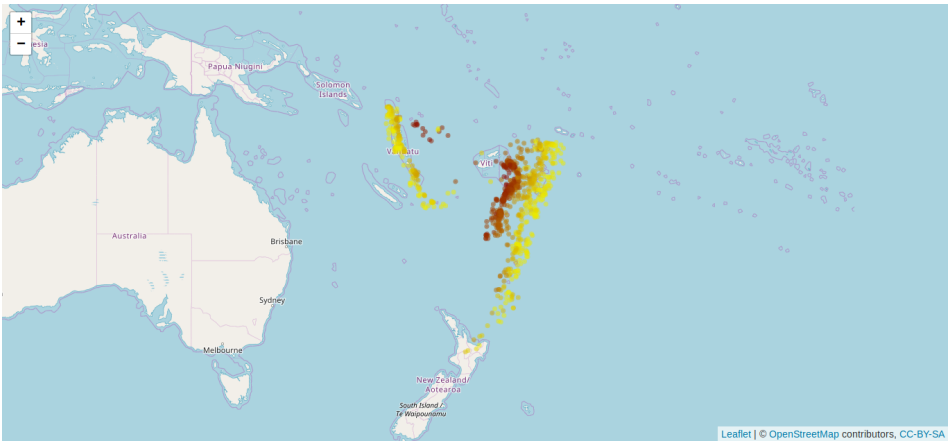


FIGURE 3.37 – Séismes des îles Tonga représentés avec le package leaflet.

Le caractère dynamique proposé par leaflet permet de construire des cartes sur lesquelles des informations pourront être proposées à l'utilisateur lorsqu'il cliquera à certains endroits de la carte. Nous illustrons cette démarche pour proposer une carte permettant de visualiser les stations velib à Paris. Les données utilisées peuvent être récupérées sur `opendataparis` à l'url `https://opendata.paris.fr/explore/dataset/velib-disponibilite-en-temps-reel/`. Nous considérons ici la base contenant les longitudes et latitudes de 68 stations velib ainsi que le nombre de vélos disponibles dans la station le 1er septembre 2018 à 12 h.

```
> sta.paris <- read.csv("velib_paris_01_09_18.csv",encoding = "UTF-8")
> summary(sta.paris)
                          name    numBikesAvailable
 Alexander Fleming - Belvédère: 1   Min.   : 1.000
 Arsonval Falguière           : 1   1st Qu.: 5.750
 Assas - Luxembourg           : 1   Median : 8.000
 Assas - Rennes               : 1   Mean   : 8.985
 Athènes - Clichy             : 1   3rd Qu.:11.000
 Bassano -  Iéna              : 1   Max.   :36.000
 (Other)                      :62

      lon             lat
 Min.   :2.264   Min.   :48.82
 1st Qu.:2.306   1st Qu.:48.85
 Median :2.338   Median :48.86
 Mean   :2.336   Mean   :48.86
 3rd Qu.:2.367   3rd Qu.:48.87
 Max.   :2.411   Max.   :48.90
```

La fonction **addCircleMarkers** permet de localiser les stations de vélos sur une carte `leaflet` en ajoutant un cercle aux longitudes et latitudes des stations de la base `sta.paris` :

```
> leaflet(data = sta.paris) %>%  addTiles() %>%
    addCircleMarkers(~ lon, ~ lat,radius=8,stroke = FALSE,
    fillOpacity = 0.5,color="red")
```

On utilisera l'argument `popup` de **addCircleMarkers** pour ajouter des informations relatives aux stations sur lesquelles l'utilisateur cliquera. On peut par exemple obtenir le nombre de vélos disponibles de la station à l'aide de :

```
> leaflet(data = sta.paris) %>% addTiles() %>%
    addCircleMarkers(~ lon, ~ lat,radius=8,stroke = FALSE,
    fillOpacity = 0.5,color="red",popup = ~ paste("<b>
    Vélos dispos: </b>",as.character(numBikesAvailable)))
```

On peut de la même façon ajouter dans le `popup` le nom de la station pour obtenir la carte 3.38.

```
> leaflet(data = sta.paris) %>%  addTiles() %>%
    addCircleMarkers(~ lon, ~ lat,radius=8,stroke = FALSE,
    fillOpacity = 0.5,color="red",popup = ~ paste(as.character(name),
    "<br/>","<b> Vélos dispos: </b>",as.character(numBikesAvailable)))
```

FIGURE 3.38 – Représentation des stations velib à Paris avec le package leaflet.

D'autres fonctions permettent d'ajouter des popups sur une carte afin de préciser certaines informations. On pourra par exemple utiliser **addPopups** pour localiser le stade vélodrome de Marseille et mettre un lien vers sa page wikipedia avec :

```
> SV <- geocode("Stade Velodrome Marseille")
> info <- paste(sep = "<br/>",
     "<b><a href='https://fr.wikipedia.org/wiki/Stade_Vélodrome'>
     Stade Velodrome</a></b>", "Marseille")
> leaflet() %>% addTiles() %>%  addPopups(SV[1]$lon, SV[2]$lat,
     info,options = popupOptions(closeButton = FALSE))
```

3.4.3 Carte avec contours : le format shapefile

Nous allons maintenant utiliser des fonds de cartes de type « shapefile » [2] qui ne font figurer que les contours (donc des polygones), par exemple ceux des départements français. Ces fonds de cartes permettent de représenter une variable quantitative ou qualitative en coloriant à l'intérieur des contours.

En guise d'exemple, nous analysons graphiquement les différences de taux de chômage par département. Pour cela, nous disposons de chaque taux mesuré aux premiers trimestres des années 2006 et 2011 (variables `TCHOMB1T06`, `TCHOMB1T11`). Ces données sont disponibles en ligne sur le site de l'INSEE. Avant d'importer ces données rappelons que, comme souvent en R, plusieurs solutions s'offrent à nous. Nous allons utiliser ici le package sf qui permet d'importer les données au format

2. Au format ESRI, voir Wikipedia.

shapefile (et d'autres formats si besoin) et de les manipuler. Une autre solution serait d'utiliser les packages rgdal pour l'importation et sp pour la manipulation. Le package sf propose d'utiliser soit le formalisme classique (les data-frames et la fonction **plot**) soit le formalisme issu des packages dplyr et ggplot2. C'est ce dernier que nous privilégions ici.

Nous importons avec le package readr qui permet d'obtenir un tibble :

```
> library(readr)
> chomage <- read_delim("tauxchomage.csv",delim=";")
> class(chomage)
[1] "tbl_df"     "tbl"        "data.frame"
```

Pour le fond de carte, nous nous servons de la carte GEOFLA® proposée par l'Institut Géographique National. Cette carte est disponible sur le site `http://professionnels.ign.fr/` au format shapefile, mais nous mettons à disposition sur le site du livre l'archive simplifiée `dpt.zip`, qu'il faut importer puis décompresser pour reproduire la carte.

Grâce au package sf, cette carte, contenue dans la série de fichiers `departement.*` du répertoire `dpt`, peut être importée dans un objet R :

```
> library(sf)
> dpt <- read_sf("dpt")
> class(dpt)
[1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

L'importation via **read_sf** prend directement en compte le système de coordonnées utilisé par l'IGN (ici Lambert 93) : ce sont des données projetées et mesurées en mètres, cette caractéristique étant stockée dans le fichier texte `departement.prj`. L'objet `dpt` est un objet `sf` qui contient plusieurs éléments : diverses variables mesurées d'un côté, et tout ce qui concerne les coordonnées de mesures de l'autre. Ce dernier aspect est contenu dans la colonne `geometry` qui est un objet (de type liste) de classe `sfc`.

Si nous souhaitons analyser les taux de chômage par département et les représenter sur la carte, nous devons fusionner le tibble `chomage` (qui contient les deux taux de chômage par département) avec les données de `dpt` (qui contiennent la carte). Pour s'assurer que la fusion est faite par département, nous effectuons une jointure :

```
> library(dplyr)
> dpt2 <- inner_join(dpt,chomage,by="CODE_DEPT")
> class(dpt2)
[1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

Enfin, pour utiliser le formalisme de ggplot2, il faut utiliser un jeu de données avec deux colonnes : l'une donnant l'année et l'autre le taux de chômage de l'année. Cette remise en forme est obtenue grâce à la fonction **gather** du package tidyr :

```
> library(tidyr)
> dpt3 <- dpt2 %>% select(A2006=TCHOMB1T06,A2011=TCHOMB1T11,geometry) %>%
      gather("Annee","TxChomage",-geometry)
> class(dpt3)
[1] "sf"          "tbl_df"      "tbl"         "data.frame"
```

La représentation des taux de chômage par département d'une ou de plusieurs années peut alors être obtenue grâce aux fonctions du package ggplot2 (Fig. 3.39) :

```
> library(ggplot2)
> ggplot() + geom_sf(data = dpt3, aes(fill = TxChomage)) +
    facet_wrap(~Annee, nrow = 1) +
    scale_fill_gradient(low="white",high="brown")+theme_bw()
```

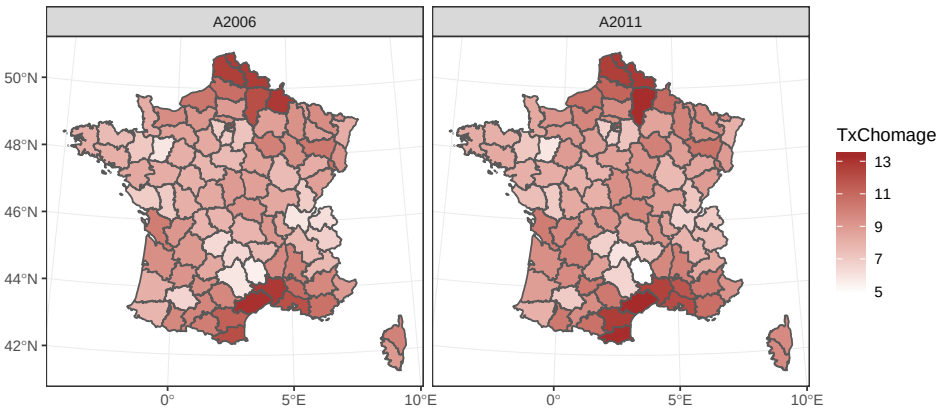


FIGURE 3.39 – Taux de chômage aux premiers trimestres des années 2006 et 2011.

Le package sf permet aussi de représenter une variable qualitative sur un fond de carte (voir exercice 3.10 p. 105).

D'autres packages très complets sur la cartographie existent et nous pouvons citer entre autres les packages tmap et cartography pour le lecteur désireux d'envisager d'autres approches.

Remarques (disponibilité de fonds de cartes)

– Un certain nombre de fonds de cartes de type « shapefile » sont disponibles dans le package maps. Pour obtenir un fond de carte des états des Etats-Unis et le tracer simplement, il suffit d'utiliser la carte `state` :

```
> library(maps)
> map("state")
```

Comme il est plus difficile de manipuler ces objets que ceux présentés précédemment (de type `sfc`), il est préférable de les transformer en objet de type `sfc` par la fonction **st_as_sfc** (voir exercice 3.11 p. 105).

– D'autres fonds de cartes « shp » sont disponibles à partir du site OpenStreetMap : `https://wiki.openstreetmap.org/wiki/Shapefiles#Obtaining_shapefiles_from_OSM_data`. Pour la France, voir l'adresse suivante : `http://wiki.openstreetmap.org/wiki/WikiProject_France/Fonds_de_cartes`.
– Les fonds de cartes de `http://www.naturalearthdata.com/` peuvent être eux aussi utilisés avec R via le package rnaturalearth.
– Enfin, une carte OSM peut être constituée élément par élément (par exemple les bâtiments, puis les autoroutes, puis les parcs, etc.) à l'aide du package osmplotr.

Le package leaflet permet aussi de représenter des polygones, éventuellement sur un fond de carte comme en figure 3.40. Cette méthode ajoute de l'interactivité mais ne permet pas de proposer deux graphiques côte à côte. Pour celles et ceux qui voudraient absolument représenter ces deux cartes côte à côte, il faudra soit utiliser le package shiny pour représenter les cartes leaflet, soit utiliser le package mapview qui propose une sur-couche à leaflet.

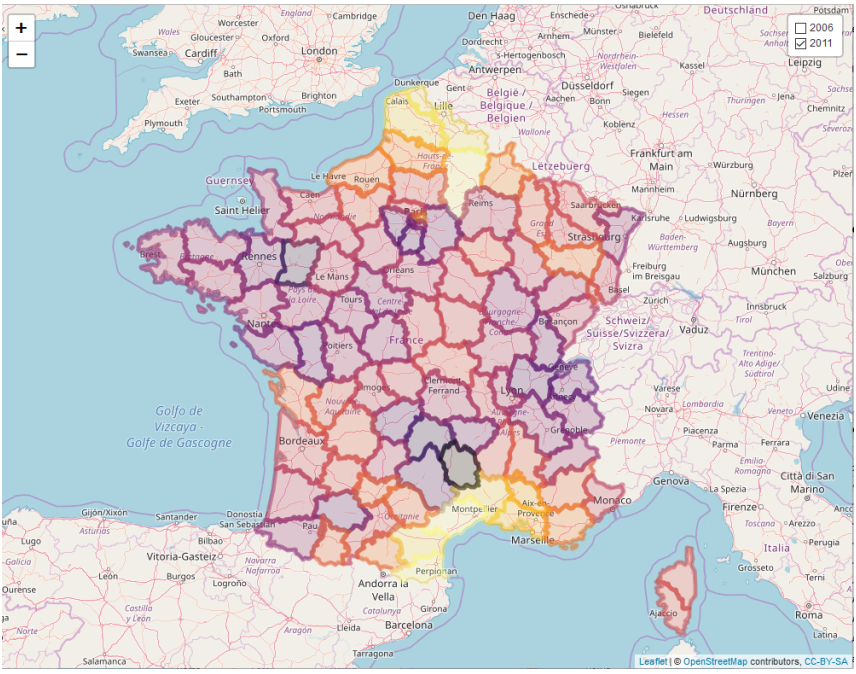


FIGURE 3.40 – Taux de chômage en 2006 et 2011 : polygones shp via leaflet.

Avec le package leaflet, nous pouvons cependant les superposer et proposer un contrôle permettant d'afficher l'une et/ou l'autre année. Pour cela, rappelons que les données d'OpenStreetMap utilisent le système de longitude et latitude mesurées avec l'ellipsoïde WGS84. Il faudra donc mettre les données de l'IGN dans

ce système de projection (qui a un code EPSG numéro 4326). Nous allons donc transformer l'objet `dpt2` en objet `dp` de classe `sf` dans le système correct :

```
dp <- st_transform(dpt2, crs="+init=epsg:4326")
```

Ensuite nous pouvons créer le fond de carte avec une première couche de tuiles centrées sur la France :

```
> library(leaflet)
> fr <- leaflet(data=dp) %>% setView(2.21,46.23,6) %>% addTiles()
```

À cet objet nous ajoutons les polygones des départements colorés selon le chômage de l'année 2006. Pour cela, nous créons la fonction permettant de renvoyer les codes couleurs (ici selon la palette `inferno` de viridis) selon l'étendue des chômages des années 2006 et 2011 :

```
> pal <- colorNumeric(palette="inferno",domain =
    c(dp$TCHOMB1T06,dp$TCHOMB1T11))
```

La première couche de polygones est rassemblée dans le groupe nommé `2006` via l'argument `group` et nous faisons de même avec l'année 2011 :

```
> frf <- fr %>% addPolygons(color=~pal(TCHOMB1T06),group ="2006") %>%
    addPolygons(color=~pal(TCHOMB1T11),group="2011")
```

Enfin, pour aboutir à la figure 3.40, nous contrôlons l'affichage des couches par les lignes suivantes :

```
> frf %>% addLayersControl(overlayGroups = c("2006", "2011"), options=
    layersControlOptions(collapsed = FALSE))
```

La fonction **layersControlOptions** permet de contrôler finement l'interface : l'option `collapsed` permettant de faire apparaître l'interface (`FALSE`) ou simplement une icône qui dévoile l'interface quand le pointeur de la souris est dessus (`TRUE`).

3.5 Exercices

Exercice 3.1 (Tracé d'une fonction)

1. Tracer la fonction sinus entre 0 et 2π (utiliser `pi`).
2. Ajouter le titre suivant (**title**) : `Graphe de la fonction sinus`.

Exercice 3.2 (Comparaison de distributions)

1. Tracer la courbe de la loi normale centrée réduite entre -4 et 4 (utiliser **dnorm**).

2. Tracer sur le même graphe les lois de Student à 5 et 30 degrés de liberté. Utiliser la fonction **curve** et une couleur différente pour chaque courbe.

3. Ajouter une légende en haut à gauche pour préciser chaque distribution.

Exercice 3.3 (Tracé de points)

1. Importer le tableau `ozone` et tracer le nuage de points du maximum d'ozone `maxO3` en fonction de la température `T12`.

2. Tracer le nuage de points `maxO3` en fonction de `T12` avec des lignes reliant les points.

3. En utilisant **order**, tracer le graphique 3.41.

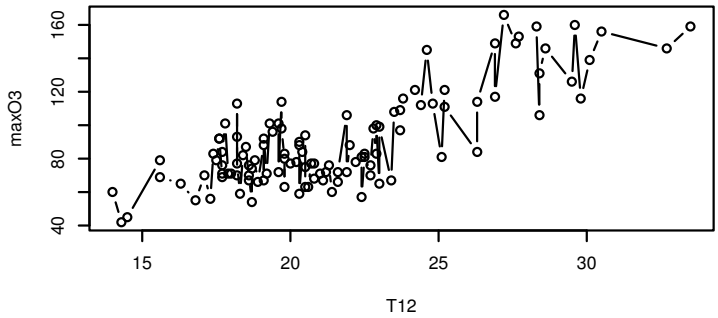


FIGURE 3.41 – Nuage de points `maxO3` en fonction de `T12`.

Exercice 3.4 (Loi des grands nombres)

1. Après avoir fixé la graine du générateur aléatoire à 123 (**set.seed**), simuler un échantillon $(x_1, ..., x_{1000})$ de longueur 1000 provenant d'une loi de Bernoulli de paramètre $p = 0.6$.

2. Calculer les moyennes successives $M_l = S_l/l$ où $S_l = \sum_{i=1}^{l} X_i$. Tracer M_l en fonction de l puis ajouter la droite horizontale d'équation $y = 0.6$.

Exercice 3.5 (Théorème central limite)

1. Soit X_1, X_2, ... X_N i.i.d. suivant une loi de Bernoulli de paramètre p. Rappeler la loi suivie par $S_N = X_1 + \ldots + X_N$. Donner sa moyenne et son écart-type.

2. On fixe $p = 0.5$. Pour $N = 10$, simuler, grâce à la fonction **rbinom**, $n = 1000$ réalisations $S_1, \ldots, S_{1000}$ d'une loi binomiale de paramètres N et p. Ranger dans un vecteur U10 les quantités $\frac{S_i - N \times p}{\sqrt{N \times p \times (1-p)}}$. Faire de même avec $N = 30$ et $N = 1000$ pour obtenir deux nouveaux vecteurs U30 et U1000.

3. Représenter sur une même fenêtre (**par**`(mfrow)`) les histogrammes de U10, U30 et U1000 en superposant à chaque fois la densité (**density**) de la loi normale centrée réduite obtenue par **dnorm**.

Exercice 3.6 (Tracé des taches solaires)

1. Importer la série `taches_solaires_date.csv` qui donne, date par date, le nombre relatif de taches solaires. Utiliser l'argument `colClasses` afin de spécifier que le nombre relatif de taches solaires est de type `numeric` et que la date est au format `Date`. Vérifier le type des variables à l'issue de l'importation.

2. Créer une variable qualitative `trenteans` égale à 1 pour la première année (1749) et qui augmente de 1 tous les trente ans. Pour cela, utiliser la fonction **cut** pour les dates (voir l'aide de cette fonction et l'argument `breaks`). Changer ensuite les intitulés des modalités en $1, 2, \ldots$ avec **levels** et **nlevels**.

3. Saisir le vecteur `couleur` qui contient les couleurs suivantes : green, yellow, magenta, orange, cyan, grey, red, green et blue. Vérifier automatiquement que ces couleurs sont bien contenues dans le vecteur **colors**`()` (instructions `%in%` et **all**).

4. Tracer la série chronologique comme en figure 3.42. Utiliser les fonctions **palette**, **plot**, **lines** et une boucle (voir aussi **unique**).

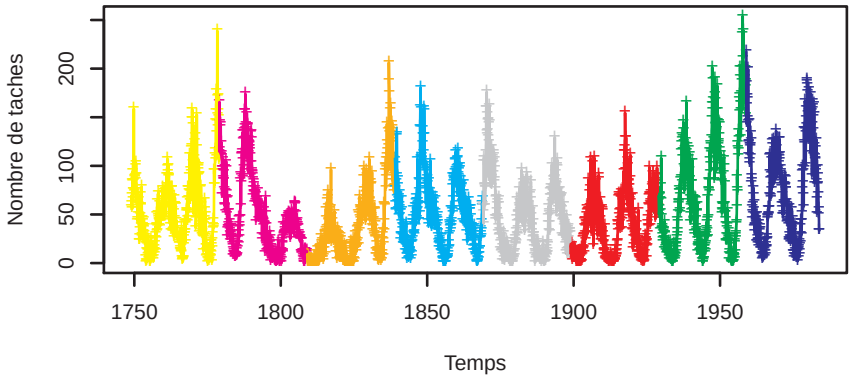


FIGURE 3.42 – Taches solaires en fonction de la date d'observation.

Exercice 3.7 (Tracé d'une densité)

1. Tracer la densité de la variable aléatoire $X \sim \mathcal{N}(0, 1)$ (voir **dnorm**).

2. Ajouter l'axe des abscisses (voir **abline**).

3. Colorier en bleu l'aire sous la courbe à droite de q correspondant à la probabilité de 5 % (**polygon**).

4. Ajouter une flèche désignant l'aire coloriée (**arrows**).

5. Ajouter enfin $\alpha = 5$ % (**text** et **expression**) pour obtenir la figure 3.43.

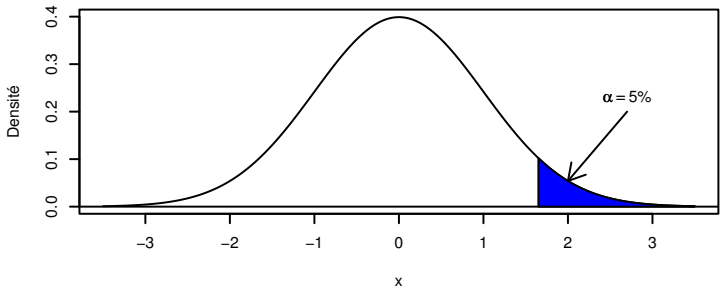


FIGURE 3.43 – Densité d'une loi normale et quantile d'ordre 95 %.

Exercice 3.8 (Plusieurs graphiques)

1. Reproduire le graphique de la figure 3.44.

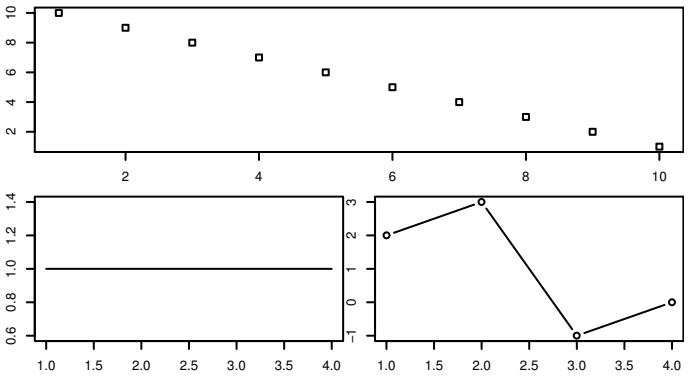


FIGURE 3.44 – Trois graphiques sur deux lignes.

2. Reproduire la même figure mais avec un graphique en bas à droite d'une largeur de 1 pour 4 par rapport à celui en bas à gauche (voir les arguments de **layout**).

Exercice 3.9 (Nombre d'étudiants par ville universitaire)

1. Importer `villes.csv` dans l'objet `villes` de classe data-frame et le résumer.

2. Créer le vecteur `decoupe` contenant les valeurs 0, 15000, 25000, 50000, 75000, 100000, **max**`(Nb.etudiant)`.

3. Découper en 6 classes la variable quantitative `Nb.etudiant` en utilisant le vecteur `decoupe` ci-dessus. La variable qualitative en résultant sera appelée `Xq` (**cut**).

4. Affecter aux modalités de `Xq` les valeurs formatées 0-15000, 15000-25000, 25000-50000, 50000-75000, 75000-100000, 100000-531904 en utilisant le vecteur `decoupe` (**format** et **levels**).

5. Trouver la « bounding box » de la carte avec **make_bbox**.

6. Ajouter la variable qualitative `Xq` au data-frame `villes`.

7. Télécharger la carte GoogleMap (**get_map**).

8. Tracer sur le fond de carte un point pour chaque ville universitaire (**ggmap**, **geom_point**). Ces points auront :
- une taille (argument `size`) fonction du nombre d'étudiants. Cette fonction est le logarithme népérien du nombre d'étudiants divisé par 5000 ;
- une couleur (argument `color`) fonction de la variable qualitative `Xq`.

Exercice 3.10 (Chômage et élection régionale)

1. Utiliser l'archive `regions-metropole-complet.tar.gz` (téléchargeable aussi via la page `http://wiki.openstreetmap.org/wiki/WikiProject_France/Fonds_de_cartes`) pour importer le fond de carte dans un objet R appelé `regions` (**read_sf**).

2. Tracer les régions avec la fonction **plot**.

3. Vérifier en utilisant **summary** que l'objet `regions` utilise le bon système de coordonnées (mesurées avec la norme WGS84, code EPSG 4326, ce qui est le cas sur OpenStreetMap ou GoogleMap) et vérifier sa classe.

4. Trouver le nom R de la variable de l'intitulé des régions et vérifier que son contenu est bien donné par `regions$NOM`.

5. Tracer la région 1 (avec la variable 1) en utilisant **plot**`(regions[1,1])`.

6. Trouver le nom de cette région en utilisant la question 4.

7. Tracer la région `Alsace` en utilisant le formalisme dplyr et ggplot2 (**filter**).

8. Charger les données de chômage 2011 (par région) contenues dans le fichier `txchom_region.csv` dans un objet R appelé `chomregion` (**read_delim**). La variable `CODGEO` est le code officiel de la région (que l'on retrouve dans la variable `NUMERO` de `regions`). Changer le nom de cette variable `CODGEO` en `NUMERO` et la transformer en caractère (**mutate**, **as.character**).

9. Grâce à **inner_join**, fusionner le tableau `chomregion` avec `regions` selon la variable `NUMERO`.

10. À l'aide des fonctions **ggplot** et **geom_sf**, représenter le chômage du premier trimestre 2011 par région.

11. Créer une variable de type facteur nommée `majorite` pour représenter les majorités régionales aux élections de 2010 (**factor**) puis la représenter avec comme couleurs `pink2` pour les divers gauche, `salmon` pour le parti radical de gauche, `pink` pour le PS et `blue` pour l'UMP (**scale_fill_manual**).

Exercice 3.11 (Représentation graphique et projection)

1. Tracer la carte des états des Etats-Unis en utilisant le package maps.

2. Affecter la carte des états des Etats-Unis dans un objet `usa` (utiliser l'option `plot=FALSE,fill=TRUE` de la fonction **map**).

3. Transformer la carte en `sfc` grâce à la fonction **st_as_sfc** du package `sf`.

4. Tracer cette carte et constater que la représentation est identique.

5. Projeter la carte dans le système de coordonnées EPSG :102008 (**st_transform**). Tracer la nouvelle carte projetée et constater la différence visuelle. Cette projection (North America Albers Equal Area Conic) conserve les surfaces.

Exercice 3.12 (Représentation de tuiles)

1. Avec la projection de Mercator [3], au zoom de 13, trouver les coordonnées `xx` et `yy` du point de longitude-latitude 6.616 et 45.852.

2. Arrondir les valeurs de `xx` et `yy` à l'entier inférieur et affecter ces deux valeurs dans `xtile` et `ytile`.

3. Télécharger le fichier au format png sur le serveur de tuile OSM à l'adresse suivante : `http://b.tile.openstreetmap.org/zoom/x/y.png` où x est remplacé par la valeur de `xtile`, y par `ytile` et z par la valeur du zoom (donc 13). Importer ce fichier avec la fonction **readPNG** du package `png` dans un array de dimension $256 \times 256 \times 3$ et transformer l'objet importé en raster (**as.raster**).

4. La matrice obtenue est ensuite transformée en objet de la classe `ggmap` et `raster`. Ajouter les attributs `source` (valeur `"OSM"`), `maptype` (valeur `"terrain"`) et `zoom` (valeur 13).

5. Ajouter le dernier attribut `bb` qui est un data-frame à 1 ligne et 4 colonnes de nom `ll.lat`, `ll.lon`, `ur.lat`, `ur.lon` qui contiennent les latitudes et longitudes des points en bas à gauche et en haut à droite (utiliser les fonctions **make_bbox** et **expand.grid**).

6. Tracer la tuile avec `ggmap`.

7. Faire la même chose en prenant deux couronnes de tuiles (l'une de 16, l'autre de 8) autour de la tuile contenant le point initial.

Exercice 3.13 (Représentations graphiques simples avec ggplot2)

Tous les graphes devront être réalisés avec le package ggplot2. On considère le jeu de données `mtcars` du package datasets.

1. Tracer l'histogramme de la variable `mpg`.

2. Tracer le diagramme en barres de la variable `cyl`.

3. Représenter le nuage de points `disp`×`mpg` en fonction des valeurs de la variable `cyl`. On représentera un nuage de points pour chaque valeur de `cyl`.

Exercice 3.14 (Représentation d'une courbe avec ggplot2)

On considère une variable discrète X dont la loi est donnée par

$$\mathbb{P}(X = \text{red}) = 0.3,\ \mathbb{P}(X = \text{blue}) = 0.2,\ \mathbb{P}(X = \text{green}) = 0.4,\ \mathbb{P}(X = \text{black}) = 0.1.$$

Représenter le diagramme en barres associé à cette distribution.

3. Voir `https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames` ou Wikipedia.

Exercice 3.15 (Représentation graphique avec ggplot2)

1. Utiliser le package ggplot2 pour tracer la fonction `sinus` sur l'intervalle $[-2\pi, 2\pi]$.

2. Ajouter en bleu épais les droites d'équations $y = -1$ et $y = 1$.

3. Ajouter la fonction `cosinus`.

4. Faire le même graphe avec une légende permettant d'identifier cosinus et sinus.

5. Représenter cosinus et sinus sur deux graphes séparés (**facet_wrap**).

Exercice 3.16 (Simulation et représentation graphique avec ggplot2)

1. Générer un échantillon $(x_i, y_i), i = 1, \ldots, 100$, selon le modèle de régression

$$Y_i = 3 + X_i + \varepsilon_i$$

où les X_i sont i.i.d. de loi uniforme sur $[0, 1]$ et les ε_i sont i.i.d. de loi $\mathcal{N}(0, 0.2^2)$.

2. Représenter le nuage de points ainsi que la droite des moindres carrés à l'aide de la fonction **geom_abline** puis de la fonction **geom_smooth**.

3. Représenter les résidus : on ajoutera un trait vertical reliant chaque point à la droite des moindres carrés.

Exercice 3.17 (Habillage selon une variable qualitative avec ggplot2)

On travaille sur le jeu de données `states` du package datasets construit selon

```
> data(state)
> states <- data.frame(state.x77,state.name=rownames(state.x77),
    state.region=state.region)
```

1. Créer une variable `revenu1` qui prend pour valeur `faible` si le revenu est dans le premier tercile, `moyen` s'il est dans le deuxième et `fort` s'il est dans le troisième (on pourra utiliser les fonctions **quantile** et **cut**).

2. Représenter à l'aide du package ggplot2 le nuage de points `Population vs Murder` pour chaque valeur de `revenu1` (il faut donc 3 nuages de points).

3. Mettre une couleur différente pour chaque point selon la variable `state.region` et ajouter la droite des moindres carrés sur chaque graphe.

Exercice 3.18 (Habillage selon une variable quantitative avec ggplot2)

On considère le jeu de données `Ozone` du package `mlbench`. Les graphiques demandés seront effectués avec `ggplot2`.

1. Charger le jeu de données et consulter l'aide.

2. Créer une variable `date` qui contient la date au format `date` (1976-01-01). On pourra utiliser **as.Date**.

3. Représenter la série contenant la concentration en ozone (variable `V4`) en fonction de la date (on appellera l'axe des ordonnées `Concentration en` O_3).

4. Représenter le nuage de points correspondant à la concentration en Ozone contre la température.

5. Créer la variable `mois` qui contient le mois de la mesure au format jan, fev, etc. (on pourra utiliser la fonction **format**).

6. Représenter les boxplots de la concentration en ozone en fonction du mois. La représentation devra tenir compte de l'aspect temporel (janvier avant février, etc.).

7. Discrétiser la variable `vent` en trois classes (selon les terciles).

8. Représenter pour chaque mois, le nuage de points correspondant à la concentration en ozone contre la température. On utilisera une couleur différente pour les points en fonction de la force du vent calculée à la question précédente.

Chapitre 4

Programmer

La programmation en R est basée sur les mêmes principes que pour d'autres logiciels de calcul scientifique. En effet, on retrouve à la fois les structures classiques de la programmation (boucles, condition `if else`, etc.) et des fonctions prédéfinies propres à la pratique statistique.

4.1 Structures de contrôle

4.1.1 Commandes groupées

Un groupe de commandes est comme une parenthèse en mathématiques : les commandes groupées sont effectuées ensemble. Sous R, le groupe de commandes est délimité par des accolades :

```
> {
    expr1
    expr2
    ...
 }
```

Deux commandes successives sont séparées par un retour à la ligne (touche Entrée). Cependant, il est possible de les séparer par un point-virgule en les conservant sur une même ligne. Ainsi l'exemple ci-dessus peut aussi s'écrire :

```
> { expr1 ; expr2 ; ... }
```

4.1.2 Les boucles (for ou while)

Les boucles classiques sont disponibles sous R. Commençons par la boucle `for`. Nous souhaitons afficher tous les entiers de 1 à 99 à l'écran. Une solution est la suivante :

```
> for (i in 1:99) print(i)
```

L'indice `i` prend comme valeurs toutes les coordonnées du vecteur choisi. Si nous souhaitons balayer les entiers de deux en deux, il suffit de constituer un vecteur qui démarre à 1 et qui va jusqu'à 99, de deux en deux. La fonction **seq** nous permet de constituer un tel vecteur. La boucle devient :

```
> for (i in seq(1,99,by=2)) print(i)
```

Cette méthode se généralise simplement à un vecteur quelconque. Ainsi, si nous choisissons un vecteur de caractères représentant les 3 premiers jours de la semaine, nous avons

```
> vecteur <- c("lundi","mardi","mercredi")
> for (i in vecteur) print(i)
[1] "lundi"
[1] "mardi"
[1] "mercredi"
```

Souvent, nous avons plusieurs ordres à effectuer à chaque itération et il est donc nécessaire de grouper les commandes. De manière générale, la boucle `for` s'écrit :

```
> for (i in vecteur) {
    expr1
    expr2
    ...
 }
```

Une autre possibilité de boucle est la condition `while`. Sa syntaxe est la suivante :

```
> while (condition) {
    expr1
    expr2
    ...
 }
```

Les ordres `expr1`, `expr2`, etc., sont effectués tant que la condition est vraie, celle-ci étant évaluée en début de boucle. Dès que la condition est fausse, la boucle est arrêtée. Ainsi,

```
> i <- 1
> while (i<3) {
   print(i)
   i <- i+1 }
[1] 1
[1] 2
```

permet d'afficher `i` et de l'augmenter de 1 tant que `i` est inférieur à 3.

Une dernière possibilité de boucle est l'ordre `repeat`. Il se comprend comme : répéter indéfiniment des ordres. La sortie de boucle est assurée par l'ordre `break`. Cet ordre peut être utilisé quelle que soit la boucle. Un exemple est donné dans le paragraphe suivant.

4.1.3 Les conditions (if, else)

Il s'agit d'exécuter un ordre sous condition : l'ordre est exécuté si et seulement si la condition est vérifiée. Dans sa forme simple il s'écrit :

```
> if (condition) {
   expr1
   expr2
   ...
 }
```

Par exemple, si nous souhaitons utiliser une boucle `repeat` pour imprimer `i` variant de 1 à 3 compris, nous devons sortir de la boucle, avant l'impression, quand `i` est supérieur à 3 :

```
> i <- 1
> repeat {
   print(i)
   i <- i+1
   if (i>3) break }
```

Ici, si `i` est supérieur à 3, nous n'avons à effectuer qu'un ordre (`break`), il n'est donc pas utile de grouper les commandes (on peut omettre les accolades).

Une autre condition peut être ajoutée à la suite du `if`, la condition `else`, qui permet de séparer les deux cas : soit la condition est vraie alors l'ordre (ou le groupement d'ordres) après `if` est exécuté, soit elle n'est pas vraie, alors l'ordre (ou le groupement d'ordres) après `else` est exécuté. Sous sa forme générale la condition (`if, else`) s'écrit :

```
> if (condition) {
    expr1
    expr2
    ...
 } else {
    expr3
    ...
 }
```

Attention, l'ordre `else` doit être sur la même ligne que la parenthèse fermante « `}` » de la clause `if` (comme ci-dessus).

S'il y a plus de deux conditions, on pourra utiliser la fonction **switch** pour associer une suite d'instructions à chaque condition. Par exemple :

```
> X <- matrix(0,5,5)
> switch(class(X),
       "matrix" = print("X est une matrice"),
       "data.frame" = print("X est un data.frame"),
       "numeric" = print("X est de classe numérique")
)
```

4.2 Construire une fonction

Une fonction permet d'effectuer un certain nombre d'ordres R. Ces ordres peuvent dépendre d'arguments fournis en entrée, mais cela n'est pas obligatoire. La fonction fournit un objet « résultat » unique (voir Fig. 4.1). Cet objet « résultat » est désigné à l'intérieur de la fonction par la fonction **return**. Par défaut, si la fonction écrite ne renvoit pas de résultat, le dernier résultat obtenu avant la sortie de la fonction est renvoyé comme résultat.

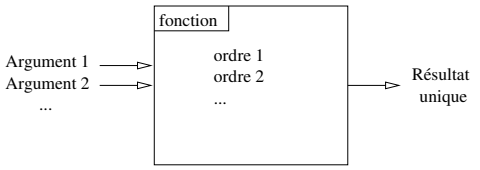


FIGURE 4.1 – Diagramme d'une fonction R.

Commençons par un exemple simple, la somme des n premiers entiers. Le nombre n est un entier qui est l'argument d'entrée, le résultat est simplement la somme demandée :

```
som <- function(n) {
    resultat <- sum(1:n)
    return(resultat)
}
```

La fonction est appelée grâce à son nom (**som**), suivi des arguments en entrée entre parenthèses. Ici, la fonction possède un argument en entrée, elle est donc appelée simplement par :

```
> som(3)
[1] 6
```

Il est possible d'affecter le résultat d'une fonction à un objet R grâce à :

```
> res <- som(3)
> res
[1] 6
```

Nous pouvons améliorer cette fonction de la façon suivante :

```
som <- function(n) {
  if (n<=0) stop("l'entier doit être strictement positif")
  if (ceiling(n)!=n) warning(paste("arrondi de",n,"en",ceiling(n)))
  resultat <- sum(1:ceiling(n))
  return(resultat)
}
```

Bien entendu, lorsque n n'est pas strictement positif, la somme n'a aucun sens. La fonction s'arrête (**stop**) en affichant un message d'erreur. D'autre part, quand n est positif mais non entier, la fonction arrondit n à l'entier immédiatement supérieur (**ceiling**) et renvoie un message indiquant que n a été remplacé. La fonction est appelée selon le même schéma :

```
> som(4.325)
[1] 15
Warning message:
In som(4.325) : arrondi de 4.325 en 5
```

Proposons maintenant une fonction avec deux arguments en entrée : `facteur1` et `facteur2`, deux variables qualitatives. Cette fonction renvoie le tableau de contingence ainsi que le vecteur de caractères des niveaux de `facteur1` et `facteur2` pour lesquels l'effectif est conjointement nul. Nous avons donc plus d'un résultat à renvoyer. Comme l'un est un tableau et l'autre un vecteur de caractères (ou une matrice de caractères), ces deux objets ne peuvent être rassemblés ni dans une matrice (pas le même type), ni dans un data-frame (pas la même longueur). Le résultat unique sera donc une liste rassemblant ces deux résultats.

La fonction va ainsi calculer le tableau de contingence (**table**) puis repérer les cellules nulles. Il faut ensuite connaître les indices correspondant aux cellules nulles du tableau de contingence (fonction **which**, option `arr.ind=TRUE`) et récupérer les noms des modalités correspondantes :

```
> mafonc <- function(facteur1,facteur2) {
   res1 <- table(facteur1,facteur2)
   selection <- which(res1==0,arr.ind = TRUE)
   res2 <- matrix("",nrow=nrow(selection),ncol=2)
   res2[,1] <- levels(facteur1)[selection[,1]]
   res2[,2] <- levels(facteur2)[selection[,2]]
   return(list(tab=res1,niveau=res2))
 }
```

Si l'on appelle la fonction avec les facteurs `laine` et `tension` définis par :

```
> tension <- factor(c(rep("Faible",5),rep("Forte",5)))
> laine <- factor(c(rep("Mer",3),rep("Ang",3),rep("Tex",4)))
```

cela donne :

```
> mafonc(tension,laine)
$tab
        facteur2
facteur1 Ang Mer Tex
  Faible   2   3   0
  Forte    1   0   4

$niveau
     [,1]     [,2]
[1,] "Forte"  "Mer"
[2,] "Faible" "Tex"
```

Pour voir comment utiliser une fonction, on peut se reporter au § 1.6.

4.3 La famille apply, des fonctions d'itération prédéfinies

Certaines fonctions ont été prédéfinies dans R pour éviter de recourir à des boucles généralement coûteuses en temps de calcul. La plus utilisée est certainement la fonction **apply**, qui permet d'appliquer une même fonction à toutes les marges d'un tableau. Considérons le tableau `X` constitué de 20 nombres entiers tirés au hasard entre 1 et 20 (**sample**). On calcule la moyenne (`FUN=mean`) par colonne (`MARGIN=2`) comme suit :

```
> set.seed(1234)
> X <- matrix(sample(1:20,20),ncol=4)
> X
     [,1] [,2] [,3] [,4]
[1,]    3   10    7    9
[2,]   12    1    5   17
[3,]   11    4   20   16
[4,]   18    8   15   19
[5,]   14    6    2   13
> apply(X,MARGIN=2,FUN=mean)
[1] 11.6  5.8  9.8 14.8
```

Il est également possible d'ajouter des arguments supplémentaires à la fonction qui est appliquée sur chaque colonne. Par exemple, s'il y a une donnée manquante

dans `X`, il sera intéressant d'utiliser l'argument `na.rm=TRUE` de la fonction **mean**, ce qui permet de calculer la moyenne uniquement sur les données présentes :

```
> X[1,1] <- NA
> apply(X,MARGIN=2,FUN=mean)
[1]   NA  5.8  9.8 14.8
> apply(X,MARGIN=2,FUN=mean,na.rm=TRUE)
[1] 13.75  5.80  9.80 14.80
```

Vu l'utilisation fréquente en statistique des moyennes par colonne (ou par ligne), il existe un raccourci sous la forme d'une fonction **colMeans** (ou **rowMeans**) :

```
> colMeans(X,na.rm=TRUE)
[1] 13.75  5.80  9.80 14.80
```

De même, il est possible d'effectuer une somme par colonne (ou par ligne) directement à l'aide de **colSums** (ou **rowSums**).

Si le tableau de données a trois dimensions (cube de données), il est possible d'exécuter la fonction par ligne (`MARGIN=1`), par colonne (`MARGIN=2`) ou par profondeur (`MARGIN=3`), mais également d'exécuter la fonction pour des croisements ligne-colonne (`MARGIN=c(1,2)`) ou ligne-profondeur (`MARGIN=c(1,3)`) ou colonne-profondeur (`MARGIN=c(2,3)`). Calculons par exemple la somme d'un tableau à trois entrées par couple ligne-colonne :

```
> set.seed(1234)
> Y <- array(sample(24),dim=c(3,4,2))
> Y
, , 1
     [,1] [,2] [,3] [,4]
[1,]    3   22    1    8
[2,]   15   18    4   10
[3,]   14   13   11   23

, , 2
     [,1] [,2] [,3] [,4]
[1,]   17   19    7   20
[2,]   16   21    6    9
[3,]   24    2    5   12
> apply(Y,MARGIN=c(1,2),FUN=sum,na.rm=TRUE)
     [,1] [,2] [,3] [,4]
[1,]   20   41    8   28
[2,]   31   39   10   19
[3,]   38   15   16   35
```

Nous avons utilisé ici des fonctions de R, **mean** et **sum**, mais on peut tout à fait utiliser des fonctions que nous avons préalablement programmées. Par exemple :

```
> MaFonction <- function(x,y) {
   z <- x**2 - y
   return(z)
 }
> set.seed(1234)
> X <- matrix(sample(12),ncol=4)
> X
     [,1] [,2] [,3] [,4]
[1,]    2    6    1    8
[2,]    7   10   12    4
[3,]   11    5    3    9
> apply(X,MARGIN=c(1,2),FUN=MaFonction, y=2)
     [,1] [,2] [,3] [,4]
[1,]    2   34   -1   62
[2,]   47   98  142   14
[3,]  119   23    7   79
```

De nombreuses fonctions sont basées sur le même principe que la fonction **apply**. Par exemple, la fonction **tapply** applique une même fonction non plus aux marges d'un tableau mais à chaque niveau d'un facteur ou combinaison de facteurs :

```
> Z <- 1:5
> Z
[1] 1 2 3 4 5
> vec1 <- c(rep("A1",2),rep("A2",2),rep("A3",1))
> vec1
[1] "A1" "A1" "A2" "A2" "A3"
> vec2 <- c(rep("B1",3),rep("B2",2))
> vec2
[1] "B1" "B1" "B1" "B2" "B2"
> tapply(Z,vec1,sum)
A1 A2 A3
 3  7  5
> tapply(Z,list(vec1,vec2),sum)
   B1 B2
A1  3 NA
A2  3  4
A3 NA  5
```

Les fonctions **lapply** et **sapply** appliquent une même fonction à chaque élément d'une liste. La différence entre ces deux fonctions est la suivante : la fonction **lapply** retourne par défaut une liste, tandis que la fonction **sapply** retourne une matrice ou un vecteur. Créons une liste contenant deux matrices puis calculons la moyenne de chaque élément de la liste, ici chaque matrice :

```
> set.seed(545)
> mat1 <- matrix(sample(12),ncol=4)
> mat1
     [,1] [,2] [,3] [,4]
[1,]    9    4    1   11
[2,]   10    2    3    6
[3,]    7    5    8   12
> mat2 <- matrix(sample(4),ncol=2)
> mat2
     [,1] [,2]
[1,]    4    3
[2,]    2    1
> liste <- list(matrice1=mat1,matrice2=mat2)
> lapply(liste,mean)
$matrice1
[1] 6.5

$matrice2
[1] 2.5
```

Il est même possible de calculer la somme par colonne de chaque élément de la liste en utilisant la fonction **apply** comme fonction `FUN` de la fonction **lapply** :

```
> lapply(liste,apply,2,sum,na.rm=T)
$matrice1
[1] 26 11 12 29

$matrice2
[1] 6 4
```

Un data-frame étant une liste, on pourra utiliser **lapply** pour répéter un calcul sur toutes les colonnes du data-frame.

La fonction **aggregate** travaille sur des data-frames. Elle sépare les données en sous-groupes, définis à partir d'un vecteur, et calcule une statistique sur l'ensemble des variables du data-frame pour chaque sous-groupe. Reprenons les données que nous avons précédemment générées et créons un data-frame avec deux variables `Z` et `T` :

```
> Z <- 1:5
> T <- 5:1
> vec1 <- c(rep("A1",2),rep("A2",2),rep("A3",1))
> vec2 <- c(rep("B1",3),rep("B2",2))
> df <- data.frame(Z,T,vec1,vec2)
> df
  Z T vec1 vec2
1 1 5   A1   B1
2 2 4   A1   B1
```

```
3 3 3   A2   B1
4 4 2   A2   B2
5 5 1   A3   B2
> aggregate(df[,1:2],list(FacteurA=vec1),sum)
  FacteurA Z T
1       A1 3 9
2       A2 7 5
3       A3 5 1
> aggregate(df[,1:2],list(FacteurA=vec1,FacteurB=vec2),sum)
  FacteurA FacteurB Z T
1       A1       B1 3 9
2       A2       B1 3 3
3       A2       B2 4 2
4       A3       B2 5 1
```

La fonction **sweep** permet d'appliquer une même procédure à toutes les marges d'un tableau. Par exemple, si on veut centrer puis réduire les colonnes d'une matrice `X`, on écrit :

```
> set.seed(1234)
> X <- matrix(sample(12),nrow=3)
> X
     [,1] [,2] [,3] [,4]
[1,]    2    6    1    8
[2,]    7   10   12    4
[3,]   11    5    3    9
> mean.X <- apply(X,2,mean)
> mean.X
[1] 6.666667 7.000000 5.333333 7.000000
> sd.X <- apply(X,2,sd)
> sd.X
[1] 4.509250 2.645751 5.859465 2.645751
> Xc <- sweep(X,2,mean.X,FUN="-")
> Xc
           [,1] [,2]      [,3] [,4]
[1,] -4.6666667   -1 -4.333333    1
[2,]  0.3333333    3  6.666667   -3
[3,]  4.3333333   -2 -2.333333    2
> Xcr <- sweep(Xc,2,sd.X,FUN="/")
> Xcr
            [,1]       [,2]       [,3]       [,4]
[1,] -1.03490978 -0.3779645 -0.7395442  0.3779645
[2,]  0.07392213  1.1338934  1.1377602 -1.1338934
[3,]  0.96098765 -0.7559289 -0.3982161  0.7559289
```

Notons que pour centrer et réduire le tableau `X`, on peut plus simplement utiliser la fonction **scale**. La fonction **by**, quant à elle, permet d'appliquer une même fonction

à un data-frame pour les différents niveaux d'un facteur ou d'une liste de facteurs. Cette fonction est donc similaire à la fonction **tapply** si ce n'est qu'elle travaille sur un data-frame plutôt qu'un vecteur. Générons quelques données :

```
> set.seed(1234)
> T <- rnorm(50)
> Z <- rnorm(50)+3*T+5
> vec1 <- c(rep("A1",20),rep("A2",30))
> don <- data.frame(Z,T)
```

On peut alors obtenir un résumé de chaque variable pour chaque modalité du facteur `vec1` :

```
> by(don,list(FacteurA=vec1),summary)
FacteurA: A1
       Z                T
 Min.   :-3.052   Min.   :-2.3457
 1st Qu.: 2.624   1st Qu.:-0.8504
 Median : 4.539   Median :-0.5288
 Mean   : 4.417   Mean   :-0.2507
 3rd Qu.: 6.205   3rd Qu.: 0.3154
 Max.   :12.584   Max.   : 2.4158
-----------------------------------
FacteurA: A2
       Z                 T
 Min.   :-0.9901   Min.   :-2.1800
 1st Qu.: 1.6098   1st Qu.:-1.0574
 Median : 3.1922   Median :-0.6088
 Mean   : 3.3560   Mean   :-0.5880
 3rd Qu.: 5.1170   3rd Qu.:-0.3309
 Max.   : 9.2953   Max.   : 1.4495
```

Attention, si on demande la somme, on a la somme pour l'ensemble des variables pour chaque modalité du facteur `vec1` et non la somme variable par variable pour chaque modalité de `vec1` :

```
> by(don,list(FacteurA=vec1),sum)
FacteurA: A1
[1] 83.32415
--------------
FacteurA: A2
[1] 83.04172
```

Il est aussi possible d'automatiser des calculs beaucoup plus compliqués. Par exemple, si on veut faire une régression pour chaque modalité de la variable `vec1`, on peut demander à **by** de répéter une fonction que nous définissons nous-mêmes (§ 4.2) et qui est ici la régression de la variable `Z` en fonction de la variable

T du jeu de données x. Cette fonction a pour unique argument x et calcule les coefficients de la régression linéaire.

```
> mafonction <- function(x){
    summary(lm(Z~T, data=x))$coef
 }
> by(don, vec1, mafonction)
vec1: A1
            Estimate Std. Error  t value     Pr(>|t|)
(Intercept) 5.166501  0.2900179 17.81442 7.030010e-13
T           2.990575  0.2844888 10.51210 4.118736e-09
------------------------------------------------------------
vec1: A2
            Estimate Std. Error  t value     Pr(>|t|)
(Intercept) 5.024887  0.2123503 23.66320 4.718878e-20
T           2.838283  0.2203563 12.88043 2.753530e-13
```

Ce type de calcul est aussi automatisé d'une manière différente par la fonction **lmList** proposée dans le package nlme.
Il existe également d'autres fonctions comme la fonction **replicate** qui permet de répéter une expression n fois (voir aussi le tableau 4.2). En général, l'expression comporte une génération de nombres aléatoires et les n évaluations de cette expression ne donnent pas le même résultat. Cela revient donc à faire une boucle `for (i in 1:n)`. Par exemple :

```
> set.seed(1234)
> replicate(n=8, mean(rnorm(100)))
[1] -0.007993349 -0.055986270 -0.060858439  0.036864175
[5]  0.109879373  0.089231186 -0.025551715 -0.073691196
```

La fonction **outer** permet de répéter une fonction sur chaque combinaison de deux vecteurs. Par exemple :

```
> Mois <- c("Jan","Fév","Mar")
> Année <- 2008:2010
> outer(Mois,Année,FUN="paste")
     [,1]       [,2]       [,3]
[1,] "Jan 2008" "Jan 2009" "Jan 2010"
[2,] "Fév 2008" "Fév 2009" "Fév 2010"
[3,] "Mar 2008" "Mar 2009" "Mar 2010"
```

Là encore, il est possible d'ajouter des arguments à la fonction utilisée. Ici, on peut séparer les mois et années par un trait d'union plutôt que par l'espace, qui est le caractère par défaut :

```
> outer(Mois,Année,FUN="paste",sep="-")
     [,1]       [,2]       [,3]
[1,] "Jan-2008" "Jan-2009" "Jan-2010"
[2,] "Fév-2008" "Fév-2009" "Fév-2010"
[3,] "Mar-2008" "Mar-2009" "Mar-2010"
```

Fonction	Description
aggregate	Applique une fonction à un sous-ensemble de lignes d'un data-frame (moyenne de toutes les variables par sexe, etc.). Retourne (en général) un data-frame.
apply	Applique une fonction à une marge (ligne, colonne, etc.) d'une matrice ou d'un tableau à plusieurs entrées (moyenne par colonne, somme par ligne, etc.).
by, **tapply**	Applique une fonction à un sous-ensemble de lignes d'un data-frame ou d'un vecteur (moyenne de toutes les variables par sexe, etc.). Retourne en général une liste.
mapply	Répète l'application d'une fonction avec comme premier argument les premières coordonnées de chaque vecteur, comme second argument les secondes coordonnées de chaque vecteur, etc. (voir aussi **vectorize**).
outer	Applique une fonction (vectorisée) à chaque couple de deux vecteurs.
replicate	Évalue n fois la même expression.
sapply, **lapply**	Applique une fonction à chaque coordonnée d'un vecteur ou chaque composante d'une liste. Retourne un vecteur, une matrice ou une liste.
sweep	Applique une fonction à une marge (ligne, colonne, etc.) d'une matrice ou d'un tableau à plusieurs entrées avec un argument `STATS` qui varie à chaque marge (diviser chaque colonne par son écart-type, soustraire à chaque colonne sa moyenne, etc.).

TABLE 4.2 – Résumé de quelques fonctions permettant d'éviter des boucles.

4.4 Calcul parallèle

4.4.1 Introduction

Cette section est une brève introduction au calcul parallèle sous R. Nous ne rentrerons pas ici dans le détail des architectures matérielles (CPU, GPU, thread, cluster, etc.), et nous nous focaliserons sur la programmation parallèle en R sur un ordinateur multi-cœurs, ce qui est le cas de (quasiment) tous les PC actuels.

Tout d'abord, le calcul parallèle se différencie du calcul séquentiel. Dans le calcul séquentiel, le problème est divisé en une série d'instructions, et ces instructions sont exécutées les unes après les autres sur une seule unité de calcul. Ainsi une seule instruction est exécutée à la fois. Dans le calcul parallèle, le problème est divisé en plusieurs séries d'instructions indépendantes pouvant s'exécuter en même temps, et ces instructions de chaque série s'exécutent simultanément sur des unités de calcul différentes. Ensuite, les résultats obtenus sur chaque unité de calcul sont renvoyés dans le processus parent. Par conséquent, plusieurs instructions sont exécutées en parallèle mais un mécanisme de contrôle et de synchronisation est nécessaire.

Si l'utilisateur souhaite résoudre un problème via une méthode statistique et qu'il souhaite gagner du temps en utilisant du calcul parallèle, il lui faudra donc diviser totalement ou en partie sa méthode en sous-calculs indépendants : il s'agit donc principalement d'un problème algorithmique. Il n'existe pas de version « parallèle » de toutes les méthodes.

Bien évidemment, si le calcul en parallèle est choisi, c'est que l'utilisateur souhaite que cela prenne moins de temps qu'avec un calcul séquentiel, ce qui n'est pas garanti. En effet, les tâches en parallèle sont contrôlées et synchronisées, ce qui est coûteux en temps de calcul. Il faudra donc que chaque tâche parallélisée prenne suffisamment de temps pour que le gain opéré par la division des tâches en parallèle ne soit pas perdu par les tâches de contrôles et de synchronisation. On s'intéressera donc à la parallélisation de notre code face à des calculs coûteux en temps, en regardant attentivement l'évolution de la performance et la gestion de la mémoire en fonction du nombre d'unités de calcul utilisées. Les problématiques classiques concernent la réalisation de simulations, le bootstrap ou encore l'estimation de plusieurs modèles (validation croisée, comparaison de méthodes, etc.).

R est à la base mono-cœur, ce qui implique que la majorité des calculs sont exécutés séquentiellement sur une unique unité de calcul. Cependant, de nombreux packages permettent de paralléliser le code. Nous nous focaliserons ici sur les deux packages parallel et foreach. Nous verrons dans les fiches, par exemple la fiche 10.1, comment certains packages utilisent le calcul parallèle.

4.4.2 Le package parallel

Le package parallel est inclus dans R depuis la version 2.14.0 et est maintenu par la *R core team*. Son interface est très proche de l'utilisation de fonctions de la famille des **apply**. Il dispose également d'une vignette très détaillée.

```
> require(parallel)
> vignette("parallel")
```

Les étapes de lancement d'un calcul parallèle sont les suivantes :

1. Ouverture d'un « cluster »

– la fonction **makeCluster** ouvre le « cluster » et déclare le nombre de cœurs utilisés (pour détecter automatiquement le nombre de CPU, on peut utiliser la fonction **detectCores**) ;
– ouverture de sessions R temporaires, fait automatiquement à l'ouverture.

2. Utilisation du « cluster »
Différentes fonctions que nous allons illustrer permettent d'utiliser le cluster, c'est-à-dire de réaliser les calculs en parallèle :
– **clusterExport**, **clusterCall**, **clusterApply**, etc. ;
– **parLapply**, **parSapply**, **parApply**, etc.

3. Fermeture du « cluster »
– C'est la fonction **stopCluster** (sinon les sessions R temporaires restent ouvertes).

Pour illustrer ces étapes, nous allons remplir une liste à 3 composantes en parallèle (autant que le nombre de cœurs utilisés). Chaque composante sera identique et aura 4 éléments de valeurs 1, 2, 3 et 4 respectivement. Cet exemple n'a bien sûr qu'une valeur pédagogique. La première étape est donc de démarrer le « cluster » sur un certain nombre de cœurs. Nous détectons le nombre de cœurs disponibles sur notre machine comme suit :

```
> require(parallel)
> nb_cores <- detectCores()
> nb_cores
[1] 4
```

Il vaut mieux éviter d'utiliser toutes les ressources, c'est pourquoi nous utilisons seulement 3 cœurs dans notre « cluster », lequel est démarré avec :

```
> cl <- makeCluster(nb_cores - 1)
```

Le calcul étant le même sur chaque cœur (création du vecteur `1:4`) nous utilisons la fonction **clusterCall** qui va exécuter notre fonction de création du vecteur `1:4`. Cette fonction prend en argument `cl`, l'objet de la classe cluster puis la fonction que l'on souhaite paralléliser, ici une fonction qui ne prend pas d'argument et qui crée un vecteur pour la séquence de 1 à 4. Par défaut **clusterCall** renvoie une liste.

```
> res <- clusterCall(cl = cl, fun = function() return(1:4))
```

Enfin nous stoppons le « cluster » et affichons les caractéristiques du résultat :

```
> stopCluster(cl)
> str(res)
List of 3
 $ : int [1:4] 1 2 3 4
 $ : int [1:4] 1 2 3 4
 $ : int [1:4] 1 2 3 4
```

Partage des données ou des packages
Par défaut, les sessions R temporaires sont vides. Il est possible pour les systèmes d'exploitation Linux/Mac de changer cette valeur par défaut (se référer à **makeCluster**(,type="FORK")). Aucune variable et aucun package de la session principale ne sont donc présents initialement. La fonction **clusterExport** permet d'exporter des objets (variables et fonctions) et la fonction **clusterEvalQ** exécute un code dans toutes les sessions, ce qui peut permettre notamment de charger un package.

Load Balancing
Généralement, les p premiers calculs sont envoyés aux p sessions temporaires. Les calculs suivants débutent lorsque tous les p calculs ont été effectués, ce qui facilite le mécanisme de contrôle et de synchronisation. Dans le cas de calculs de durées différentes, on perd donc de la performance en attendant la fin du calcul le plus long. Des versions LB, pour *Load Balancing*, existent pour enchaîner sur un nouveau calcul dès que le précédent se termine au niveau de chaque unité, par exemple en utilisant la fonction **clusterApplyLB**.

Générateur de nombres aléatoires
Une génération de nombres aléatoires (ou plus précisément pseudo-aléatoires) peut être programmée dans le code exécuté dans chaque cœur. Afin de pouvoir garantir que les résultats soient reproductibles et que les générations de nombres aléatoires soient différentes dans chaque cœur, il est nécessaire de contrôler la graine et la génération dans chaque partie du « cluster ». Pour cela, la fonction **clusterSetRNGStream** permet d'utiliser un générateur adapté et de gérer la graine globalement avec l'argument iseed.

Nous donnons un exemple avancé d'utilisation de ces fonctions dans la section 4.4.4.

4.4.3 Le package foreach

Le package foreach propose une alternative à l'utilisation des boucles for.

```
> require(foreach)
> vignette("foreach")
```

Exemples d'utilisation
La syntaxe est assez intuitive et on utilise **foreach** avec %do% pour exécuter une expression R de façon répétée, et retourner les résultats dans une liste par défaut.

```
> require(foreach)
> x <- foreach(i = 1:3) %do% (round(sqrt(i),2))
> str(x)
List of 3
 $ : num 1
```

```
 $ : num 1.41
 $ : num 1.73
```

Une fois calculée la racine carré des nombres 1, 2 et 3, on peut combiner les résultats et calculer la somme des trois valeurs en utilisant `.combine` comme suit :

```
> require(foreach)
> x <- foreach(i = 1:3, .combine = "+") %do% sqrt(i)
> x
[1] 4.146264
```

Le package dispose aussi de fonctionnalités pour appliquer des filtres avec `%:% when`. Dans l'exemple suivant, on parcours la séquence de 1 à 50 et quand c'est un nombre premier on affiche ce nombre. Les résulats sont ensuite combinés dans un vecteur (ils sont concaténés avec **c**).

```
> require(numbers)
> foreach(n = 1:50, .combine = "c") %:% when (isPrime(n)) %do% n
[1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47
```

Il permet également de paralléliser simplement des calculs en utilisant l'opérateur `%dopar%` à la place de `%do%` et des packages complémentaires. Nous utiliserons ici doParallel, qui fait le pont avec parallel (voir section 4.4.2). Les étapes sont les mêmes qu'avec l'utilisation directe du package parallel (ouverture d'un « cluster », utilisation et fermeture), à la seule différence qu'il faut préalablement « enregistrer » le cluster pour le lier à foreach. Cela se fait avec la fonction **registerDoParallel** du package doParallel.

```
> require(doParallel)
> cl <- makeCluster(3)
> registerDoParallel(cl) # enregistrement du cluster
> res <- foreach(n = 1:3) %dopar% rnorm(1000)
> stopCluster(cl)
```

Partage des données et packages
Contrairement au package parallel, toutes les variables de l'environnement courant sont exportées par défaut. Il faudra donc faire attention à ne pas exporter les objets inutiles, d'autant plus s'ils sont volumineux. Ceci se contrôle avec l'argument `.noexport`. L'argument `.export` permet l'exportation d'une variable présente dans l'environnement parent. Le chargement des packages se fait quant à lui simplement avec `.packages`.

4.4.4 Exemple avancé

Le jeu de données que nous allons utiliser se trouve dans le package kernlab.

```
> require(kernlab)
> data(spam)
```

Il se compose de 4 601 courriels, identifiés comme `spam` ou `nonspam`, et de 58 variables explicatives. Nous voulons modéliser le fait d'être un spam ou non à l'aide de forêts aléatoires (voir fiche 10.2 et le package randomForest) en mettant en place une validation croisée sur 4 blocs : apprentissage du modèle estimé sur (environ) 3/4 des données, validation sur le 1/4 restant, et cela répété 4 fois en permutant les blocs. Pour cela, nous créons une variable `spam$fold` qui contient le numéro du bloc, numéro choisi aléatoirement avec la fonction **sample** :

```
> set.seed(125)
> spam$fold <- sample(1:4, nrow(spam), replace = TRUE)
```

Nous pouvons vérifier la répartition des spams dans chaque bloc :

```
> table(spam$type, spam$fold)
            1   2   3   4
  nonspam 669 700 688 731
  spam    452 489 439 433
```

Nous créons également la fonction suivante, qui estime notre modèle sur un jeu d'apprentissage, calcule les prévisions ainsi que l'erreur associée sur le jeu de validation, et retourne l'ensemble des résultats dans une liste :

```
> cv_rf <- function(data_app, data_val){
    rf <- randomForest(type ~ ., data=data_app)
    y_val <- data.frame(type=data_val$type, y=predict(rf, newdata=data_val))
    list(rf=rf, y_val, err_rate=mean(y_val$y != y_val$type))
 }
```

Estimation avec parallel

Nous créons le « cluster » et contrôlons la graine du générateur qui sera utilisée par randomForest :

```
> require(parallel)
> cl <- makeCluster(2)
> clusterSetRNGStream(cl,iseed=78)
```

Ici il faut ne pas oublier d'exporter les variables nécessaires (les données `spam` ainsi que la fonction **cv_rf**) et de charger le package randomForest :

```
> clusterExport(cl, varlist = c("spam", "cv_rf"))
> clusterEvalQ(cl, {require(randomForest)})
```

Pour chaque bloc (de 1 à 4), nous le mettons de côté (jeu de validation), nous estimons le modèle sur les autres blocs (jeu d'apprentissage) puis calculons son taux d'erreur (sur la validation), et ceci grâce à la fonction **cv_rf**. Cette exécution est faite en parallèle grâce à **clusterApply** qui prend pour argument `x` qui varie de 1 à 4 et exécute une fonction de `x` faisant les opérations que nous venons d'énumérer. Rappelons que la fonction **cv_rf** utilise par construction toutes les variables pour expliquer `type`. Cependant, nous avons ajouté la variable `fold`, qui n'est pas une variable explicative, et nous la supprimons donc avant d'utiliser la fonction :

```
> res <- clusterApply(cl = cl, x = 1:4, fun = function(fold){
  spam_app <- spam[spam$fold != fold, ] # creation apprentissage
  spam_val <- spam[spam$fold == fold, ] # creation validation
  spam_app$fold <- NULL # suppression fold
  cv_rf(spam_app, spam_val) # calculs
})
```

Enfin nous stoppons le « cluster » et affichons les taux de mal classés par bloc :

```
> stopCluster(cl)
> sapply(res, function(x) x$err_rate)
[1] 0.05441570 0.05130362 0.05057675 0.04381443
```

Estimation avec foreach

On utilise l'argument `.packages` pour charger randomForest, et on évite d'exporter des variables inutiles avec `.noexport`. Ici la liste de tous les objets de la session, obtenue avec **ls**, est raccourcie de **c("spam","cv_rf")**, les objets à exporter, grâce à la fonction **setdiff**. Cette liste est fournie comme argument à `.noexport`. Pour terminer, on n'oublie pas d'enregistrer le cluster avec **registerDoParallel** :

```
> require(foreach)
> require(doParallel)
> cl <- makeCluster(2)
> clusterSetRNGStream(cl,iseed=78)
> registerDoParallel(cl)
> res <- foreach(fold = 1:4, .packages = "randomForest",
    .noexport = setdiff(ls(), c("spam", "cv_rf"))) %dopar% {
     spam_app <- spam[spam$fold != fold, ]
     spam_val <- spam[spam$fold == fold, ]
     spam_app$fold <- NULL
     cv_rf(spam_app, spam_val)
  }
> stopCluster(cl)
```

Pour aller plus loin

Pour en savoir plus sur le calcul parallèle, nous conseillons l'ouvrage *Calcul parallèle avec R* (Miele et Louvet, 2016). Vous pouvez aussi lire les vignettes des

packages parallel et foreach (si ce n'est déjà fait...) ou encore le task view `https://cran.r-project.org/web/views/HighPerformanceComputing.html`

4.5 Faire une application shiny

Objet

Le package shiny, créé en 2012 par les auteurs de RStudio, permet de construire des applications shiny, c'est-à-dire des interfaces web interactives qui facilitent l'utilisation de fonctions ou programmes R. Plutôt que de lancer des fonctions depuis R à l'aide de lignes de commandes, l'utilisateur pourra obtenir immédiatement et automatiquement les résultats des programmes en modifiant les paramètres des fonctions à partir d'une interface web. La construction d'une application shiny fait donc appel aux notions de programmation présentées précédemment. Des exemples d'applications shiny sont présentés ici : `http://shiny.rstudio.com/gallery/`.

Toutes les applications shiny, qu'elles soient simples ou complexes, nécessitent d'une part de construire une interface, et d'autre part d'écrire un programme R pour faire des calculs et générer des sorties (graphiques, tableaux, etc.). Les applications doivent respecter la structure suivante :

- un fichier `ui.R` (ui pour User Interface) qui définit l'interface, i.e. la mise en page et l'apparence de l'application. Il n'est pas nécessaire de connaître le langage html pour paramétrer ce fichier. Cette interface est visible par l'utilisateur, elle lui permet de définir les paramètres d'entrée et de visualiser les résultats ;
- un fichier `server.R` qui donne les instructions permettant les calculs sous R afin de créer les sorties et ainsi mettre à jour les résultats de l'interface.

Ces deux fichiers contiennent toutes les informations nécessaires pour définir l'application. On peut tester l'application grâce à la fonction `runApp` qui va la générer et l'ouvrir dans une page web (navigateur ou viewer de RStudio). Il sera finalement possible de mettre à disposition l'application pour d'autres utilisateurs sur Internet.

La construction d'application est facilitée par l'utilisation de RStudio. C'est pourquoi nous montrons comment construire une application shiny depuis RStudio. Après avoir lancé RStudio, faire `File` → `New project` → `New directory` → `Shiny Web Application`. Mettre un nom, par exemple `MonAppliShiny`, définir le répertoire où les fichiers seront sauvegardés et cliquer sur `Créer l'application`. Une fenêtre s'ouvre alors avec deux onglets : le premier contient un fichier `ui.R` et le second contient un fichier `server.R`. Suivant la version de RStudio, nous pouvons nous retrouver avec une application codée en un seul fichier `app.R`. Il est alors possible d'initier la structure en deux fichiers via `File` → `New file` → `Shiny Web App` en sélectionnant une application de type `Multiple File`.

Ces deux fichiers contiennent les lignes de code d'une application shiny proposée par défaut par RStudio. Il « suffit » de les modifier tout en gardant leur structure pour construire une nouvelle application. En cliquant sur le bouton `Run App`, on

peut donc la tester en la lançant sur son propre ordinateur. Si l'un des deux fichiers a été modifié, ce bouton est renommé `Reload App`.

Étapes

1. Construction de l'interface web avec le fichier `ui.R`
2. Écriture du programme dans le fichier `server.R`
3. Mise à disposition de l'application

Exemple simple

Nous allons détailler comment construire une application qui permet de choisir une variable d'un jeu de données et qui retourne la médiane ainsi qu'un histogramme des valeurs de cette variable. Pour l'histogramme, le nombre de classes et la couleur peuvent être choisis par l'utilisateur. L'interface est proposée en figure 4.2 et l'application est disponible à l'adresse `https://statavecr.shinyapps.io/MonAppliShiny1/`.

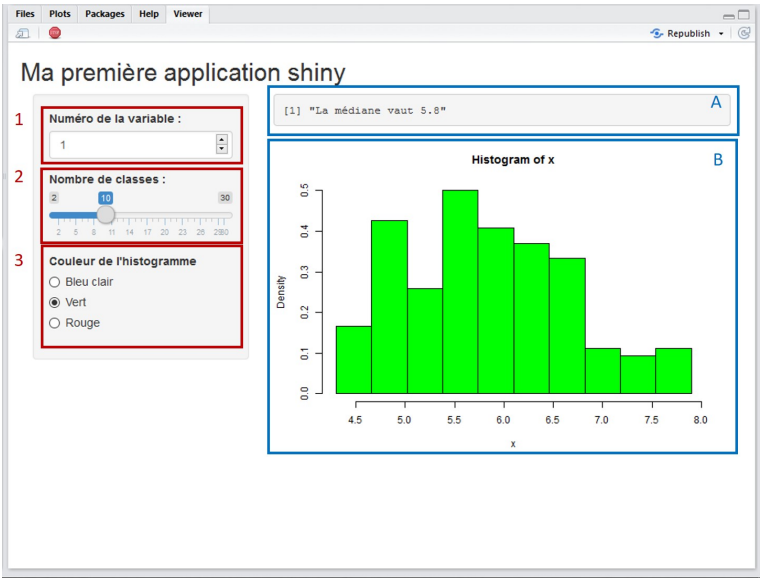


FIGURE 4.2 – Exemple simple d'application shiny.

1. Construction de l'interface web avec le fichier `ui.R`

L'interface web (Fig. 4.2) est définie à partir du fichier `ui.R`. L'ensemble des instructions pour la construction de l'interface est défini par **shinyUI** et la structure

de la page web est définie par **fluidPage**. On trouve ensuite le titre de la page web (**titlePanel**) puis la définition de la partie paramétrage, à gauche, et la définition de la partie résultat à droite. Le fichier `ui.R` est donc ici structuré comme suit :

```
library(shiny)
shinyUI(fluidPage(
  titlePanel("Titre de l'application"),
  sidebarPanel(
     Définition de la partie gauche de la page web (paramètres)
  ),
  mainPanel(
     Définition de la partie droite de la page web (sorties)
  )
))
```

sidebarPanel gère la partie gauche de la page web et permettra à l'utilisateur de choisir les valeurs des paramètres. Cette fonction contient une suite d'instructions, chacune permettant de définir un paramètre. Les paramètres sont positionnés verticalement dans l'ordre où ils sont définis. Chaque instruction est un argument de la fonction ce qui explique qu'elles soient séparées par des virgules. **mainPanel** gère la partie droite de la page web, c'est-à-dire l'affichage des résultats. Là encore, la fonction contient une suite d'instructions, séparées par des virgules, chaque instruction permettant d'afficher un résultat.

La figure 4.3 donne le fichier `ui.r` puis le fichier `server.R` de notre exemple d'application.

Les instructions de **sidebarPanel** définissent les paramètres `NumVar` (numéro de la variable), `NbCla` (nombre de classes de l'histogramme), `Coul` (la couleur de l'histogramme) :

```
> numericInput(inputId="NumVar", label="Numéro de la variable :",
    min=1, max=4, value=1),
> sliderInput(inputId="NbCla", label="Nombre de classes :",
    min=2, max=30, value=10),
> radioButtons(inputId="Coul", label="Couleur de l'histogramme",
    choices = c("Bleu clair"="lightblue","Vert"="green","Rouge"="red"),
    selected="green")
```

On voit que shiny propose un ensemble de fonctions pour différents types de paramètres, par exemple ici : **numericInput** si le paramètre est défini dans une suite d'entiers, **sliderInput** pour une sélection à partir d'un curseur, **radioButtons** à partir d'une liste, etc. Pour chacune de ces fonctions, le premier argument (`inputId`) identifie le paramètre et c'est ce nom qui sera utilisé dans le fichier `server.R`. Comme pour tout objet R, les caractères accentués sont à proscrire ici. Le second argument, `label`, donne le libellé affiché dans l'interface et il est possible ici d'utiliser des caractères accentués ou plusieurs mots. Les arguments suivants diffèrent selon la fonction shiny utilisée : `choices` propose des

Fichier ui.R

```
library(shiny)
shinyUI(fluidPage(
  titlePanel("Ma première application shiny"),

    sidebarPanel(
1 → numericInput("NumVar", "Numéro de la variable :", min=1, max=4, value=1),
2 → sliderInput("NbCla", "Nombre de classes :", min=2, max=30, value=10),
3 → radioButtons("Coul", "Couleur de l'histogramme",
        choices = c("Bleu clair"="lightblue","Vert"="green","Rouge"="red"),
        selected="green")
    ),

    mainPanel(
A' → verbatimTextOutput("sortie1"),
B' → plotOutput("sortie2")
    )
))
```

Fichier server.R

```
library(shiny)
shinyServer(function(input, output) {

A → output$sortie1 <- renderPrint({                    (1)
      paste("La médiane vaut",median(iris[,input$NumVar]))
    })

B → output$sortie2 <- renderPlot({
      x <- iris[,input$NumVar](1)                      (2)
      Classes <- seq(min(x), max(x), length.out = input$NbCla + 1)
      hist(x, breaks = Classes, freq=FALSE, col=input$Coul)
    })                                                 (3)
})
```

FIGURE 4.3 – Fichiers `ui.R` et `server.R` de l'application shiny. Les paramètres d'entrée sont en rouge, les sorties sont en bleu. La définition des paramètres d'entrée ou des résultats est en gras tandis que l'utilisation est en police normale.

choix de valeurs possibles pour le paramètre, `value` ou `selected` définissent une valeur par défaut, etc. Voir les aides sur les fonctions à partir du site shiny `http://shiny.rstudio.com/reference/shiny/latest/`.

Tous ces objets `NumVar`, `NbCla`, `Coul` sont regroupés dans un objet nommé `input` qui sera utilisé en entrée dans le programme `server.R`. Ainsi, le nombre de classes sera défini par l'utilisateur grâce à l'interface et utilisé ensuite dans le fichier `server.R` avec le code `input$NbCla`.

La fonction **mainPanel** définit la partie droite de la page web et affiche tous les résultats obtenus depuis `server.R`. Les résultats peuvent être de formats variables, d'où l'utilisation ici encore de différentes fonctions. Le nom de la fonction est la concaténation du type de résultat et de `Output` : **tableOutput** pour afficher un tableau, **verbatimTextOutput** pour afficher une sortie console R, **plotOutput** pour afficher un graphe (voir les lignes A' et B' de la figure 4.3).

2. Écriture du programme dans le fichier `server.R`

Voyons maintenant comment est structuré le fichier `server.R`. Ce fichier peut tout d'abord contenir des instructions, par exemple le chargement d'un package, d'un

jeu de données, le téléchargement d'une carte ou encore des calculs qui sont toujours effectués quelle que soit l'utilisation de l'application. En d'autres termes, des lignes de commande qui définissent l'environnement à partir duquel l'application est lancée.

Ensuite, ce fichier `server.R` reçoit les valeurs des paramètres d'entrée choisies via l'interface et contenues dans `input`, fait les calculs, puis renvoie au fichier `ui.R` un objet nommé `output` qui contient tous les résultats à afficher dans l'interface. Les résultats pouvant être de natures différentes, les fonctions sont adaptées : **renderPlot** pour renvoyer un graphe, **renderTable** pour un tableau, **renderText** pour du texte à écrire, **renderImage** pour une image, etc. Les fonctions **renderPrint** (sortie console) et **renderPlot** sont utilisées dans l'exemple (voir les points A et B de la figure 4.3)). À chaque fois, le résultat de la fonction est affecté à un objet dont le nom est `output$MaSortie` et c'est l'objet `MaSortie` qui est affiché par `ui.R`. Les fonctions sont programmées comme habituellement dans R si ce n'est qu'elles utilisent des paramètres d'entrée dont les valeurs sont définies via l'interface (`input$NumVar` par exemple). Elles peuvent être le résultat de nombreux calculs, utiliser plusieurs fonctions de différents packages, etc. La complexité de la programmation n'augmente en rien la difficulté de la construction de l'application shiny.

3. Mise à disposition de l'application

L'application est maintenant prête. Elle peut être testée sur l'ordinateur du développeur en cliquant sur `RunApp` ou `ReloadApp`. Si l'application convient, elle peut être mise à disposition de n'importe quel utilisateur sur Internet. Ainsi, toute personne connectée à Internet pourra utiliser l'application, sans utiliser R et sans même avoir besoin d'installer le logiciel R sur son ordinateur. En effet, les calculs seront effectués sur le serveur de shiny et les résultats renvoyés sur la page web.

Pour publier l'application depuis RStudio, il vous faut cliquer sur le bouton `Publish Application` à droite du bouton `Run App`. Lors de la première publication d'une application shiny, vous devrez vous créer un compte ShinyApps et choisir une clé (appelée `Token`). Vous allez donc cliquer sur `Go to your account on shinyApps and log in` et choisir la formule que vous voulez (la formule gratuite permet de proposer jusqu'à 5 applications au jour où ces lignes sont écrites). Vous pouvez passer à l'étape 2 et copier le Token en cliquant sur `Copy to Clipboard` avant de coller ce Token dans le cadre réservé à cet effet dans RStudio. Vous pouvez ensuite cliquer sur `OK` et publier votre application. Le package rsconnect est nécessaire pour cette procédure. Il est normalement intégré dans RStudio mais si ce n'est pas le cas, vous devez l'installer.

La publication de l'application peut prendre du temps si votre application fait appel à plusieurs packages car les packages seront installés sur le serveur shiny. Une fois l'application publiée, vous pouvez fermer RStudio. L'application restera disponible sur le serveur shiny via une page Internet jusqu'à ce que vous « dépubliez » l'application. Lorsque vous souhaitez que votre application ne soit plus accessible sur Internet, il faut la « dépublier » par :

```
> rsconnect::terminateApp("MonAppliShiny1")
```

Remarque
Il est possible de construire les fichiers `ui.R` et `server.R` sans utiliser RStudio. Il faut alors mettre ces deux fichiers dans un même répertoire que l'on appellera pour l'exemple `MonAppliShiny1` et lancer l'application par :

```
> runApp("MonAppliShiny1")
```

Il est possible d'intégrer une application shiny dans un package, sans utiliser de compte ShinyApps. Le répertoire `MonAppliShiny1` devra être copié dans le répertoire `inst` et appelé par la fonction **shiny::runApp**.

L'application peut aussi être déployée sur un serveur externe via l'utilisation de shiny-server. Ce service propose une partie opensource et une autre payante (cf. `https://www.rstudio.com/products/shiny/shiny-server/`).

Exemple avancé

Il est possible de construire des applications plus complexes. Nous considérons un second exemple qui permet de choisir un jeu de données, de sélectionner une variable quantitative puis, sur les données de cette variable, calcule sa médiane et construit un histogramme ainsi qu'un estimateur de la densité. Pour l'histogramme, on pourra faire varier le nombre de classes, la couleur et ajouter ou non une courbe de lissage en choisissant le mode de lissage. L'interface de cette application est disponible en figure 4.4 et l'application est disponible à l'adresse `https://statavecr.shinyapps.io/MonAppliShiny2/`.
La figure 4.5 donne le fichier `ui.R` puis le fichier `server.R` de notre application. On retrouve ici de nombreuses instructions déjà présentes dans l'exemple précédent. Les instructions numérotées 0, 2, 3 et 4 définissent les paramètres d'entrée de l'application. Ces objets sont regroupés dans l'objet `input` qui sera utilisé en entrée dans le programme `server.R`. Les résultats obtenus depuis `server.R` sont affichés avec la fonction **mainPanel** (voir les lignes A', B' et C' de la figure 4.5).

Détaillons trois nouveautés de ce fichier : la mise en forme de l'interface, l'utilisation de la fonction **reactive** pour éviter de refaire certains calculs, et l'utilisation conjointe des fonctions **renderUI** et **uiOutput** pour modifier l'apparence de l'interface en fonction des choix de l'utilisateur.

1. Mise en forme de l'interface
Les lignes suivantes (cf. lignes html de la figure 4.5)

```
> h3("Choix des données"),
> h3("Paramètres du graphe"),
```

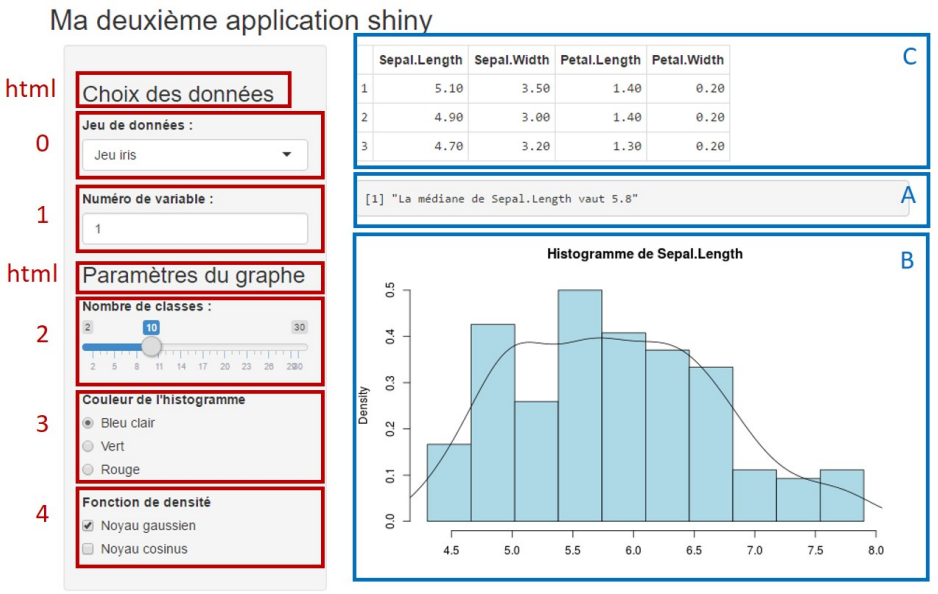


FIGURE 4.4 – Deuxième application shiny.

permettent de mettre en forme la page web en ajoutant du texte avec les instructions `h1`, `h2`, etc. qui définissent le niveau des titres en langage html (`h1` étant le titre de niveau le plus élevé donc écrit en plus gros caractères). shiny arrive en effet avec un ensemble de balises html disponibles dans l'objet `tags` (cf. **names**(`tags`).

2. Lorsque certains calculs du programme `server.R` sont longs, on peut construire une fonction dite **reactive** afin de partager le résultat entre plusieurs sorties. Cette fonction est lancée normalement la première fois et le résultat est alors sauvegardé dans la mémoire de l'ordinateur. Pour les utilisations suivantes, le résultat sauvegardé est réutilisé s'il est à jour, sans que les calculs de la fonction **reactive** ne soient réexécutés. En revanche, si l'utilisateur modifie certains paramètres d'entrée depuis la page web, et que ces paramètres sont utilisés par cette fonction, alors le code est relancé et le nouveau résultat sauvegardé en mémoire. Il faut donc y penser dès que le résultat d'un calcul est utilisé dans plusieurs sorties, afin d'accélérer la réactivité de l'application en réduisant le temps de génération des résultats.

Dans l'exemple, les lignes suivantes créent une fonction appelée **don**, fonction qui retourne un extrait du jeu de données choisi par l'utilisateur avec uniquement les variables quantitatives :

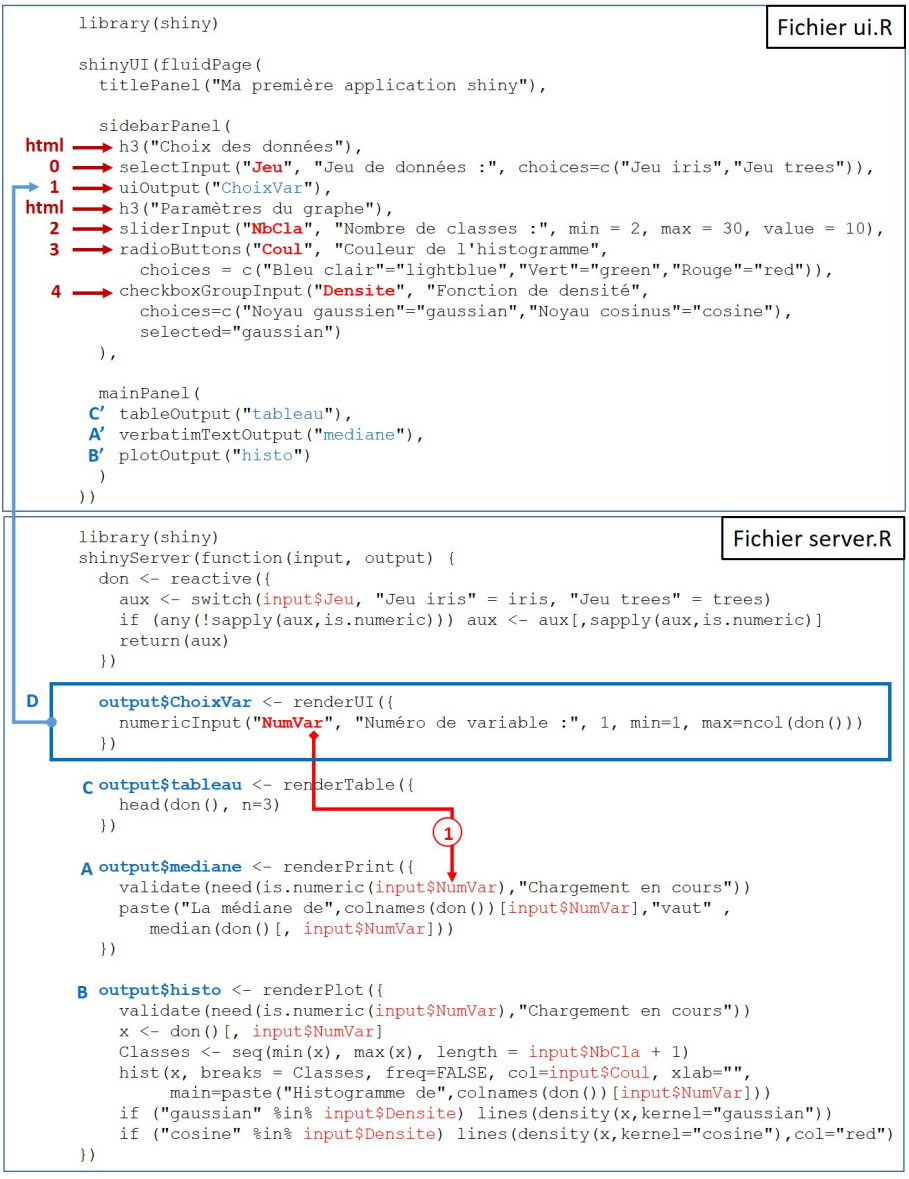


FIGURE 4.5 – Fichiers `ui.R` et `server.R` de l'application shiny. Les paramètres d'entrée sont en rouge, les sorties sont en bleu. La définition des paramètres d'entrée ou des résultats est en gras tandis que l'utilisation est en police normale.

```
don <- reactive(
  aux <- switch(input$Jeu, "Jeu iris" = iris, "Jeu trees" = trees)
  if (any(!sapply(aux,is.numeric))) aux <- aux[,sapply(aux,is.numeric)]
  return(aux)
)
```

C'est parce que **don** est une fonction que l'on utilise **don()**, avec des parenthèses, pour appeler le jeu de données. Par exemple pour le calcul de la médiane, on utilise **median(don()[,input$NumVar])**.

3. Voyons maintenant comment mettre à jour l'interface en fonction de choix faits par l'utilisateur sur d'autres paramètres. Dans l'exemple simple, nous choisissions un numéro de variable entre 1 et 4 avec l'instruction 1 présente dans le fichier `ui.R` :

```
> numericInput("NumVar", "Numéro de la variable :", min=1, max=4, value=1)
```

Cette ligne de code construit l'encadré 1 de la figure 4.2, dont l'affichage est équivalent à l'encadré 1 de la figure 4.4.

Nous allons faire en sorte que ce paramètre `NumVar` puisse maintenant être choisi entre 1 et un maximum qui n'est plus systématiquement égal à 4 mais au nombre de variables quantitatives du jeu de données choisi. Ainsi, selon le jeu de données, on doit déterminer automatiquement, grâce à R, le maximum. Le remplacement de `max=4` par `max=ncol(don())` doit donc se faire côté `server.R` car il variera dans le temps en fonction des paramètres rentrés par l'utilisateur. L'objet `ChoixVar` est donc construit dans `server.R` avec les lignes du bloc D de la figure 4.5, en calculant le nombre de variables du jeu de données choisi et en générant le bloc qui sera affiché dans l'interface :

```
output$ChoixVar <- renderUI(
  numericInput("NumVar", "Numéro de variable :", 1, min=1, max=ncol(don()))
)
```

Il faut ensuite afficher ce bloc `ChoixVar` dans l'interface grâce à la commande suivante dans le fichier `ui.R` (voir l'instruction 1 de la figure 4.5) :

```
> uiOutput("ChoixVar"),
```

On fait appel à la fonction **uiOutput** pour signifier qu'on affiche ici un résultat (un output) composé d'éléments normalement présents dans le fichier `ui.R` (paramètres, affichage et placement des sorties). Lorsque l'utilisateur choisit un numéro de variable à étudier, la valeur est bien affectée à `input$NumVar` afin d'être utilisée dans les autres fonctions de `server.R`.

La ligne suivante est utilisée dans `output$median` et `output$histo` :

```
validate(need(is.numeric(input$NumVar),"Chargement en cours"))
```

Elle permet d'éviter qu'une erreur apparaisse au début, quand le numéro de variable n'est pas encore défini parce que le jeu de données n'est pas encore choisi. Le calcul du nombre de variables n'est donc pas encore possible. Cette erreur n'est pas visible par l'utilisateur car l'interface se met rapidement à jour seule, mais cette erreur est visible par le concepteur de l'interface.

Pour en savoir plus et trouver des exemples d'applications shiny, visiter le site `http://shiny.rstudio.com`.

4.6 Exercices

Exercice 4.1 (Factorielle)

1. Programmer factorielle n : $n! = n \times (n-1) \times \cdots \times 2 \times 1$ en utilisant **prod**.

2. Programmer factorielle n en utilisant une boucle `for`.

Exercice 4.2 (Ventilation)

1. Reprendre l'exercice 2.6 (p. 55) et programmer une fonction de ventilation. Cette fonction aura pour arguments une variable et le seuil à partir duquel une modalité doit être ventilée. Mettre 5 % comme seuil par défaut.

2. Écrire une fonction permettant de ventiler toutes les variables qualitatives d'un tableau (utiliser la question précédente).

Exercice 4.3 (Ventilation sur facteur ordonné)

1. Reprendre l'exercice 2.7 (p. 56) et programmer une fonction de ventilation sur facteur ordonné. Cette fonction aura pour arguments une variable et le seuil à partir duquel une modalité doit être ventilée. Mettre 5 % comme seuil par défaut.

2. Écrire une fonction permettant de ventiler toutes les variables qualitatives ordonnées d'un tableau (utiliser la question précédente).

Exercice 4.4 (Parallélisation)

On souhaite estimer la loi de la longueur maximale des sous-suites strictement croissantes d'une suite de n observations i.i.d. issues d'une loi uniforme en estimant les proportions empiriques sur k simulations (on rappelle que **runif**`(n)` permet de simuler une loi uniforme). On se donne la fonction suivante qui permet de déterminer la longueur de la plus grande sous-suite strictement croissante :

```
> max_inc <- function(seq) {
   dseq <- seq[-1] > seq[-length(seq)]; cur_l <- 1+dseq[1]; max_l <- cur_l
   for (i in 1:(length(dseq)-1)) {
     if (dseq[i] && (dseq[i] == dseq[i+1])) {
```

```
      cur_l <- cur_l + 1;
    } else { cur_l <- 1+dseq[i+1] }
  max_l <- max(cur_l, max_l)
  }
  return(max_l)
}
```

1. Utiliser **sapply** et **table** pour déterminer les proportions empiriques pour $n = 1000$ et $k = 100\,000$ simulations.

2. Paralléliser ce code à l'aide de parallel puis de foreach.

3. Quel gain de vitesse peut-on obtenir ? Que faire si c'est encore trop lent ?

Exercice 4.5 (Test de permutation)
On veut tester la nullité d'un coefficient de corrélation entre deux variables x et y par un test de permutation.

1. Simuler deux variables aléatoires x et y de longueur 50 et suivant une loi normale à l'aide des lignes de code suivantes :

```
> set.seed(1234)
> x <- rnorm(50)
> epsilon <- rnorm(50)
> y <- 2+ 0.5*x + epsilon
```

2. A l'aide d'une boucle `for` construire un test de permutation. Pour ce faire, calculer la corrélation entre x et un vecteur $y_{\text{permuté}}$ (utiliser la fonction **sample** par exemple). Si cette corrélation est plus grande que la corrélation observée entre x et y, alors ajouter 1. Répéter ce calcul `nb.simul=100 000` fois puis donner le pourcentage de fois où la corrélation dépasse la corrélation observée entre x et y. Comparer à la probabilité critique obtenue avec la fonction **cor.test** (nous rappelons que le test est bilatéral).

3. Refaire la question 2 en parallélisant le calcul et donner les temps de calcul avec et sans parallélisation.

4. Reprendre les questions 2 et 3 en simulant non plus des lois normales mais des lois du χ^2 à 1 degré de liberté pour x et pour ε. Que pouvez-vous dire ?

Chapitre 5

Outils avancés pour la préparation des données

Les données évoluent... R aussi !
Les origines de R remontent à la création du langage S au milieu des années 1970. À cette époque, la volumétrie et l'organisation des données étaient radicalement différentes de ce que l'on connaît aujourd'hui. Cette dernière décennie, sous l'impulsion des géants du *web*, l'évolution s'est encore accélérée. Pour répondre aux nouveaux besoins, de nombreux packages ont vu le jour, ciblant des problématiques spécifiques ou apportant un point de vue original.
Aujourd'hui, deux packages majeurs se partagent la manipulation des données : data.table, qui vise avant tout l'efficacité en étant très optimisé, et dplyr qui réalise plutôt un compromis entre efficacité et facilité d'utilisation [1]. Les principales fonctionnalités de ces deux packages sont présentées dans les sections suivantes.
Vu la diversité des données à traiter, leur importation depuis des sources aussi variées que des bases de données relationelles, des fichiers structurés (au format JSON par exemple), des bases de données NoSQL ou des sites *web* devient un enjeu majeur. Les packages actuels les plus utilisés sont également présentés.

5.1 Le package data.table

Le package data.table est conçu pour traiter de manière efficace, tant du point de vue du temps d'exécution que de l'empreinte mémoire utilisée, des data-frames volumineux, voire très volumineux. A cette fin, le package définit une nouvelle syntaxe de manipulation des données qui se veut à la fois souple et puissante, et introduit certains concepts informatiques peu ou pas utilisés dans un usage traditionnel de R. Par exemple, nous reviendrons largement dans la suite sur la

1. Tout un écosystème de packages partageant la même *philosophie* gravite autour de ce package pour former le tidyverse et s'étend même au-delà.

notion de modification d'objets par référence. Toutefois, en guise d'illustration du soin apporté par les développeurs de ce package à l'optimisation du temps d'exécution, nous allons présenter l'exemple de la fonction **fread**.

5.1.1 Importation avec fread

Dédiée à la lecture de fichiers « plats », **fread** est si efficace que même les utilisateurs de R qui n'utilisent pas les autres fonctionnalités de data.table s'en servent souvent lors de l'importation de données volumineuses.
Pour illustrer notre propos, commençons par construire un jeu de données qui servira dans les exemples illustratifs de ce chapitre et du suivant.

```
> library(data.table)
> set.seed(1234)
> dt <- data.table(group1 = c("A", "B"),
                   group2 = rep(c("C", "D"), each = 5000),
                   value = rnorm(10000),
                   weight = sample(1:10, 10000, replace = TRUE))
> dt
       group1 group2       value weight
    1:      A      C -1.20706575      1
    2:      B      C  0.27742924      9
    3:      A      C  1.08444118      8
    4:      B      C -2.34569770      7
    5:      A      C  0.42912469      7
   ---
 9996:      B      D  0.01973902      2
 9997:      A      D -2.12674529     10
 9998:      B      D -0.05022201      4
 9999:      A      D -0.23817408      6
10000:      B      D  0.77640531      8
```

Remarquons au passage que la création d'un data-table est similaire à celle d'un data.frame. Les fonctions **as.data.frame** et **as.data.table** permettent d'ailleurs de passer d'un type d'objet à l'autre [2]. On remarque également que l'affichage est restreint aux premières et dernières lignes, ce qui permet d'éviter des mauvaises surprises en cas de **print** d'un gros data-frame dans la console R.

Nous allons d'abord sauvegarder cet objet de petite taille. Ensuite nous allons le lire d'une part avec la fonction usuelle de lecture **read.table**, d'autre part avec la fonction de lecture **fread**.

```
> write.table(dt,"dt.csv",sep=",",row.names=FALSE,quote=FALSE)
> cat("File size (MB):", round(file.info("dt.csv")$size/1024^2),"\n")
Taille en  (MB): 0
```

2. En réalité les data-tables sont aussi des data-frames, comme on peut s'en convaincre à l'aide de la commande **class**(dt).

Le fichier est donc petit. Néanmoins, le rapport des temps d'exécution est significatif (attention, ces temps dépendent de votre ordinateur) :

```
> system.time(df <- read.table("dt.csv", sep=",", header=TRUE,
    stringsAsFactors=FALSE))
utilisateur     système      écoulé
       0.02        0.00        0.018
> system.time(dt <- fread("dt.csv"))
utilisateur     système      écoulé
      0.008       0.000       0.007
> all.equal(as.data.table(df),dt)
[1] TRUE
```

Sur ce petit exemple, **fread** va déjà bien plus vite. Pour pouvoir comparer les deux lectures, nous avons précisé que les caractères (qui sont transformés par défaut en facteurs avec **read.table** mais pas avec **fread**) ne devaient pas être considérés comme des facteurs.

Voyons maintenant un exemple plus représentatif des données actuelles puisqu'il contient un million de lignes et 5 colonnes (numériques et caractères). La taille du fichier est de 40 Mb.

```
> set.seed(1234)
> n <- 1e6
> dt2 <- data.table(a=sample(1:1000,n,replace=TRUE),
                    b=runif(n),
                    c=rnorm(n),
                    d=sample(c("A","B","C","D"),n,replace=TRUE))
> write.table(dt2,"dt2.csv",sep=",",row.names=FALSE,quote=FALSE)
> cat("Taille en (MB):", round(file.info("dt2.csv")$size/1024^2),"\n")
Taille en (MB): 40
> system.time(df2 <- read.table("dt2.csv", sep=",", header=TRUE,
    stringsAsFactors=FALSE))
utilisateur     système      écoulé
      8.036       0.076       8.114
> system.time(dt2 <- fread("dt2.csv"))
utilisateur     système      écoulé
      0.364       0.000       0.373
```

Il y a désormais un facteur 20 en gain de temps ! Si l'on reprend cette procédure avec 10 millions de lignes, ce qui donne un fichier d'environ 400 Mb, la lecture avec **read.table** laisse le temps de prendre un café... Avec **fread**, le gain de temps est cette fois d'un facteur 30. Nous vous laissons imaginer le temps gagné pour lire des fichiers de quelques gigaoctets.

L'ensemble des fonctions du package a été optimisé avec le même souci d'efficacité que la fonction **fread**. Le temps d'apprentissage, non négligeable, est ainsi largement rentabilisé par la suite, ne serait-ce que par les temps d'importation économisés !

5.1.2 Syntaxe

La syntaxe pour manipuler un data-table peut sembler proche, au premier abord, de celle utilisée pour les data-frames car elle utilise des crochets `[...]`. Elle est néanmoins très différente en pratique. De façon générique la manipulation d'un data-table revêt la forme suivante :

```
> data[i, j, by]
```

où `i` définit une sous-sélection des lignes, `j` un calcul (opération, modification, sélection) sur les colonnes et `by` un groupement des lignes [3]. Avant de donner les détails de la syntaxe relative à chaque partie `i`, `j` et `by`, donnons un exemple et calculons, pour les individus `"A"` du `group1`, la moyenne des valeurs pour chaque modalité du `group2` :

```
> dt[group1 == "A", mean(value), by = group2]
   group2           V1
1:      C 0.0008501418
2:      D 0.0539426361
```

On constate ici l'appel aux différentes colonnes (`group1`, `value`, `group2`) sans avoir besoin de les préfixer par `dt$` comme c'est le cas en R de base. Ceci permet une syntaxe un peu plus légère.

5.1.3 Sélection

Nous allons maintenant préciser la syntaxe pour les sélections.

Sur les lignes : `i`
Un data-table ne dispose pas de **rownames**. La sélection peut se faire de différentes manières. Par exemple, en utilisant les indices des lignes, comme pour un data-frame :

```
> dt[1:2, ]
   group1 group2      value weight
1:      A      C -1.2070657      1
2:      B      C  0.2774292      9

> dt[c(1,5)]              # virgule optionnelle
   group1 group2      value weight
1:      A      C -1.2070657      1
2:      A      C  0.4291247      7

> dt[order(value), ]
```

3. Une analogie est souvent faite avec le langage SQL : `i` correspond à *WHERE*, `j` à *SELECT* et `by` à *BY*.

```
       group1 group2     value weight
    1:      B      C -3.396064      6
    2:      B      D -3.335108      8
    3:      A      C -3.233152      9
    4:      A      D -3.200732      7
    5:      B      D -3.185662      6
   ---
 9996:      B      D  3.357021      4
 9997:      B      D  3.456972      3
 9998:      A      D  3.560775      6
 9999:      B      D  3.606510      6
10000:      B      D  3.618107     10
```

La fonction **order** renvoie la permutation (sous la forme d'un vecteur) qui permet d'ordonner le vecteur qu'elle reçoit en paramètre selon les valeurs croissantes. On peut également utiliser un filtre sur les colonnes :

```
> dt[weight > 9, ]        # pas besoin de guillemets
> dt[weight > 9 & group2 == "C", ]
```

Dans le premier exemple, seuls les individus (ou les lignes) dont le poids (variable ou colonne `weight`) est strictement supérieur à 9 sont sélectionnés. Dans le second, les individus dont le poids dépasse 9 et pour lesquels la variable catégorielle `group2` vaut `C` sont sélectionnés.

Pour l'instant, hormis le fait qu'on économise la lourdeur d'écrire `dt$weight` et qu'on peut se contenter de mentionner directement le nom de la variable, tout ce qui précède fonctionne de façon similaire pour les data-frames.

Sur les colonnes : `j`

Il existe au moins trois façons de sélectionner des colonnes (ou variables) d'un data-frame. La première est d'utiliser leurs indices, comme dans l'exemple suivant :

```
> dt[, 1]
       group1
    1:      A
    2:      B
    3:      A
    4:      B
    5:      A
   ---
 9996:      B
 9997:      A
 9998:      B
 9999:      A
10000:      B
```

```
# plusieurs colonnes
> dt[, c(1,3)]
      group1       value
    1:      A -1.20706575
    2:      B  0.27742924
    3:      A  1.08444118
    4:      B -2.34569770
    5:      A  0.42912469
   ---
 9996:      B  0.01973902
 9997:      A -2.12674529
 9998:      B -0.05022201
 9999:      A -0.23817408
10000:      B  0.77640531
```

Avant d'aller plus loin, on peut souligner un point commun avec les data-frames : la première virgule est évidemment obligatoire... sinon, on sélectionnerait des lignes comme au-dessus ! Cette similarité évoquée, il est sans doute plus intéressant de pointer une différence importante : la commande `dt[, 1]` renvoie un data-table et non pas un vecteur comme ce serait le cas pour un data-frame. Il y a donc une stabilité du type de résultat indépendamment du nombre de colonnes sélectionnées.

Il est également possible de sélectionner des variables en utilisant leurs noms sous la forme d'un vecteur de chaînes de caractères. L'exemple suivant clarifie la manière de s'y prendre :

```
> dt[, "group1"]
> dt[, c("group1", "value")]
```

Enfin, on peut construire la liste des variables qu'on souhaite sélectionner. Regardons un exemple :

```
> dt[, list(group1)]
> dt[, list(group1, value)]
> dt[, .(group1, value)]
```

La dernière ligne mérite une explication : data.table faisant historiquement un usage intensif des listes, le raccourci **.** a été créé pour se substituer à **list** dans un soucis de compacité du code à écrire et à lire. Ce raccourci n'est défini qu'à l'intérieur des `[...]` d'un data-table.

La sélection de variables à l'aide de **list** présente plusieurs intérêts. L'un d'entre eux est de permettre de renommer les variables pour l'affichage des résultats (cela ne change pas le nom des variables dans l'objet `dt`) :

```
> dt[, list(mygroup = group1, myvalue = value)][1:2]
   mygroup    myvalue
1:       A -1.2070657
2:       B  0.2774292
```

Cette ligne de code montre qu'il est aussi possible d'enchaîner des opérations sur un data-table en concaténant plusieurs expressions `[i, j, by]`. Dans l'exemple précédent, on commence par sélectionner (et renommer) deux variables puis on sélectionne les deux premières lignes du data-table résultant de cette opération. On comprend ainsi l'intérêt d'obtenir un data-table même lorsqu'on ne sélectionne qu'une seule variable : le chaînage est toujours permis. Ce ne serait pas le cas si l'on obtenait un vecteur dans cette situation.

Ces méthodes de sélection de colonnes retourne toujours un data-table, y compris lorsque l'on extrait une seule colonne. Si on souhaite obtenir le résultat sous forme d'un vecteur, il faut utiliser `$` ou `[[...]]` :

```
> dt$value
> dt[["value"]]
```

Notons que `dt[,value]` fonctionne également mais n'est pas recommandé par les auteurs de data.table.

5.1.4 Manipulation

Modification des colonnes

Toutes les opérations de manipulations des colonnes (ajout, modification, suppression) se font avec le nouvel opérateur « `:=` ». On commence par créer une nouvelle colonne `tval` qui sera rajoutée à la fin du data-table et qui correspond à la colonne `value` mais tronquée (voir l'aide la fonction **trunc**). On affiche ici les deux premières lignes du nouveau jeu de données :

```
> dt[, tval := trunc(value)][1:2]
   group1 group2      value weight tvalue
1:      A      C -1.2070657      1     -1
2:      B      C  0.2774292      9      0
```

Notons que contrairement aux data-frames, il n'est pas nécessaire de créer un nouvel objet qui corresponde à l'ancien de jeu de données concaténé avec la nouvelle colonne. L'objet initial est modifié. Ici, la nouvelle colonne `tval` fait maintenant partie du data-table `dt`.

Il est possible de créer plusieurs colonnes de deux façons :

```
# par référence
> dt[, c("tvalue", "rvalue") := list(trunc(value), round(value, 2))]
# syntaxe alternative
> dt[, ':=' (tvalue = trunc(value), rvalue = round(value ,2))]
```

La modification d'une colonne existante est très facile, quant à sa suppression elle se fait en affectant la valeur `NULL` à la colonne :

```
> dt[, tvalue := tvalue + 10]   # modification
> dt[, rvalue := NULL]          # suppression
```

En combinant les deux paramètres `i` et `j`, il est donc possible de modifier directement des colonnes pour une sous-population :

```
> dt[ group1 %in% "A", weight := weight * 10L][1:2]
   group1 group2      value weight tvalue
1:      A      C -1.2070657     10     -1
2:      B      C  0.2774292      9      0
```

Remarque
Dans la première ligne de code, on utilise L pour indiquer que c'est un entier et non un réel (double), sans quoi on reçoit un avertissement.

Avec cette syntaxe, on peut par exemple remplacer facilement l'ensemble des valeurs manquantes d'une colonne par sa moyenne :

```
> dt[is.na(value), value := mean(value, na.rm = TRUE)]
```

Calculs et agrégations
Il est possible d'effectuer des calculs directement dans la partie `j`, soit en utilisant une expression R simple, par exemple `dt[, sum(value)]` pour récupérer la somme de la colonne `value`, soit en utilisant une **list** (ou le raccourci **.**), avec ou sans nom, dans le cas de plusieurs opérations. Pour des opérations plus complexes, nous pouvons également utiliser les accolades **{}**.

```
> dt[, sum(value)]        # un vecteur
> dt[, list(sum(value))]  # un data-table
# calculs multiples + nouveau nom
> dt[, list(somme = sum(value), moyenne = mean(value))]
      somme     moyenne
1: 61.15893 0.006115893
```

Pour effectuer un calcul sur une sous-population, on peut ajouter une sélection préalable des lignes avec `i` :

```
> dt[group1 == "B" & group2 == "C", list(sum(value), mean(value))]
       V1     V2
1: -33.89884 -0.01355953
```

Sans noms explicites, les colonnes du résultat s'appellent `V1`, `V2`, etc.

Calculs et agrégations avec by

Le calcul d'agrégats est facilité avec la troisième partie by, jusqu'à présent non utilisée. Agrégats sur :
- une seule variable : utilisation directe du nom, avec ou sans guillemets :

```
> dt[, sum(value), by = group1]
> dt[, sum(value), by = "group1"]
```

- plusieurs variables : **list** (ou **.**), ou un vecteur de noms :

```
> dt[, list(somme = sum(value)), by = list(group1, group2)]
> dt[, list(somme = sum(value)), by = .(group1, group2)]
> dt[, list(somme = sum(value)), by = c("group1", "group2")]
```

Nous pouvons renommer les variables utilisées et nous servir d'expressions R :

```
> dt[, list(somme = sum(value)), by = list(pop = group1, poids = weight>5)]
   pop poids     somme
1:   A  TRUE 136.98194
2:   B  TRUE -58.17120
3:   B FALSE -17.65182
```

L'ensemble des opérations détaillées ci-dessus retourne un data-table et réalise un affichage dans la console. Nous pouvons affecter le résultat à une variable en utilisant <- pour le conserver en mémoire dans un objet R. Si on souhaite modifier le jeu de données disponible, il faut utiliser l'opérateur « := ». L'argument by permet d'effectuer un calcul par groupe et de le réaffecter directement à l'ensemble des lignes :

```
> dt[, mean_group1 := mean(value), by = list(group1)][1:3]
   group1 group2      value weight tvalue mean_group1
1:      A      C -1.2070657     10     -1  0.02739639
2:      B      C  0.2774292      9      0 -0.01516460
3:      A      C  1.0844412     80      1  0.02739639
```

L'exemple suivant illustre l'utilisation de l'opérateur « .N », très utile pour compter les occurrences suite à un regroupement par by (cela revient à utiliser la fonction **table** dans R base). On sélectionne ici les sous-populations identifiées avec les colonnes group1 et group2 présentant un poids supérieur à 5, et on trie les résultats :

```
> dt[weight > 5, .N, by = list(group1, group2)][order(-N)]
   group1 group2    N
1:      A      C 2500
2:      A      D 2500
3:      B      C 1305
4:      B      D 1268
```

Une manipulation très fréquente, utilisée avec by, consiste à appliquer la même opération d'agrégation à toutes les variables sauf celles qui ont servi à effectuer un groupement. Par exemple, on peut vouloir calculer la moyenne de toutes les variables quantitatives pour chaque combinaison possible des modalités des variables qualitatives. Dans notre exemple le code suivant produit le résultat escompté :

```
> dt[, list(mean(value), mean(weight)), by = list(group1, group2)]
   group1 group2           V1     V2
1:      A      C  0.024472851 5.4472
2:      B      C -0.023296178 5.5212
3:      A      D  0.012778894 5.4924
4:      B      D  0.007089771 5.4520
```

Remarquons toutefois qu'une telle façon de procéder peut être source d'erreurs. Dans notre exemple très simple, le nombre de variables quantitatives est peu important. Le code est lisible et facilement maintenable en cas de modification du nom de ces variables. Dans des situations réelles, avec un grand nombre de variables quantitatives, le code serait inutilement long et pénible à maintenir. Pour pallier ces problèmes, la commande .SD, qui doit être utilisée dans la partie j, permet un accès, sous forme d'un data-table à l'ensemble des colonnes présentes hors variables de groupement. L'exemple donné ci-dessus peut donc être réécrit de la façon suivante, plus compacte :

```
> dt[, lapply(.SD, mean), by = list(group1, group2)]
   group1 group2        value weight
1:      A      C  0.024472851 5.4472
2:      B      C -0.023296178 5.5212
3:      A      D  0.012778894 5.4924
4:      B      D  0.007089771 5.4520
```

Dans ce cas .SD contient donc list(value, weight) et l'on comprend l'utilisation de la fonction lapply qui renvoie une liste : la même que dans le premier exemple.

Notons enfin qu'il est possible de restreindre .SD à un sous-ensemble de colonnes à l'aide du paramètre .SDcols. Un cas typique d'utilisation est le suivant : on souhaite calculer la moyenne de toutes les variables quantitatives pour un sous-ensemble des variables catégorielles. Il est alors possible d'utiliser l'une des méthodes suivantes :

```
> dt[, lapply(.SD, mean), by = group1, .SDcols = c("value", "weight")]
   group1        value weight
1:      A  0.018625873 5.4698
2:      B -0.008103203 5.4866
```

ou, peut-être encore mieux dans ce contexte :

```
> dt[, lapply(.SD, mean), by = group1, .SDcols = -c("group2")]
   group1        value weight
1:      A  0.018625873 5.4698
2:      B -0.008103203 5.4866
```

Enfin, notons qu'il est également possible de créer ou modifier simultanément plusieurs colonnes avec l'utilisation conjointe de l'opérateur `:=` :

```
> sd_col <- c("value", "weight")
> new_col <- c("t_value", "t_weight")
> dt[,c(new_col) := lapply(.SD, trunc), .SDcols = sd_col]
```

5.1.5 Pour aller plus loin

Le package data.table propose également un certain nombre de fonctions optimisées, présentes dans R, et conservant la même syntaxe. On peut citer par exemple **merge** (fusion de tables), **subset** (sous-ensemble), **melt** et **dcast** (transformation/pivot des données) ou encore **split**.

Quelques fonctions utiles
Nous avons vu en détail l'intérêt d'utiliser **fread** pour lire des données. La fonction **fwrite** fournit une alternative avantageuse à **write.csv** en apportant des gains de vitesse substantiels. La fonction **tables** permet d'accéder à des informations sur l'ensemble des tables existantes :

```
> tables()
   NAME   NROW NCOL MB                          COLS KEY
1:   dt 10,000    4  0 group1,group2,value,weight
Total: 0MB
```

Le package data.table fournit également de nombreuses fonctions dont le nom débute par `set`. Ces fonctions modifient [4] le data-table passé comme premier argument. Par exemple **setorder** (et sa variante **setorderv**) permettent de trier une table, **setcolorder** est dédiée au ré-ordonnancement des colonnes et **setnames** autorise la modification des noms de colonnes. Mentionnons rapidement **rbindlist**, dédiée à la concaténation de data-tables et **shift** qui permet de faire du décalage de vecteurs comme dans l'exemple :

```
> shift(1:10, type="lag", fill = NA, n=2L) # "lag" ou "lead"
[1] NA NA  1  2  3  4  5  6  7  8
```

Terminons notre inventaire en mentionnant **IDate** et **IDateTime**, utiles pour gérer efficacement les dates et les heures.

4. La modification se fait par référence afin d'améliorer la rapidité et l'empreinte mémoire utilisée.

Indexation : les clés

Le filtrage d'une table selon des colonnes est nettement plus rapide si celles-ci sont pré-triées dans un ordre compatible avec le filtrage. Cette idée est exploitée de manière efficace dans les bases de données sous le nom d'indexation par clés. Le package data.table dispose d'une telle d'indexation où les clés sont des colonnes choisies par l'utilisateur. Le tableau est alors ordonné suivant ces clés (il peut y en avoir plusieurs), et les filtres sur ces clés deviennent beaucoup plus rapides, avec un appel simplifié, en respectant l'ordre des clés. Cela peut se faire de deux façons :

- à travers argument `key` dans la fonction **data.table** ;
- à l'aide des fonctions **setkey** & **setkeyv** : noms de colonnes, avec ou sans guillemets.

La fonction **key** liste quant à elle les clés effectives.

```
# clé unique
> setkeyv(dt, "group1")
> key(dt)
[1] "group1"
> dt["A"] # une valeur
> dt[c("A", "B")] # plusieurs valeurs
# clés multiples
> setkey(dt, group1, group2)
> dt["A"]
> dt[list("A", c("C", "D"))]
# supressions des clés
> setkeyv(dt, NULL)
```

Attention, la mise en place de clés peut impacter certaines fonctions (**unique** par exemple).

Fonction copy et gestion de la mémoire

Un objet data-table peut se voir comme un pointeur mémoire, et il ne possède pas les mêmes propriétés que la plupart des autres objets R. En effet, si on affecte un data-table existant à une nouvelle variable et si l'on modifie cette dernière, la variable contenant le data-table initial sera également impactée par cette modification.

```
> dt <- data.table(x = 1, y = 1)
> dt2 <- dt     # nouvelle affectation
> dt2[, y := 2] # modification de y
> dt
   x y
1: 1 2
> dt2
   x y
1: 1 2
```

Pour éviter cela, il faut copier explicitement l'objet avec la fonction **copy** :

```
> dt <- data.table(x = 1, y = 1)
> new_dt <- copy(dt)
> new_dt[, y := 2] # modification de y
> dt
   x
1: 1
> new_dt
   x y
1: 1 2
```

Notons que ce comportement est différent des data-frames pour lesquels une copie est effectuée lors de l'affectation :

```
> df <- data.frame(x = 1, y = 1)
> df2 <- df  # nouvelle affectation
> df2$y <- 2 # modification de y
> df
  x y
1 1 1
> df2
  x y
1 1 2
```

5.2 Le package dplyr et le tidyverse

L'une des forces de R réside dans ses packages. Ceux-ci permettent d'étendre ses fonctionnalités. Le package dplyr propose ainsi, à la manière de data.table, une nouvelle manière de travailler sur les tableaux dans R. Ces packages proposent en fait une alternative à la syntaxe de base qu'on est libre d'utiliser ou non.

Plus précisément, dplyr est le précurseur d'une famille de packages, appelée tidyverse, qui cherche à proposer un écosystème de fonctions développées avec des principes similaires de sorte que l'on puisse facilement les combiner entre elles.

5.2.1 Le package dplyr

L'objectif de dplyr est de proposer une grammaire permettant de travailler sur des tableaux, les data-frames de R ou plutôt une variante appelée tibble. Les tibbles se comportent comme des data-frames à quelques exceptions près : essentiellement, les lignes d'un tibble n'ont pas de nom, un tibble reste un tibble lorsqu'on extrait une unique colonne et, à l'affichage, seules quelques lignes sont montrées.

Ces choix correspondent à trois idées centrales : les tibbles sont l'analogue des tables dans une base de données et les lignes n'y ont pas de nom ; une fonction

doit toujours retourner le même type d'objet et le résultat de l'extraction de colonnes ne doit pas dépendre du nombre de colonnes ; enfin, l'affichage ne doit pas être trop long car on est susceptible d'utiliser de gros tableaux. De manière générale, les fonctions de dplyr (ou du tidyverse) ne font que peu de transformations implicites : tout doit se faire de manière explicite, ce qui facilite l'utilisation dans un cadre programmatique. Le tibble est l'un des cœurs du tidyverse et l'objet utilisé prioritairement pour stocker des informations.

Afin de permettre la comparaison avec data.table, nous allons construire la même table.

```
> library(tidyverse)
> set.seed(1234)
> tbl <- tibble(group1 = rep(c("A", "B"), 5000),
                group2 = rep(c("C", "D"), each = 5000),
                value = rnorm(10000),
                weight = sample(1:10, 10000, replace = TRUE))
> tbl
# A tibble: 10,000 x 4
   group1 group2  value weight
   <chr>  <chr>   <dbl>  <int>
 1 A      C      -1.21       1
 2 B      C       0.277      9
 3 A      C       1.08       8
 4 B      C      -2.35       7
 5 A      C       0.429      7
 6 B      C       0.506      6
 7 A      C      -0.575      6
 8 B      C      -0.547      6
 9 A      C      -0.564     10
10 B      C      -0.890      3
# ... with 9,990 more rows
```

L'affichage est plus compact que celui d'un data-frame, tout en fournissant le maximum d'informations pour un nombre fixé de lignes. On notera que le type des colonnes est indiqué juste en dessous du nom de celles-ci. On peut modifier le nombre de lignes écrites en utilisant l'argument `n` et la fonction **print** :

```
> print(tbl, n = 4)
# A tibble: 10,000 x 4
  group1 group2  value weight
  <chr>  <chr>   <dbl>  <int>
1 A      C      -1.21       1
2 B      C       0.277      9
3 A      C       1.08       8
4 B      C      -2.35       7
# ... with 9,996 more rows
```

On peut sélectionner des lignes et des colonnes par les indexations classiques :

```
> tbl[1:2, ]
# A tibble: 2 x 4
  group1 group2  value weight
  <chr>  <chr>   <dbl>  <int>
1 A      C      -1.21       1
2 B      C       0.277      9
> tbl[c(1,5), ]
# A tibble: 2 x 4
  group1 group2  value weight
  <chr>  <chr>   <dbl>  <int>
1 A      C      -1.21       1
2 A      C       0.429      7
> print(tbl[, c("group1", "value")], n=4)
# A tibble: 10,000 x 2
   group1  value
   <chr>   <dbl>
 1 A      -1.21
 2 B       0.277
 3 A       1.08
 4 B      -2.35
# ... with 9,996 more rows
```

On peut utiliser des commandes spécifiques à dplyr comme **slice** et **select** qui permettent, respectivement, de sélectionner des lignes et des colonnes :

```
> slice(tbl, 1:2)
# A tibble: 2 x 4
  group1 group2  value weight
  <chr>  <chr>   <dbl>  <int>
1 A      C      -1.21       1
2 B      C       0.277      9
> print(select(tbl, group1, value), n=4)
# A tibble: 10,000 x 2
  group1  value
  <chr>   <dbl>
1 A      -1.21
2 B       0.277
3 A       1.08
4 B      -2.35
# ... with 9,996 more rows
```

Pour extraire le contenu d'une colonne, on peut utiliser :

```
> tbl[["value"]]
> pull(tbl, value)
```

Les commandes du tidyverse ont toujours comme premier argument l'objet sur lequel on souhaite travailler. C'est effectivement le cas des commandes **slice** et **select** comme de la commande **filter** qui permet de filtrer des lignes en fonction de conditions.

L'un des points forts de dplyr est l'utilisation de l'évaluation non standard de R qui permet d'utiliser les noms des colonnes d'un tibble sans avoir à les transformer en chaîne de caractères : il suffit de les lister dans les arguments en les séparant par des virgules. On peut ainsi sélectionner les colonnes `group1` et `weight` puis filtrer le résultat pour ne garder que les lignes dont le poids est supérieur à 9 à l'aide des commandes suivantes :

```
> tmp <- select(tbl, group1, weight)
> print(filter(tmp, weight > 9), n=4)
# A tibble: 1,011 x 2
  group1 weight
  <chr>   <int>
1 A          10
2 B          10
3 B          10
4 A          10
# ... with 1,007 more rows
```

Le même résultat peut être obtenu en composant les fonctions :

```
> filter(select(tbl, group1, weight), weight > 9)
```

Cette notation compositionnelle est rapidement illisible car l'ordre de lecture est l'ordre inverse de l'ordre d'exécution. Le tidyverse propose une alternative, le tuyau `%>%` (pipe en anglais), qui permet d'enchaîner des commandes en appliquant des fonctions qui utilisent tout ce qui précède comme premier argument :

```
> tbl %>% select(group1, weight) %>% filter(weight > 9) %>% print(,n=4)
# A tibble: 1,011 x 2
  group1 weight
  <chr>   <int>
1 A          10
2 B          10
3 B          10
4 A          10
# ... with 1,007 more rows
```

Ce pipe peut se traduire par « et ensuite » : on prend le tableau `tbl` et ensuite on sélectionne deux colonnes et ensuite on filtre les lignes. Le *pipeline* de traitement

apparaît ainsi de manière explicite dans le bon ordre. Ceci permet de composer de manière élégante des fonctions élémentaires pour effectuer des tâches complexes.

5.2.2 Manipulation de tables

Le langage le plus utilisé de manipulation de tables est sans aucun doute le SQL, ce langage déclaratif utilisé par une grande majorité des bases de données. Le package dplyr propose une syntaxe différente, pensée pour la programmation, qui permet essentiellement d'effectuer les mêmes tâches que celles possibles avec une base de données. Ce lien fort permet même d'utiliser la syntaxe de dplyr pour concevoir des requêtes SQL comme expliqué à la section 5.3.

Trois commandes de base de dplyr permettent des manipulations élémentaires sur les lignes et les colonnes d'un tibble :
- **select** : sélectionne des colonnes par leur nom (ou leur position) ;
- **mutate** : modifie ou crée des colonnes en fonction des autres ;
- **filter** : filtre les lignes en fonctions de conditions sur les colonnes.

Ces trois commandes sont complétées par deux commandes agissant globalement sur la table :
- **summarize** : résume une table en la réduisant à une unique ligne ;
- **arrange** : ordonne une table en fonction d'une variable.

Ces commandes permettent de manipuler de manière efficace les tables. En reprenant les exemples de la section précédente, on peut ainsi :
- sélectionner des colonnes et extraire certaines lignes à l'aide de :

```
tbl %>% select(mygr = group1, myval = value) %>%  slice(1:2)
# A tibble: 2 x 2
  mygroup myvalue
  <chr>     <dbl>
1 A        -1.21
2 B         0.277
```

- créer une ou des nouvelles colonnes (on notera que, à l'inverse de data.table, le tibble original n'est modifié que si l'on affecte explicitement le résultat) :

```
> tbl %>% mutate(tval = trunc(value)) %>% slice(1:2)
# A tibble: 2 x 5
  group1 group2  value weight  tval
  <chr>  <chr>   <dbl>  <int> <dbl>
1 A      C      -1.21       1    -1
2 B      C       0.277      9     0
> tbl <- tbl %>% mutate(tvalue = trunc(value), rvalue = round(value,2))
```

- modifier une colonne existante :

```
> tbl %>% mutate(tvalue = tvalue + 10) %>% print(,n=4)
# A tibble: 10,000 x 6
  group1 group2  value weight tvalue rvalue
  <chr>  <chr>   <dbl>  <int>  <dbl>  <dbl>
```

```
1 A      C      -1.21       1      9  -1.21
2 B      C       0.277      9     10   0.28
3 A      C       1.08       8     11   1.08
4 B      C      -2.35       7      8  -2.35
# ... with 9,996 more rows
```

– éliminer une colonne, voici deux façons de faire :

```
> tbl %>% select(-rvalue) %>% print(,n=4)
# A tibble: 10,000 x 5
  group1 group2  value weight tvalue
  <chr>  <chr>   <dbl>  <int>  <dbl>
1 A      C      -1.21       1     -1
2 B      C       0.277      9      0
3 A      C       1.08       8      1
4 B      C      -2.35       7     -2
# ... with 9,996 more rows
> tbl %>% mutate(rvalue = NULL) %>% print(,n=4)
# A tibble: 10,000 x 5
  group1 group2  value weight tvalue
  <chr>  <chr>   <dbl>  <int>  <dbl>
1 A      C      -1.21       1     -1
2 B      C       0.277      9      0
3 A      C       1.08       8      1
4 B      C      -2.35       7     -2
# ... with 9,996 more rows
```

– modifier de manière conditionnelle une colonne à l'aide de **if_else** :

```
> tbl %>% mutate(weight = if_else(group1 %in% "A",
            weight * 10L, weight)) %>% slice(1:2)
# A tibble: 2 x 6
  group1 group2  value weight tvalue rvalue
  <chr>  <chr>   <dbl>  <int>  <dbl>  <dbl>
1 A      C      -1.21      10     -1  -1.21
2 B      C       0.277      9      0   0.28
```

– calculer un résumé d'une table :

```
> tbl %>% summarize(sum(value))
# A tibble: 1 x 1
  `sum(value)`
         <dbl>
1         61.2
> tbl %>% summarize(somme = sum(value), moyenne = mean(value))
# A tibble: 1 x 2
  somme moyenne
  <dbl>   <dbl>
  1  61.2 0.00612
```

Notons qu'une erreur courante est d'utiliser le classique **summary**, qui permet d'obtenir des statistiques sur un data-frame, au lieu du **summarize** de dplyr.

– trier un tableau :

```
> tbl %>% arrange(desc(weight)) %>% slice(1:2)
# A tibble: 2 x 6
  group1 group2  value weight tvalue rvalue
  <chr>  <chr>   <dbl>  <int>  <dbl>  <dbl>
1 A      C      -0.564     10      0  -0.56
2 B      C      -0.911     10      0  -0.91
```

La puissance de ces commandes est décuplée par la possibilité de faire des traitements par groupe à l'aide de la fonction **group_by**. Ceci permet de :

– déterminer des résumés par groupes :

```
> tbl %>% group_by(group1, group2) %>% summarize(sum(value))
# A tibble: 4 x 3
# Groups:   group1 [?]
  group1 group2 `sum(value)`
  <chr>  <chr>          <dbl>
1 A      C               2.13
2 A      D             135.
3 B      C             -33.9
4 B      D             -41.9
> tbl %>% group_by(pop = group1, poids = weight > 5) %>%
    summarize(somme = sum(value))
# A tibble: 4 x 3
# Groups:   pop [?]
  pop   poids  somme
  <chr> <lgl>  <dbl>
1 A     FALSE  103.
2 A     TRUE    33.8
3 B     FALSE  -17.7
4 B     TRUE   -58.2
```

– modifier des colonnes par un calcul par groupe :

```
> tbl %>% group_by(group1) %>% mutate(mean_group1 = mean(value)) %>%
    print(,n=4)
# A tibble: 10,000 x 7
# Groups:   group1 [2]
   group1 group2  value weight tvalue rvalue mean_group1
   <chr>  <chr>   <dbl>  <int>  <dbl>  <dbl>       <dbl>
 1 A      C      -1.21       1     -1 -1.21       0.0274
 2 B      C       0.277      9      0  0.28      -0.0152
 3 A      C       1.08       8      1  1.08       0.0274
 4 B      C      -2.35       7     -2 -2.35      -0.0152
# ... with 9,996 more rows
```

– compter les effectifs par groupe grâce à la commande **n()** :

```
> tbl %>% filter(weight > 5) %>% group_by(group1, group2) %>%
      summarize(N = n()) %>% arrange(desc(N))
# A tibble: 4 x 3
# Groups:   group1 [2]
  group1 group2     N
  <chr>  <chr>  <int>
1 B      C       1305
2 A      C       1282
3 B      D       1268
4 A      D       1244
```

Ces exemples ne présentent qu'une petite partie des commandes disponibles dans le package dplyr. Pour plus de détails, le lecteur intéressé pourra se référer au site `https://dplyr.tidyverse.org/`.

5.2.3 Pour aller plus loin

dplyr et fusion de tableaux

Lorsque les données sont assez complexes et qu'elles présentent un certain degré de redondance, elles peuvent être organisées dans plusieurs tables différentes liées entre elles : c'est le principe des bases de données relationnelles. Nous allons voir, à travers quelques exemples, comment utiliser dplyr pour fusionner des tables entre elles et en extraire de l'information structurée.

Sur le site du livre, on peut trouver le fichier `chanson-française.xlsx`[5] qui contient plusieurs tables (ou feuilles dans la terminologie des tableurs). Pour les lister, on utilise la commande **excel_sheets** du package readxl :

```
> xlsx <- "chanson-française.xlsx"
> readxl::excel_sheets(xlsx)
[1] "chanteurs" "albums"
```

Pour créer les tibbles qui serviront à illustrer nos propos, il suffit d'utiliser les instructions :

```
> chanteurs <- readxl::read_excel(xlsx, sheet="chanteurs")
> albums <- readxl::read_excel(xlsx, sheet="albums")
```

Regardons les données :

```
> chanteurs
# A tibble: 4 x 4
  prenom  nom      naissance  mort
```

5. Fichier pour tableurs au format Office Open XML. Ce format est principalement utilisé par les versions de Micorsoft Office (Excel) mais est également lisible par LibreOffice ou Apache OpenOffice.

```
  <chr>   <chr>        <dbl> <dbl>
1 Georges Brassens      1921  1981
2 Léo     Ferré         1916  1993
3 Jacques Brel          1929  1978
4 Renaud  Séchan        1952    NA
```

et

```
> albums
# A tibble: 76 x 4
   titre                              annee prenom  nom
   <chr>                              <dbl> <chr>   <chr>
 1 La Mauvaise Réputation              1952 Georges Brassens
 2 Le Vent                             1953 Georges Brassens
 3 Les Sabots d'Hélène                 1954 Georges Brassens
 4 Je me suis fait tout petit          1956 Georges Brassens
 5 Oncle Archibald                     1957 Georges Brassens
 6 Le Pornographe                      1958 Georges Brassens
 7 Les Funérailles d'antan             1960 Georges Brassens
 8 Le temps ne fait rien à l'affaire   1961 Georges Brassens
 9 Les Trompettes de la renommée       1962 Georges Brassens
10 Les Copains d'abord                 1964 Georges Brassens
# ... with 66 more rows
```

On remarque immédiatement que ces tables sont liées entre elles : chaque album de la table `albums` est enregistré par un chanteur identifié à l'aide de son prénom et de son nom. Par ailleurs, la table `chanteurs` contient des informations biographiques sur chaque chanteur, également identifié par son prénom et son nom. Le couple `c("prenom", "nom")` est appelé « clé unique » et permet de fusionner les différentes informations contenues dans les deux tables de façon cohérente.

Le package dplyr propose une famille de fonctions réalisant ce genre d'opérations de fusion, les fonctions ***_join** qui prennent toutes comme arguments deux tableaux et la liste des clés selon laquelle faire la fusion. Ces fonctions diffèrent néanmoins sur la manière de réaliser les fusions. Nous allons donner un exemple utilisant la fonction **left_join** et nous expliquerons brièvement l'utilité de quelques autres fonctions.

La fonction **left_join** essaie de mettre en relation toutes les lignes du premier tableau avec des lignes du second. En cas de correspondance multiple dans le second tableau, toutes les lignes sont conservées :

```
> left_join(chanteurs, albums, by=c("prenom", "nom"))
# A tibble: 65 x 6
  prenom  nom      naissance  mort titre                                annee
  <chr>   <chr>        <dbl> <dbl> <chr>                                <dbl>
1 Georges Brassens      1921  1981 La Mauvaise Réputation                1952
```

```
2 Georges Brassens      1921  1981 Le Vent                              1953
3 Georges Brassens      1921  1981 Les Sabots d'Hélène                  1954
4 Georges Brassens      1921  1981 Je me suis fait tout petit           1956
# ... with 61 more rows
```

En cas d'absence de correspondance dans le second tableau, la ligne du premier tableau est conservée et des valeurs manquantes sont ajoutées. On peut s'en rendre compte sur l'exemple ci-dessous avec la chanteuse Juliette Noureddine :

```
> albums %>%
    filter(annee>1968) %>%
    group_by(prenom, nom) %>%
    summarise(post_soixante_huit=n()) %>%
    left_join(chanteurs, by=c("prenom", "nom")) %>%
    select(prenom, nom, naissance, mort, post_soixante_huit)
# A tibble: 4 x 5
# Groups:   prenom [4]
  prenom   nom        naissance  mort post_soixante_huit
  <chr>    <chr>          <dbl> <dbl>              <int>
1 Georges  Brassens         1921  1981                  3
2 Jacques  Brel            1929  1978                  2
3 Juliette Noureddine        NA    NA                 12
4 Léo      Ferré            1916  1993                 21
```

La fonction **right_join** fait la même chose en inversant le rôle des deux tableaux. Pour ne conserver que les lignes ayant des correspondances, il faut utiliser **inner_join**. Enfin, **full_join** permet de rajouter à ce dernier tableau toutes les lignes des deux tableaux n'ayant pas eu de correspondance (en les complétant par des valeurs manquantes). Il existe deux autres fonctions dans la même famille **semi_join** et **anti_join** qui se contentent de sélectionner des lignes dans le premier tableau en fonction respectivement de l'existence et de la non existence d'une correspondance dans le second.

dplyr et nest

Comme data.table, le package dplyr permet d'avoir accès aux sous-tables correspondant à chacun des groupes afin de permettre des traitements avancés. La commande **nest** transforme ainsi un tibble groupé en un tibble groupé avec une ligne par groupe, des colonnes correspondant aux colonnes des groupes et une colonne supplémentaire contenant sur chaque ligne le tibble correspondant aux lignes du groupe avec toutes les colonnes autres que celles utilisées pour le groupement. Il s'agit véritablement de l'équivalent de l'objet accessible via **.SD** dans data.table. En utilisant la commande **map** ou **lapply**, on peut alors appliquer des commandes arbitraires à ces tibbles pourvu que ces commandes donnent des résultats compatibles avec des tibbles. Ces résultats peuvent être conservés sous cette forme ou

extraits à l'aide de la colonne **unnest**. Cette dernière commande permet d'obtenir un nouveau tibble avec pour chaque groupe autant de lignes et de colonnes supplémentaires que dans les tibbles obtenus précédemment.

On peut par exemple calculer la moyenne par groupe de certaines colonnes à l'aide du code suivant :

```
> tbl %>% group_by(group1) %>% select(value, weight) %>% nest() %>%
  mutate(data = map(data, ~ map_df(., mean))) %>% unnest()
```

Philosophie du tidyverse

Le langage R bénéficie de fonctionnalités avancées (évaluation non standard, méta-programmation, etc.) qui en font un outil idéal pour créer des langages spécialisés à un domaine (Domain Specific Language ou DSL en anglais). Les formules utilisées en R de base pour décrire le lien entre réponse et variables explicatives en sont un exemple, de même que lattice ou ggplot2.

Le tidyverse développé autour des tibbles propose une nouvelle manière de travailler sous R. Elle repose sur des fonctions élémentaires dont le premier élément est toujours l'objet sur lequel s'applique la fonction et qui retourne des nouveaux objets dont le type est connu à l'avance. L'objectif est de proposer une grammaire la plus expressive possible afin d'obtenir des codes les plus faciles à gérer. Il faut noter que l'objectif n'est pas nécessairement l'optimalité en terme de vitesse d'exécution. Le package data.table présenté en Section 5.1 affiche par exemple de meilleurs performances en proposant des commandes qui modifient les tables sans aucune copie. Les packages du tidyverse s'interdisent ce type d'optimisation afin de rester plus proche des principes de la programmation fonctionnelle.

Un bon point de départ pour savoir ce qui est disponible est la page de référence `https://www.tidyverse.org/`. Le package tidyverse est un méta-package qui permet d'installer en une seule commande tous les packages du tidyverse. On y trouve par exemple :

- ggplot2 qui a déjà été présenté pour les graphes ;
- tidyr qui permet notamment des opérations de type pivot ;
- readr et haven qui permettent de lire des fichiers de manière efficace ;
- forcats qui permet de travailler avec les facteurs ;
- lubridate qui permet de travailler sur les dates ;
- stringr qui permet de travailler sur les chaînes de caractères ;
- purrr qui propose un remplacement de la famille **apply** par des fonctions dont le type en sortie est stable.

De nombreux autres packages sont développés dans cet esprit. On peut citer par exemple skimr qui permet d'obtenir des résumés statistiques du type de **summary** sous la forme de table, janitor pour nettoyer des tables, etc.

Quelques fonctions utiles

Nous concluons cette section par une petite liste de fonctions que nous utilisons dans cet ouvrage et qui peuvent être utiles ailleurs.

- Les fonctions **row_number**, **min_rank** et **dense_rank** de dplyr qui définissent le rang d'une observation dans une série avec respectivement une priorité à l'ordre d'apparition, un rang égal en cas d'égalité et des rangs contigus même en cas d'égalité.
- La fonction **top_n** de dplyr qui extrait les lignes correspondants aux plus grandes valeurs d'une variable sans les classer.
- Les fonctions **gather** et **spread** du package tidyr qui permettent de passer ou de pivoter des tableaux : **gather** regroupe les valeurs de différentes colonnes en deux colonnes, une de clé et une de valeurs, tandis que **spread** fait l'opération inverse.
- Les fonctions **str_c** et **str_sub** du package **stringr** qui permettent respectivement de coller des chaines de caractères ou d'extraire des sous parties de ces chaines de caractères, le tout de manière vectoriser.
- La fonction **map_df**, ou plus généralement **map_***, du package purrr qui fonctionne de manière similaire à **apply** tout en garantissant le type en sortie.

5.3 Bases de données

Les données requises pour les analyses sont souvent stockées dans des bases de données. La manière d'y accéder avec R dépend bien sûr du type de base utilisée. Nous verrons dans cette section les deux exemples les plus classiques : les bases de données de type SQL, qui répondent à des requêtes par des tables à récupérer dans R, et les bases où les données sont stockées sous forme JSON, un format plus souple qu'une table. Ce dernier format est utilisé par certaines bases de données NoSQL, comme MongoDB, et dans de nombreuses API web, par exemple celle du réseau de vélos de Rennes que nous allons utiliser.

5.3.1 SQL : Structured Query Language

DBI : un package pour les gouverner tous

Le package DBI (`https://www.r-dbi.org/`) offre une interface de communication entre R et différentes bases de données de type SQL (SQLite, MariaDB, PostgreSQL, etc.) à l'aide de pilotes dédiés. SQL est un langage commun permettant de commander de nombreuses bases de données. Bien que standardisé, il varie légèrement d'une base de données à l'autre par la présence d'extensions et l'absence de certaines fonctionnalités.

Le package DBI permet de se connecter à une base de données en précisant le nom de la base de données, le nom de l'hôte, le port à écouter ainsi que l'identifiant et

le mot de passe de l'utilisateur. Par exemple, dans le cas d'une base de données PostGreSQL, on peut utiliser :

```
> library(DBI)
> con <- dbConnect(RPostgres::Postgres(), dbname = 'DATABASE_NAME',
  host = 'HOST', port = 5432, user = 'USERNAME', password = 'PASSWORD')
```

Si la base de données est de type MariaDB, il suffit de remplacer dans ce qui précède la ligne `RPostgres::Postgres()` par `RMariaDB::MariaDB()`.

Dans la suite, nous privilégierons l'utilisation de SQLite [6]. Le code suivant permet de se connecter à une base de données éphémère qui est créée en mémoire vive (à l'aide du nom spécial `:memory:`) :

```
> library(RSQLite)
> con <- dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

On peut constater que cette base de données ne contient aucune table à l'aide de la commande :

```
> dbListTables(con)
character(0)
```

Il est possible d'en créer une simplement à partir d'un data-frame :

```
> df <- data.frame(x = runif(25),
                   label = sample(c("A", "B"), size = 25, replace = TRUE))
> dbWriteTable(con, name = "Exemple", value = df)
> dbListTables(con)
[1] "Exemple"
```

On peut également consulter la liste des colonnes (aussi appelées champs) disponibles dans une table donnée :

```
> dbListFields(con, "Exemple")
[1] "x" "label"
```

Enfin, il est possible de voir les données qui sont dans cette table à l'aide de la commande `dbReadTable(con, "Exemple")`.

6. L'avantage de travailler avec SQLite est que cette base de données n'est pas basée sur le paradigme client/serveur. Il est donc inutile d'installer un logiciel tiers de base de données pour expérimenter les fonctionnalités de base fournies par des systèmes plus sophistiqués. SQLite permet de travailler sur des bases de données stockées dans des fichiers voire directement en mémoire vive. C'est donc très simple à mettre en œuvre.

Premières fonctions utiles

Entrons maintenant dans le vif du sujet et voyons comment effectuer des requêtes SQL sur la base de données. Précisons tout de suite qu'il ne s'agit pas d'expliquer en détail la syntaxe SQL mais plutôt de voir comment s'en servir en interaction avec R.

Pour simplifier, SQL propose une syntaxe déclarative où l'on décrit ce que l'on souhaite obtenir et pas la manière de le faire. Le pseudo-code SQL `"SELECT j FROM df WHERE i GROUP BY by"` signifie ainsi : récupère `j` dans la table `df` si la ligne vérifie `i`, tout ceci se fait en groupant les lignes par `k`. On retrouve des motifs déjà vus : `df[i, j, by]` de data.table ou `df %>% group_by(by) %>% select(i) %>% transmute(j)` de dplyr. Cela n'est pas une surprise car ces deux packages sont fortement inspirés par les possibilités de traitements offertes par ce langage.

SQL propose bien sûr la fusion de tableaux et utilise pour cela le mot-clé `JOIN` que l'on retrouvera plus tard. Un exemple d'utilisation est le suivant :

```
> res <- dbSendQuery(con, "SELECT * FROM Exemple WHERE label = 'A'")
> res
<SQLiteResult>
  SQL  SELECT * FROM Exemple WHERE label = 'A'
  ROWS Fetched: 0 [incomplete]
       Changed: 0
> dbClearResult(res)
```

La dernière commande est utilisée afin de libérer les ressources locales et distantes liées au résultat de la requête. On notera d'ailleurs que `res` est un objet d'un type particulier : `SQLiteResult`. Il est possible de collecter (ramener vers R) les données à partir de cet objet à l'aide de la commande **dbFetch**. Un paramètre optionnel permet de limiter la collecte des données à un nombre choisi par l'utilisateur. Une utilisation classique revêt souvent la forme suivante :

```
> res <- dbSendQuery(con, "SELECT * FROM Exemple WHERE label = 'A'")
> while(!dbHasCompleted(res)){
  chunk <- dbFetch(res, n = 5)
  print(res)
  print(chunk[,2])
}
<SQLiteResult>
  SQL  SELECT * FROM Inutile WHERE label = 'A'
  ROWS Fetched: 5 [incomplete]
       Changed: 0
[1] "A" "A" "A" "A" "A"
<SQLiteResult>
  SQL  SELECT * FROM Inutile WHERE label = 'A'
  ROWS Fetched: 9 [complete]
```

```
       Changed: 0
[1] "A" "A" "A" "A"
> dbClearResult(res)
```

On remarque l'utilisation de la commande **dbHasCompleted**`(res)` pour vérifier si toutes les lignes ont été traitées. En effet, l'objet `res` contient l'information sur le nombre de lignes déjà collectées comme on peut s'en rendre compte dans l'exemple ci-dessus.

Pour terminer, il faut veiller à fermer la connexion à la base de données par la commande **dbDisconnect**`(con)`.

Quelques fonctions supplémentaires

Nous venons de présenter le « minimum vital ». Cependant d'autres fonctions sont fort utiles lors de l'utilisation de DBI. En voici une liste commentée et non exhaustive :

- **dbExistsTable**`(conn, name, ...)` permet de vérifier si la table nommée `name` existe pour la connexion `conn`. Le résultat est `TRUE` ou `FALSE`.
- **dbRemoveTable**`(conn, name, ...)` est utile pour effacer la table `name` de la connexion `conn`. Un booléen est retourné pour pouvoir s'assurer de la réussite de l'opération.
- **dbGetQuery**`(conn, statement)` permet de faire plusieurs opérations élémentaires successivement sur la connexion `conn`. Ainsi la requête `statement` est soumise et exécutée. Enfin, si la requête produit des données, elles sont collectées et retournées sous la forme d'un data-frame.
- **dbGetRowsAffected**`(res, ...)` : si `res` est le résultat d'une requête (effectuée par exemple via la fonction **dbGetQuery** vue ci-dessus), le nombre de lignes affectées (extraction, effacement, modification) est retourné.
- **dbGetRowCount**`(res, ...)` retourne le nombre de lignes collectées lors de la requête qui a produit le résultat `res`.

Exemple d'utilisation d'une base de données

Nous allons maintenant traiter un exemple complet. Le réseau STAR [7] propose un service de location de vélos collectifs appelé *LE vélo STAR*. Des données sur la topologie de l'ensemble des stations ainsi que sur la disponibilité des vélos dans ces stations sont disponibles en ligne (nous y reviendrons un peu plus tard). Pour les besoins de cet exemple, nous avons regroupé ces informations dans une base de données au format SQLite, appelée `LEveloSTAR.sqlite3` et disponible en ligne sur le site du livre.

Les lignes qui suivent permettent de se connecter à la base de données et de voir les tables qui la composent :

7. Le *Service des Transports en commun de l'Agglomération Rennaise* est le réseau de transport public de Rennes Métropole.

```
> con <- dbConnect(RSQLite::SQLite(), dbname = "LEveloSTAR.sqlite3")
> dbListTables(con)
[1] "Etat"      "Topologie"
```

Nous pouvons également consulter la liste des variables contenues dans chacune des deux tables :

```
> dbListFields(con, "Etat")
[1] "id"                 "nom"              "latitude"
[4] "longitude"          "nb_emplacements"  "emplacements_disponibles"
[7] "velos_disponibles"  "date"             "en_fonctionnement"
```

```
> dbListFields(con, "Topologie")
 [1] "id"                "nom"                 "adresse_numero"
 [4] "adresse_voie"      "commune"             "latitude"
 [7] "longitude"         "id_correspondance"   "mise_en_service"
[10] "nb_emplacements"   "id_proche_1"         "id_proche_2"
[13] "id_proche_3"       "terminal_cb"
```

Nous constatons que les deux tables contiennent des informations complémentaires, même si elles sont parfois redondantes comme la longitude et la latitude. Nous remarquons également la présence d'une variable `id` qui est un identifiant unique qui servira lors de jointures entre les tables.

À titre d'exercice nous allons afficher des informations — ici le nom et l'adresse — sur les trois stations les plus proches d'un usager qui arriverait à Rennes par la gare ferroviaire dont les coordonnées GPS sont $(48.103712, -1.672342)$. Pour que l'information soit utile, nous allons demander à ce que la station soit en fonctionnement et qu'au moins un vélo y soit disponible. Pour simplifier, nous nous contenterons de regarder la distance à vol d'oiseau.

```
> res <- dbGetQuery(con,
"SELECT left.id AS id,
        right.nom AS nom,
        (COALESCE(right.adresse_numero, '') ||
          ' ' ||
          COALESCE(right.adresse_voie, '')
        ) AS adresse,
        left.distance AS distance
FROM (SELECT id,
        POWER((latitude - 48.103712), 2.0) +
          POWER((longitude + 1.672342), 2.0) AS distance
        FROM Etat
        WHERE ((etat = 'En fonctionnement') AND (velos_disponibles > 0))
      ) AS left
LEFT JOIN Topologie AS right
```

```
ON (left.id = right.id)
ORDER BY distance
LIMIT 3")
  id               nom                   adresse     distance
1 15 Gares - Solférino      18 Place de la Gare 1.350434e-06
2 45 Gares Sud - Féval    19 B Rue de Châtillon 2.761371e-06
3 84  Gares - Beaumont 22 Boulevard de Beaumont 2.783296e-06
```

Remarquons que tous les calculs sont effectués du côté de la base de données, par exemple le calcul du carré de la distance, la concaténation du numéro de voie avec le nom de la rue, etc. L'utilisation de DBI ne dispense donc pas de la maîtrise du langage SQL et des spécificités de la base de données utilisée. En revanche, l'interface est simplifiée et unifiée entre les différentes bases pour lesquelles des pilotes sont disponibles. Pour terminer, n'oublions pas de nous déconnecter :

```
> dbDisconnect(con)
```

Et si je ne connais pas SQL ?

L'idéal est bien sûr de l'apprendre, c'est un langage très utile en science des données. Cependant, vous pouvez trouver un traducteur inattendu à l'aide de dplyr. En effet, dplyr permet en réalité de travailler sur des objets plus variés que les data-frames ou les tibbles. À l'aide d'extensions, il est compatible avec data.table et, ce qui va nous intéresser plus particulièrement, avec des bases de données SQL.

Tout commence en chargeant le package dplyr puis le package dbplyr. On le fait au travers de la commande usuelle :

```
> library(dplyr)
> library(dbplyr)
```

On initie ensuite une connexion avec la base de données et l'on construit, pour chaque table de la base, un objet avec lequel dplyr va pouvoir travailler :

```
> con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "LEveloSTAR.sqlite3")
> etat_db <- tbl(con, "Etat")
> topologie_db <- tbl(con, "Topologie")
```

Il est instructif de regarder la classe d'un objet construit à l'aide de la fonction **tbl** :

```
> class(etat_db)
[1] "tbl_dbi"  "tbl_sql"  "tbl_lazy" "tbl"
```

Il n'est pas très surprenant de voir que nous avons un objet de classe `tbl`, i.e. un tibble. Nous pouvons aussi facilement comprendre les deux chaînes de caractères

`tbl_dbi` et `tbl_sql` vu l'utilisation que nous avons faite du package DBI afin de nous connecter à une base de données de type SQL. Enfin nous comprendrons un peu plus tard la présence du mot paresseux (lazy) dans `tbl_lazy`.

Il est ensuite possible de travailler avec les deux `tbl` nouvellement créés, `etat_db` et `topologie_db`, presque comme avec des tibbles usuels (l'équivalent des data-frames dans le monde dplyr). Donnons un exemple très simple pour comprendre le principe en affichant les deux stations les plus au sud :

```
> req <- etat_db %>% arrange(latitude) %>% select(nom,latitude) %>% head(2)
> req
# Source:     lazy query [?? x 2]
# Database:   sqlite 3.22.0 [/path/to/LEveloSTAR.sqlite3]
# Ordered by: latitude
                nom latitude
              <chr>    <dbl>
1              Alma 48.08410
2            Italie 48.08674
```

Nous avons simplement utilisé la syntaxe habituelle de dplyr sans nous soucier si l'objet `etat_db` était un vrai tibble ou une table distante. Le résultat est cependant différent dans les deux cas. Ici, au lieu d'un tibble, on obtient un objet de type `lazy_query`. Celui-ci ressemble à une table mais son nombre de lignes ne semble pas connu. Pour comprendre ce résultat, il faut voir comment fonctionne dbplyr. Lorsqu'elle s'applique à un tableau distant, une suite de commande dplyr ne fait rien dans l'immédiat. Ce n'est que lorsque l'on demande un résultat que dplyr fait quelque chose : il prépare une requête SQL correspondant à la suite de commandes et l'envoie vers la base de données. Cette phase est retardée le plus possible afin de minimiser le nombre d'accès à la base. Ici, la requête n'est envoyé que lorsqu'on demande l'affichage `res`. Ce principe d'évaluation retardée se retrouve fréquemment dans R sous le nom d'évaluation paresseuse, d'où le lazy dans `tbl_lazy` pour les tables de dplyr se comportant de la sorte.

Le fait que le nombre de lignes du résultat ne soit pas connu s'explique également par le fait de minimiser les communications avec la base. En effet, par défaut, dbplyr se contente d'envoyer la requête et de récupérer les premières lignes du résultat, ce qu'il souligne par les points d'interrogation dans le nombre de lignes de la table. Si l'on souhaite récupérer le résultat complet, il faut le faire de manière explicite : avec **collect** :

```
> req %>% collect()
# A tibble: 2 x 2
  nom    latitude
  <chr>     <dbl>
1 Alma       48.1
2 Italie     48.1
```

On obtient alors un tibble dont la taille est bien sûr connue.

Notons qu'on peut voir quelle requête est réellement construite à l'aide de la fonction **show_query**. Sur le même exemple nous obtenons :

```
> req %>% show_query()
<SQL>
SELECT `nom` AS `nom`, `latitude` AS `latitude`
FROM (SELECT *
FROM `Etat`
ORDER BY `latitude`)
LIMIT 2
```

Nous constatons, même sur cet exemple simple, que la requête construite est plus complexe que nécessaire puisque la requête suivante produit le même résultat :

```
<SQL>
SELECT nom, latitude
FROM Etat
ORDER BY latitude
LIMIT 2
```

C'est le prix à payer pour la traduction. Il est toutefois complètement envisageable de faire des requêtes complexes et, dans une certaine mesure, de ne pas avoir à se soucier du code SQL généré. Ceci est d'autant plus vrai que la complexité du code n'impacte guère que sa lisibilité : des optimisations internes des bases permettent d'avoir des vitesses de traitement quasiment identiques.

Reprenons l'exemple plus complexe traité ci-dessus qui nécessitait une jointure des deux tables.

```
> etat_db %>%
  filter(etat=="En fonctionnement", velos_disponibles>0L) %>%
  mutate(distance2 = (latitude-48.103712)**2+(longitude+1.672342)**2) %>%
  arrange(distance2) %>% head(3) %>% select(id, distance2) %>%
  left_join(topologie_db, by=c("id" = "id")) %>%
  mutate(adresse = adresse_voie) %>%
select(id, nom, adresse, distance2) %>% collect()
# A tibble: 3 x 4
     id nom               adresse                distance2
  <int> <chr>             <chr>                      <dbl>
1    15 Gares - Solférino Place de la Gare      0.00000135
2    45 Gares Sud - Féval Rue de Châtillon      0.00000276
3    84 Gares - Beaumont  Boulevard de Beaumont 0.00000278
```

Il serait intéressant de regarder la requête générée par dplyr mais les contraintes d'une édition physique du livre nous obligent à vous priver de ce résultat !

Notons pour terminer que nous avons un peu triché. En effet l'adresse n'est pas formatée de la même manière que dans la version écrite directement en SQL. Il semble pourtant assez simple de le faire avec dplyr. Par exemple en remplaçant l'avant-dernière ligne :

```
mutate(adresse = adresse_voie) %>%
```

par la ligne :

```
mutate(adresse = paste(adresse_numero, adresse_voie)) %>%
```

Malheureusement on se heurte alors à une erreur et R proteste :

```
Error in result_create(conn@ptr, statement) : no such function: PASTE
```

Même si tout n'est pas absolument clair au premier coup d'œil, il semble que la fonction **paste** n'existe pas ! En effet, cette fonction n'existe pas... en SQL ! Et l'on comprend une des limites de la traduction : on ne peut pas utiliser toutes les fonctions usuelles de R comme on le ferait avec les `tbl_df`. Puisque la requête est exécutée côté base de données, il faut que les fonctions existent ou qu'elles aient été traduites.

Nous ne nous étendrons pas sur ce sujet mais il est préférable de connaître le langage SQL (et même la version spécifique à la base de données qu'on utilise) pour ne pas se retrouver face à une erreur difficilement compréhensible. Mentionnons seulement un problème classique qui a été contourné plus haut. On ne peut pas trouver la station la plus au sud à l'aide de la commande suivante car la fonction **min** n'existe pas en SQLite [8] :

```
> req <- etat_db %>% filter(latitude == min(latitude)) %>%
    select(nom, latitude)
```

Pour conclure, il nous semble que l'approche proposée par dplyr et son *backend* `dbplyr` est très intéressante et que, dans bien des cas, elle permet de se passer d'imbriquer du code R avec du langage SQL. Néanmoins, la compréhension du fonctionnement des bases de données permet d'appréhender certaines limites de dbplyr. De plus, la création de requêtes performantes passe nécessairement par un codage manuel.

8. Pour les lecteurs les plus tatillons, précisons que la fonction de fenêtrage **min** n'existe pas alors que la fonction d'agrégation **min**, elle, existe. Il est donc possible d'effectuer un code du type `etat_db %>% summarise(min = min(latitude))`. Précisons également que PostgreSQL permet d'utiliser des fonctions de fenêtrage comme **min**.

5.3.2 JSON : JavaScript Object Notation

Il existe bien d'autres façons de sauvegarder des informations que d'utiliser des bases de données relationnelles. L'une d'elle est d'encoder ces informations dans des fichiers textes structurés. Comme toujours, il existe maintes manières de répondre à ce problème depuis des conventions personnelles utilisées pour de tout petits projets jusqu'à l'utilisation de formats structurés normalisés. Les principaux formats qui se partagent cette tâche sont sans doute XML [9], YAML [10] et JSON [11].

XML est apparenté au très célèbre langage de balisage HTML, standard du web. YAML est bien connu des utilisateurs des notebooks R puisque les entêtes de ces fichiers sont précisément écrits dans ce format. JSON [12] est de son côté utilisé dans le célèbre logiciel `MongoDB`, base de données de type NoSQL. C'est aussi dans ce format qu'on peut récupérer les données disponibles dans le cadre de l'ouverture des données publiques de l'État et des administrations sur le site `https://www.data.gouv.fr/fr/` [13]. De même, *LE vélo STAR* propose des données en accès libre, dont celles utilisées dans l'exemple précédent. On peut même y accéder via une interface de programmation applicative (API) et obtenir certaines informations au format JSON en temps réel !

Nous utiliserons à nouveau les données du site *LE vélo STAR* pour comprendre comment utiliser le package jsonlite afin de manipuler au sein de R des données au format JSON.

À quoi ressemble un fichier JSON ?

Le format JSON se veut à la fois lisible par les humains et facilement interprétable par les machines. Il est donc articulé autour d'une syntaxe simple et d'un petit nombre de types de données. Ces types sont prédéfinis et non extensibles. Deux d'entre eux permettent d'organiser des données structurées, les autres sont des types simples.

Les deux types structurés sont les objets (proches des dictionnaires `Python` ou des listes de R) et les tableaux (proches des listes `Python` ou des vecteurs de R). Les objets permettent de structurer les données suivant un principe clé/valeur. Ils sont codés de la façon suivante :

```
{
  "clé1": valeur1,
  "clé2": valeur2,
  ...
}
```

9. Acronyme de *Extensible Markup Language*.
10. Acronyme récursif de *YAML Ain't Markup Language*.
11. Acronyme de *JavaScript Object Notation*.
12. Ou plus précisément sa version compilée BSON.
13. Des collectivités publiques et certaines sociétés participent également à ce mouvement d'accès large aux données appelé parfois « OpenData ».

Les listes permettent simplement de structurer des données de façon ordonnée et sont codées de la façon suivante :

```
[
  valeur1,
  valeur2,
  ...
]
```

Notons que, dans les listes, les clés sont entourées de double quotes. Pour compléter la description, il reste à préciser les types que peuvent prendre les valeurs dans les deux situations précédentes. Outre les deux types déjà présentés qui – à condition d'être patient ! – peuvent être imbriqués à l'infini, les types suivants sont licites :

– Chaîne de caractères : elle est entre `"..."`. Donnons un exemple :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach"
}
```

– Nombre : on l'écrit directement, comme dans l'exemple ci-dessous :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "nombre_enfants": 20
}
```

– `true` ou `false` : attention, les valeurs booléennes sont écrites en minuscules :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "compositeur": true
}
```

– `null` : si l'on ne souhaite pas donner de valeur à une clé il faut lui donner la valeur `null`. L'exemple ci-dessous n'est pas valide :

```
{
  "prénom": "Jean-Sébastien",
  "particule": ,
  "nom": "Bach"
}
```

Le suivant l'est :

```
{
  "prénom": "Jean-Sébastien",
  "particule": null,
  "nom": "Bach"
}
```

Donnons un exemple un peu plus évolué où les deux épouses de J. S. Bach ont été indiquées :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "épouses": [
    {
      "prénom": ["Maria", "Barbara"],
      "nom": "Bach"
    },
    {
      "prénom": ["Anna", "Magdalena"],
      "nom": "Wilcke"
    }
  ]
}
```

Comme on le voit, les règles de construction d'un objet JSON sont simples. Néanmoins, la possibilité d'imbriquer les différents types JSON entre eux rend le format très riche.

API *LE vélo STAR*

Dans la suite de cette section, nous allons présenter l'interface de programmation (API) qui permet, en temps réel, d'obtenir des données sur le service *LE vélo STAR*. De telles API se rencontrent fréquemment et ce qui est expliqué ici se généralise à de nombreuses situations.

Une rapide recherche sur le site `https://data.rennesmetropole.fr/` montre que deux bases de données vont nous intéresser : l'une contient les informations sur l'état des stations et l'autre sur leur topographie. En suivant l'un des liens [14] on remarque qu'on peut avoir accès à l'API. Une interface graphique permet de construire une requête et le fichier JSON généré par cette requête s'affiche en vis-à-vis de cette interface. Plus intéressant pour nous, on constate que cette requête prend la forme d'une URL spécifiquement formatée (en bas de la page). En visitant le lien fourni, le fichier JSON s'affiche dans une nouvelle page du navigateur.

Regardons de plus près la structure de l'URL. Elle est composée de trois parties :
- le nom de domaine : `https://data.rennesmetropole.fr/` ;
- le chemin d'accès à l'API : `api/records/1.0/search/` ;
- la requête, elle-même composée de plusieurs parties :
  - une indication du jeu de données à utiliser (le site en contient beaucoup...) : `?dataset=etat-des-stations-le-velo-star-en-temps-reel`,

14. `https://data.rennesmetropole.fr/explore/dataset/etat-des-stations-le-velo-star-en-temps-reel/`.

– séparées par des esperluettes `&`, une liste de facettes à inclure dans les résultats. Par exemple `&facet=nom&facet=etat` permet d'inclure le nom et l'état des stations dans les résultats.

Pour pouvoir utiliser ces données, la stratégie est claire : accéder à l'URL pour récupérer les données au format JSON puis utiliser un outil capable, sous R, de travailler avec de format. C'est là qu'entre en jeu le package jsonlite.

Le package jsonlite

Nous supposerons, dans les exemples suivants, que les packages tidyverse et jsonlite ont été chargés à l'aide de la fonction **library** :

```
> library(tidyverse)
> library(jsonlite)
```

Les principales commandes que nous allons utiliser sont **fromJSON**, qui sert à importer un fichier JSON en un data-frame, et **toJSON** qui permet l'opération inverse en transformant un data-frame en fichier JSON. Ces deux commandes seront complétées par les variantes **stream_in** et **stream_out** dont nous reparlerons plus loin. Pour l'instant regardons l'effet des deux commandes de base sur des exemples très simples.

```
> df <- tibble(
  x = c(0, pi),
  y = cos(x)
)
> toJSON(df)
[{"x":0,"y":1},{"x":3.1416,"y":-1}]
```

Le résultat est bien un fichier JSON qui contient un tableau de deux objets. Dans le paradigme individus/variables utilisé avec les data-frames, on constate que chaque individu est représenté par un objet JSON dont les clés sont les noms des variables. Le tableau contient autant d'objets que le data-frame contient d'individus.

Nous venons d'expliquer la correspondance effectuée entre les data-frames et les fichiers JSON. Vérifions que ça fonctionne bien en utilisant la commande **fromJSON** sur le résultat obtenu :

```
> df %>% toJSON() %>% fromJSON()
       x  y
1 0.0000  1
2 3.1416 -1
```

On retrouve bien le data-frame initial, ce qui est le résultat escompté. On perçoit en revanche que le format JSON permet plus de souplesse que les data-frames.

Voyons comment s'en sort jsonlite dans des situations plus délicates pour convertir certains fichiers JSON exotiques.

La situation la plus simple est celle où tous les objets JSON ne possèdent pas les mêmes clés.

```
> fromJSON('[{"x":1},{"y":2}]')
   x  y
1  1 NA
2 NA  2
```

Le data-frame créé contient toutes les variables qui apparaissent pour les différents individus. Des valeurs manquantes sont attribuées si nécessaire. Voyons un exemple un peu plus complexe :

```
> df1 <- fromJSON('[{"x":[1, 2, 3]},{"x":4}]')
> df1
        x
1 1, 2, 3
2       4
> df1$x
[[1]]
[1] 1 2 3

[[2]]
[1] 4
```

Ici, jsonlite a pu construire un data-frame car le tableau `[1,2,3]` a pu être interprété comme un vecteur de R. Ceci est lié au fait que, par défaut, la fonction **fromJSON** tente de simplifier les vecteurs. On peut forcer le comportement inverse :

```
> lll <- fromJSON('[{"x":[1, 2, 3]},{"x":4}]', simplifyVector = FALSE)
> lll
[[1]]
[[1]]$'x'
[[1]]$'x'[[1]]
[1] 1

[[1]]$'x'[[2]]
[1] 2

[[1]]$'x'[[3]]
[1] 3


[[2]]
```

```
[[2]]$'x'
[1] 4
```

On obtient alors une liste qui a une structure bien plus complexe. Donnons un dernier exemple et comparons les deux résultats obtenus :

```
> df2 <- fromJSON('[{"x":{"xa":1, "xb":2}},{"x":3}]')
> df2
     x
1 1, 2
2    3
```

```
> l12 <- fromJSON('[{"x":{"xa":1, "xb":2}},{"x":3}]',
    simplifyDataFrame = FALSE)
> l12
[[1]]
[[1]]$x
[[1]]$x$xa
[1] 1
[[1]]$x$xb
[1] 2

[[2]]
[[2]]$x
[1] 3
```

Si le nom des variables n'a pas d'importance, la première solution est sans doute préférable mais elle ne l'est plus si l'on souhaite conserver cette information.

Extraction de données à partir de l'API *LE vélo STAR*

Nous allons voir comment traiter les données fournies par l'API du site *LE vélo STAR*. Une première étape est de consulter le fichier fourni lors de la requête. On peut le faire directement en ligne en consultant l'URL spécialement formatée [15] donnée dans la section décrivant l'API juste au-dessus. Après avoir installé l'extension JSONView, dans Chrome ou Firefox, on obtient un formatage du code qui facilite la lecture du fichier :

```
{
  "nhits":55,
  "parameters":{...},
```

15. https://data.rennesmetropole.fr/api/records/1.0/search/?dataset=etat-des-stations-le-velo-star-en-temps-reel&rows=100&facet=nom&facet=etat&facet=nombreemplacementsactuels&facet=nombreemplacementsdisponibles&facet=nombrevelosdisponibles.

```
  "records":[
    {
      "datasetid":"etat-des-stations-le-velo-star-en-temps-reel",
      "recordid":"88a8871cd455b991aae541e2d60ebdab34840e3e",
      "fields":{
        "etat":"En fonctionnement",
        "lastupdate":"2018-08-29T06:52:05+00:00",
        "nombrevelosdisponibles":0,
        "nombreemplacementsactuels":16,
        "nom":"Musée Beaux-Arts",
        "nombreemplacementsdisponibles":16,
        "idstation":"5510",
        "coordonnees":[48.109601,-1.67408]},
      "geometry":{
        "type":"Point",
        "coordinates":[-1.67408,48.109601]},
      "record_timestamp":"2018-08-29T06:53:00+00:00"
    },
    ...
  ],
  "facet_groups": [...]
}
```

On réalise rapidement que le champs `records` contient un tableau de `nhits` objets présentant l'essentiel des informations qui nous intéressent (plus particulièrement dans le champs `fields`, sous la forme d'un objet). De plus ces `nhits` objets possèdent une même structure compatible avec le paradigme individus/variables.

La commande suivante permet donc d'aller chercher, en temps réel, l'information en ligne :

```
> url <- paste0(
  "https://data.rennesmetropole.fr/api/records/1.0/search/",
  "?dataset=etat-des-stations-le-velo-star-en-temps-reel",
  "&rows=100",
  "&facet=nom",
  "&facet=etat",
  "&facet=nombreemplacementsactuels",
  "&facet=nombreemplacementsdisponibles",
  "&facet=nombrevelosdisponibles"
)
> ll <- jsonlite::fromJSON(url)
> df <- ll$records$fields
```

Il resterait un peu de travail à effectuer sur le data-frame résultant pour obtenir exactement la même structure que la base de données `sqlite3` vue à la section précédente. Il est néanmoins possible de vérifier que l'on dispose bien des informations souhaitées sous le bon format :

```
> class(df)
[1] "data.frame"
> glimpse(df)
Observations: 55
Variables: 8
$ etat                          <chr>  "En fonctionnement", ...
$ lastupdate                    <chr>  "2018-08-29T07:34:05+00:00", ...
$ nombrevelosdisponibles        <int>  5, ...
$ nombreemplacementsactuels     <int>  16, ...
$ nom                           <chr>  "Musée Beaux-Arts",
$ nombreemplacementsdisponibles <int>  11, ...
$ idstation                     <chr>  "5510", ...
$ coordonnees                   <list> [<48.10960, -1.67408>, ...
```

Pour aller plus loin

ndjson

On constate que, dans de nombreux cas, le résultat d'une requête n'est pas exactement un fichier JSON valide. En effet, ce résultat contient un objet JSON (valide) par ligne. Comme dans l'exemple ci-dessous :

```
{"x":1,"y":2}
{"x":3,"y":4}
...
```

Ce format particulier est parfois appelé `ndjson` pour *Newline Delimited JSON*. La fonction **stream_in** est dédiée à la lecture de tels fichiers utilisés principalement par des serveurs de données. On doit passer comme argument obligatoire une connexion (par exemple sous la forme d'une URL) et tout argument valide pour la fonction **fromJSON**. L'utilisation typique de cette commande se fait donc sous l'une des formes :

```
> url <- "http://jeroen.github.io/data/diamonds.json"
> diamonds <- jsonlite::stream_in(url(url))
```

ou

```
> url <- "http://jeroen.github.io/data/nycflights13.json.gz"
> flights <- jsonlite::stream_in(gzcon(url(url)))
```

selon que le fichier est compressé ou pas !

mongolite

MongoDB est une base de données de type NoSQL : les données n'y sont plus présentées dans des tables liées entre elles (modèle relationnel). MongoDB utilise

une version compilée du format JSON [16] pour stocker l'information et les requêtes peuvent être formulées en Javascript en lieu et place de SQL.

L'objectif n'est absolument pas d'apprendre à construire des requêtes afin d'utiliser MongoDB mais simplement de voir qu'il est possible, au sein de R, de se connecter à une base de données MongoDB et d'en extraire des données au format JSON afin de les manipuler avec les outils vus dans cette section.

Il est assez simple de se connecter à une base de données MongoDB en spécifiant une URL :

```
> library(mongolite)
> username <- "readwrite"
> password <- "test"
> host <- "mongo.opencpu.org"
> port <- "43942"
> path <- "/jeroen_test"
> url <- paste0(
  "mongodb://",
  username, ":", password, "@",
  host, ":", port,
  path)
> m <- mongo("mtcars", url = url)
```

Cette base de données de test, disponible en ligne [17], peut avoir été utilisée par une autre personne effectuant des essais. Nous allons donc l'initialiser avec les données disponibles dans `mtcars`.

```
> if(m$count() > 0) m$drop()
> m$insert(mtcars)
```

Il est possible d'exporter l'ensemble des données vers un fichier JSON en utilisant la fonction **export** :

```
> m$export(file("dump.json"))
```

En réalité il s'agit plutôt d'un fichier au format `ndjson`. Il est donc tout à fait adapté de lire ce fichier à l'aide de la fonction **stream_in** du package jsonlite :

```
> df <- stream_in(file("dump.json"))
```

L'intérêt de cette méthode reste assez faible car MongoDB est surtout utilisé pour des bases de données très volumineuses. Il n'est alors ni très judicieux ni même parfois possible de copier l'intégralité de la base localement. Dans cette situation, il est préférable de faire des requêtes directement via MongoDB en utilisant la fonction **find** de mongolite. Néanmoins, pour des bases de données de taille raisonnable, ces méthodes peuvent vous épargner l'apprentissage de MongoDB !

16. Appelé BSON pour Binary JSON.
17. Nous invitons ceux d'entre vous qui disposent localement d'un serveur MongoDB à l'utiliser.

5.4 Web scraping

5.4.1 Introduction

Parfois nous n'avons pas la chance d'avoir à disposition une base de données bien formatée et bien pensée. Il est alors possible de se tourner vers le *web* pour trouver les précieuses informations dont nous avons besoin. Bien entendu, il importe avant tout de considérer des sites de confiance et de passer un peu de temps à vérifier l'information.

Dans la suite, nous allons faire appel à des données disponibles sur *Wikipedia* et nous nous bornerons [18] à vérifier que la page utilisée n'a pas été vandalisée avant de l'utiliser. L'objectif que nous nous fixons est de construire la liste de tous les acteurs (importants) ayant tourné avec Louis de Funès dans les années 1950 et de préciser dans quels films ils ont joué ensemble. Il s'agira de construire un data-frame, nommé `liste_acteurs`, contenant, pour chaque film, une ligne par acteur. Nous souhaitons enregistrer le prénom et le nom de chaque acteur ainsi que le titre du film. Nous définirons donc les variables `nom` et `titre`. Les deux premières lignes devraient être proches de l'exemple suivant :

```
> liste_acteurs %>% head(2)
# A tibble: 2 x 4
  nom           titre
  <chr>         <chr>
1 Henri Vidal   Quai de Grenelle
2 Maria Mauban  Quai de Grenelle
```

Ces informations peuvent facilement être trouvées en consultant la page `https://fr.wikipedia.org/wiki/Filmographie_de_Louis_de_Funès` et en suivant les liens de chaque film. On pourrait ainsi construire, à la main, une base de données adaptée à nos besoins. Outre le fait que la tâche de copier et coller dans un tableau les informations contenues dans ces pages est pour le moins fastidieuse, le risque de faire des erreurs n'est pas négligeable. Et si le besoin se faisait sentir d'obtenir une base de données similaire pour d'autres acteurs (disons plusieurs dizaines), il est pratiquement certain que l'intégrité des données ne pourrait être assurée.

Il faut donc trouver un moyen d'automatiser le plus possible la tâche. À ce stade il faut bien comprendre qu'une automatisation totale n'est pas possible. Chaque site possède sa propre structure et la façon dont sont présentées les données peut changer d'un site à l'autre voire d'une page à l'autre d'un même site ! Il va donc falloir étudier le code source des pages *web* qui contiennent les données qui nous intéressent et adopter une stratégie pour aller les récupérer. Mais avant de voir comment s'y prendre avec le package `rvest`, considérons une approche naïve pour comprendre ensemble la structuration d'une page *web*.

18. En réalité nous nous contenterons de vous inviter à le faire !

5.4.2 Approche naïve

Rappelons que notre navigateur *web* se *contente*[19] de transformer des fichiers textes en pages lisibles et bien présentées. Pour simplifier un peu, les informations structurées sont contenues dans un fichier `HTML` tandis que la présentation est codée dans des feuilles de style `css`. C'est donc dans le fichier source `HTML` que nous allons trouver les renseignements que nous voulons.

Une première tentative consiste donc à lire le fichier source et à le manipuler pour extraire les données recherchées. Puisque la fonction **readLines** permet d'accéder à une `URL`, nous pouvons simplement écrire :

```
> url_wikipedia <- "https://fr.wikipedia.org/"
> url_filmographie <- "wiki/Filmographie_de_Louis_de_Funès"
> url <- paste0(url_wikipedia, url_filmographie)
> data_html <- readLines(url, encoding="UTF-8")
```

La variable `data_html` contient alors un vecteur de chaînes de caractères. On peut regarder les premiers élements pour se convaincre qu'il s'agit bien du code `HTML`.

```
> data_html[1:5]
[1] "<!DOCTYPE html>"
[2] "<html class=\"client-nojs\" lang=\"fr\" dir=\"ltr\">"
[3] "<head>"
[4] "<meta charset=\"UTF-8\"/>"
[5] "<title>Filmographie de Louis de Funès - Wikipédia</title>"
```

Nous constatons même qu'il s'agit de lignes de l'en-tête du document. Il convient de trouver le début et la fin du corps du document, par exemple à l'aide des instructions :

```
> begin <- grep("<body", data_html)
> end <- grep("</body>", data_html)
> data_html <- data_html[(begin +1):(end-1)]
```

En inspectant le fichier `HTML`, nous constatons que la partie du code qui contient les informations qui nous intéressent est à peu près structurée de la façon suivante[20] :

```
<h3>Longs métrages</h3>
  <h4>Années 40<h4>
    <ul>
      <li>...</li>
      <li>...</li>
    </ul>
```

19. Au moins en première approximation et pour nos besoins.
20. On ne tient pas compte des attributs (identifiants et classes) des diverses balises.

Le but est donc de récupérer les informations (titre du film et lien vers la page dédiée qui contient notamment la distribution) qui sont à l'intérieur des balises `<li>...</li>`. On peut s'amuser à utiliser, comme ci-dessus, des expressions régulières pour arriver à nos fins. Par exemple la commande `li<- grep("<li>", data_html)` retourne un vecteur de tous les éléments qui contiennent `<li>`. À titre d'exemple, regardons :

```
> li<- grep("<li>", data_html)
> data_html[li[1]]
[1] "<ul><li><a href=\"/wiki/1946_au_cin%C3%A9ma\"
 title=\"1946 au cinéma\">1946</a> : <i><a
 href=\"/wiki/La_Tentation_de_Barbizon\"
 title=\"La Tentation de Barbizon\">La Tentation de
 Barbizon</a></i> de <a href=\"/wiki/Jean_Stelli\"
 title=\"Jean Stelli\">Jean Stelli</a> : le
 portier du cabaret <i>Le Paradis</i></li>"
```

Il s'agit bien du premier film dans la liste ! Mais c'est fastidieux et le résultat n'est pas très bien structuré car le document a été lu ligne à ligne : on a donc un bout de balise `<ul>` au début qui ne nous intéresse nullement. On se rend également vite compte que si plusieurs balises `<li>` se trouvaient sur la même ligne, des erreurs pourraient sans doute survenir [21]. Il existe une manière plus efficace de procéder grâce à un package adapté qui prend mieux en compte la structuration imposée par le code `HTML`.

5.4.3 Le package rvest

Installation

L'installation de rvest peut s'avérer plus délicate que la simple installation d'un package usuel. En effet rvest nécessite la présence d'outils dans le système d'exploitation, par exemple celle de `curl`. Heureusement les éventuels messages d'erreur de la commande

```
> install.packages("rvest")
```

indiquent assez clairement les outils nécessaires et les moyens de les installer pour les systèmes d'exploitation les plus usuels (Windows, MacOS et différentes distributions Linux). La lecture attentive des messages d'erreur est donc *obligatoire* !

Parcourir le DOM

Les données contenues dans un fichier `HTML` peuvent être représentées de façon arborescente. L'interface de programmation DOM [22] permet d'examiner et de mo-

21. Il est bien entendu possible de créer des expressions régulières plus sophistiquées pour pallier ces problèmes, mais ce n'est pas le but poursuivi ici...
22. Document Object Model, normalisée par le W3C.

difier (par exemple en insérant un nœud dans l'arbre) le contenu d'une page *web*. C'est de cette façon que des pages *web* dynamiques sont créées.

Le package rvest est capable de parcourir cet arbre et donc de récupérer des informations. La question qui se pose est donc de savoir comment parcourir cet arbre. Commençons par le commencement et lisons à nouveau notre URL :

```
> library(rvest)
> data_html <- read_html(url)
```

Contrairement à ce qui se passait avec la fonction **readLines**, la variable data_html contient des objets plus sophistiqués que de simples chaînes de caractères. Il est alors facile de sélectionner tous les titres des sections de niveau <h4> en utilisant la fonction **html_nodes** qui permet de retourner une liste de nœuds [23] en utilisant des sélecteurs css :

```
> data_html %>% html_nodes("h4") %>% head(1)
xml_nodeset (1)
[1] <h4>\n<span id="Ann.C3.A9es_1940"></span> ...
```

Le résultat semble encore abscons mais il est possible d'améliorer sa lisibilité en utilisant la fonction **html_text** qui formate correctement notre objet :

```
> data_html %>% html_nodes("h4") %>% head(1) %>% html_text()
[1] "Années 1940[modifier | modifier le code]"
```

Sélection de nœuds : css et XPath

Pour obtenir les noms et liens vers les pages *Wikipedia* de tous les films des années 40 avec Louis de Funès, on peut procéder comme suit :

```
> data_html %>%
  html_nodes('#mw-content-text > div >
              ul:nth-of-type(1) > li > i > a') %>%
  html_attrs()
[[1]]
                            href                               title
"/wiki/La_Tentation_de_Barbizon"       "La Tentation de Barbizon"
```

Par souci de concision, nous n'avons donné que le premier résultat mais la liste des 19 films est bien affichée. Si l'on souhaite obtenir la liste des 82 films des années 1950, on peut utiliser l'instruction :

23. On remarquera dans la sortie suivante que l'objet retourné est de type xml_nodeset.

```
> data_html %>%
  html_nodes('#mw-content-text > div >
              ul:nth-of-type(2) > li > i > a') %>%
  html_attrs()
[[1]]
                    href                    title
"/wiki/Quai_de_Grenelle"       "Quai de Grenelle"
```

Expliquons, sans trop entrer dans les détails, comment nous avons utilisé les sélecteurs `css` dans notre contexte. Les listes (balise `<ul>`) se trouvent toutes à l'intérieur de la balise `<div id="mw-content-text" ... >` qui possède un identifiant unique et qui est donc facilement... identifiée. Il y a ensuite une autre balise `<div>` qui *entoure* les balises `<ul>`. Ceci explique la première partie du sélecteur `#mw-content-text > div > ul`. Comme il y a plusieurs balises `<ul>` il faut être précis. En l'absence d'identifiant unique permettant de sélectionner sans coup férir la liste qui nous intéresse, il faut dire si l'on souhaite travailler avec la première liste, la deuxième, la troisième, etc. C'est ce que permet de faire le terme `:nth-of-type(2)` pour sélectionner la deuxième liste. On sélectionne enfin la balise `<a>` qui nous intéresse en indiquant le chemin à suivre pour y accéder à partir du nœud correspondant à la balise de liste. D'où la partie finale du sélecteur : `> li > i > a`. Enfin la fonction **html_attrs** permet d'obtenir les attributs du nœud sélectionné.

Notons qu'il est aussi possible d'utiliser `XPath` qui est un langage de requêtes conçu pour localiser des portions d'un document `XML`. Malgré une syntaxe plus lourde, cet outil permet de réaliser des sélections plus complexes que les sélecteurs `css`. Donnons un exemple simple afin de voir comment employer `XPath` avec rvest :

```
> data_html %>%
    html_nodes(xpath =
        '//*[@id="mw-content-text"]
        /div/ul[
          preceding::h4[span/@id="Années_1950"]
          and
          following::h4[span/@id="Années_1960"]]
        /li/i/a'
    ) %>%
    head(5) %>%
    html_text()
[1] "Quai de Grenelle"      "Le Roi du bla bla bla" "La Rue sans loi"
[4] "L'Amant de paille"     "Bibi Fricotin"
```

Si la plupart du temps on peut se contenter des sélecteurs `css`, il semble utile d'apprendre à utiliser `XPath` si l'on a un besoin intensif de faire du *web scraping*.

La distribution du film *Quai de Grenelle*

Rendons-nous sur la page dédiée au premier film de la liste : *Quai de Grenelle*. C'est assez simple :

```
> url_film <- "wiki/Quai_de_Grenelle"
> url <- paste0(url_wikipedia,url_film)
> data_html <- read_html(url)
```

La liste des acteurs est la liste qui suit immédiatement le titre de section *Distribution*. Puisque c'est la même chose pour tous les films, il va être simple d'écrire un code de sélection générique, dans le sens où il pourra servir pour d'autres films. Remarquons que ce titre de section est contenu dans une balise `<h2>` qui contient elle-même une balise `<span>` qui possède comme identifiant unique le mot-clé `Distribution`. On peut réaliser une telle sélection assez simplement en utilisant `XPath` :

```
> data_html %>%
  html_nodes(xpath = '(
        //*[@id="mw-content-text"]
        //ul[preceding::h2[span/@id="Distribution"]]
      )[1]/li/a[1]'
  ) %>%
  head(6) %>%
  html_text()
[1] "Henri Vidal"   "Maria Mauban"   "Françoise Arnoul"
[4] "Jean Tissier"  "Robert Dalban"  "Micheline Francey"
```

Donnons quelques explications. Le code `//*[@id="mw-content-text"]` permet de sélectionner tous les nœuds descendants d'une balise ayant pour identifiant `mw-content-text`. Le code `//ul[preceding::h2[span/@id="Distribution"]]` sélectionne alors tous les `<ul>` dans ce sous-arbre qui suivent une balise `<h2>` contenant une balise `<span>` identifiée par le mot-clé `Distribution`. Enfin, On sélectionne la première liste avec `[1]` puis, dans cette liste, toutes les premières balises `<a>` qui suivent une balise `<li>` avec le code `/li/a[1]`. Ceci est nécessaire pour ne pas sélectionner d'autres liens (regarder la ligne qui correspond à l'acteur Jean Tissier sur la page *Wikipedia*).

Solution du problème initial

Tout est maintenant en place pour résoudre notre problème. Nous allons donner directement le code complet qui permet d'obtenir le data-frame qu'on souhaitait construire. Les parties relatives à rvest ont été expliquées ci-dessus, le reste est très standard et ne mérite pas d'explication particulière.

```
> url_wikipedia <- "https://fr.wikipedia.org"
> url_de_funes <- "/wiki/Filmographie_de_Louis_de_Funès"
> url <- paste0(url_wikipedia, url_de_funes)
> data_html <- read_html(url)
> films <- data_html %>%
  html_nodes('#mw-content-text > div >
              ul:nth-of-type(2) > li > i > a') %>%
  html_attrs()
> library(tidyverse)
> liste_acteurs <- tibble()
> for(i in seq_along(films)){
  titre <- films[[i]][2]
  url_film <- films[[i]][1]
  url <- paste0(url_wikipedia, url_film)
  data_html <- read_html(url)
  acteurs <- data_html %>%
  html_nodes(xpath = '
    (
      //*[@id="mw-content-text"]
      //ul[preceding::h2[span/@id="Distribution"]]
    )[1]/li/a[1]'
  ) %>%
  html_text()
  liste_acteurs <- rbind(liste_acteurs,tibble(nom = acteurs,titre = titre))
}
```

Le résultat est un `tibble` contenant 2710 lignes et 2 colonnes. Si l'on souhaite savoir avec qui Louis de Funès a le plus joué, il suffit de le demander !

```
> liste_acteurs %>%
  group_by(nom) %>%
  summarise(n = n()) %>%
  arrange(desc(n)) %>%
  head(4)
# A tibble: 6 x 2
  nom                n
  <chr>          <int>
1 Louis de Funès    78
2 Paul Demange      16
3 Albert Michel     14
4 Paul Faivre       14
5 Bernard Musson    13
6 Charles Bayard    13
```

On constate qu'il s'agit de Paul Demange !

5.4.4 Pour aller plus loin

Données tabulaires

Lorsque des données sont encodées dans des tables `HTML`, il est facile de les extraire à l'aide de rvest. Il suffit d'identifier précisément le nœud où commence la table (balise `<table>`) et d'utiliser la fonction **html_table** qui retourne une liste de data-frames.

```
> url_wikipedia <- "https://fr.wikipedia.org"
> url_de_funes <- "/wiki/Louis_de_Funès"
> url <- paste0(url_wikipedia, url_de_funes)
> read_html(url) %>%
  html_nodes('#mw-content-text> div > div.infobox_v3.large > table') %>%
  html_table()
[[1]]
                X1                                                  X2
1 Nom de naissance            Louis Germain David de Funès de Galarza
2        Naissance                  31 juillet 1914Courbevoie (Seine)
3      Nationalité                                           Français
4            Décès 27 janvier 1983(à 68 ans)Nantes (Loire-Atlantique)
5       Profession                                    Acteur Pianiste
6   Films notables                                 voir filmographie
```

L'extrait de code précédent permet de récupérer les informations contenues dans l'encadré situé sur la page principale dédiée à Louis de Funès.

Simulateur de navigation

Il est également possible d'utiliser rvest pour simuler la navigation sur Internet. On peut donc par exemple essayer d'obtenir une représentation de la structure d'un site web complexe. Nous nous conterons de présenter les fonctions les plus utiles. Tout commence par une commande du type :

```
> session <- html_session(url_wikipedia)
> session
<session> https://fr.wikipedia.org/wiki/Wikip%C3%A9dia:Accueil_principal
  Status: 200
  Type:   text/html; charset=UTF-8
  Size:   135551
```

La fonction **html_session** sert à simuler le comportement d'un navigateur *web* et renvoie un objet de type `session`. Quelques informations sont indiquées comme l'adresse vers laquelle on a été redirigé.
Les fonctions de navigation **back** (qui simule un click sur le bouton *page précédente*), **jump_to** (qui sert à naviguer vers une URL relative ou absolue) et la fonction plus originale **follow_link** reçoivent comme premier argument un objet

de type `session`, ce qui fait qu'elle peuvent très simplement être chaînées avec la fonction **html_session**.

Donnons simplement quelques exemples :

```
> session %>% jump_to("Emanuel_Macron")
<session> https://fr.wikipedia.org/wiki/Emanuel_Macron
  Status: 404
  Type:   text/html; charset=UTF-8
  Size:   27286
```

renvoie comme statut 404, signe que la page n'existe pas. Effectivement, le prénom a mal été orthographié. L'erreur est corrigée dans le code ci-dessous :

```
> session %>% jump_to("Emmanuel_Macron")
<session> https://fr.wikipedia.org/wiki/Emmanuel_Macron
  Status: 200
  Type:   text/html; charset=UTF-8
  Size:   1145315
```

On remarque que, pour que le chaînage avec l'opérateur `%>%` fonctionne bien, ces fonctions retournent un objet de type `session`. Le résultat produit par le code ci-dessous est sans surprise :

```
> session %>% jump_to("Emmanuel_Macron") %>% back()
<session> https://fr.wikipedia.org/wiki/Wikip%C3%A9dia:Accueil_principal
  Status: 200
  Type:   text/html; charset=UTF-8
  Size:   135831
```

Enfin, la fonction **follow_link** est plus complexe car elle permet de suivre un lien en donnant une *référence* à ce lien. Cette référence peut être donnée de trois façons différentes. Avec simplement un nombre entier pour identifier le premier, le deuxième ou le six cent soixante sixième lien d'une page. On peut également utiliser des sélecteurs `css` ou des `XPath`. Enfin, on peut chercher une séquence de mots qui corresponde au texte affiché par le navigateur entre les balises `<a>` et `</a>`. Pour terminer, donnons un exemple de chaque utilisation :

```
> session <- html_session("https://fr.wikipedia.org/wiki/Emmanuel_Macron")
> session %>% follow_link(5)
Navigating to /wiki/Aide:Homonymie
<session> https://fr.wikipedia.org/wiki/Aide%3AHomonymie
  Status: 200
  Type:   text/html; charset=UTF-8
  Size:   317728
> session %>% follow_link("Mouvement des citoyens")
Navigating to /wiki/Mouvement_des_citoyens_(France)
```

```
<session> https://fr.wikipedia.org/wiki/Mouvement_des_citoyens_(France)
  Status: 200
  Type:   text/html; charset=UTF-8
  Size:   45722
> session %>% follow_link(css = "#mw-content-text >
                    div > p:nth-child(9) > a:nth-child(1)")
Navigating to /wiki/La_R%C3%A9publique_en_marche
<session> https://fr.wikipedia.org/wiki/La_R%C3%A9publique_en_marche
  Status: 200
  Type:   text/html; charset=UTF-8
  Size:   437129
```

5.5 Exercices

Exercice 5.1 (Les prénoms aux USA)
Le package babynames contient une table du même nom recensant, pour chaque année depuis 1880 et pour chaque sexe, les prénoms données à plus de 5 enfants aux USA.

1. Déterminer la liste des 10 prénoms les plus utilisés pour les garçons et les filles en l'an 2000 à l'aide des packages data.table et dplyr.

2. Déterminer la proportion représentée par les 10 prénoms les plus utilisés par année et sexe à l'aide de R, data.table et dplyr.

3. Déterminer l'évolution au cours des années de la proportion des 10 prénoms les plus utilisés en 2000 pour chacun des sexes à l'aide de R, data.table et dplyr.

4. Comment faire si les données sont stockées dans une base distante ?

Exercice 5.2 (Les tournois majeurs au tennis en 2013)
Des fichiers `csv` donnant des informations sur les quatre tournois majeurs de 2013 sont disponibles sur le site du livre. Les questions suivantes peuvent être traitées avec dplyr ou data.table. Les lecteurs les plus tenaces feront les deux versions.

1. Lire le fichier `FrenchOpen-men-2013.csv` qui contient les informations sur le tournoi masculin de Roland-Garros de 2013. Inspecter le data-frame obtenu. En faire un tibble ou un data-table en fonction du package que vous utiliserez pour les manipulations.

2. Construire la liste des joueurs `Player1`

3. Idem pour `Player2`.

4. Afficher tous les matches de Roger Federer.

5. Afficher le nom des demi-finalistes.

6. Combien y a t-il eu de points disputés en moyenne par match ?

7. On s'intéresse aux *aces*. Pour chaque tour afficher le plus petit nombre d'*aces* réalisé dans un match, le plus grand nombre et le nombre moyen.

8. Combien y a t-il eu de doubles fautes au total dans le tournoi.

9. Faire un histogramme du nombre de doubles fautes par match.

10. Représenter l'évolution du nombre moyen de doubles fautes par match au cours des différents tours.

11. Peut-on dire que le pourcentage de premier service a une influence sur le résultat ? On pourra faire une boîte à moustaches ainsi qu'un test pour répondre à la question ?

Exercice 5.3 (LE vélo STAR, encore !)

1. À l'aide du package jsonlite, importer dans `R` les données disponibles en ligne (voir plus haut dans ce chapitre page 165) dans les deux tables, topologie et état des stations.

2. Créer une table qui contient la liste des stations (avec l'identifiant et le nom), le nom de la station la plus proche et la distance (euclidienne) à cette station.

3. On se trouve au point de coordonnées (48.1179151,-1.7028661). Créer une table avec le nom des trois stations les plus proches classées par ordre de distance et le nombre d'emplacements libres dans ces stations.

Exercice 5.4 (Se cultiver par hasard)

Un honnête homme souhaite parfaire sa culture générale et décide à cette fin de parcourir les pages de *Wikipedia* au hasard. Sa stratégie est la suivante : tous les jours, en partant de la page d'accueil (`https://fr.wikipedia.org/`) il suivra un premier lien pris au hasard parmi les liens internes à wikipedia. Il lira attentivement la page que le sort lui aura permis de visiter puis renouvellera l'opération à partir cette page.
Curieux, il se demande combien de pages distinctes il aura lu, en moyenne, au bout de dix ans (sous l'hypothèse que le site ne soit pas modifié durant cette période). Nous allons l'aider grâce au package rvest :

1. Ouvrir une session vers la page d'accueil de *Wikipedia*.

2. Construire une table avec tous les liens contenus dans la page (ces liens sont dans une balise `<a>` et l'adresse du lien est donnée par l'attribut `href`).

3. Ne conserver que les liens internes qui ne débutent pas par `http`.

4. Aller sur une page prise au hasard (on pourra utiliser la fonction **sample**) et renouveler l'opération afin de construire un vecteur contenant les adresses des deux pages visitées.

5. À l'aide de ce qui précède construire un data-frame contenant les adresses visitées pendant 10 ans (soit 3652 jours).

6. En déduire le nombre de pages différentes visitées.

7. Renouveler ces opérations (combien de fois ?) pour se faire une idée de la distribution de ce nombre (en particulier de sa moyenne).

Exercice 5.5 (Un peu de musique)

Sur le site `https://github.com/lerocha/chinook-database`, on peut trouver des bases de données de bibliothèques musicales. Une copie de la base `sqlite` est disponible sur le site du livre.

1. Se connecter à la base de données.

2. En utilisant `dbplyr`, construire une table contenant les informations suivantes sur la `Playlist` appelée *classical* : le titre de chaque piste ainsi que le titre de l'album dont cette piste est tirée.

3. Même question en écrivant directement la requête en SQL.

Exercice 5.6 (Du trampoline sur Wikipedia)

Le trampoline est un sport olympique depuis les jeux de Sydney en 2000. La page `https://fr.wikipedia.org/wiki/Liste_des_m%C3%A9daill%C3%A9s_olympiques_au_trampoline`. donne accès à la liste de tous les médaillés.

1. Dans un premier data-frame, récupérer le tableau des médaillées féminines.

2. À partir de ce tableau, créer un nouveau data-frame contenant, pour chaque pays, le nombre de médailles d'or, d'argent et de bronze obtenues lors des différentes olympiades.

3. Classer ce data-frame dans l'ordre usuel en fonction tout d'abord du nombre de médailles d'or obtenues puis, pour départager les ex-aequo en fonction du nombre de médailles d'argent et enfin du nombre de médailles de bronze.

4. Mêmes questions pour le tableau masculin et enfin pour le tableau mixte.

On pourra comparer les résultats obtenus avec la page `https://fr.wikipedia.org/wiki/Trampoline_aux_Jeux_olympiques`

Exercice 5.7 (Débouchés dans l'enseignement supérieur français)

On peut trouver sur le site `https://data.gouv.fr` [24] un jeu de données sur l'insertion professionnelle des diplômés de l'enseignement supérieur français. Un fichier au format JSON est disponible sur le site du livre.

1. Importer les données contenues dans le champs `fields` sous la forme d'un data-table. Créer un nouveau data-table où, pour chaque établissement (variable `etablissement`), on renseignera :
- le salaire net médian des emplois à temps plein, toutes disciplines confondues (on fera par exemple la moyenne en veillant à prendre en compte le fait qu'il y a parfois des valeurs manquantes) ;
- le salaire net mensuel régional.

2. Classer le data-frame obtenu en fonction du ratio des deux valeurs renseignées.

24. Plus précisément à l'adresse `https://www.data.gouv.fr/fr/datasets/insertion-professionnelle-des-diplomes-de-master-en-universites-et-etablissements-assimil-0/`

Seconde partie

Fiches thématiques

Chapitre 6

Intervalles de confiance et tests d'hypothèses

Ce chapitre expose quelques méthodes classiques en statistique inférentielle : intervalle de confiance d'une moyenne (fiche 6.1), test du χ^2 d'indépendance entre deux variables qualitatives (fiche 6.2), test de comparaison de deux moyennes (fiche 6.3), test de conformité d'une proportion à une valeur donnée et test d'égalité de plusieurs proportions (fiche 6.4).

Dans chacune des fiches, on présente succinctement la méthode, puis un jeu de données et une problématique avant d'énumérer les principales étapes de l'analyse. L'exemple est alors traité via l'enchaînement des différentes instructions R et les résultats numériques et graphiques sont brièvement commentés.

Le site du livre `https://r-stat-sc-donnees.github.io/` permet de retrouver les jeux de données. On peut les sauvegarder sur sa machine avant de les importer en R ou les importer directement comme suit :

```
> read.table("https://r-stat-sc-donnees.github.io/Fiche.csv",
    header=TRUE, dec=".",sep=";")
```

Chaque fiche se voulant autonome, certaines redondances sont inévitables.

6.1 Intervalle de confiance d'une moyenne

Objet

L'objectif de cette fiche est de construire un intervalle de confiance pour la moyenne μ d'une variable quantitative X à partir des données $x_1, \ldots, x_n$ d'un échantillon. Dans ce qui est ici présenté, les données sont supposées être des réplications i.i.d. de X suivant une loi normale de moyenne μ et d'écart-type σ, paramètres tous deux inconnus. L'intervalle de confiance de niveau $(1-\alpha)$ pour μ s'écrit :

$$\left[\bar{x} - \frac{\hat{\sigma}}{\sqrt{n}} \times t_{1-\alpha/2}(n-1) \; ; \; \bar{x} + \frac{\hat{\sigma}}{\sqrt{n}} \times t_{1-\alpha/2}(n-1)\right] \tag{6.1}$$

avec $\bar{x}$ la moyenne d'échantillon, $\hat{\sigma} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$ l'estimation de σ et $t_{1-\alpha/2}(n-1)$ le quantile d'ordre $1-\alpha/2$ de la loi de Student à $(n-1)$ degrés de liberté. Rappelons que la validité de cette formule pour tout n suppose que les observations X_i sont issues d'une distribution gaussienne. En pratique, pour un petit échantillon, on teste la normalité des données avant d'appliquer la formule. Pour les grands échantillons (typiquement $n > 30$), le théorème central limite assure que la formule reste satisfaisante : il faut alors voir l'équation (6.1) comme un intervalle de confiance asymptotique.

La méthode présentée peut s'étendre au calcul de l'intervalle de confiance d'une proportion (voir la section « Pour aller plus loin »).

Exemple

Nous nous intéressons aux poids de poulpes femelles au stade adulte. Nous disposons pour cela d'un échantillon de données de 240 poulpes femelles pêchées au large des côtes mauritaniennes. Nous souhaitons connaître, pour la population mère, une estimation du poids moyen et un intervalle de confiance pour cette moyenne de niveau 95 %.

Étapes

1. Importer les données
2. Estimation des paramètres (moyenne, écart-type)
3. Distribution des données
4. Construire l'intervalle de confiance

Traitement de l'exemple

1. Importer les données

Nous importons les données avec la fonction **read.table** puis vérifions cette importation :

```
> don <- read.table("poulpeF.csv",header=T)
> summary(don)

     Poids
 Min.   :  40.0
 1st Qu.: 300.0
 Median : 545.0
 Mean   : 639.6
 3rd Qu.: 800.0
 Max.   :2400.0
```

La variable `Poids` est bien considérée comme quantitative puisque la fonction **summary** donne quelques statistiques de base : le minimum, la moyenne, les quartiles et le maximum des observations.

2. Estimation des paramètres (moyenne, écart-type)
Calculons la moyenne d'échantillon $\bar{x}$, estimation de μ, et $\hat{\sigma}$, estimation de σ :

```
> mean(don$Poids)
639.625
> sd(don$Poids)
445.8965
```

3. Distribution des données
Avant toute analyse, il est conseillé de représenter les données. Celles-ci étant quantitatives, on représente l'histogramme ainsi qu'un estimateur de la densité, avec la représentation graphique conventionnelle et avec ggplot2 (voir Fig. 6.1) :

```
> hist(don$Poids,main="Distribution des poids",nclass=12,freq=FALSE,
      xlab="Poids",ylab="")
> lines(density(don$Poids),col="red")
> require(ggplot2)
> ggplot(don)+aes(x=Poids,y=..density..)+geom_histogram(bins=12)+ylab("")+
    geom_line(stat="density",col="red")+ggtitle("Distribution des poids")
```

L'histogramme donne une idée de la distribution des poids : cette distribution asymétrique ne ressemble pas à celle d'une loi normale. Comme mentionné en introduction, ceci ne pose cependant pas de problème pour estimer μ vu la taille importante de la population étudiée ($n = 240$).

4. Construire l'intervalle de confiance
Le calcul de l'intervalle de confiance se fait à l'aide de la fonction **t.test** :

```
> t.test(don$Poids,conf.level=0.95)$conf.int
[1] 582.9252 696.3248
attr(,"conf.level")
[1] 0.95
```

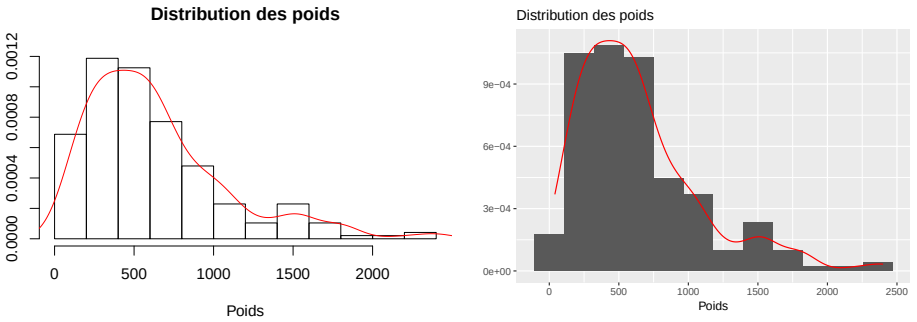


FIGURE 6.1 – Histogramme et estimateur de la densité pour les poids des poulpes ; à gauche la représentation graphique conventionnelle, à droite avec ggplot2.

La théorie de l'estimation garantit que, si on répète l'expérience (obtention d'un échantillon i.i.d. selon une loi $\mathcal{N}(\mu, \sigma^2)$ de taille n puis calcul de l'intervalle de confiance avec la formule (6.1)) un nombre « infini » de fois, alors asymptotiquement 95 % des intervalles calculés contiendront la moyenne μ. Cependant, il serait absurde de prétendre que l'intervalle $[583, 696]$ contient ici la « vraie » moyenne μ avec probabilité 0.95 : cette probabilité ne peut en effet valoir que 0 ou 1. Disons plutôt que la confiance que l'on a pour que cet intervalle contienne μ est de 95 %.

Pour aller plus loin

Pour construire un intervalle de confiance d'une proportion, on utilisera la fonction **prop.test** (voir aussi la fiche 6.4, p. 210, sur le test d'égalité de deux proportions). Ces méthodes classiques sont exposées dans de nombreux ouvrages comme Dagnelie (2007), Pagès (2010), Saporta (2011).

6.2 Test du χ^2 d'indépendance

Objet

L'objectif de cette fiche est de tester l'indépendance entre deux variables qualitatives. Plus précisément, disposant d'observations conjointes sur deux variables qualitatives, on teste l'hypothèse nulle H_0 : « les deux variables sont indépendantes » contre l'hypothèse alternative H_1 : « les deux variables ne sont pas indépendantes ». La mise en œuvre de ce test utilise la fonction **chisq.test** qui prend en entrée le tableau de contingence de dimension $I \times J$ qui contient les effectifs n_{ij} observés conjointement pour la i^{e} modalité de la première variable et pour la j^{e} modalité de l'autre.

Le test repose sur la confrontation des effectifs observés n_{ij} aux effectifs T_{ij} définis sous l'hypothèse H_0 d'indépendance : $T_{ij} = \frac{n_{i\bullet} n_{\bullet j}}{n}$ avec $n_{i\bullet} = \sum_{j=1}^{J} n_{ij}, i = 1, \ldots, I$ et $n_{\bullet j} = \sum_{i=1}^{I} n_{ij}, j = 1, \ldots, J$.

Cette hypothèse d'indépendance peut aussi s'énoncer comme suit : les distributions conditionnelles, qui donnent la répartition d'une variable dans les modalités de l'autre, sont toutes égales à la distribution marginale de cette variable. Par exemple, pour les lignes : $\forall i,\ T_{ij}/n_{\bullet j} = n_{i\bullet}/n$.

La statistique de test est

$$\chi^2_{obs} = \sum_{i=1}^{I} \sum_{j=1}^{J} \frac{(n_{ij} - T_{ij})^2}{T_{ij}}.$$

Sous réserve que les n_{ij} sont assez grands, χ^2_{obs} suit, sous l'hypothèse H_0, une loi du χ^2 à $(I-1) \times (J-1)$ degrés de liberté.

Exemple

Les données proviennent d'un exemple historique dû à Fisher : il donne la couleur des cheveux de garçons et de filles d'un district écossais :

	Blond	Roux	Châtain	Brun	Noir de jais
Garçon	592	119	849	504	36
Fille	544	97	677	451	14

Nous allons tester l'indépendance entre les variables `couleur des cheveux` et `sexe` avec une erreur de première espèce choisie égale à 5 %. Si nous rejetons l'hypothèse d'indépendance, nous aimerions avoir une idée des couleurs qui sont les plus dépendantes du sexe.

Étapes

1. Saisir les données
2. Visualiser les données
3. (Facultatif) Calculer les profils lignes et les profils colonnes
4. Construire le test du χ^2
5. Calculer les contributions au χ^2

Traitement de l'exemple

1. Saisir les données
Saisissons le jeu de données manuellement dans une matrice. Affectons-lui des noms de lignes (**rownames**) et de colonnes (**colnames**) :

```
> tab <- matrix(c(592,544,119,97,849,677,504,451,36,14),ncol=5)
> rownames(tab) <- c("Garçon","Fille")
> colnames(tab) <- c("Blond","Roux","Châtain","Brun","Noir de jais")
```

2. Visualiser les données
Nous choisissons de représenter la distribution de la couleur des cheveux par sexe, aussi appelée distribution conditionnelle de la variable `couleur des cheveux`. Pour cela, nous dessinons (voir Fig. 6.2) deux diagrammes en barres dans une même fenêtre graphique (cf. fonction **par** § 3.1.6, p. 69). Le vecteur `couleur` permet d'attribuer une couleur à chaque catégorie :

```
> par(mfrow=c(2,1))
> couleur <- c("Gold","OrangeRed","Goldenrod","Brown","Black")
> barplot(tab[1,],main="Garçon",col=couleur)
> barplot(tab[2,],main="Fille",col=couleur)
```

3. (Facultatif) Calculer les profils lignes et les profils colonnes
Il est intéressant de calculer la répartition en fréquences des couleurs de cheveux par sexe (le graphique 6.2 donne cette répartition en effectifs) : c'est ce qu'on appelle ici les profils lignes. Pour ce faire, on utilise la fonction **prop.table**, le paramètre `margin = 1` précisant que le conditionnement se fait selon les lignes :

```
> round(100 * prop.table (tab, margin = 1), 1)
       Blond Roux Châtain Brun Noir de jais
Garçon  28.2  5.7    40.4 24.0          1.7
Fille   30.5  5.4    38.0 25.3          0.8
```

Précisons que cette opération suppose que le tableau de contingence soit une matrice. Si ce n'était pas le cas, on transformerait le tableau via la fonction **as.matrix**.

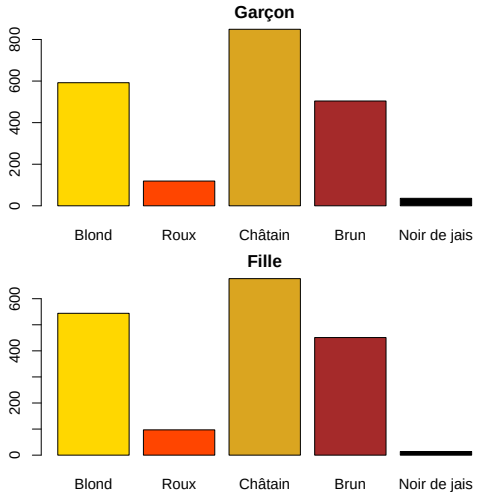


FIGURE 6.2 – Distributions des couleurs de cheveux par sexe.

En stipulant l'argument `margin = 2` dans l'application de la fonction **prop.table**, on obtient les profils colonnes décrivant la distribution conditionnelle de la variable `sexe`, c'est-à-dire la répartition Garçon/Fille pour chaque couleur de cheveux :

```
> round(100 * prop.table (tab, margin = 2), 1)
       Blond Roux Châtain Brun Noir de jais
Garçon  52.1 55.1    55.6 52.8          72
Fille   47.9 44.9    44.4 47.2          28
```

Il y a peu de différence entre les profils lignes si ce n'est que 1.7% des garçons ont des cheveux noirs de jais contre moitié moins pour les filles (0.8%). Les profils colonnes sont aussi assez semblables si ce n'est, à nouveau, pour la modalité `Noir de jais` : dans l'échantillon, 72% des individus ayant des cheveux noirs de jais sont des garçons et 28% sont des filles.

4. Construire le test du χ^2

C'est en effectuant le test que l'on décide si ces différences sont en contradiction avec l'hypothèse d'indépendance H_0, ce qui révélerait un lien entre les variables (rejet de H_0) ou au contraire si les différences peuvent être attribuées au hasard de l'échantillonnage (conservation de H_0).

Pour effectuer le test d'indépendance entre les variables `sexe` et `couleur de cheveux`, nous calculons la valeur de la statistique χ^2_{obs} sur les données et déterminons la probabilité critique :

```
> resultat <- chisq.test(tab)
> resultat

        Pearson's Chi-squared test

data:  tab X-squared = 10.4674, df = 4, p-value = 0.03325
```

Pour nos données, $\chi^2_{obs} \approx 10.47$. La probabilité critique, p-value, indique que, sous H_0, et pour un échantillon de même taille, la probabilité d'obtenir une valeur de χ^2_{obs} au moins aussi grande que celle-ci est d'environ 3 %. Cette probabilité critique étant inférieure au seuil fixé à 5 %, nous considérons de telles données comme peu probables sous H_0 et rejetons cette hypothèse. On considère donc que la couleur des cheveux dépend du sexe de l'enfant.

5. Calculer les contributions au χ^2
La fonction **chisq.test** produit une liste dont les composantes sont :

```
> names(resultat)
[1] "statistic" "parameter" "p.value" "method" "data.name" "observed"
[7] "expected"  "residuals" "stdres"
```

On retrouve en particulier la valeur χ^2_{obs} dans la composante `statistic` et le tableau fictif traduisant H_0 dans la composante `expected` :

```
> resultat$statistic
X-squared
 10.46745
> round(resultat$expected,1)
       Blond  Roux Châtain  Brun Noir de jais
Garçon 614.4 116.8   825.3 516.5           27
Fille  521.6  99.2   700.7 438.5           23
```

Quand on rejette l'hypothèse d'indépendance, il est intéressant de calculer la contribution des couples de modalités à la statistique du χ^2 pour voir les associations entre modalités qui contribuent aux écarts à l'indépendance.

On peut étudier plus en détail cette liaison en calculant les contributions $\frac{(n_{ij}-T_{ij})^2}{T_{ij}}$ à la statistique χ^2_{obs}. Les carrés des éléments fournis dans la composante `residuals` donnent ces contributions. En divisant chacun de ces carrés par le total χ^2_{obs} on obtient le pourcentage relatif à chaque cellule :

```
> round(100 * resultat$residuals^2 / resultat$statistic, 1)
       Blond Roux Châtain Brun Noir de jais
Garçon   7.8  0.4     6.5  2.9         28.4
Fille    9.2  0.5     7.7  3.4         33.4
```

Sans surprise, les combinaisons qui contribuent le plus aux écarts à l'indépendance sont celles concernant la couleur `Noir de jais`. Les contributions contenues dans la composante `residuals` sont par ailleurs associées à un signe qui indique si les effectifs observés sont supérieurs ou non aux effectifs attendus sous H_0 :

```
> round(resultat$residuals, 3)
        Blond   Roux Châtain   Brun Noir de jais
Garçon -0.903  0.202   0.825 -0.549        1.723
Fille  -0.979 -0.219  -0.896  0.596       -1.870
```

Le nombre de garçons ayant des cheveux noir de jais est plus important qu'attendu sous H_0 et le nombre de filles plus faible.

Pour aller plus loin

Les données peuvent provenir d'un tableau individus × variables. Pour notre exemple, on aurait en ligne un enfant, i.e. un individu statistique, et en colonnes les variables qualitatives, ici `couleur de cheveux` et `sexe`. On commence alors par construire le tableau de contingence, préférentiellement à l'aide de la fonction **xtabs** (voir § 2.6, p. 52). Une fois le tableau construit, on l'analyse comme précédemment :

```
> tab.cont <- xtabs(~cheveux+sexe, data=donnees)
> chisq.test(tab.cont)
```

Cette procédure est utile lorsqu'on effectue une ACM (i.e. lors de l'étude de la liaison entre plusieurs variables qualitatives, voir la fiche 7.3 p. 228) et qu'on étudie la liaison entre deux variables particulières.

Il est possible de visualiser les tableaux de contingence grâce à une Analyse Factorielle des Correspondances (AFC) en utilisant la fonction **CA** du package FactoMineR (fiche 7.2, p. 223).

Des rappels théoriques sur le test du χ^2 sont disponibles dans beaucoup d'ouvrages comme Dagnelie (2007), Pagès (2010), Saporta (2011).

6.3 Comparaison de deux moyennes

Objet

Nous présentons ici le test classique de comparaison de deux moyennes. Pour cela nous disposons d'observations d'une variable quantitative qui sont supposées avoir été tirées aléatoirement dans deux populations (notées 1 et 2) et ce de manière indépendante. Les moyennes de la variable dans chaque population, notées μ_1 et μ_2, sont inconnues et nous allons tester l'égalité de ces deux moyennes au regard des données collectées. Formellement, nous testons l'hypothèse $H_0 : \mu_1 = \mu_2$ contre l'hypothèse alternative $H_1 : \mu_1 \neq \mu_2$ dans le cas d'un test bilatéral. Le cas d'un test unilatéral de type $H_0 : \mu_1 \leq \mu_2$ contre $H_1 : \mu_1 > \mu_2$ sera également abordé.

Exemple

Nous allons comparer les poids de poulpes mâles et femelles au stade adulte. Nous disposons pour cela des données de poids pour 13 poulpes femelles (population notée 1) et 15 poulpes mâles (population notée 2) pêchés au large des côtes mauritaniennes. Le tableau 6.1 donne un extrait du jeu de données.

Poids	Sexe
300	Femelle
700	Femelle
850	Femelle
⋮	⋮
5400	Male

TABLE 6.1 – Extrait du jeu de données sur les poulpes (poids en grammes).

Dans le test présenté ici, les poids sont supposés être i.i.d. selon une loi $\mathcal{N}(\mu_1, \sigma_1^2)$ pour les femelles et $\mathcal{N}(\mu_2, \sigma_2^2)$ pour les mâles. En pratique, le respect de cette hypothèse s'accompagne des mêmes précautions liées à la taille des échantillons que celles mentionnées en fiche 6.1. Pour de petits échantillons ($n_1 < 30$ ou $n_2 < 30$), il convient de commencer par tester ces hypothèses de normalité, d'autres tests existant dans le cas où celles-ci ne sont pas vérifiées (cf. point 4). Pour des échantillons plus grands ($n_1 > 30$ et $n_2 > 30$), vérifier les hypothèses de normalité n'est pas nécessaire, la loi de la statistique de test étant approchée par une loi de Student comme dans le cas gaussien.

La statistique de test s'appuie également sur l'estimation des variances σ_1^2 et σ_2^2. Il est ainsi d'usage, en préalable au test de comparaison de moyennes, d'effectuer le test de comparaison des variances : $H_0 : \sigma_1^2 = \sigma_2^2$ contre $H_1 : \sigma_1^2 \neq \sigma_2^2$. En effet, si l'on conserve H_0 à cette étape, toutes les données sont utilisées pour estimer une variance commune.

Étapes

1. Importer les données
2. Comparer graphiquement les deux sous-populations
3. Estimer les statistiques de base dans chaque groupe
4. Tester la normalité des données
5. Tester l'égalité des variances
6. Tester l'égalité des moyennes

Traitement de l'exemple

1. Importer les données

```
> don <- read.table("poulpe.csv",header=T,sep=";")
> summary(don)
     Poids           Sexe
 Min.   : 300   Femelle:13
 1st Qu.:1480   Male   :15
 Median :1800
 Mean   :2099
 3rd Qu.:2750
 Max.   :5400
```

Le résumé du jeu de données fournit les statistiques de base, nous vérifions que la variable `Poids` est quantitative tandis que la variable `Sexe` est qualitative. Attention, si les modalités `Sexe` étaient codées numériquement (1 et 2 par exemple), il serait nécessaire de transformer cette variable en facteur préalablement à l'analyse, R la considérant comme quantitative (voir § 1.5.2, p. 10).

2. Comparer graphiquement les deux sous-populations
Commençons par visualiser les données. Les boîtes à moustaches permettent de comparer la distribution des poids dans chaque modalité de la variable `Sexe`. Les commandes suivantes permettent de le faire pour un graphe conventionnel et via ggplot2 :

```
> boxplot(Poids ~ Sexe, ylab="Poids", xlab="Sexe", data=don)
> require(ggplot2)
> ggplot(don)+aes(x=Sexe,y=Poids)+geom_boxplot()
```

La figure 6.3 montre que les mâles sont souvent plus lourds que les femelles puisque médiane et quartiles de poids sont supérieurs chez les mâles.

3. Estimer les statistiques de base dans chaque groupe
L'application de la fonction **aggregate** avec l'argument `FUN=summary` permet d'obtenir le résumé des données par modalité de `Sexe` :

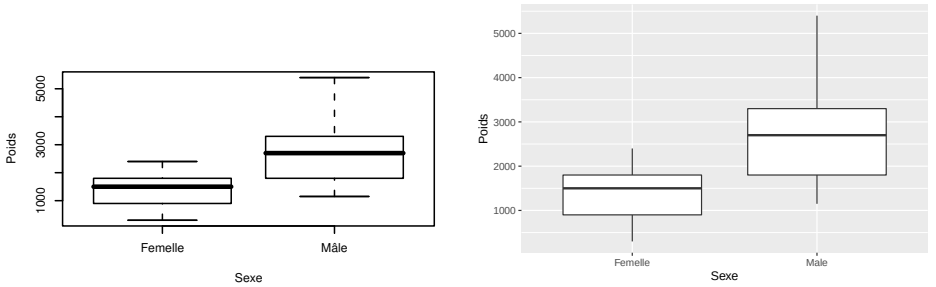


FIGURE 6.3 – Boîtes à moustaches des poids des poulpes par sexe ; à gauche la représentation graphique conventionnelle, à droite avec ggplot2.

```
> aggregate(don$Poids,by=list(don$Sexe),FUN=summary)
  Group.1   x.Min. x.1st Qu. x.Median   x.Mean x.3rd Qu.   x.Max.
1 Femelle  300.000   900.000 1500.000 1405.385  1800.000 2400.000
2    Male 1150.000  1800.000 2700.000 2700.000  3300.000 5400.000
```

Les résumés numériques sont toujours plus grands pour les mâles. La fonction **summary** ne renvoyant pas l'estimation de l'écart-type, nous combinons les fonctions **tapply** et **sd** pour calculer l'écart-type de `Poids` dans chaque modalité de `Sexe`. Précisons que l'argument `na.rm = TRUE` est simplement donné ici pour l'exemple car il n'y a pas de données manquantes :

```
> tapply(don$Poids,don$Sexe,sd,na.rm=TRUE)
  Femelle      Male
 621.9943 1158.3547
```

La variabilité des poids est presque deux fois plus élevée chez les mâles. On peut donc s'attendre à rejeter l'hypothèse d'égalité des variances au point 5.

4. Tester la normalité des données
Les tailles d'échantillons étant inférieures à 30, nous testons la normalité des données de poids dans chaque groupe. Détaillons ce test pour les mâles, la procédure étant analogue pour les femelles. On utilise ici le test de Shapiro-Wilk en sélectionnant au préalable les poids des mâles dans le jeu de données :

```
> select.males <- don[,"Sexe"]=="Male"
> shapiro.test(don[select.males,"Poids"])
        Shapiro-Wilk normality test

data:  don[select.males, "Poids"]
W = 0.93501, p-value = 0.3238
```

La probabilité critique associée au test étant supérieure à 5 %, on conserve l'hypothèse de normalité du poids des mâles. Le listing pour les femelles n'est pas fourni mais l'hypothèse de normalité est également conservée dans ce cas.

Quand l'hypothèse de normalité est rejetée, le test d'égalité des moyennes peut être effectué à l'aide de tests non-paramétriques tels que celui de Wilcoxon (**wilcox.test**) ou celui de Kruskal-Wallis (**kruskal.test**).

5. Tester l'égalité des variances
La statistique de test sur laquelle s'appuie le test de comparaison de moyennes diffère selon que les variances σ_1^2 et σ_2^2 peuvent être considérées comme égales ou non. Le test de l'hypothèse $H_0 : \sigma_1^2 = \sigma_2^2$ contre l'hypothèse alternative $H_1 : \sigma_1^2 \neq \sigma_2^2$ s'écrit comme suit :

```
> var.test(Poids ~ Sexe,conf.level=.95,data=don)
        F test to compare two variances

data:  Poids by Sexe
F = 0.2883, num df = 12, denom df = 14, p-value = 0.03713
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.0945296 0.9244467
sample estimates:
ratio of variances
         0.2883299
```

La probabilité critique associée à ce test est 0.037 : au seuil 5 %, nous rejetons H_0 et considérons que les variances sont significativement différentes. Notons que le listing fournit une estimation du rapport σ_1^2/σ_2^2 (`ratio of variances=0.2883299`), le numérateur correspondant par défaut à la variance empirique associée au premier niveau du facteur (ici `Femelle`). On dispose aussi d'un intervalle de confiance pour ce rapport (`95 percent confidence interval: 0.0945296 0.9244467`)

6. Tester l'égalité des moyennes
L'hypothèse de normalité ayant été vérifiée sur ces échantillons de petite taille, on utilise la fonction **t.test** en spécifiant que les variances sont différentes (argument `var.equal=FALSE`). Dès lors, c'est le test de Welch qui est mis en œuvre. Par défaut, c'est le test bilatéral (`alternative="two.sided"`) qui est construit par la fonction **t.test**, c'est-à-dire $H_0 : \mu_1 = \mu_2$ contre l'hypothèse alternative $H_1 : \mu_1 \neq \mu_2$. Comme au point précédent, la modalité prise comme référence est celle associée au premier niveau du facteur (ici `Femelle`).

```
> t.test(Poids~Sexe, alternative="two.sided", conf.level=.95,
      var.equal=FALSE, data=don)

        Welch Two Sample t-test
```

```
data:  Poids by Sexe
t = -3.7496, df = 22.021, p-value = 0.001107
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2010.624  -578.607
sample estimates:
mean in group Femelle    mean in group Male
             1405.385              2700.000
```

La probabilité critique associée au test (`p-value = 0.001107`) nous conduit à rejeter, au seuil 5 %, $H_0 : \mu_1 = \mu_2$ au profit de $H_1 : \mu_1 \neq \mu_2$. Ainsi, compte tenu des écarts-types, les moyennes observées dans l'échantillon (1405 g pour les femelles, 2700 g pour les mâles) sont suffisamment différentes pour que nous puissions considérer que les « vraies » moyennes μ_1 et μ_2 sont significativement différentes. On peut aussi tester si les femelles sont plus légères que les mâles contre l'alternative opposée, c'est-à-dire $H_0 : \mu_1 \leq \mu_2$ contre $H_1 : \mu_1 > \mu_2$. Ceci se fait en spécifiant `alternative="greater"`.

```
> t.test(Poids~Sexe, alternative="greater", conf.level=.95,
      var.equal=FALSE, data=don)

        Welch Two Sample t-test

data:  Poids by Sexe
t = -3.7496, df = 22.021, p-value = 0.9994
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -1887.471       Inf
sample estimates:
mean in group Femelle    mean in group Male
             1405.385              2700.000
```

Sans surprise, on accepte l'hypothèse H_0 selon laquelle les femelles sont plus légères que les mâles.

Pour aller plus loin

Si l'on souhaite comparer plus de deux moyennes, plusieurs tests sont envisageables. Le choix du test dépend de l'égalité ou non des variances dans chaque sous-population. Si les variances sont égales, on peut se reporter à la fiche sur l'analyse de variance à 1 facteur (p. 276) ; si elles sont différentes, on peut utiliser la fonction **oneway.test** (Welch, 1951). Le test de Bartlett (**bartlett.test**) permet quant à lui de tester l'égalité des variances.

Dans certaines situations, avant le recueil et l'analyse des données, on veut déterminer le nombre d'observations à effectuer dans chaque groupe de façon à

contrôler à la fois les erreurs de première et de seconde espèce. Si l'écart entre les vraies moyennes est suffisamment grand (prenons ici `delta=1`), alors on veut que le test de comparaison de moyennes mette en évidence la différence entre les sous-populations. Pour estimer cet effectif (`n`), il est nécessaire d'avoir une idée de l'écart-type intra-groupe `sd` (estimé par des expériences antérieures, supposons ici qu'il soit autour de 3) et il faut définir la puissance du test, prenons ici `power = 0.8`, i.e. la probabilité de détecter une différence de moyennes si cette différence existe. On utilise alors la fonction **power.t.test** :

```
> power.t.test(delta = 1, sd = 3, sig.level = 0.05, power = 0.8)
     Two-sample t test power calculation

              n = 142.2466
          delta = 1
             sd = 3
      sig.level = 0.05
          power = 0.8
    alternative = two.sided

NOTE: n is number in *each* group
```

Dans cet exemple, il faudrait donc avoir $n = 143$ observations par groupe pour avoir 80 % de chance de mettre en évidence une différence de moyennes si les vraies moyennes ont au moins un écart de 1 et si tous les écarts-types intra-groupes sont les mêmes et égaux à 3.

Si, au contraire, on envisage de faire seulement `n=100` observations par groupe, alors la puissance du test se calcule ainsi :

```
> power.t.test(n = 100, delta = 1, sd = 3, sig.level = 0.05)
     Two-sample t test power calculation

              n = 100
          delta = 1
             sd = 3
      sig.level = 0.05
          power = 0.6501087
    alternative = two.sided

NOTE: n is number in *each* group
```

Avec seulement 100 observations par groupe, on a donc 65 % de chance que le test de comparaison de moyennes mette en évidence une différence significative.

6.4 Tests sur les proportions

Objet

L'objet de cette fiche est de présenter le test de conformité d'une proportion à une valeur donnée ainsi que le test d'égalité de plusieurs proportions. Une situation associée au premier test est celle où l'on s'intéresse à la probabilité d'observer un caractère particulier dans une population et que l'on examine, au regard des données, l'hypothèse d'égalité de cette probabilité à une valeur donnée. Dans le second test, on cherche à comparer les probabilités d'observation d'un caractère particulier selon l'appartenance des individus à différentes catégories.

Exemple

Reprenons l'exemple historique concernant la couleur des cheveux de garçons et de filles d'un district écossais (voir fiche 6.2, p. 199) :

	Blond	Roux	Châtain	Brun	Noir de jais	Total
Garçon	592	119	849	504	36	2100
Fille	544	97	677	451	14	1783
Total	1136	216	1526	955	50	3883

Nous nous intéressons dans un premier temps à la répartition Garçon/Fille chez les individus Blonds. Dans ce cadre, nous testons l'égalité des proportions de filles et de garçons (ou l'égalité de la proportion de garçons à 50%, ce qui revient au même) sur la base de cet échantillon de taille $n = 1136$. Pour cela, nous appliquons la fonction **binom.test** qui met en œuvre le test dit « binomial exact » faisant intervenir la seule loi binomiale. Dans un second temps, nous nous intéresserons à tester l'égalité de la proportion de garçons dans les cinq catégories définies par la couleur des cheveux. Nous utilisons pour cela la fonction **prop.test** qui fait intervenir une statistique du χ^2 résultant d'une approximation asymptotique.

Étapes

1. Tester, chez les individus blonds, si la proportion de garçons est égale à 50 %
2. Tester l'égalité des proportions de filles pour les différentes couleurs de cheveux

Traitement de l'exemple

1. Tester, chez les individus blonds, si la proportion de garçons est égale à 50 %
Nous utilisons la fonction **binom.test** en lui donnant comme arguments le nombre de garçons (`x=592`), la taille de l'échantillon (`n=1136`), la proportion testée (`p=0.5`) et choisissons l'hypothèse alternative H_1 selon laquelle la proportion de garçons est différente de 50% :

```
> binom.test(x=592,n=1136,p=0.5,alternative="two.sided")

Exact binomial test

data:  592 and 1136
number of successes = 592, number of trials = 1136, p-value = 0.1631
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.4916142 0.5505297
sample estimates:
probability of success
             0.5211268
```

La probabilité critique est plus élevée que 5 % : pour ce seuil, nous conservons donc l'hypothèse H_0 selon laquelle la proportion de garçons est égale à 50 %. Nous obtenons aussi un intervalle de confiance sur cette proportion au niveau de confiance 95 % : $[0.492; 0.551]$.

2. Tester l'égalité des proportions de filles pour les différentes couleurs de cheveux
Ce test peut se faire grâce à la fonction **prop.test**. Lors de son appel, il suffit de fournir une liste avec le nombre de filles pour chaque catégorie puis une liste avec le nombre total d'individus par catégorie :

```
> prop.test(c(544,97,677,451,14),n=c(1136,216,1526,955,50))

5-sample test for equality of proportions without continuity correction

data:  c(544, 97, 677, 451, 14) out of c(1136, 216, 1526, 955, 50)
X-squared = 10.467, df = 4, p-value = 0.03325
alternative hypothesis: two.sided
sample estimates:
   prop 1    prop 2    prop 3    prop 4    prop 5
0.4788732 0.4490741 0.4436435 0.4722513 0.2800000
```

Au seuil 5 %, nous rejetons l'égalité de ces proportions (`p-value = 0.03325`). Nous voyons que la proportion de filles dans le groupe noir de jais est beaucoup plus faible que dans les autres groupes (voir la fiche 6.2 p. 199 pour plus de détails). Clairement, comme la variable `Sexe` n'a que deux modalités (Fille ou Garçon), ce test est en fait équivalent au test du χ^2 de la fiche 6.2 sur l'indépendance entre la couleur des cheveux et le sexe. Les résultats sont d'ailleurs les mêmes (voir par exemple la p-value).

Par défaut la fonction **prop.test** teste l'égalité des proportions dans les différentes catégories. Notons que l'on peut aussi tester l'égalité de ces proportions à une suite de proportions théoriques. Si nous souhaitons tester l'hypothèse $p = (p_1, p_2, p_3, p_4, p_5) = (0.5, 0.5, 0.4, 0.5, 0.3)$, il suffit de préciser dans l'appel de la fonction l'argument `p=c(0.5,0.5,0.4,0.5,0.3)`.

Chapitre 7

Analyses factorielles

Ce chapitre expose les principales méthodes d'analyse factorielle et leur mise en œuvre avec R. Ces techniques ont toutes pour objectif de synthétiser et de visualiser un tableau de données. Le choix de la méthode dépend alors du type de données à analyser : analyse en composantes principales pour des tableaux de variables quantitatives (fiche 7.1), analyse des correspondances pour les tableaux de contingence (fiche 7.2), analyse des correspondances multiples pour les tableaux de variables qualitatives (fiche 7.3), et enfin analyse factorielle multiple pour les tableaux dans lesquels les variables sont structurées en groupes (fiche 7.4).

Dans chacune des fiches, on présente succinctement la méthode, puis un jeu de données et une problématique avant d'énumérer les principales étapes de l'analyse. L'exemple est alors traité via l'enchaînement des différentes instructions R et les résultats numériques et graphiques sont brièvement commentés.

Le site `https://r-stat-sc-donnees.github.io/` permet de retrouver les jeux de données. On peut les sauvegarder sur sa machine avant de les importer en R ou les importer directement comme suit :

```
> read.table("https://r-stat-sc-donnees.github.io/Fiche.csv",
    header=TRUE, dec=".",sep=";")
```

Chaque fiche se voulant autonome, certaines redondances sont inévitables.

7.1 Analyse en Composantes Principales

Objet

L'objectif d'une Analyse en Composantes Principales (ACP) est de résumer et visualiser un tableau de données individus × variables. L'ACP permet d'étudier les ressemblances entre individus du point de vue de l'ensemble des variables et dégage des profils d'individus. Elle permet également de réaliser un bilan des liaisons linéaires entre variables à partir des coefficients de corrélations. Ces études étant réalisées dans un même cadre, elles sont liées, ce qui permet de caractériser les individus ou groupes d'individus par les variables et d'illustrer les liaisons entre variables à partir d'individus caractéristiques.

Nous conseillons l'utilisation de la fonction **PCA** du package FactoMineR plutôt que la fonction **princomp** qui est très simpliste. La fonction **PCA** permet l'ajout d'éléments supplémentaires et la construction simple et automatisée de graphiques. Elle est aussi accessible via une interface graphique disponible dans le package Factoshiny via la fonction **PCAshiny**. Cette interface permet de paramétrer la méthode et de construire des graphes interactifs. Nous y revenons en fin de fiche page 221.

Exemple

Le jeu de données concerne les résultats aux épreuves du décathlon lors de deux compétitions d'athlétisme qui ont eu lieu à un mois d'intervalle : les Jeux Olympiques d'Athènes (23 et 24 août 2004) et le Décastar (25 et 26 septembre 2004). Lors d'une compétition, les athlètes participent à cinq épreuves (100 m, longueur, poids, hauteur, 400 m) le premier jour, puis aux épreuves restantes (110 m haies, disque, perche, javelot, 1 500 m) le lendemain. Dans le tableau 7.1, on a recueilli, pour chaque athlète, ses performances dans chacune des 10 épreuves, son classement final, son nombre de points final et la compétition à laquelle il a participé.

Nom	100m	Long	Poids	Haut	400m	110m	Disq	Perc	Jave	1500m	Class	Nb pts	Comp
Sebrle	10.85	7.84	16.36	2.12	48.36	14.05	48.72	5.00	70.52	280.01	1	8893	JO
Clay	10.44	7.96	15.23	2.06	49.19	14.13	50.11	4.90	69.71	282	2	8820	JO
Karpov	10.5	7.81	15.93	2.09	46.81	13.97	51.65	4.60	55.54	278.11	3	8725	JO
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE 7.1 – Performances des athlètes lors des 10 épreuves du décathlon.

L'objectif de l'ACP sur ce jeu de données est de déterminer des profils de performances similaires : y a-t-il des athlètes plus endurants, plus explosifs... ? Et de voir

si certaines épreuves se ressemblent : lorsqu'un athlète est performant pour une épreuve, est-il plutôt performant pour une autre ?

Étapes

1. Importer le jeu de données
2. Choisir les variables et les individus actifs
3. Standardiser ou non les variables
4. Choisir le nombre d'axes
5. Analyser les résultats
6. Décrire de façon automatique les principales dimensions de variabilité
7. Retour aux données brutes

Traitement de l'exemple

1. Importer le jeu de données
Il est important ici de préciser que la première colonne correspond aux noms des individus (`row.names=1`) ; on spécifie aussi que l'on veut conserver le nom des variables tel que dans le tableau initial (`check.names=FALSE`) :

```
> decath <- read.table("decathlon.csv",sep=";",dec=".", header=TRUE,
    row.names=1, check.names=FALSE)
> summary(decath)
```

2. Choisir les variables et les individus actifs
Comme nous souhaitons déterminer des profils de performances, nous mettons en actif les variables correspondant aux performances des athlètes (les 10 premières variables). Le choix des variables actives est très important : ce sont ces variables, et uniquement ces variables, qui participent à la construction des axes de l'ACP, autrement-dit, seules ces variables sont utilisées pour calculer les distances entre individus. Nous pouvons ajouter en variables supplémentaires les variables quantitatives `nombre de points` et `classement`, ainsi que la variable qualitative `compétition`. Les variables supplémentaires sont très utiles pour aider à interpréter les axes.

On choisit aussi les individus actifs, i.e. ceux qui participent à la construction des axes. Ici, comme fréquemment, tous les individus sont considérés comme actifs.

3. Standardiser ou non les variables
Lors d'une ACP, nous pouvons centrer-réduire les variables ou seulement les centrer. Pour le jeu de données traité dans cette fiche, nous n'avons pas le choix, la réduction est indispensable car les variables ont des unités différentes. Quand les variables ont les mêmes unités, les deux solutions sont envisageables et impliquent deux analyses différentes. Cette décision est donc cruciale. La réduction permet

d'accorder la même importance à chacune des variables alors que ne pas réduire revient à donner à chaque variable un poids correspondant à son écart-type. Ce choix est d'autant plus important que les variables ont des variances très différentes.

Pour réaliser l'ACP, la fonction **PCA** peut être utilisée par lignes de commandes à l'aide du package FactoMineR ou grâce à une interface graphique via le package Factoshiny.

```
> library(FactoMineR)
> res.pca <- PCA(decath, quanti.sup=11:12, quali.sup=13)
```

Les variables 11 et 12 sont quantitatives supplémentaires et la variable 13 est qualitative supplémentaire. Ces variables ne sont donc pas prises en compte dans la construction des axes. Par défaut, les variables sont centrées-réduites : on parle alors d'ACP normée. Pour éviter la réduction des variables, on utilise l'argument `scale.unit=FALSE`. L'objet `res.pca` contient l'ensemble des résultats.

4. Choisir le nombre d'axes

Plusieurs solutions existent pour déterminer le nombre d'axes à analyser en ACP. La plus courante consiste à représenter le diagramme en barres des valeurs propres ou des inerties associées à chaque axe :

```
> barplot(res.pca$eig[,2],names=paste("Dim",1:nrow(res.pca$eig)))
```

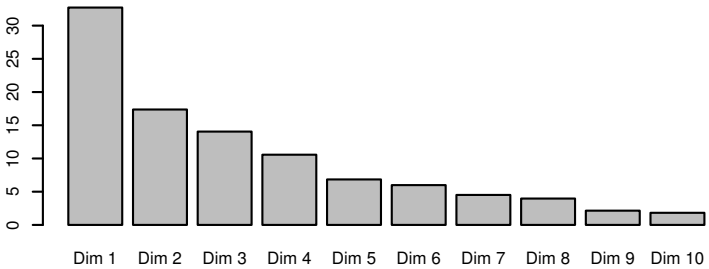


FIGURE 7.1 – Pourcentage d'inertie associée à chaque dimension de l'ACP.

Nous cherchons alors une décroissance ou une cassure apparente sur le diagramme. Ici, nous pouvons analyser les quatre premières dimensions. En effet, nous constatons après 4 axes une décroissance régulière des inerties et nous observons un petit « saut » entre 4 et 5.

Le tableau des pourcentages d'inertie expliquée par chaque axe est donné par la fonction **summary.PCA** (cf. section suivante). Les deux premiers axes expriment 50 % de l'inertie totale, c'est-à-dire que 50 % de l'information du tableau de données est contenue dans les deux premières dimensions. Cela signifie que la diversité des profils de performances ne peut être résumée par deux dimensions. Ici, les quatre premiers axes permettent d'expliquer 75 % de l'inertie totale.

5. Analyser les résultats

Pour interpréter les résultats d'une ACP, l'usage est d'étudier simultanément les résultats sur les individus et sur les variables. En effet, on veut caractériser les différents profils de performances à partir des épreuves.

La fonction **PCA** fournit par défaut le graphique des variables (Fig. 7.2) et celui des individus (Fig. 7.3) pour les deux premières dimensions. Les variables qualitatives supplémentaires sont renseignées sur le graphe des individus. Chaque modalité de la variable qualitative `compétition` est représentée au barycentre des individus qui prennent cette modalité. Les variables quantitatives supplémentaires apparaissent en pointillés sur le graphe des variables.

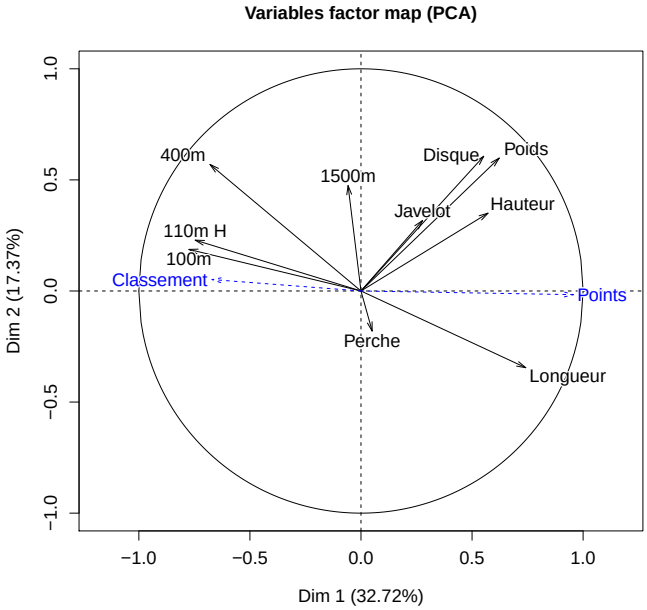


FIGURE 7.2 – ACP sur le décathlon : graphe des variables.

Dans l'ensemble, le premier axe oppose des profils de performance « uniformément élevée » (i.e. des athlètes « forts partout »), par exemple Karpov aux J.O., à des profils « (relativement !) faibles partout », comme Bourguignon au Décastar (cf. Fig. 7.3).

Notons sur la figure 7.2 que la corrélation négative entre la variable `longueur` et la variable `100m` (par exemple) s'interprète de façon particulière : celle-ci indique qu'un athlète qui court vite le 100 mètres, donc qui réalise un temps faible, réalise souvent de bonnes performances en longueur, en sautant loin.

La première composante principale est la combinaison linéaire des variables qui synthétise le mieux l'ensemble des variables. Dans cet exemple, la synthèse auto-

matique fournie par l'ACP coïncide presque avec le nombre de points, c'est-à-dire avec la synthèse officielle !

La position des modalités de la variable `compétition` sur le graphique des individus (voir Fig. 7.3) s'interprète comme suit : la modalité J.O. a une coordonnée positive sur l'axe 1 tandis que la modalité Décastar a une coordonnée négative. Ainsi, les athlètes ont en moyenne de meilleures performances aux Jeux Olympiques qu'au Décastar.

L'interprétation peut être affinée à l'aide des résultats numériques fournis par la fonction **summary.PCA**, qui peut être appelée indifféremment par **summary** ou **summary.PCA**. On précise ici que l'on veut les résultats pour les 2 premières dimensions (`ncp=2`) et les 3 premières lignes de chaque tableau (`nbelements=3`). Si on voulait les résultats pour tous les individus et toutes les variables, on écrirait `nbelements=Inf`.

```
> summary(res.pca, ncp=2, nbelements=3)
Call:
PCA(X = decathlon, quanti.sup = 11:12, quali.sup = 13)

Eigenvalues
                       Dim.1   Dim.2   Dim.3   Dim.4   Dim.5   Dim.6
Variance               3.272   1.737   1.405   1.057   0.685   0.599
% of var.             32.719  17.371  14.049  10.569   6.848   5.993
Cumulative % of var.  32.719  50.090  64.140  74.708  81.556  87.548

Individuals (the 3 first)
               Dist    Dim.1    ctr   cos2    Dim.2    ctr   cos2
SEBRLE      |  2.369 |  0.792  0.467  0.112 |  0.772  0.836  0.106 |
CLAY        |  3.507 |  1.235  1.137  0.124 |  0.575  0.464  0.027 |
KARPOV      |  3.396 |  1.358  1.375  0.160 |  0.484  0.329  0.020 |

Variables (the 3 first)
             Dim.1    ctr   cos2    Dim.2    ctr   cos2
100m       | -0.775 18.344  0.600 |  0.187  2.016  0.035 |
Longueur   |  0.742 16.822  0.550 | -0.345  6.869  0.119 |
Poids      |  0.623 11.844  0.388 |  0.598 20.607  0.358 |

Supplementary continuous variables
             Dim.1   cos2    Dim.2   cos2
Classement | -0.671  0.450 |  0.051  0.003 |
Points     |  0.956  0.914 | -0.017  0.000 |

Supplementary categories
              Dist    Dim.1   cos2 v.test    Dim.2   cos2 v.test
Decastar   |  0.946 | -0.600  0.403 -1.430 | -0.038  0.002 -0.123 |
JO         |  0.439 |  0.279  0.403  1.430 |  0.017  0.002  0.123 |
```

Pour chaque élément (individus, variables actives, variables qualitatives et quantitatives supplémentaires), nous avons, dimension par dimension, sa coordonnée, son cosinus carré (qui mesure la qualité de sa projection sur l'axe), et sa contribution à la construction de l'axe (quand l'élément est actif).

La fonction **plot.PCA**, appelée indifféremment par **plot** ou **plot.PCA**, permet d'améliorer la lisibilité des graphiques. Par exemple, en figure 7.3, on a modifié le graphe des individus (`choix="ind"`), colorié les individus en fonction d'une variable qualitative (`habillage=13`), augmenté la taille de la police des libellés (`cex = 1.1` au lieu de 1 par défaut), choisi de mettre un libellé uniquement aux individus ayant une qualité de projection sur le plan supérieure à 0.6 (`select="cos2 0.6"`) et modifié le titre du graphique :

```
> plot(res.pca, choix="ind", habillage=13, cex = 1.1,
    select="cos2 0.6", title="Graphe des individus")
```

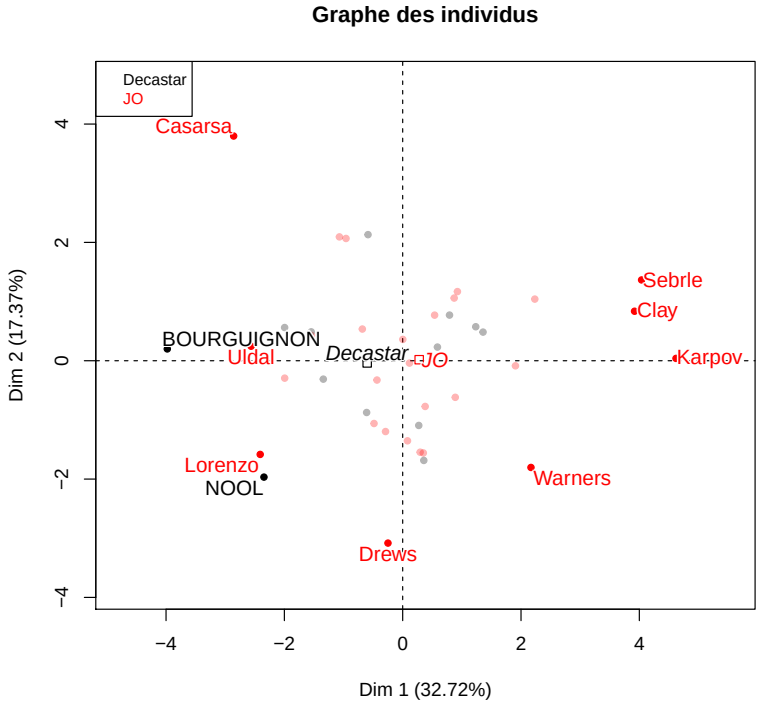


FIGURE 7.3 – ACP sur le décathlon : graphe des individus.

Après s'être intéressé aux résultats du premier plan factoriel (i.e. des deux premières dimensions), on peut construire les graphiques sur les axes 3 et 4 :

```
> plot(res.pca, choix="ind", habillage=13, axes=3:4, cex=0.7)
> plot(res.pca, choix="var", habillage=13, axes=3:4, new.plot=TRUE)
```

Pour une interprétation détaillée de ces axes, voir Husson et Pagès (2013a).

6. Décrire de façon automatique les principales dimensions de variabilité
La fonction **dimdesc** de FactoMineR permet de décrire automatiquement les axes de l'ACP, ce qui est très utile lorsqu'il y a beaucoup de variables. La fonction trie les variables quantitatives en fonction du coefficient de corrélation entre la variable quantitative et la composante de l'ACP et elle conserve les coefficients de corrélation significatifs.

Elle trie également les variables qualitatives ainsi que les modalités des variables qualitatives. Pour cela, pour chaque variable, un modèle d'analyse de variance à un facteur est construit, la variable correspondant à la composante principale (les coordonnées des individus) étant expliquée par la variable qualitative considérée. Le rapport de corrélation et la probabilité critique du test global (test de Fisher) sont calculés et les probabilités critiques associées à chaque test global sont triées par ordre croissant. Ainsi, les variables qualitatives sont triées de la plus caractérisante à la moins caractérisante.

Enfin, des tests sont construits sur chaque modalité de chaque variable qualitative (test de H_0 : $\alpha_i = 0$ avec la contrainte $\sum_i \alpha_i = 0$, voir la fiche 9.3, p. 276). Une estimation du coefficient α_i est calculée ainsi que la probabilité critique du test H_0 associé. Les probabilités critiques des tests sont triées par ordre croissant afin de trier les modalités de la plus caractérisante à la moins caractérisante.

Par défaut, les variables sont conservées quand la probabilité critique associée au test est inférieure à 5 % mais ce seuil peut être modifié. Ici, on donne les résultats pour la première dimension et un seuil de 20 % (`proba = 0.2`) :

```
> dimdesc(res.pca, proba = 0.2)
$Dim.1
$Dim.1$quanti
           correlation      P-value
Points       0.9561543 2.099191e-22
Longueur     0.7418997 2.849886e-08
Poids        0.6225026 1.388321e-05
Hauteur      0.5719453 9.362285e-05
Disque       0.5524665 1.802220e-04
Classement  -0.6705104 1.616348e-06
400m        -0.6796099 1.028175e-06
110m H      -0.7462453 2.136962e-08
100m        -0.7747198 2.778467e-09

$Dim.1$quali
                    R2   p.value
Competition 0.05110487 0.1552515
```

```
$Dim.1$category
           Estimate   P-value
JO         0.4393744 0.1552515
Decastar -0.4393744 0.1552515
```

On retrouve que le premier axe est lié à la variable `nombre de points` (coefficient de corrélation de 0.96), puis à la variable `100m` (corrélation négative de -0.77), etc. Avec un niveau de confiance de 80 %, la variable `Competition` caractérise le premier axe (ce qui ne serait pas le cas au niveau 95 %).

7. Retour aux données brutes
Il est utile de conforter son interprétation en revenant aux données brutes ou aux données centrées-réduites (sur les variables quantitatives) calculées comme suit :

```
> round(scale(decath[,1:12]),2)
         100m  Long Poids  Haut  400m ...
Sebrle  -0.56  1.83  2.28  1.61 -1.09 ...
Clay    -2.12  2.21  0.91  0.94 -0.37 ...
  ⋮       ⋮     ⋮     ⋮     ⋮     ⋮   ⋮
```

Par exemple, Sebrle lance le poids plus loin que la moyenne (valeur centrée-réduite positive) et ce de façon remarquable (valeur extrême car supérieure à 2).

De même on peut calculer la matrice des corrélations pour avoir les liaisons exactes entre les variables, et non plus les approximations fournies par le graphique :

```
> round(cor(decath[1:12,1:12]),2)
       100m  Long Poids  Haut  400m ...
100m   1.00 -0.60 -0.36 -0.25  0.52 ...
Long  -0.60  1.00  0.18  0.29 -0.60 ...
  ⋮     ⋮     ⋮     ⋮     ⋮     ⋮   ⋮
```

L'interface graphique Factoshiny

Le package Factoshiny offre une interface graphique permettant de construire des graphiques interactifs. Cette interface permet d'une part de paramétrer la méthode, en choisissant par exemple les variables supplémentaires, et d'autre part d'améliorer les graphiques en jouant sur la taille de la police, la sélection des éléments pour lesquels les libellés sont écrits, l'habillage des individus par les modalités d'une variable qualitative, etc. Après avoir chargé le package, il suffit d'appliquer la fonction **PCAshiny** sur le jeu de données pour ouvrir l'interface graphique et accéder aux paramétrages.

```
> library(Factoshiny)
> res.shiny <- PCAshiny(decath)
```

Pour aller plus loin

Il est possible de construire des ellipses de confiance autour des modalités d'une variable qualitative supplémentaire (cf. Fig. 7.4). Ces ellipses donnent alors des régions de confiance sur la position de chacune des modalités. Pour construire ces ellipses, on écrit :

```
> plotellipses(res.pca)
```

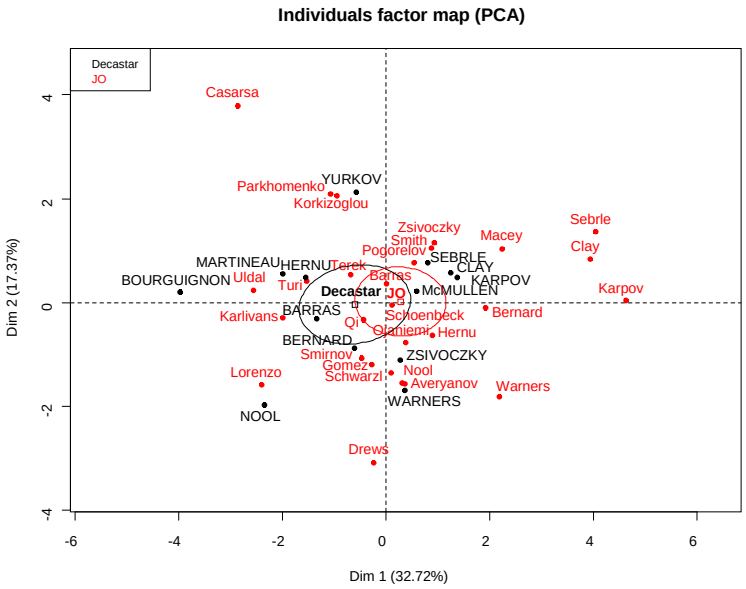


FIGURE 7.4 – Ellipses de confiance autour des modalités d'une variable qualitative.

Notons que par défaut la fonction **PCA** remplace les données manquantes d'une variable par la moyenne de cette variable calculée sur les données observées. Cette stratégie n'est pas idéale et nous renvoyons le lecteur à la fiche 11.1 page 374 et le package missMDA sur la gestion des données manquantes.

Des rappels théoriques et des applications pratiques sur l'ACP sont disponibles dans beaucoup d'ouvrages comme Escofier et Pagès (2016) ou Lebart *et al.* (2006). Dans Husson *et al.* (2016) des études de cas effectuées avec FactoMineR sont également détaillées.

7.2 Analyse Factorielle des Correspondances

Objet

L'Analyse Factorielle des Correspondances (AFC) permet de résumer et de visualiser un tableau de contingence, c'est-à-dire un tableau croisant deux variables qualitatives. Ce tableau donne, au croisement de la ligne i et de la colonne j, le nombre d'individus prenant la modalité i de la première variable et j de la seconde. Les objectifs de l'AFC sont de comparer les lignes entre elles, de comparer les colonnes entre elles et d'interpréter les positions entre les lignes et les colonnes, autrement dit de visualiser les associations des modalités des deux variables.

Pour l'AFC, on utilise la fonction **CA** du package FactoMineR, package dédié à l'analyse factorielle.

Exemple

Le jeu de données représente le nombre d'étudiants des universités françaises par discipline et par cursus selon le sexe lors de l'année 2007-2008. Le tableau croise les variables qualitatives `Discipline` et `Niveau-sexe`. Il comprend en lignes les 10 disciplines de l'université et en colonnes les croisements des variables `niveau` (licence, master et doctorat) et `sexe` (homme et femme). L'AFC est alors appliquée entre une variable (`Discipline`) et le croisement de deux variables (`Niveau-sexe`), ce qui est fréquent en AFC. Nous disposons de plus par discipline du nombre total d'étudiants par niveau, par sexe et du total global (cf. Tab. 7.2).

	Licence		Master		Doctorat		...	Total
	F	H	F	H	F	H		
Droit, sc. politiques	69373	37317	42371	21693	4029	4342	...	179125
Sc. eco., gestion	38387	37157	29466	26929	1983	2552	...	136474
Administration eco. et sociale	18574	12388	4183	2884	0	0	...	38029
Lettres, sc. du langage, arts	48691	17850	17672	5853	4531	2401	...	96998
Langues	62736	21291	13186	3874	1839	907	...	103833
Sc. humaines et sociales	94346	41050	43016	20447	7787	6972	...	213618
Pluri-lettres-langues-sc. humaines	1779	726	2356	811	13	15	...	5700
Sc. fondamentales et applications	22559	54861	17078	48293	4407	11491	...	158689
Sc. de la nature et de la vie	24318	15004	11090	8457	5641	5232	...	69742
STAPS	8248	17253	1963	4172	188	328	...	32152

TABLE 7.2 – Données sur les universités.

Étapes

1. Importer le jeu de données
2. Choisir les lignes et les colonnes actives

3. Réaliser l'AFC
4. Choisir le nombre d'axes
5. Visualiser les résultats

Le but de cette étude est d'avoir une image de l'université. Quelles sont les disciplines pour lesquelles le profil des étudiants est le même ? Quelles sont les disciplines privilégiées par les femmes (resp. les hommes) ? Quelles sont les disciplines pour lesquelles les études sont plus longues ?

Traitement de l'exemple

1. Importer le jeu de données

Il est important ici de préciser que la première colonne correspond aux noms des lignes (`row.names=1`). On s'assure de la bonne importation des données grâce à la fonction **summary** ou par la visualisation d'un extrait du jeu de données.

```
> univ <- read.table("universite.csv",sep=",", header=T, row.names=1)
> summary(univ)
> univ[1:4,1:3]
                                    Licence.F Licence.H Master.F
Droit, sciences politiques              69373     37317    42371
Sciences economiques, gestion           38387     37157    29466
Administration economique et sociale    18574     12388     4183
Lettres, sciences du langage, arts      48691     17850    17672
```

2. Choisir les lignes et les colonnes actives

Le choix des lignes et colonnes actives est très important car seules ces lignes et colonnes participent à la construction des axes. Comme nous souhaitons comparer l'ensemble des disciplines, nous allons considérer les disciplines comme lignes actives. Les colonnes de la variable `Niveau-sexe` seront considérées comme actives et les colonnes 7 à 12, correspondant aux divers totaux, seront considérées comme illustratives (l'information contenue dans ces variables est déjà prise en compte par les colonnes actives).

3. Réaliser l'AFC

Construisons maintenant l'AFC à l'aide de la fonction **CA** du package FactoMineR en précisant que les colonnes 7 à 12 sont illustratives :

```
> library(FactoMineR)
> res.ca <- CA(univ, col.sup=7:12)
```

L'objet `res.ca` contient l'ensemble des résultats qui peuvent être résumés par la fonction **summary.CA**, qui peut être appelée indifféremment par **summary** ou **summary.CA**. On précise ici que l'on veut les résultats pour les 2 premières dimensions (`ncp=2`) et pour les 3 premières lignes de chaque tableau (`nbelements=3`).

Pour avoir les résultats sur tous les individus et toutes les variables, on écrirait `nbelements=Inf`.

```
> summary(res.ca, nb.dec=2, ncp=2, nbelements=3)
Call: CA(X = univ, col.sup = c(7:12))

The chi square of independence between the two variables is equal
to 170789.2 (p-value =  0 ).

Eigenvalues
                       Dim.1  Dim.2  Dim.3  Dim.4  Dim.5
Variance                0.12   0.03   0.02   0.00   0.00
% of var.              70.72  15.51  10.90   2.63   0.25
Cumulative % of var.   70.72  86.23  97.13  99.75 100.00

Rows (the 3 first)
                       Iner*1000 Dim.1  ctr cos2   Dim.2   ctr cos2
Droit, sc. politiques |  5.72 | -0.10 1.45 0.30 |  0.07  2.86 0.13 |
Sciences éco. gestion |  9.78 |  0.18 3.85 0.46 | -0.02  0.12 0.00 |
Admin. éco. et soc.   |  6.43 | -0.19 1.10 0.20 | -0.37 20.03 0.80 |

Columns (the 3 first)
               Iner*1000   Dim.1   ctr  cos2   Dim.2   ctr  cos2
Licence.F      |   48.35 | -0.35 39.72  0.96 | -0.04  2.27  0.01 |
Licence.H      |   24.31 |  0.23 11.51  0.55 | -0.20 37.49  0.39 |
Master.F       |   16.11 | -0.11  1.99  0.14 |  0.17 20.90  0.33 |

Supplementary columns (the 3 first)
                 Dim.1  cos2   Dim.2  cos2
Total.F        | -0.26  0.96 |  0.05  0.03 |
Total.H        |  0.37  0.96 | -0.07  0.03 |
Licence        | -0.12  0.55 | -0.10  0.40 |
```

Par défaut, un test du χ^2 est construit pour tester l'indépendance des variables `Discipline` et `Niveau-sexe` sur les modalités (lignes et colonnes) actives. Sur notre exemple, l'hypothèse d'indépendance est rejetée puisque la probabilité critique est très proche de 0 : `The chi-square of independence between the two variables is equal to 170789.2 (p-value = 0).`

Ainsi il existe des associations entre certaines disciplines et certaines combinaisons niveau-sexe, qu'une AFC permet de visualiser.

4. Choisir le nombre d'axes

Pour déterminer le nombre d'axes à analyser, on se base sur la décomposition de l'inertie associée à chaque axe. Celle-ci est donnée en début de sortie précédente et l'on peut représenter le diagramme en bâtons de cette décomposition (Fig. 7.5).

```
> barplot(res.ca$eig[,2],names=paste("Dim",1:nrow(res.ca$eig)))
```

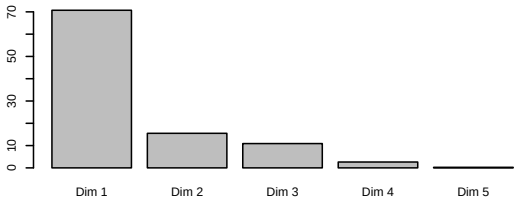


FIGURE 7.5 – Pourcentage d'inertie associée à chaque dimension de l'AFC.

Les trois premiers axes expriment presque 97 % de l'inertie totale : autrement dit, 97 % de l'information du tableau de données est résumée par les trois premières dimensions. Nous pouvons donc nous contenter de décrire ces trois premiers axes.

5. Visualiser les résultats
La fonction **CA** fournit par défaut le graphique avec la représentation simultanée des lignes et des colonnes (actives et illustratives) sur le plan principal. La figure 7.6 permet d'apprécier les grandes tendances qui se dégagent de l'analyse.

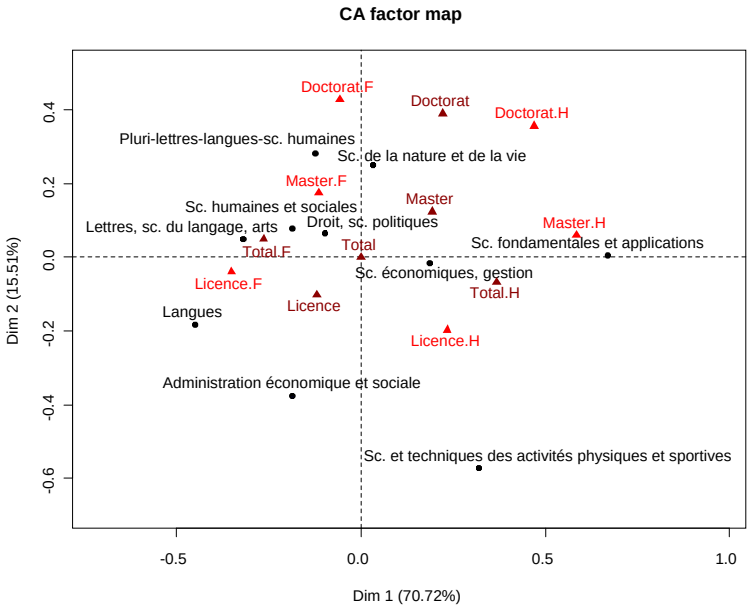


FIGURE 7.6 – AFC sur les données des universités.

Si le graphique est surchargé, on peut par exemple ne représenter que les lignes en utilisant l'option `invisible` :

```
> plot(res.ca, invisible=c("col","col.sup"))
```

On peut s'intéresser aux proximités entre les disciplines. Rappelons que deux disciplines sont proches si elles ont les mêmes profils (elles attirent les étudiants d'un même sexe et pour des études d'une même durée). Le graphe montre par exemple que les disciplines `Langues` et `Lettres, sc. du langage, arts` attirent surtout des femmes en licence. Les colonnes supplémentaires aident à interpréter le graphique de l'AFC. Ainsi, le premier axe oppose les hommes aux femmes tandis que le deuxième axe classe les niveaux, de la licence (en bas) au doctorat (en haut). Les disciplines à gauche (resp. droite) du graphique sont préférentiellement suivies par les femmes (resp. les hommes) : les disciplines littéraires sont plutôt à gauche du graphique (suivies par les femmes) et les disciplines scientifiques plutôt à droite du graphique (suivies par les hommes). Les disciplines en bas du graphique concernent des études courtes (niveau licence sur-représenté pour administration économique et sociale et STAPS par exemple) tandis que les disciplines en haut du graphique concernent des études longues (niveau doctorat sur-représenté pour les sciences de la nature et de la vie par exemple). Signalons qu'il n'est pas toujours aisé d'interpréter les axes de l'AFC, auquel cas on se focalise plutôt sur les proximités entre modalités.

Précisons enfin qu'il est possible de construire le graphique des axes 3 et 4 :

```
> plot(res.ca, axes = 3:4)
```

Interface graphique Factoshiny

Une interface graphique est disponible dans le package Factoshiny, via la fonction **CAshiny**, permettant d'effectuer les analyses précédentes à l'aide d'un menu déroulant convivial et de modifier le graphe de façon interactive.

Pour aller plus loin

L'Analyse Factorielle des Correspondances est un outil très utile pour effectuer des analyses textuelles. L'analyse textuelle consiste à comparer des textes, de différents auteurs par exemple, à partir des mots qui sont utilisés dans ces textes. Le principe est alors le suivant : on liste l'ensemble des mots qui sont utilisés dans les textes puis on dresse un tableau avec en lignes les textes et en colonnes les mots. À l'intérieur d'une case, on donne le nombre de fois où un mot a été utilisé dans un texte. L'AFC appliquée à ce type de tableau permet alors de comparer les textes entre eux, de voir quels sont les mots qui sont sur-employés ou sous-employés dans certains textes.

7.3 Analyse des Correspondances Multiples

Objet

L'objectif de l'Analyse des Correspondances Multiples (ACM) est de résumer un tableau de données où les individus sont décrits par des variables qualitatives. L'ACM permet d'étudier les ressemblances entre individus du point de vue de l'ensemble des variables et de dégager des profils d'individus. Elle permet également de faire un bilan des liaisons entre variables et d'étudier les associations de modalités. Enfin, à l'instar de l'ACP et de l'AFC, les individus ou groupes d'individus (lignes) peuvent être caractérisés par les modalités des variables (colonnes). Si certaines variables du tableau de données sont quantitatives, il est possible de les intégrer dans l'analyse en les découpant en classes (§ 2.3.2, p. 41). Pour effectuer une ACM, on utilise la fonction **MCA** du package FactoMineR, package dédié à l'analyse factorielle.

Exemple

Le jeu de données contient 66 clients ayant souscrit un crédit à la consommation dans un organisme de crédit. Les 11 variables qualitatives et les modalités sont les suivantes :

- `Marché` : rénovation d'un bien, voiture, scooter, moto, mobilier, side-car ; indique le bien pour lequel les clients ont fait un emprunt.
- `Apport` : oui, non ; indique si les clients possèdent un apport personnel avant de réaliser l'emprunt.
- `Impayé` : 0, 1, 2, 3 et plus ; nombre d'échéances impayées par le client.
- `Taux d'endettement` : 1 (faible), 2, 3, 4 (fort) ; le taux a été discrétisé en 4 classes.
- `Assurance` : sans assurance, AID (assurance invalidité et décès), AID + Chômage, Senior (pour les plus de 60 ans) ; type d'assurance.
- `Famille` : union libre (concubinage), marié, veuf, célibataire, divorcé.
- `Enfants à charge` : 0, 1, 2, 3, 4 et plus.
- `Logement` : propriétaire, accédant à la propriété, locataire, logé par la famille, logé par l'employeur.
- `Profession` : ouvrier non qualifié, ouvrier qualifié, retraité, cadre moyen, cadre supérieur.
- `Intitulé` : M, Mme, Melle.
- `Age` : 20 (18 à 29 ans), 30 (30 à 39), 40 (40 à 49), 50 (50 à 59), 60 et plus.

Le but de cette étude est de caractériser la clientèle de l'organisme de crédit. Nous voulons dans un premier temps mettre en évidence différents profils de comportements bancaires, c'est-à-dire effectuer une typologie des individus. Nous voulons ensuite étudier la liaison entre le signalétique (CSP, âge, etc.) et les principaux facteurs de variabilité des profils de comportements bancaires (i.e. caractériser les clients aux comportements particuliers).

Étapes

1. Importer le jeu de données
2. Choisir les variables et les individus actifs
3. Choisir le nombre d'axes
4. Analyser les résultats
5. Décrire de façon automatique les principales dimensions de variabilité
6. Retour aux données brutes par des tableaux croisés

Traitement de l'exemple

1. Importer le jeu de données

```
> credit <- read.table("credit.csv",sep=";", header=TRUE)
> summary(credit)
```

Les résultats de la fonction **summary** (non fournis) montrent que la variable `Age` n'est pas considérée comme qualitative. Il est donc nécessaire de la transformer :

```
> credit[,"Age"] <- factor(credit[,"Age"])
```

En ACM, il est important de vérifier qu'il n'y a pas de modalités rares, i.e. avec un faible effectif, car l'ACM accorde beaucoup d'importance à ces modalités (forte inertie). Pour cela, nous préconisons de représenter chaque variable :

```
> for (i in 1:ncol(credit)){   # permet d'avoir les graphes un à un
   par(ask=TRUE)               # cliquer sur la fenêtre graphique
   plot(credit[,i]) }          # pour voir le graphe
```

Si certaines variables admettent des modalités à faibles effectifs, plusieurs solutions sont envisageables pour éviter que ces modalités influencent trop l'analyse :
- regroupement naturel de certaines modalités, solution préconisée dans le cas de modalités ordonnées (cf. exercice 2.7) ;
- ventilation (répartition aléatoire) des individus associés aux modalités rares dans les autres modalités (cf. exercice 2.6) : certains programmes permettent une ventilation automatique (voir ci-dessous) ;
- suppression des individus qui prennent des modalités rares.

Dans l'exemple, un seul individu prend la modalité `Side-car` de la variable `Marche`. Il est naturel de regrouper cette modalité avec `Moto` (voir § 2.3.3, p. 43) :

```
> levels(credit[,"Marche"])[5] <- "Moto"
```

2. Choisir les variables et les individus actifs
Puisque nous souhaitons déterminer les profils de comportements bancaires, nous allons mettre en actif les variables correspondant aux informations bancaires (les cinq premières). Le choix des variables actives est très important car elles seules participent à la construction des axes de l'ACM, autrement dit seules ces variables sont utilisées pour calculer les distances entre individus. Nous ajoutons en variables supplémentaires les autres variables qualitatives (correspondant aux questions sur le signalétique). Il est également possible de mettre des variables quantitatives en supplémentaire (sans préalablement les découper en classes).

Nous choisissons aussi les individus actifs, i.e. ceux qui participent à la construction des axes. Ici, comme fréquemment, tous les individus sont considérés comme actifs.

Pour réaliser l'ACM, nous utilisons la fonction **MCA** du package FactoMineR.

```
> library(FactoMineR)
> res.mca <- MCA(credit, quali.sup=6:11, level.ventil=0)
```

L'ACM est construite à partir des 5 premières variables (les variables actives) alors que les variables 6 à 11 sont supplémentaires. L'argument `level.ventil` est par défaut égal à 0 ce qui signifie qu'aucune ventilation n'est effectuée. Si cet argument vaut par exemple 0.05, cela signifie que les modalités d'effectif inférieur ou égal à 5 % du total des individus sont ventilées de façon automatique par la fonction **MCA** avant de construire l'ACM. L'objet `res.mca` contient l'ensemble des résultats.

3. Choisir le nombre d'axes
Plusieurs solutions existent pour déterminer le nombre d'axes à analyser. La plus connue consiste à représenter le diagramme en barres des valeurs propres ou des inerties associées à chaque axe.

```
> barplot(res.mca$eig[,2],names=paste("Dim",1:nrow(res.mca$eig)))
```

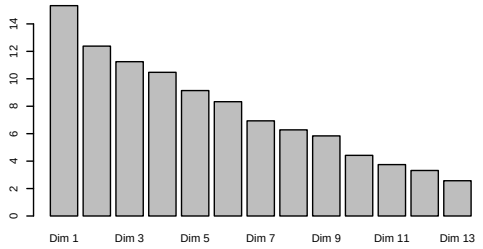


FIGURE 7.7 – Pourcentage d'inertie associée à chaque dimension de l'ACM.

La décroissance des valeurs propres (Fig. 7.7) est régulière. Nous n'observons pas de cassure ou décrochage flagrant, il devient donc difficile de choisir le nombre d'axes.

Cependant, ces faibles taux d'inertie et cette décroissance sont très classiques en ACM. Le tableau des pourcentages d'inertie expliquée par chaque axe est donné par la fonction **summary.MCA** (voir section suivante).

Les deux premiers axes expriment 28 % de l'inertie totale : autrement dit, 28 % de l'information du tableau de données est résumée par les deux premières dimensions, ce qui est relativement important dans le cadre d'une ACM. On ne détaille ici que l'interprétation des deux premiers axes factoriels. Cependant, les axes suivants sont intéressants à analyser.

4. Analyser les résultats

La fonction **MCA** fournit par défaut le graphique des individus (Fig. 7.8), celui des modalités (Fig. 7.9) et celui des variables (Fig. 7.10). Nous voyons dans ce qui suit comment retrouver et améliorer ces graphes. Notons que la simple application de la fonction **plot** (ou **plot.MCA**) sur l'objet résultat `res.mca` conduit à la représentation simultanée des individus et des modalités des variables (actives et illustratives) sur le plan principal. Cette représentation est vite surchargée et il est donc nécessaire de réaliser des représentations séparées des individus et des modalités des variables. On utilise pour cela l'argument `invisible` de la fonction **plot.MCA**. Pour avoir uniquement les individus, on utilise `invisible=c("var","quali.sup")` :

```
> plot(res.mca, invisible=c("var","quali.sup"))
```

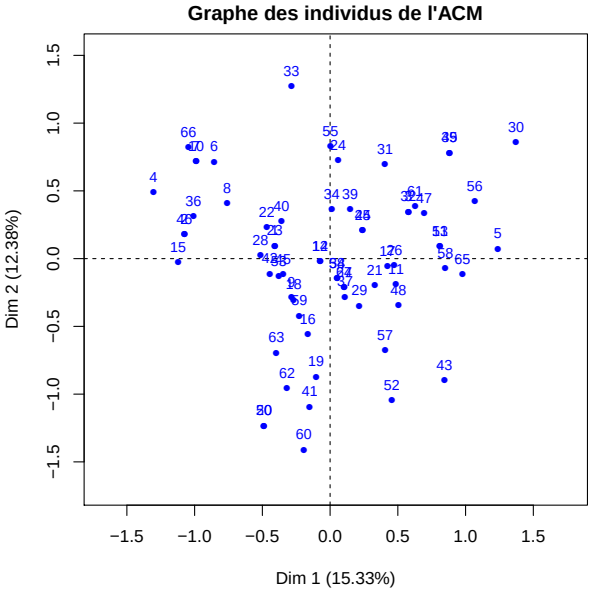


FIGURE 7.8 – ACM sur les données de crédit : représentation des individus.

Ce graphique des individus (Fig. 7.8) permet d'apprécier l'allure générale du nuage de points et de voir s'il se dégage par exemple des groupes d'individus particuliers. Ce n'est pas le cas ici. Pour aider à l'interprétation des résultats, il est intéressant de colorier les individus en fonction d'une variable, une couleur étant utilisée pour chaque modalité de celle-ci. On utilise alors l'argument `habillage` en indiquant le nom ou le numéro de la variable qualitative (graphiques non fournis) :

```
> plot(res.mca, invisible=c("var","quali.sup"),habillage="Marche")
> plot(res.mca, invisible=c("var","quali.sup"),habillage=1)
```

Le graphique avec l'ensemble des modalités des variables qualitatives actives et illustratives (Fig. 7.9) permet d'apprécier les grandes tendances qui se dégagent de l'analyse. Pour construire ce graphique et représenter uniquement les modalités, on utilise l'argument `invisible="ind"` :

```
> plot(res.mca, invisible="ind",
     title="Graphe des modalités actives et illustratives")
```

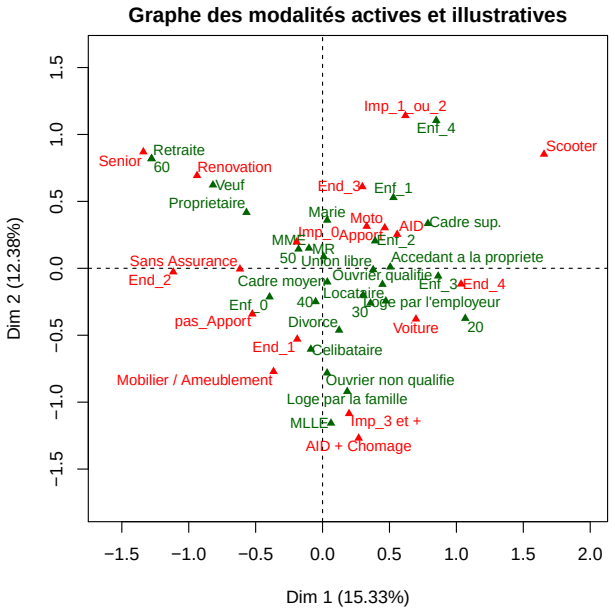


FIGURE 7.9 – Représentation des modalités actives et illustratives.

Le premier axe de la figure 7.9 oppose un profil « jeunes » (à droite) à un profil « vieux » (à gauche). On retrouve ainsi des personnes plus âgées, propriétaires, qui ont contracté un crédit pour financer des travaux de rénovation, opposées à

des personnes plus jeunes qui contractent un crédit pour acheter un scooter par exemple. Les « jeunes » ont tendance à avoir un taux d'endettement assez élevé.

L'axe 2 oppose principalement les personnes qui rencontrent de grandes difficultés financières (en bas) aux autres (en haut). On retrouve ainsi des personnes qui ont des difficultés à rembourser leur crédit à la consommation (`Impaye` 3 et plus) et qui ont souscrit à l'assurance AID + Chômage. Cet axe peut alors être qualifié d'axe de difficultés financières.

Nous pouvons rappeler sur quelques exemples les règles générales d'interprétation de proximité entre modalités de deux variables différentes ou non. Par exemple, la modalité `Senior` de la variable (active) `Assurance` est du côté de la modalité `Retraite` de la variable (illustrative) `Profession`, ce qui signifie que beaucoup d'individus qui prennent la modalité `Senior` prennent aussi la modalité `Retraite`. Nous pouvons aussi constater que la modalité `Loge par l'employeur` et `Locataires` de la variable `logement` sont du même côté, ce qui signifie que ces personnes prennent en général les mêmes modalités pour les autres variables. Les individus qui prennent ces modalités ont donc le même profil.

L'interprétation peut être affinée à l'aide des résultats numériques fournis par la fonction **summary.MCA**, que l'on appelle simplement par **summary**. Ici on veut les résultats pour les 2 premières dimensions (`ncp=2`) et les 2 premières lignes de chaque tableau (`nbelements=2`). Si on voulait les résultats pour tous les individus, toutes les variables et toutes les modalités, on écrirait `nbelements=Inf`.

```
> summary(res.mca, nbelements=2, ncp=2, nb.dec=2)
Call:
MCA(X = credit, quali.sup = 6:11)

Eigenvalues
                       Dim.1  Dim.2  Dim.3  Dim.4  Dim.5  Dim.6  ...
Variance                0.40   0.32   0.29   0.27   0.24   0.22  ...
% of var.              15.33  12.38  11.25  10.47   9.14   8.33  ...
Cumulative % of var.   15.33  27.71  38.96  49.43  58.57  66.90  ...

Individuals (the 2 first)
               Dim.1   ctr  cos2   Dim.2   ctr  cos2
1           | -0.41  0.63  0.10 |  0.10  0.04  0.01 |
2           | -1.07  4.38  0.47 |  0.18  0.16  0.01 |

Categories (the 2 first)
               Dim.1   ctr  cos2 v.test   Dim.2   ctr  cos2 v.test
Mobilier     | -0.37  1.73  0.05  -1.74 | -0.77  9.52  0.21  -3.66 |
Moto         |  0.33  0.74  0.02   1.06 |  0.31  0.83  0.02   1.00 |

Categorical variables (eta2)
               Dim.1 Dim.2
Marche       |  0.62  0.39 |
```

```
Apport       |  0.24  0.10 |

Supplementary categories (the 2 first)
              Dim.1  cos2 v.test   Dim.2  cos2 v.test
Celibataire | -0.09  0.00  -0.41 | -0.60  0.13  -2.87 |
Divorce     |  0.12  0.00   0.29 | -0.46  0.02  -1.07 |

Supplementary categorical variables (eta2)
              Dim.1 Dim.2
Famille      |  0.09  0.19 |
Enfants      |  0.24  0.11 |
```

Nous obtenons alors les résultats des inerties, puis les résultats (coordonnées, qualités de représentation et contributions à la construction des axes) des individus, des modalités et des variables. La coordonnée d'une variable sur une dimension est le rapport de corrélation η^2 entre la composante principale (les coordonnées des individus sur la dimension) et la variable qualitative.
Pour connaître les variables les plus liées aux axes, i.e. interpréter les axes par les variables, on dessine le graphique des rapports de corrélation entre les dimensions et les variables :

```
> plot(res.mca, choix="var")
```

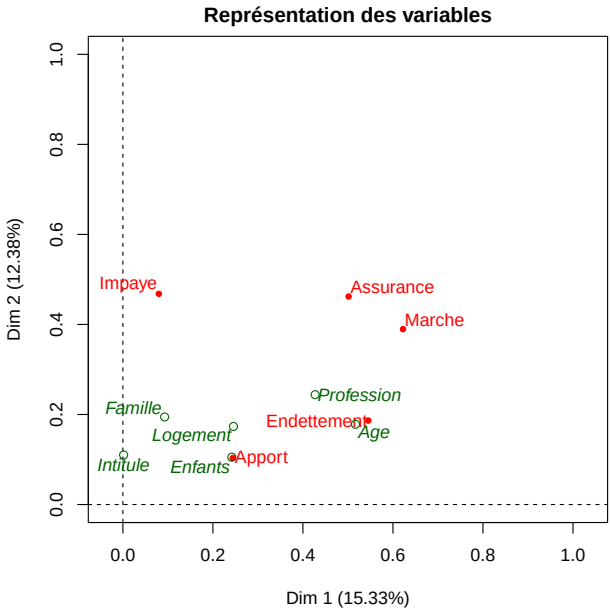


FIGURE 7.10 – Représentation des variables actives et illustratives.

La variable qui contribue le plus à la création de l'axe 1 est la variable `Marche`, celle qui contribue le plus à la création de l'axe 2 est la variable `Impaye`. Cette information résume l'influence globale de toutes les modalités de chacune des variables sur la construction des axes ; ainsi, les modalités de la variable `Marche` contribuent fortement à la création de l'axe 1 et les modalités de la variable `Impaye` à la création de l'axe 2. Nous pouvons ensuite examiner les résultats concernant les variables qualitatives supplémentaires contenus dans l'objet `res.mca$quali.sup`. Par défaut les graphes proposés concernent les axes 1 et 2. Mais il est possible de choisir d'autres axes : pour sélectionner les axes 3 et 4 on précisera l'argument `axes=3:4` dans l'appel de la fonction **plot** par exemple.

5. Décrire de façon automatique les principales dimensions de variabilité
Il existe dans le package FactoMineR une fonction permettant de décrire automatiquement les axes de l'ACM et des autres méthodes d'analyse factorielle : **dimdesc**. Cette fonction aide à l'interprétation puisqu'elle permet de décrire rapidement les principales dimensions de variabilité. Elle trie les variables quantitatives (supplémentaires) en fonction des coefficients de corrélation entre la composante et chaque variable. Elle trie également les variables qualitatives ainsi que les modalités des variables qualitatives. Pour cela, un modèle d'analyse de variance à un facteur est construit pour chaque variable qualitative : la variable à expliquer correspond à la composante principale (les coordonnées des individus) et est expliquée en fonction d'une variable qualitative. La probabilité critique du test global (test de $H_0 : \forall i,\ \alpha_i = 0$, cf. fiche 9.3, p. 276) est calculée ainsi que les probabilités critiques des tests de chaque modalité (test de $H_0 : \alpha_i = 0$ avec la contrainte $\sum_i \alpha_i = 0$, cf. fiche 9.3, p. 276). Les probabilités critiques associées à chaque test global sont triées par ordre croissant. Ainsi, les variables qualitatives sont triées de la plus caractérisante à la moins caractérisante. De même, les probabilités critiques des tests par modalités sont triées (par ordre croissant quand le coefficient est positif, par ordre décroissant quand le coefficient est négatif). Ceci permet de mettre en évidence les modalités les plus caractérisantes. On donne ci-dessous une description de la deuxième dimension de l'ACM par les variables et les modalités.

```
> dimdesc(res.mca)
$`Dim 2`
$`Dim 2`$quali
                P-value
Impaye      2.330816e-09
Assurance   1.979222e-08
Marche      3.732164e-06
........... ............

$`Dim 2`$category
                        Estimate      P-value
Imp_1_ou_2             0.5997078 6.148708e-07
Senior                 0.5149935 9.168824e-06
```

```
Retraite                0.4477685 1.245334e-03
...                      ......... ...........
AID + Chomage          -0.6985157 4.229885e-09
Imp_3 et +             -0.6634535 7.988517e-10
```

Cette fonction est très utile lorsqu'il y a beaucoup de modalités. On retrouve ici que le deuxième axe est surtout lié aux modalités `AID+Chomage` et `Imp_3 et +`.

6. Retour aux données brutes par des tableaux croisés
Il est intéressant de revenir aux données brutes pour analyser plus finement la liaison entre deux variables et notamment la proximité entre les modalités de ces variables. On peut ainsi construire des tableaux croisés avec la commande **table**, calculer des pourcentages en lignes, en colonnes et effectuer un test du χ^2 (voir la fiche 6.2) ou une AFC (voir la fiche 7.2).

Interface graphique Factoshiny

Une interface graphique est disponible dans le package Factoshiny, via la fonction **MCAshiny**, permettant d'effectuer les analyses précédentes à l'aide d'un menu déroulant convivial et de modifier les graphes de façon interactive.

Pour aller plus loin

L'ACM comme prétraitement d'une classification : les composantes principales de l'ACM sont des variables quantitatives qui résument un ensemble de variables qualitatives. Ce sont donc des variables synthétiques qui peuvent être utilisées comme prétraitement avant une classification des individus. Ce prétraitement à la classification peut être vu comme un changement de base permettant de passer de variables qualitatives à des variables quantitatives : les coordonnées des individus sur les axes de l'ACM. On peut éventuellement supprimer les dernières dimensions pour ne conserver que 95 % de l'information. Pour la mise en œuvre de la classification sur les composantes principales de l'ACM, voir la fiche 8.1.

Il peut aussi être intéressant d'utiliser la fonction **catdes** (voir la fiche 8.1) pour décrire spécifiquement une variable qualitative en fonction de variables quantitatives et/ou qualitatives.

La fonction **MCA** crée par défaut une nouvelle modalité `NA` pour chaque variable ayant des données manquantes. D'autres stratégies peuvent être envisagées pour gérer les données manquantes en ACM et sont implémentées dans la fonction **imputeMCA** du package missMDA.

7.4 Analyse Factorielle Multiple

Objet

L'objectif de l'Analyse Factorielle Multiple est d'étudier des tableaux de données complexes où un même ensemble d'individus est décrit par des variables structurées en groupes et provenant éventuellement de différentes sources d'information. L'AFM équilibre l'influence de chaque groupe dans l'analyse, les variables pouvant être de nature quantitative dans un groupe et qualitative dans un autre. L'AFM permet, comme l'ACP et l'ACM, d'étudier les similarités entre individus et les liaisons entre variables. Sa spécificité et sa richesse sont de prendre en compte simultanément plusieurs groupes de variables et de comparer l'information apportée par chacun de ces groupes. On utilisera la fonction **MFA** de FactoMineR.

Exemple

Le jeu de données provient de l'OCDE et concerne des indicateurs de qualité de vie en 2015. Pour les 34 pays de l'OCDE à cette date, plus la Russie et le Brésil, on dispose de 22 indicateurs regroupés en cinq thèmes, et d'une variable qualitative correspondant à la région. Les cinq thèmes sont les suivants, on donne entre parenthèses le nombre d'indicateurs pour le thème : Bien-être matériel (5), Emploi (5), Satisfaction (3) ; Santé et sécurité (6), Enseignement (3).

On souhaite comparer les 36 pays en tenant compte de ces cinq groupes d'indicateurs. Quels pays se ressemblent du point de vue de l'ensemble des indicateurs (pour tous les thèmes) ? Certains pays sont-ils particuliers ? Certains pays sont-ils particuliers pour un ou plusieurs thèmes ? La position relative des pays est-elle la même d'un groupe d'indicateurs à l'autre ?

Étapes

1. Importer le jeu de données
2. Choisir les groupes de variables actifs
3. Choisir de standardiser ou non les variables des groupes quantitatifs
4. Choisir le nombre d'axes
5. Analyser les résultats
6. Décrire de façon automatique les axes

Traitement de l'exemple

1. Importer le jeu de données

Lors de l'importation, on précise que la première colonne correspond aux noms des pays (`row.names=1`), que l'on conserve le nom des variables tel qu'il est dans

le fichier de données (`check.names=FALSE`) et quel est l'encodage du fichier (pour gérer les accents) :

```
> QteVie <- read.table("QteVie.csv", header=TRUE, sep=";",
   quote="", check.names=FALSE, row.names=1, encoding="latin1")
> summary(QteVie)
```

2. Choisir les groupes de variables actifs
Un groupe de variables correspond à un sous-tableau, donc le plus souvent à un type d'information : par exemple, les notes à l'école peuvent être regroupées par pôle (littéraire, scientifique, etc.), dans un questionnaire un groupe correspond à toutes les questions d'un même thème, en économie les groupes peuvent correspondre aux indicateurs d'une même année... La constitution des groupes est essentielle car l'AFM attribue un poids à chaque groupe pour équilibrer leur influence dans l'analyse. Dans notre exemple, le choix des groupes est dicté par les thèmes des indicateurs de qualité de vie.

Ici, les cinq groupes d'indicateurs sont actifs et participent à la construction des dimensions de l'analyse. Le groupe constitué d'une seule variable (Région) est mis en illustratif. Notons que pour ajouter des variables, il faut les regrouper dans un ou plusieurs groupes supplémentaires. Tous les individus (i.e. pays) sont actifs.

3. Choisir si les variables des groupes quantitatifs sont standardisées ou non
Pour chaque groupe de variables quantitatives, il est nécessaire de préciser si les variables sont réduites ou non. Ce choix se fait par groupe, et la décision est prise selon les mêmes critères qu'en ACP. Ici, la réduction est indispensable car les variables ont des unités différentes.

Pour réaliser l'AFM, on peut utiliser la fonction **MFA** du package FactoMineR ou l'interface graphique du package Factoshiny.

```
> library(FactoMineR)
> afm <- MFA(QteVie, group=c(5,5,3,6,3,1), type=c(rep("s",5),"n"),
   name.group=c("Bien-être matériel","Emploi","Satisfaction",
   "Santé et sécurité","Enseignement","Région"),num.group.sup=6)
```

L'argument `group` indique le nombre de variables par groupe : le premier groupe est constitué des 5 premières variables, le 2ème groupe des 5 suivantes, ..., et le dernier groupe de la dernière variable. Le `type` indique la nature des variables du groupe : ici les 5 premiers groupes sont standardisés `"s"` et le dernier est qualitatif (`"n"` pour nominal) ; les autres options sont `"c"` pour non standardisés et `"f"` pour des tableaux de fréquences. L'argument `name.group` précise le nom des groupes et `num.group.sup` le(s) numéro(s) du(des) groupe(s) supplémentaire(s).

4. Choisir le nombre d'axes
Comme pour l'ACP, on peut déterminer le nombre d'axes à interpréter en AFM en dessinant le diagramme des pourcentages d'inertie (ou celui des valeurs-propres) :

```
> barplot(afm$eig[,2],names=paste("Dim",1:nrow(afm$eig)))
```

5. Analyser les résultats
Les indicateurs numériques (coordonnées, qualités de représentation et contributions des individus, des variables et des groupes) peuvent être résumés à l'aide de la fonction **summary.MFA** que l'on appelle par **summary** :

```
> summary(afm, nbelement=10, nb.dec=2, ncp=2)
```

On peut obtenir des résultats spécifiques de l'AFM comme les coefficients RV et Lg qui sont des mesures de liaison entre groupes de variables. Pour les RV :

```
> round(afm$group$RV, 2)
                  Bien-être                          Santé et
                   matériel Emploi Satisfaction  sécurité ...
Bien-être matériel     1.00   0.39         0.32      0.56 ...
Emploi                 0.39   1.00         0.46      0.35 ...
Satisfaction           0.32   0.46         1.00      0.34 ...
Santé et sécurité      0.56   0.35         0.34      1.00 ...
Enseignement           0.20   0.26         0.15      0.51 ...
Région                 0.22   0.23         0.25      0.30 ...
MFA                    0.71   0.71         0.65      0.79 ...
```

Ces coefficients RV varient entre 0 et 1. Le RV de 0.15 entre `Enseignement` et `Satisfaction` montre que les variables entre ces deux groupes sont très peu liées. La valeur la plus extrême, 0.56, entre `Santé et sécurité` et `Bien-être matériel` est relativement éloignée de 1, ce qui montre que des liaisons existent entre les variables de ces deux groupes, mais ne sont pas très fortes. L'information apportée par les différents groupes n'est donc pas trop redondante. La ligne (ou colonne) intitulée MFA correspond à un groupe contenant toutes les dimensions de l'AFM. La valeur de 0.79 entre MFA et `Santé et sécurité` montre que ce dernier groupe donne une configuration des individus assez proche de celle fournie par l'AFM.

L'AFM délivre par défaut le graphe des groupes, celui des individus, celui des axes partiels, et celui des individus avec des points partiels (graphe non représenté ici). Ensuite, selon la nature des variables, les graphes suivants sont représentés : le graphe des variables avec le cercle des corrélations si des variables quantitatives sont présentes, et un graphe avec les modalités des variables qualitatives et ses points partiels si des variables qualitatives sont présentes. Ici des variables quantitatives et qualitatives sont présentes et nous retrouvons ces deux graphes.

Dans le graphe des groupes (Fig. 7.11 en haut à gauche), tous les groupes ont une coordonnée assez élevée, supérieure à 0.5, sur la première dimension. Ainsi, tous les groupes séparent les individus, ici les pays, comme le fait la première dimension de l'AFM. Selon la deuxième dimension, seuls les groupes `Emploi`, et dans une moindre mesure `Satisfaction`, différencient les pays. La deuxième dimension de l'AFM est donc commune à ces deux groupes, mais pas aux autres groupes.

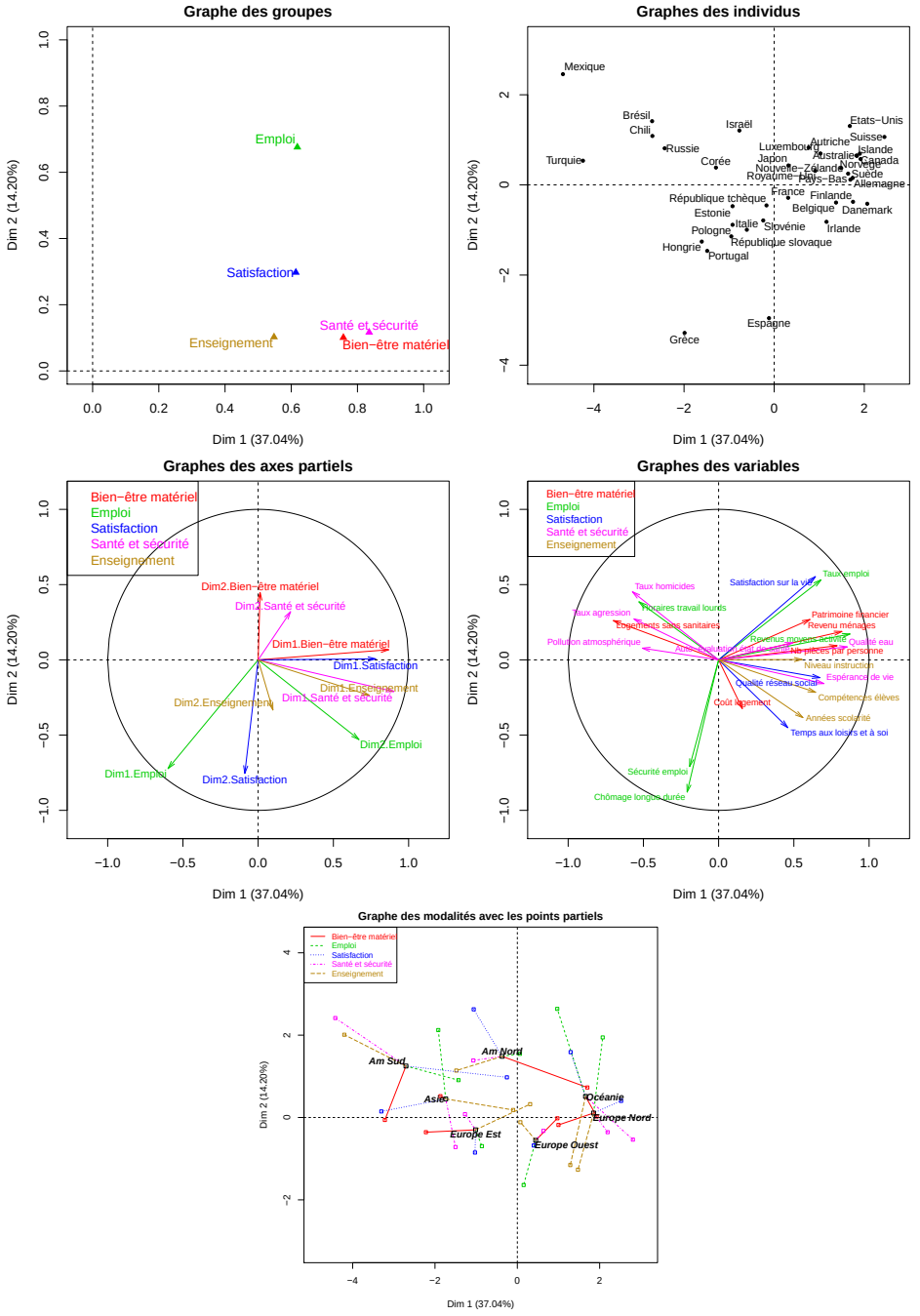


FIGURE 7.11 – Graphes de l'AFM pour : les groupes, les individus, les axes partiels, les variables quantitatives, les modalités avec les points partiels.

Le graphe des variables quantitatives (Fig. 7.11 ligne du milieu à droite) et celui des individus (Fig. 7.11 en haut à droite) s'interprètent conjointement, comme en ACP. Ainsi, les individus à droite du graphe (comme l'Allemagne) prennent des valeurs plus fortes que la moyenne pour les variables qui ont une forte coordonnée sur la première dimension, comme `Qualité eau` et `Revenu ménages`. A contrario, ces pays prennent de faibles valeurs pour les variables qui ont une faible coordonnée sur l'axe 1, comme `Pollution atmosphérique` ou `Taux d'agression`. C'est l'inverse pour les pays situés sur la gauche (Turquie et Mexique). Cette opposition Turquie et Mexique vs les autres pays est présente, de façon plus ou moins forte, dans tous les groupes de variables (cf. graphe des groupes).

Le graphe des axes partiels (Fig. 7.11 ligne du milieu à gauche) montre que la dimension 1 de l'ACP de chacun des groupes (obtenue en effectuant l'ACP sur les variables du groupe) se projette relativement bien sur la première dimension de l'AFM et donc ressemble à cette dimension. Ainsi la position des individus sur la première dimension de l'AFM est similaire à celle des individus sur les premières dimensions des ACP séparées. La deuxième dimension de l'AFM est principalement liée à la dimension 2 de `Satisfaction` et à la dimension 1 d'`Emploi`.

Le graphe des points partiels (Fig. 7.12) permet quant à lui de voir comment un individu est vu par les variables de chacun des groupes pris séparément. Par défaut les points partiels sont dessinés pour les 2 individus qui ont les points partiels avec l'inertie intra la plus faible et les 2 individus ayant les points partiels avec l'inertie intra la plus forte. On peut dessiner le graphe des points partiels pour tous les individus grâce à l'argument `partial="all"`. Pour représenter quelques individus particuliers, par exemple la France et l'Autriche, on utilise l'argument `partial` :

```
> plot(afm, partial=c("France","Autriche"),invisible="quali")
```

Le point moyen de la France (en noir) est proche du barycentre du nuage, ce qui indique des valeurs moyennes sur l'ensemble des critères. Dans le détail, le point partiel associé aux variables sur le `Bien-être matériel` a une forte coordonnée sur la première dimension, ce qui signifie que les indicateurs de bien-être matériel sont bons. En revanche, le point partiel `Enseignement` est sur la gauche, ce qui indique que les indicateurs d'enseignement sont insuffisants.

On peut aussi visualiser les points partiels d'un seul groupe pour voir la représentation des individus vus par les variables de ce groupe dans le repère de l'AFM. On modifie alors la `palette` de couleurs et on utilise la couleur `transparent` pour tous les groupes sauf un (la première couleur, `black`, correspond à la couleur de l'individu moyen). On redéfinit également les bornes des axes avec `xlim` pour les abscisses et `ylim` pour les ordonnées :

```
> plot(afm, invisible="quali", partial="all", xlim=c(-5,5), ylim=c(-5,5),
     palette = palette(c("black", "transparent", "transparent", "blue",
     rep("transparent",2))), title="Points partiels sur la satisfaction")
```

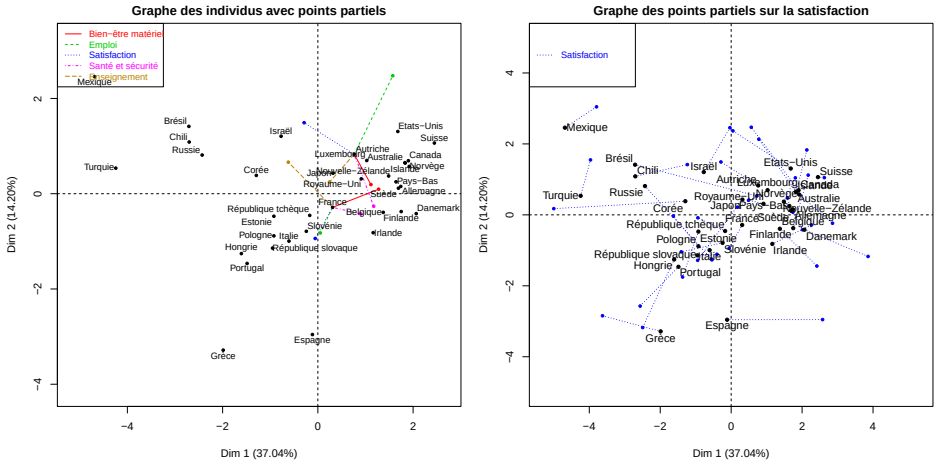


FIGURE 7.12 – Graphes des points partiels. À gauche, les points partiels de la France et l'Autriche pour tous les groupes ; à droite les points partiels du groupe de variables Satisfaction.

Le graphe des modalités avec les points partiels (Fig. 7.11 en bas) s'interprète comme les deux graphes précédents. Ainsi, le point Asie sur la gauche du graphe indique que les pays asiatiques ont des indicateurs globalement mauvais. Cependant, la position des points partiels `Enseignement` et `Emploi` est encourageante.

6. Décrire de façon automatique les principales dimensions de variabilité
La fonction **dimdesc** de FactoMineR décrit automatiquement les axes de l'AFM, ce qui est utile quand il y a beaucoup de variables (cf. fiches sur l'ACP et l'ACM).

Interface graphique Factoshiny

Une interface graphique est disponible dans le package Factoshiny, via la fonction **MFAshiny**, permettant d'effectuer les analyses précédentes à l'aide d'un menu déroulant convivial et de modifier les graphes de façon interactive.

Pour aller plus loin

Par défaut, la fonction **MFA** remplace les données manquantes d'une variable quantitative par la moyenne de cette variable calculée sur les données observées, et code les données manquantes d'une variable qualitative comme une nouvelle modalité « NA ». Cette stratégie n'est pas idéale et il est préférable d'utiliser par exemple la fonction **imputeMFA** du package missMDA pour compléter le tableau de données (voir fiche 11.1 page 374 pour plus de détails).

Chapitre 8

Classification non supervisée

Ce chapitre présente trois méthodes standards de classification non supervisée. À partir d'un tableau individus-variables, elles consistent à créer un arbre hiérarchique ou une partition. La première méthode, la classification ascendante hiérarchique (fiche 8.1), construit un arbre permettant de visualiser les distances entre individus et groupes d'individus. Les deux autres méthodes ont pour principe de former une partition. La méthode k-means (fiche 8.2) construit une partition de façon géométrique, tandis que les modèles de mélange (fiche 8.3) s'appuient sur une modélisation probabiliste.

Dans chacune des fiches, on présente succinctement la méthode et les principales étapes de sa mise en œuvre. Un même exemple est alors traité via l'enchaînement des différentes instructions R et les résultats numériques et graphiques sont brièvement commentés.

Nous utiliserons le jeu de données décathlon détaillé dans la fiche d'ACP (voir fiche 7.1). Le jeu de données concerne les résultats aux épreuves du décathlon lors de deux compétitions d'athlétisme qui ont eu lieu à un mois d'intervalle : les Jeux Olympiques d'Athènes (23 et 24 août 2004) et le Décastar (25 et 26 septembre 2004). Nous importons et résumons le jeu de données par les lignes de code suivantes :

```
> decath <- read.table("https://r-stat-sc-donnees.github.io/decathlon.csv",
    sep=";",dec=".", header=TRUE, row.names=1, check.names=FALSE)
> summary(decath)
```

8.1 Classification Ascendante Hiérarchique

Objet

La Classification Ascendante Hiérarchique (CAH) a pour objectif de construire une hiérarchie sur les individus et se présente sous la forme d'un dendrogramme. Cette classification permet de regrouper des individus dans des classes les plus homogènes possibles à partir d'un jeu de données individus × variables. Cette méthode nécessite de choisir : une distance entre individus, ou plus généralement une dissimilarité et un indice d'agrégation entre les classes.

Nous présentons ici le cas où les individus sont caractérisés par des mesures sur des variables quantitatives, le cas des variables qualitatives étant abordé dans la section « Pour aller plus loin ». La distance choisie est la distance euclidienne et l'indice d'agrégation celui de Ward. Ce dernier choix est naturel quand la classification est couplée avec des analyses factorielles puisqu'il utilise aussi la notion d'inertie (voir Saporta, 2011).

Exemple

Reprenons le jeu de données sur les athlètes qui participent au décathlon (cf. fiche 7.1) et construisons une Classification Ascendante Hiérarchique des profils de performances des athlètes. Nous nous intéressons donc uniquement aux dix variables de performances.

Étapes

1. Importer les données
2. Standardiser ou non les données
3. Construire la Classification Ascendante Hiérarchique
4. Couper l'arbre de classification
5. Caractériser les classes

Traitement de l'exemple

1. Importer les données

```
> decath <- read.table("decathlon.csv",sep=";",header=TRUE,row.names=1,
    check.names=FALSE)
```

2. Standardiser ou non les données

Avant de construire la CAH, nous devons choisir de réduire ou non les variables. Pour le jeu de données de cette fiche, nous n'avons pas le choix, la réduction est indispensable car les variables ont des unités différentes. Quand les variables ont

les mêmes unités, les deux solutions sont envisageables et impliquent deux analyses différentes. Cette décision est donc cruciale. La réduction permet d'accorder la même importance à chacune des variables dans le calcul de la distance entre individus. Ne pas réduire revient à donner plus d'importance aux variables ayant un grand écart-type. Ici, nous standardisons les données grâce à la fonction **scale**.

3. Construire la Classification Ascendante Hiérarchique
Pour réaliser la CAH à l'aide du package cluster, il faut charger le package préalablement installé (voir section 1.7). Nous utilisons ensuite la fonction **agnes**.

```
> library(cluster)
> classif <- agnes(scale(decath[,1:10]),method="ward")
> plot(classif,xlab="Individu",which.plot=2,main="Dendrogramme")
```

La fonction **agnes** prend comme arguments le jeu de données (réduit ou non), la méthode d'agrégation (ici la méthode de `Ward` qui minimise la diminution de l'inertie inter-classes lorsque deux classes sont regroupées en une seule), la métrique (par défaut la distance euclidienne). La fonction **plot** dessine l'arbre hiérarchique, ou dendrogramme.

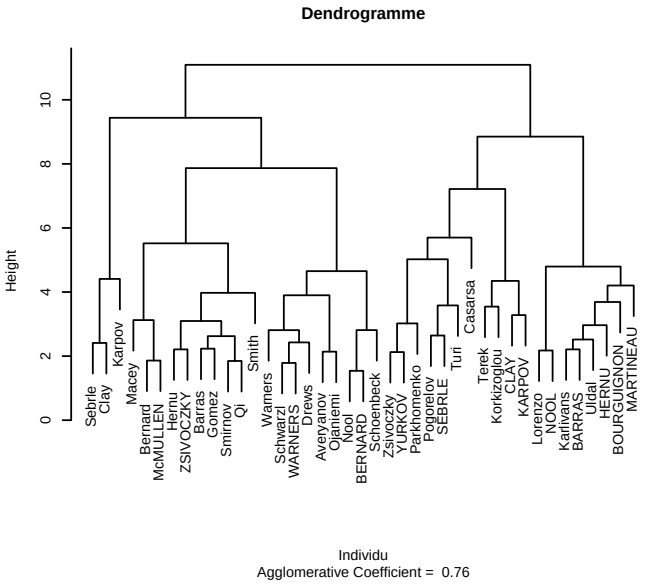


FIGURE 8.1 – Classification Ascendante Hiérarchique.

4. Couper l'arbre de classification
Au vu de l'arbre hiérarchique (Fig. 8.1), nous pouvons « tailler » l'arbre pour construire des classes d'individus. Pour cela, il faut définir une hauteur de coupe.

Par exemple, une hauteur de coupe de 10 (Fig. 8.1) définit deux classes alors qu'une hauteur de 8 définit quatre classes.

Afin de choisir judicieusement une hauteur de coupe, nous allons nous servir des hauteurs où deux classes sont fusionnées ensemble. Pour obtenir ces hauteurs dans un vecteur, il suffit de transformer l'objet `classif` en un objet de type `hclust` et d'en extraire la hauteur. Ces hauteurs étant rangées par ordre croissant, nous inversons leur ordre (**rev**) :

```
> classif2 <- as.hclust(classif)
> plot(rev(classif2$height),type = "h", ylab = "hauteurs")
```

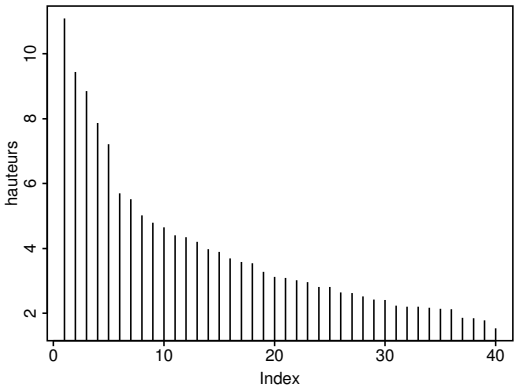


FIGURE 8.2 – Hauteurs de coupes.

La hauteur du 40e et dernier bâton (Fig. 8.2) donne une idée de la difficulté de réunir les deux premiers individus et de former une classification à 40 classes. Ensuite, la hauteur du 39e donne une idée de la difficulté de passer de cette classification à 40 classes en une classification à 39 classes. Ce passage est obtenu soit en fusionnant deux individus, soit en agrégeant un individu à la classe existante. De la même façon, la hauteur du premier bâton donne donc une idée de la difficulté d'agréger les deux classes restantes pour grouper les $n = 41$ individus ensemble.

Comme nous cherchons des classes homogènes et peu nombreuses, nous sommes intéressés par les hauteurs des premiers bâtons. Le saut le plus évident sépare le cinquième et le sixième bâton : il est donc difficile de passer de 6 classes à 5 classes. Une première possibilité de coupure consisterait donc à prendre six groupes.

Cependant, entre le troisième et le quatrième bâton se trouve un autre saut, moins prononcé. Afin d'obtenir un effectif suffisant dans chacune des classes, nous utilisons ce saut qui donne quatre classes. Nous souhaitons de plus connaître les individus de chacune de ces quatre classes. Pour cela, nous pouvons utiliser la fonction **cutree** et indiquer la coupure de l'arbre, c'est-à-dire le nombre de classes entre lesquelles sont répartis les individus (par exemple `k=4`).

```
> classe <- cutree(classif, k = 4)
> classe
 [1] 1 1 1 2 2 3 2 2 2 2 3 2 2 2 2 2 2 2 2 3 3
[22] 2 3 4 4 3 4 3 3 3 3 2 3 2 2 2 4 4 4 4 4
```

5. Caractériser les classes

Après avoir regroupé les individus en classes, il est intéressant de décrire ces classes et d'interpréter les similitudes et différences entre ces groupes d'individus. Plutôt que d'énumérer tous les individus qui appartiennent à une même classe, il est préférable de décrire les classes à l'aide des variables. La fonction **catdes** du package FactoMineR permet de caractériser chaque classe à l'aide de variables quantitatives et qualitatives. Afin de pouvoir utiliser cette fonction, nous allons ajouter la variable `classe` au jeu de données initial. La variable `classe` est préalablement transformée en facteur et on lui affecte le nom `"classe"`.

```
> decath.comp <- cbind.data.frame(decath,classe=as.factor(classe))
```

La fonction **catdes** fournit dans un premier temps les variables les plus liées à la variable de classe. Pour les variables quantitatives, un modèle d'analyse de variance est construit entre la variable quantitative et la variable qualitative de classe et on récupère le R^2 (`Eta2`) ainsi que la probabilité critique du test de nullité associé. Seules les variables significatives, triées par probabilités croissantes, sont données. Ici, la variable `Points` est la plus intéressante pour expliquer les classes. Le test n'est qu'indicatif car les variables ont été utilisées pour construire les classes. Puis chaque classe est décrite en détail. La fonction trie les variables quantitatives de la plus à la moins caractérisante en positif (variables pour lesquelles les individus de la classe prennent des valeurs significativement supérieures à la moyenne de l'ensemble des individus) puis de la moins à la plus caractérisante en négatif (variables pour lesquelles les individus de la classe prennent des valeurs significativement inférieures à la moyenne de l'ensemble des individus). Pour les variables qualitatives, ce sont les modalités des variables qui sont triées. Elles sont triées de la plus à la moins caractérisante lorsque la modalité est sur-représentée dans la classe (par rapport aux autres classes) et de la moins à la plus caractérisante lorsque la modalité est sous-représentée dans la classe. Les résultats sont ici fournis pour les classes 2 et 3.

```
> library(FactoMineR)
> catdes(decath.comp, num.var = 14)
Link between the cluster variable and the quantitative variables
================================================================
                  Eta2      P-value
Points     0.7181466 2.834869e-10
X100m      0.5794940 4.206021e-07
Poids      0.4746713 2.358915e-05
```

```
Longueur   0.4405023 7.319114e-05
110m.H     0.4254154 1.178937e-04
400m       0.4124012 1.759714e-04
1500m      0.4078248 2.021278e-04
......     ......... ............

Description of each cluster by quantitative variables
======================================================
'2'
                      Mean in   Overall      sd in    Overall      p.value
            v.test   category      mean   category         sd
100m     -2.265855   10.89789  10.99805  0.1701572  0.2597956  0.023460250
110m.H   -2.397231   14.41579  14.60585  0.3097931  0.4660000  0.016519515
400m     -2.579590   49.11632  49.61634  0.5562394  1.1392975  0.009891780
1500m    -2.975997  273.18684 279.02488  5.6838942 11.5300118  0.002920378

'3'
                      Mean in   Overall     sd in     Overall      p.value
            v.test   category      mean  category         sd
1500m     3.899044 290.763636 279.02488 12.6274652 11.5300118 9.657328e-05
400m      2.753420  50.435455  49.61634   1.272588  1.1392975 5.897622e-03
Longueur -2.038672   7.093636   7.26000   0.283622  0.3125193 4.148279e-02
```

Les individus de la classe 2 courent vite le 1 500 m (ils ont un temps plus faible que les autres). Le temps moyen pour les individus de cette classe est de 273.2 s contre 279.0 s pour l'ensemble de tous les individus. Trois autres variables caractérisent de façon significative les individus de cette classe (valeur-test supérieure à 2 en valeur absolue) : le 400 m, le 110 m haies et le 100 m. Ces athlètes courent vite (temps plus faible) dans ces disciplines. Une valeur-test supérieure à 2 en valeur absolue signifie ici que la moyenne de la classe est significativement différente de la moyenne générale : un signe positif (resp. négatif) de la valeur-test indique que la moyenne de la classe est supérieure (resp. inférieure) à la moyenne générale. Notons qu'aucune variable qualitative ne caractérise ces classes.

Remarque

Une autre fonction (**hclust**) permet également de construire une classification :

```
> classif <- hclust(dist(scale(decath[,1:10])), method = "ward.D2")
```

Cette fonction prend comme argument une matrice de distances entre individus : avec la méthode de Ward, il s'agit de la distance euclidienne au carré.

Pour aller plus loin

Dans cette section, nous détaillons deux extensions de la classification.

1. Classification sur des variables qualitatives
Si les individus sont décrits par des variables qualitatives, deux solutions sont envisageables :
- construire une distance ou une dissimilarité adaptée (voir Kaufman et Rousseeuw, 1990) ;
- se ramener à des données quantitatives en effectuant au préalable une Analyse des Correspondances Multiples (voir fiche 7.3). Il suffit pour cela de récupérer les coordonnées des individus sur l'intégralité ou sur une partie des axes factoriels et d'effectuer la CAH sur ces coordonnées (voir l'exemple d'enchaînement ACP-CAH ci-dessous).

2. Classification sur les composantes principales d'une analyse factorielle
Il peut être intéressant de réaliser une analyse factorielle (ACP ou ACM) avant de construire une classification de façon à supprimer l'information contenue sur les derniers axes, laquelle peut souvent être considérée comme du bruit. Dans l'exemple précédent, les données étant toutes quantitatives, l'analyse factorielle considérée est une ACP. On conserve ici l'information contenue sur les 8 premiers axes avec l'argument `ncp=8` de l'ACP, ce qui représente 96 % de l'inertie totale. Pour rester fidèle à l'information contenue dans les données, il importe de conserver un pourcentage d'inertie très élevé, l'objectif ici étant de débruiter et non de résumer au mieux l'information. On effectue ensuite la CAH sur les coordonnées des individus sur les 8 axes factoriels. Cette démarche est implémentée dans la fonction **HCPC** du package FactoMineR pour les situations où on s'intéresse au critère de Ward (critère fondé sur l'inertie donc classiquement utilisé lors d'un enchaînement analyse factorielle - CAH) et à des distances euclidiennes. La fonction est utilisée sur les résultats d'une analyse factorielle, ici une ACP (la même procédure est utilisée en ACM), grâce aux lignes de commandes suivantes :

```
> res.pca<-PCA(decath,quanti.sup=11:12,ncp=8,quali.sup=13,graph=F)
> res.hcpc<-HCPC(res.pca,consol=FALSE)
```

On effectue donc l'ACP en précisant que les variables 11 et 12 sont quantitatives supplémentaires et que la variable 13 est qualitative supplémentaire. L'argument `ncp=8` de la fonction **PCA** permet de spécifier le nombre de composantes retenues. Noter que l'ensemble des composantes principales peut être conservé avec `ncp=Inf`, ce qui reviendrait à construire la classification sur les données brutes.

Si l'argument `consol` de la fonction **HCPC** vaut `TRUE`, cela signifie qu'il y a une étape de consolidation par K-means après le découpage de l'arbre. L'algorithme K-means (voir fiche 8.2) est initialisé avec les centres de gravité des classes obtenues par la CAH.

La fonction **HCPC** retourne un graphe interactif (Fig. 8.3) avec l'arbre hiérarchique permettant de choisir un niveau de coupure ainsi qu'un graphe des gains d'inertie intra-classe. Un niveau de coupure optimal est suggéré par un trait horizontal (ici à la hauteur 0.80 environ).

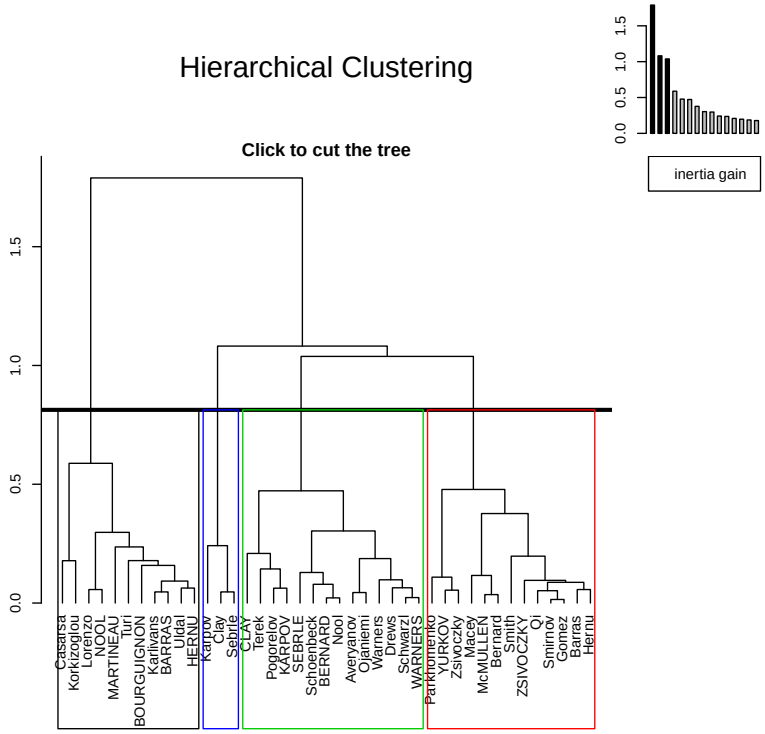


FIGURE 8.3 – Classification Ascendante Hiérarchique fournie par **HCPC**.

Par rapport au graphe fourni par la fonction **agnes**, les individus et les classes sont regroupés en fonction de l'inertie tandis qu'avec la fonction **agnes** le regroupement est effectué en fonction de la racine carrée de l'inertie. Ainsi, l'arbre hiérarchique de la fonction **HCPC** a tendance à être plus aplati : on distingue moins bien les classes pour les premiers regroupements d'individus mais mieux les derniers regroupements de classes (i.e. le haut de l'arbre hiérarchique), ce qui rend le choix du niveau de coupure plus aisé.

Après avoir cliqué sur un niveau de coupure, l'arbre hiérarchique est représenté en trois dimensions sur le premier plan factoriel (Fig. 8.4).
Un dernier graphe donne le premier plan factoriel dans lequel les individus sont représentés avec une couleur différente selon leur classe d'appartenance. Cela revient à considérer la variable de classe comme une variable qualitative supplémentaire en ACP. Les classes obtenues après coupure de l'arbre hiérarchique sont décrites par les variables, par les axes de l'analyse factorielle et par les individus grâce aux commandes suivantes :

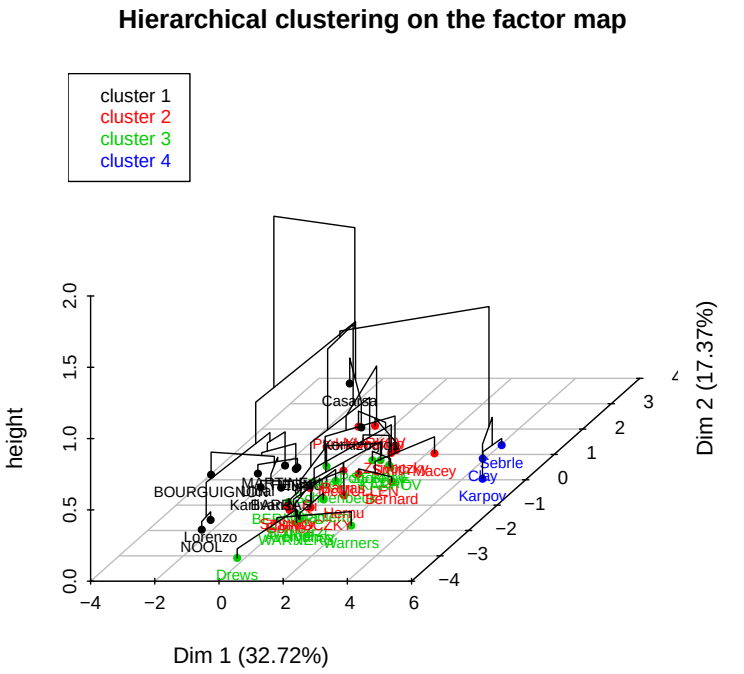


FIGURE 8.4 – Classification Ascendante Hiérarchique fournie par **HCPC**.

```
> res.hcpc$desc.var
> res.hcpc$desc.axes
> res.hcpc$desc.ind
```

On retrouve la description des classes de la fonction **catdes**. La description par les individus donne le parangon de chaque classe, i.e. l'individu le plus proche du centre de chaque classe avec sa distance au centre de la classe dans l'objet `para`.

Des rappels théoriques sur la classification ascendante hiérarchique sont disponibles dans Escofier et Pagès (2016), Lebart *et al.* (2006) et Husson *et al.* (2016). Ce dernier ouvrage détaille des études de cas et les résultats d'une classification effectuée après une analyse factorielle.

8.2 Méthode des K-means

Objet

Cette fiche présente l'algorithme K-means qui permet de construire une partition en K classes, aussi homogènes que possible, d'individus décrits par des variables quantitatives. Le nombre de classes K est supposé connu, déterminé par exemple par un niveau de coupure « naturel » dans un arbre hiérarchique (voir fiche 8.1). Il faut choisir une distance entre individus et un représentant de classe. Souvent, on utilise la distance euclidienne et comme représentant la moyenne du groupe. On peut aussi partitionner autour d'un individu médian par groupe, ce que permet la fonction **pam** du package cluster.

La version la plus simple de l'algorithme K-means consiste d'abord à initialiser l'algorithme en choisissant K individus au hasard qui seront considérés comme les centres des classes. On calcule ensuite la distance entre un individu et chaque centre de classe, et on affecte l'individu à la classe dont il est le plus proche. Une fois tous les individus affectés, on recalcule le centre de gravité de chaque classe, en calculant simplement la moyenne des variables des individus de la classe. Ce processus (calcul des distances des individus aux centres de classe et mise à jour des centres de gravité) est répété jusqu'à convergence.

Le cas où les individus sont décrits par des variables qualitatives est abordé dans la section « Pour aller plus loin ».

Exemple

Reprenons le jeu de données sur les décathloniens (cf. fiche 7.1) et construisons une classification des profils de performances des athlètes. Nous utilisons donc uniquement les dix variables de performances ainsi que les arguments classiques : distance euclidienne et moyenne de la classe comme représentant.

Étapes

1. Importer les données
2. Standardiser ou non les variables
3. Construire la partition
4. Caractériser les classes

Traitement de l'exemple

1. Importer les données

```
> decath <- read.table("decathlon.csv",sep=";",header=TRUE,row.names=1,
    check.names=FALSE)
```

2. Standardiser ou non les variables
Avant de construire la partition, nous pouvons réduire ou non les variables. Pour le jeu de données traité dans cette fiche, nous n'avons pas le choix, la réduction est indispensable car les variables ont des unités différentes. Quand les variables ont les mêmes unités, les deux solutions sont envisageables et impliquent deux analyses différentes. Cette décision est donc cruciale. La réduction permet d'accorder la même importance à chacune des variables dans le calcul de la distance entre individus. Ne pas réduire revient à donner plus d'importance, dans le calcul de la distance entre individus, aux variables ayant un fort écart-type. Ici, nous standardisons les données grâce à la fonction **scale**.

3. Construire la partition
Compte tenu de la classification ascendante hiérarchique effectuée sur ce jeu de données (cf. fiche 8.1), nous construisons ici une partition en $K = 4$ classes :

```
> set.seed(123)
> classe <- kmeans(scale(decath[,1:10]), centers = 4, nstart = 100)
```

La fonction **kmeans** prend comme argument le jeu de données (réduit ou non) et le nombre de classes (ou les centres des classes). Nous spécifions aussi le nombre d'initialisations aléatoires réalisées dans `nstart`. Par défaut, la fonction utilise `nstart=1` mais nous recommandons d'utiliser plus d'initialisations. Ainsi la classification qui a la variance intra-classe la plus faible est retenue. Par défaut, le nombre d'itérations de l'algorithme est de `iter.max = 10`, on peut avoir besoin d'augmenter ce nombre selon les données.

```
> classe
K-means clustering with 4 clusters of sizes 12, 13, 3, 13

Cluster means:
        100m   Longueur       Poids    Hauteur ...
1 -0.2713911 -0.06847836  0.11372756  0.3635437 ...
2  1.0222463 -0.80958444 -0.43964769 -0.3362115 ...
3  -1.5260343  1.92792930  1.65317910  1.2722886 ...
4-0.4195697  0.42788847 -0.04683446 -0.2929723 ...

Clustering vector:
 Sebrle     Clay     Karpov     Macey     Warners   ...
    3           3           3           1           4   ...

Within cluster sum of squares by cluster:
55.90563 100.18299  12.62547  73.25410
```

En sortie, la fonction donne la moyenne des variables pour chaque classe, un vecteur donnant le numéro de classe de chaque individu, ainsi que les variabilités intra-classe, inter-classe et totale.

4. Caractériser les classes

Après avoir regroupé les individus en classes, il est intéressant de décrire ces classes et d'interpréter les similitudes et différences entre ces groupes d'individus. Plutôt que d'énumérer tous les individus qui appartiennent à une même classe, il est préférable de décrire les classes à l'aide des variables. La fonction **catdes** du package FactoMineR permet de caractériser chaque classe à l'aide de variables quantitatives et qualitatives. Afin de pouvoir utiliser cette fonction, nous allons ajouter la variable `classe` au jeu de données initial en dernière (quatorzième) colonne. La variable `classe` est préalablement transformée en facteur et on lui affecte le nom `"classe"`.

```
> decath.comp <- cbind.data.frame(decath,classe=factor(classe$cluster))
```

La fonction **catdes** fournit dans un premier temps les variables les plus liées à la variable de classe. Pour les variables quantitatives, un modèle d'analyse de variance est construit entre la variable quantitative et la variable qualitative de classe et on récupère le R^2 (`Eta2`) ainsi que la probabilité critique du test de nullité associé. Seules les variables significatives, triées par probabilité croissante, sont données. Ici, la variable `Points` est la plus intéressante pour expliquer les classes. Le test n'est qu'indicatif car les variables ont été utilisées pour construire les classes. Puis chaque classe est décrite en détail. La fonction trie les variables quantitatives de la plus à la moins caractérisante en positif (variables pour lesquelles les individus de la classe prennent des valeurs significativement supérieures à la moyenne de l'ensemble des individus) puis de la moins à la plus caractérisante en négatif (variables pour lesquelles les individus de la classe prennent des valeurs significativement inférieures à la moyenne de l'ensemble des individus). Pour les variables qualitatives, ce sont les modalités des variables qui sont triées. Elles sont triées de la plus à la moins caractérisante lorsque la modalité est sur-représentée dans la classe (par rapport aux autres classes) et de la moins à la plus caractérisante lorsque la modalité est sous-représentée dans la classe. Les résultats sont fournis pour les classes 1 et 4 (aucune variable qualitative ne caractérise les classes).

```
> library(FactoMineR)
> catdes(decath.comp, num.var = 14)

Link between the cluster variable and the quantitative variables
================================================================
               Eta2      P-value
Points     0.7959632 7.543476e-13
Perche     0.6072763 1.213468e-07
100m       0.5935880 2.263340e-07
Longueur   0.5526927 1.290912e-06
400m       0.5006104 9.468719e-06
110m.H     0.4773576 2.150925e-05
.....      ......... ............
```

```
Description of each cluster by quantitative variables
======================================================
‘1‘
                   Mean in    Overall     sd in     Overall
          v.test  category    mean        category  sd         p.value
1500m   -2.893339 270.825000  279.024878  5.8957039 11.5300118 0.003811701
Perche  -3.715512   4.511667    4.762439  0.1635967  0.2745887 0.000202792

‘4‘
                   Mean in   Overall     sd in      Overall
          v.test  category   mean        category   sd         p.value
Perche   4.452686 5.046154   4.762439    0.1763536  0.2745887  8.4802e-06
```

Ce qui caractérise le plus les individus de la classe 1 est le fait qu'ils sautent moins haut à la perche que les autres (valeur-test inférieure à -2) et qu'ils courent vite le 1 500 m (ils ont un temps plus faible que les autres). Les individus de la classe 4 sautent haut à la perche. Pour les individus de cette classe, la hauteur moyenne est de 5.05 m contre 4.76 m pour l'ensemble des individus (y compris ceux de la classe 4). Une valeur-test supérieure à 2 en valeur absolue signifie ici que la moyenne de la classe est significativement différente de la moyenne générale : un signe positif (resp. négatif) de la valeur-test indique que la moyenne de la classe est supérieure (resp. inférieure) à la moyenne générale.

Pour aller plus loin

Des rappels théoriques sur les méthodes de partitionnement sont disponibles dans Lebart *et al.* (2006) ou Saporta (2011).

L'algorithme K-means est sensible aux points initiaux. Pour éviter l'influence des outliers, il est possible d'utiliser l'algorithme K-means++, où l'étape d'initialisation est modifiée comme suit : le premier centre de gravité est choisi au hasard parmi les individus, les suivants sont sélectionnés avec une probabilité proportionnelle à la distance au centre le plus proche parmi ceux déjà construits. Cet algorithme K-means++ est disponible via la fonction **kmeanspp** du package LICORS.

Si les individus sont décrits par des variables qualitatives, il n'est pas possible de construire une classification avec la méthode décrite dans cette fiche. Cependant, à l'image de la CAH, il est possible de se ramener à des données quantitatives en effectuant au préalable une Analyse des Correspondances Multiples (cf. fiche 7.3) et en récupérant les coordonnées des individus sur les axes factoriels. Si on ne conserve pas toutes les dimensions, mais seulement les premières, cela permet de « débruiter » les données, les dernières dimensions étant considérées comme du bruit, et de stabiliser la classification. Dans cette optique, pour des variables quantitatives, il est également possible de réaliser l'algorithme K-means sur les composantes principales de l'Analyse en Composantes Principales.

8.3 Modèles de mélange

Objet

L'objectif de cette fiche est d'effectuer une partition ou classification en K classes de n individus décrits par p variables quantitatives $X = (X_1, ..., X_p)$ en utilisant une approche probabiliste basée sur les modèles de mélange. On note $Z \in \{1, \dots, K\}$ la classe de l'observation k qui est inconnue et à estimer à partir des données $X_{n\times p}$. Chaque classe est décrite par sa distribution de probabilité, qui est supposée normale :

$$f_k(x) = \mathbb{P}(X = x|Z = k) \sim \mathcal{N}(\mu_k, \Sigma_k).$$

On fait aussi l'hypothèse d'une distribution multinomiale : $\tilde{Z} \sim \mathcal{M}(1; p_1, \dots, p_K)$ où $p_k = \mathbb{P}(Z = k) = \mathbb{P}(\tilde{Z}_k = 1)$ est la probabilité d'appartenir à la classe k. La distribution marginale des données X qui en résulte est ainsi un mélange de lois normales, d'où le nom de modèle de mélange : $X \sim \sum_{k=1}^{K} p_k f_k(x) = f_X(x)$ et la formule de Bayes donne la distribution conditionnelle

$$t_k(x) = \mathbb{P}(Z = k|X = x) = \frac{p_k f_k(x)}{f_X(x)},$$

c'est-à-dire la probabilité d'affectation à la classe k sachant que $X = x$.

Si on suppose que tous les paramètres $\theta = (p_k, \mu_k, \Sigma_k)_{1\leq k\leq K}$ du mélange sont connus (nous les estimerons à partir des données), alors chaque observation x est classée dans le groupe k qui maximise la distribution conditionnelle $t_k(x)$, c'est-à-dire $\hat{k} = \underset{k}{\text{argmax}}\ t_k(x)$.

Comme les paramètres θ sont inconnus, on les estime par maximisation de la vraisemblance $L(x_1, \dots, x_n, \theta)$. Cependant, en général, il n'y a pas de solution explicite et on utilise un algorithme itératif de type EM (expectation-maximization, voir par exemple Dempster *et al.* (1977)) qui vise à maximiser la vraisemblance des observations via des maximisations itérées de la vraisemblance dite « complétée » $L(x_1, \dots, x_n, z_1, \dots, z_n, \theta)$.

La méthodologie précédente suppose le nombre K de classes connu. Néanmoins, si tel n'est pas le cas, on peut appliquer des critères basés sur la vraisemblance pénalisée, par exemple BIC ou AIC : l'idée est alors, pour une certaine plage de valeurs de $K \in \{1, \dots, K_{\max}\}$, d'optimiser ce critère pour chaque K, puis de trouver le $\hat{K} \in \{1, \dots, K_{\max}\}$ qui lui-même maximise ce critère parmi les $K_{\max}$ valeurs envisagées.

Notons que le nombre de paramètres à estimer dans ces modèles est assez conséquent : $K \times p(p + 1)/2$ pour les Σ_k, $K \times p$ pour les μ_k et $K - 1$ pour les p_k (car $\sum_{k=1}^{K} p_k = 1$). Pour pallier ce problème, différentes simplifications sont envisageables. On peut par exemple supposer que les matrices de variances sont identiques dans chaque groupe, i.e. $\Sigma_k = \Sigma$. On se rapproche alors de l'analyse

discriminante linéaire (fiche 9.6 p. 294), si ce n'est que l'appartenance des individus au groupe est inconnue. On peut aussi considérer des matrices de variances sphériques $\Sigma_k = \lambda_k I_p$, ce qui signifie que les variables sont indépendantes au sein de chaque groupe. Notons que la classification par K-means (fiche 8.2 p. 252) peut être présentée comme un cas particulier des modèles de mélange avec des matrices de variances sphériques. Les autres paramétrisations des matrices de variances les plus utilisées consistent à décomposer, en utilisant les notations des auteurs du package mclust, $\Sigma_k = \lambda_k D_k A_k D_k'$ avec D_k matrice unitaire des vecteurs propres (orientation), A_k matrice diagonale dont les éléments sont proportionnels aux valeurs propres (forme), et λ_k une constante de proportionnalité associée (volume). Il est alors possible de spécifier ce qui est commun ou spécifique à chaque groupe : mêmes directions mais volumes différents, etc.

De nombreux packages implémentent la classification par modèles de mélange : citons mclust pour les variables quantitatives, Rmixmod pour des variables quantitatives et qualitatives, HDclassif pour des données en grandes dimension, etc.

Exemple

Nous utilisons le jeu de données sur les crabes de l'espèce Leptograpsus variegatus du package HDclassif qui comporte 8 colonnes dont les 5 dernières sont des mesures morphologiques. Ces variables sont renseignées pour 50 crabes, de deux couleurs et des deux sexes soit 200 crabes en tout.

Les 5 variables considérées sont :
- `FL` : taille du lobe frontal (mm) ;
- `RW` : largeur arrière (mm) ;
- `CL` : longueur de la carapace (mm) ;
- `CW` : largeur de la carapace (mm) ;
- `BD` : profondeur du corps (mm).

Le but est de faire des groupes de crabes en fonction de leurs caractéristiques.

Étapes

1. Importer les données
2. Choix du modèle et du nombre de classes
3. Construire la classification
4. Caractériser les classes

Traitement de l'exemple

1. Importer les données

```
> library(HDclassif)
> data(crabs)
> don <- crabs[ , 4:ncol(crabs)]
> summary(don)
```

2. Choix du modèle et du nombre de classes
La fonction **mclustBIC** de la librairie mclust construit les modèles associés aux différentes paramétrisations des matrices de variances et ce pour un nombre de classes allant par défaut de 1 à 9.

```
> library(mclust)
> res.BIC <- mclustBIC(don, verbose = F)
> plot(res.BIC)
> summary(res.BIC)
```

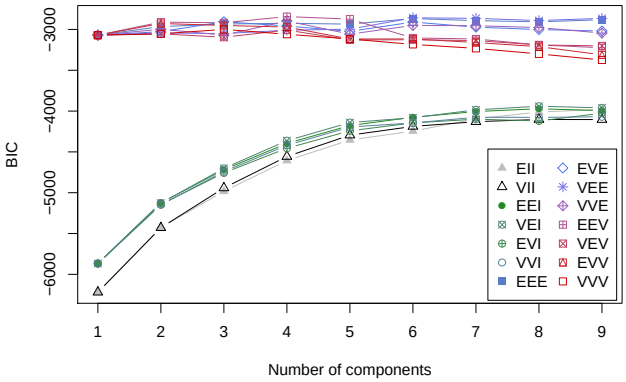


FIGURE 8.5 – Représentation des valeurs du BIC pour 14 paramétrisations des matrices de variances.

La figure 8.5 donne les valeurs du critère BIC pour les 14 paramétrisations des matrices de variances disponibles et jusqu'à 9 groupes d'individus. La lettre « V » correspond à variable , « E » à égal et « I » à la matrice identité. Les modèles sont définis par des codes à trois lettres pour volume, forme et orientation (dans cet ordre). Par exemple, EVI désigne un modèle dans lequel les volumes de toutes les classes sont égaux (E), les formes des classes peuvent varier (V), et l'orientation est l'identité (I), c'est-à-dire $\Sigma_k = \lambda A_k$. Le « meilleur » modèle est celui qui a le BIC le plus élevé.

3. Construction de la classification
La fonction **Mclust** permet de réaliser la classification par modèle de mélange en utilisant les contraintes spécifiées par l'argument `x`.

```
> mod <- Mclust(don, x = res.BIC)
> summary(mod)
----------------------------------
Gaussian finite mixture model
fitted by EM algorithm
----------------------------------

Mclust EEV (ellipsoidal, equal
volume and shape) model with 4
components:

 log.likelihood   n df       BIC
      -1241.006 200 68 -2842.298
      ICL
 -2854.29

Clustering table:
 1  2  3  4
60 55 39 46
```

Le modèle retenu ici est celui avec 4 classes et des matrices de variances par groupe qui ont les mêmes valeurs propres mais des vecteurs propres spécifiques à chacun des groupes, soit $\Sigma_k = \lambda D_k A D_k'$. L'effectif de chaque classe est également spécifié. On peut le retrouver comme suit :

```
> table(mod$classification)
```

Les estimations des paramètres μ_k et Σ_k sont quant à elles disponibles dans la sortie `mod$parameters` : on retrouve les 4 centres de gravité des classes (4 vecteurs de dimension 5) et les 4 matrices de variances. Les probabilités d'appartenance à chaque classe $t_k(x)$ sont dans l'objet `mod$z` sous la forme d'une matrice de taille 200 par 4 dont chaque cellule (i, k) correspond à la probabilité que l'individu i appartienne à la classe k.

4. Caractériser les classes

Après avoir regroupé les individus en classes, il est intéressant de décrire ces classes afin d'interpréter les similitudes et différences entre ces groupes d'individus. Nous pouvons utiliser la fonction **catdes** du package FactoMineR. Cette fonction permet de caractériser chaque classe à l'aide de variables quantitatives et qualitatives. Pour utiliser cette fonction, nous ajoutons la variable `classe` au jeu de données initial en dernière colonne. La variable `classe` est préalablement transformée en facteur et on lui affecte le nom `"classe"`.

```
> don.comp <- cbind.data.frame(don,classe=factor(mod$classification))
```

La fonction **catdes** fournit dans un premier temps les variables les plus liées à la variable de classe. Pour les variables quantitatives, on ajuste un modèle d'analyse de

variance entre la variable quantitative et la variable qualitative de classe et on récupère le R^2 (`Eta2`) ainsi que la probabilité critique du test de nullité associé. Seules les variables significatives sont données. Ici, la variable `FL` est très importante pour expliquer les classes. Le test n'est qu'indicatif car les variables ont été utilisées pour construire les classes. Puis chaque classe est décrite plus en détail. La fonction trie les variables quantitatives de la plus caractérisante à la moins caractérisante en positif (variables pour lesquelles les individus de la classe prennent des valeurs significativement supérieures à la moyenne de l'ensemble des individus) puis de la moins caractérisante à la plus caractérisante en négatif (variables pour lesquelles les individus de la classe prennent des valeurs significativement inférieures à la moyenne de l'ensemble des individus). Pour les variables qualitatives, ce sont les modalités des variables qui sont triées. Ces modalités sont triées de la plus caractérisante à la moins caractérisante lorsque la modalité est sur-représentée dans la classe (par rapport aux autres classes) et de la moins caractérisante à la plus caractérisante lorsque la modalité est sous-représentée dans la classe. Les résultats sont fournis pour la description des classes 1 et 2.

```
> library(FactoMineR)
> catdes(don.comp, num.var = 6)
Link between the cluster variable and the quantitative variables
================================================================
        Eta2      P-value
FL 0.3063318 1.700502e-15
RW 0.2819780 4.806351e-14
BD 0.2791240 7.055350e-14
CL 0.2262530 6.562003e-11
CW 0.2025411 1.198917e-09

Description of each cluster by quantitative variables
=====================================================
$`1`
      v.test   Mean in  Overall     sd in  Overall       p.value
               category     mean  category       sd
RW -4.173975  11.57833  12.7385  2.587154  2.566898 2.993307e-05
CW -5.864227  31.42833  36.4145  6.924500  7.852251 4.512294e-09
CL -6.513962  27.09667  32.1055  5.998416  7.101163 7.319396e-11
BD -7.219174  11.36000  14.0305  2.758575  3.416200 5.230442e-13
FL -7.322878  12.81833  15.5830  2.668926  3.486576 2.427082e-13

$`2`
      v.test   Mean in  Overall     sd in  Overall       p.value
               category     mean  category       sd
BD 2.567361      15.04  14.0305  3.542654   3.4162    0.01024758
```

Les individus de la classe 1 ont des caractéristiques morphologiques plus petites que les autres crabes. Ils ont une largeur à l'arrière RW de 31.42, contre 36.41 pour

l'ensemble des crabes. Les autres variables caractérisent aussi de façon significative les individus de cette classe (valeur-test supérieure à 2 en valeur absolue) : un signe positif (resp. négatif) de la valeur-test indique que la moyenne de la classe est supérieure (resp. inférieure) à la moyenne générale. Cette classe est donc une classe de « petits » crabes. La classe 2 représente des crabes qui ont une profondeur du corps grande par rapport aux autres : 15.04 en moyenne contre 14.03 pour la population de crabes.

Pour aller plus loin

En grande dimension, il est possible de contraindre encore plus les matrices de variances. Une solution implémentée dans le package HDclassif consiste à combiner des techniques de réduction de la dimension (comme l'analyse en composantes principales) et de classification. Par exemple, en utilisant ici les notations du package HDclassif, on écrit $\Sigma_k = Q_k D_k Q_k'$ avec Q_k la matrice des vecteurs propres et D_k la matrice diagonale des valeurs propres. Il est possible de contraindre la matrice D_k en imposant que les $p - d_k$ dernières valeurs propres soient égales. Ceci s'apparente à de la réduction de la dimension et revient à supposer que les individus de la classe k peuvent être bien représentés dans un espace à d_k dimensions. De nombreuses paramétrisations sont disponibles et 28 modèles sont implémentés dans ce package. L'avantage de cette stratégie est qu'elle permet de visualiser les classes sur leurs sous-espaces propres, ce qui peut aider à l'interprétation : on décrit et interprète chaque classe comme en ACP (voir fiche 7.1). Cependant elle nécessite de choisir plus de paramètres : le nombre de classes K et la dimension d_k de chaque classe. Notons que ces techniques s'apparentent à des versions probabilistes des méthodes de type « subspace clustering ».

Nous illustrons brièvement l'utilisation du package sur le jeu de données de reconnaissance optique de caractères utilisé pour l'étude du service postal des États-Unis (USPS), qui consiste à reconnaître des écritures manuscrites. Nous disposons de l'ensemble de données USPS358 qui correspond à 1756 images de chiffres écrits et représentées sous la forme d'une matrice de taille 1756 par 256. L'ensemble de données USPS358 est un sous-ensemble des données USPS original où seuls les chiffres 3, 5 et 8 ont été conservés. Chaque chiffre est une image de dimensions 16 par 16 avec des niveaux de gris qui a été transformée sous la forme d'un vecteur de dimension 256.

Nous effectuons la classification par modèle de mélange sur ces données. Nous chargeons l'environnement `USPS358.Rdata` qui contient la matrice de données dans une matrice `X` mais aussi un vecteur `cls` qui contient la classe des images (3, 5 ou 8). En effet, même si c'est un problème de classification non-supervisée (nous ne connaissons théoriquement pas le label de chaque image), nous disposons ici de l'information sur l'objectif qu'on aimerait atteindre (classer ensemble toutes les images correspondant à un même chiffre), ce qui nous permettra d'évaluer la procédure de classification.

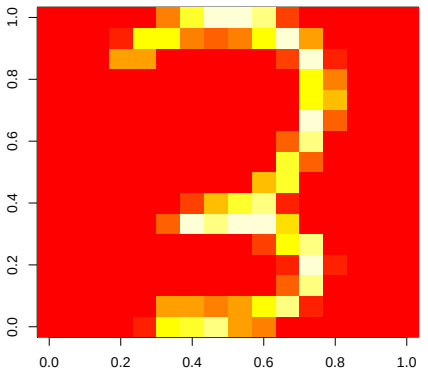


FIGURE 8.6 – Cinquième image du jeu de données `USPS358`.

Pour visualiser l'image correspondant à la cinquième ligne de la matrice, nous utilisons la fonction **image** qui s'applique sur l'image de dimension 16 par 16 :

```
> load("USPS358.Rdata")
> image(matrix(t(X[5,]), ncol = 16))
```

Nous utilisons ensuite la fonction **hddc** en lui spécifiant ici le nombre de classes souhaité. La fonction choisit le meilleur modèle (la meilleure paramétrisation) à 3 classes. L'appartenance à chaque classe est donnée dans la sortie `class`. Il est alors possible de comparer les résultats aux données labellisées et de calculer par exemple un indice de Rand pour évaluer l'accord entre les deux classifications. Nous utilisons ici la fonction **adjustedRandIndex** du package `mclust` :

```
> res.hddc <- hddc(X, 3)
> table(res.hddc$class,cls)
cls
      1   2   3
  1 596   5  24
  2  43 549  26
  3  19   2 492
> adjustedRandIndex(res.hddc$class,cls)
[1] 0.8055
```

L'indice de Rand correspond au pourcentage de paires d'observations en accord (c'est-à-dire dans la même classe ou dans des classes séparées dans les deux partitions comparées). Sous l'hypothèse nulle de deux partitions aléatoires, sa version espérée n'est pas nulle et l'indice de Rand ajusté corrige cela. Il est au plus égal à 1 quand les partitions sont négatives, vaut 0 quand on est à l'hypothèse nulle et peut être négatif.

Pour plus de détails sur l'utilisation du package, voir Bergé *et al.* (2012).

Chapitre 9

Méthodes usuelles de régression

Ce chapitre expose quelques méthodes de régression permettant d'expliquer une variable Y en fonction de variables quantitatives et/ou qualitatives. Les fiches 9.1 à 9.4 concernent les méthodes les plus classiques du modèle linéaire : régression simple (une seule variable explicative quantitative, fiche 9.1), régression multiple (plusieurs variables explicatives quantitatives, fiche 9.2), analyse de la variance (une ou plusieurs variables explicatives qualitatives, fiche 9.3) et analyse de la covariance (des variables explicatives à la fois quantitatives et qualitatives, fiche 9.4).

Les deux méthodes suivantes, régression logistique (fiche 9.5) et analyse discriminante linéaire (fiche 9.6), permettent d'expliquer une variable qualitative Y, le plus souvent binaire, en fonction de variables quantitatives et qualitatives. Les arbres de régression ou de classification expliquent quant à eux une variable Y quantitative ou qualitative en fonction de variables quantitatives et/ou qualitatives (fiche 9.7). Enfin, la régression PLS permet d'expliquer une variable quantitative Y en fonction de nombreuses variables explicatives quantitatives (fiche 9.8).

Dans chacune des fiches, on présente succinctement la méthode, puis un jeu de données et une problématique avant d'énumérer les principales étapes de l'analyse. L'exemple est alors traité via l'enchaînement des différentes instructions R et les résultats numériques et graphiques sont brièvement commentés.

Le site `https://r-stat-sc-donnees.github.io` permet de retrouver les jeux de données. On peut les sauvegarder sur sa machine avant de les importer en R ou les importer directement comme suit :

```
> read.table("https://r-stat-sc-donnees.github.io/Fiche.csv",
    header=TRUE, dec=".",sep=";")
```

9.1 Régression simple

Objet

La régression linéaire simple est une méthode statistique permettant de modéliser la relation linéaire entre deux variables quantitatives dans un objectif explicatif et/ou prévisionnel. Nous disposons d'une variable explicative (notée X) et une variable à expliquer (notée Y), liées par le modèle suivant :

$$Y = \beta_0 + \beta_1 X + \varepsilon,$$

où ε est la variable de bruit ou erreur de mesure. Les paramètres β_0 et β_1 sont inconnus. L'objectif est de les estimer à partir d'un échantillon de n couples $(x_1, y_1), \ldots, (x_n, y_n)$. Le modèle s'écrit sous forme indicée :

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i.$$

Le coefficient β_0 correspond à l'ordonnée à l'origine (intercept pour les anglo-saxons) et β_1 à la pente. Pour estimer les paramètres, on minimise la somme des carrés des écarts entre la variable à expliquer et la droite de régression. On minimise donc la quantité suivante :

$$\min_{\beta_0,\beta_1} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2.$$

Une fois les paramètres estimés (notés $\hat{\beta}_0$ et $\hat{\beta}_1$), on obtient la droite de régression :

$$f(x) = \hat{\beta}_0 + \hat{\beta}_1 x,$$

ce qui permet éventuellement d'effectuer des prévisions. Les valeurs ajustées ou lissées sont définies par :

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i,$$

et les résidus sont alors estimés par :

$$\hat{\varepsilon}_i = y_i - \hat{y}_i.$$

L'analyse des résidus est primordiale car elle permet de vérifier l'ajustement individuel du modèle (point aberrant) et l'ajustement global en vérifiant par exemple qu'il n'y a pas de structure.

Exemple

La pollution de l'air constitue actuellement une des préoccupations majeures de santé publique. De nombreuses études épidémiologiques ont permis de mettre en

évidence l'influence sur la santé de certains composés chimiques comme le dioxyde de souffre (SO_2), le dioxyde d'azote (NO_2), l'ozone (O_3) ou des particules sous forme de poussières contenues dans l'air.

Des associations de surveillance de la qualité de l'air existent sur tout le territoire français et mesurent la concentration des polluants. Elles enregistrent également les conditions météorologiques comme la température, la nébulosité, le vent, etc.

Nous souhaitons analyser ici la relation entre le maximum journalier de la concentration en ozone (en $\mu g/m^3$) et la température. Nous disposons de 112 données relevées durant l'été 2001 à Rennes.

Étapes

1. Importer les données
2. Représenter le nuage de points (x_i, y_i)
3. Estimer les paramètres
4. Tracer la droite de régression
5. Analyser les résidus
6. Prévoir une nouvelle valeur

Traitement de l'exemple

1. Importer les données

Nous importons les données et résumons les variables d'intérêt, ici `maxO3` et `T12` :

```
> ozone<-read.table("ozone.txt",header=TRUE)
> summary(ozone[,c("maxO3","T12")])
     maxO3             T12
 Min.   : 42.00   Min.   :14.00
 1st Qu.: 70.75   1st Qu.:18.60
 Median : 81.50   Median :20.55
 Mean   : 90.30   Mean   :21.53
 3rd Qu.:106.00   3rd Qu.:23.55
 Max.   :166.00   Max.   :33.50
```

2. Représenter le nuage de points (x_i, y_i)

```
> plot(maxO3~T12,data=ozone,pch=15,cex=.5)
```

Chaque point du graphique (Fig. 9.1) représente, pour un jour donné, une mesure de la température à 12h et le pic d'ozone de la journée. Au vu de ce graphique, la liaison entre la température et la concentration d'ozone semble plutôt linéaire.

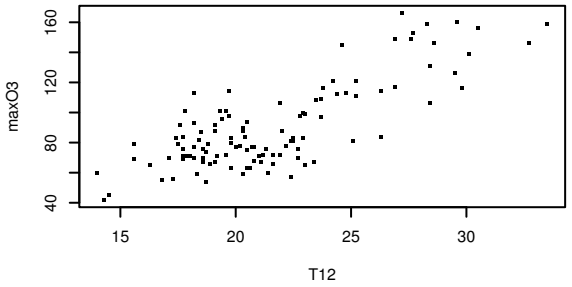


FIGURE 9.1 – Représentation des couples (x_i, y_i).

3. Estimer les paramètres
La fonction **lm** (linear model) permet d'ajuster un modèle linéaire :

```
> reg.s <- lm(maxO3~T12,data=ozone)
> summary(reg.s)
Call:
lm(formula = maxO3 ~ T12, data = ozone)

Residuals:
     Min       1Q   Median       3Q      Max
-38.0789 -12.7352   0.2567  11.0029  44.6714

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -27.4196     9.0335  -3.035    0.003 **
T12           5.4687     0.4125  13.258   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 17.57 on 110 degrees of freedom
Multiple R-Squared: 0.6151,     Adjusted R-squared: 0.6116
F-statistic: 175.8 on 1 and 110 DF,  p-value: < 2.2e-16
```

On obtient entre autres une matrice `Coefficients` qui comporte pour chaque paramètre (chaque ligne) 4 colonnes : son estimation (colonne `Estimate`), son écart-type estimé (`Std. Error`), la valeur observée de la statistique du test d'hypothèse $H_0 : \beta_i = 0$ contre $H_1 : \beta_i \neq 0$. Enfin, la probabilité critique (`Pr(>|t|)`) donne, pour la statistique de test sous H_0, la probabilité de dépasser la valeur estimée.

Les coefficients β_0 et β_1 sont estimés par -27.4 et 5.5. Les tests de significativité des coefficients donnent ici des probabilités critiques de 0.003 et environ $2\ 10^{-16}$. Ainsi l'hypothèse nulle de chacun des tests est rejetée au profit de l'hypothèse alternative. La probabilité critique inférieure à 5 % pour la constante indique que

la constante doit apparaître dans le modèle. La probabilité critique inférieure à 5 % pour la pente indique une liaison significative entre `maxO3` et `T12`.

Le résumé de l'étape d'estimation fait figurer l'estimation de l'écart-type résiduel σ, qui vaut ici 17.57, ainsi que le nombre de degrés de liberté associé $n - 2 = 110$.

La valeur du R^2 est également donnée, ainsi que le R^2_a ajusté. La valeur du R^2 est assez élevée ($R^2 = 0.6151$), ce qui corrobore l'intuition de la relation linéaire entre les deux variables. En d'autres termes, 61 % de la variabilité de `maxO3` est expliquée par `T12`.

La dernière ligne, surtout utile en régression multiple, indique le résultat du test de comparaison entre le modèle utilisé et le modèle n'utilisant que la constante comme variable explicative.

Nous pouvons consulter la liste des différents résultats (composantes de la liste, cf. § 1.5.7, p. 23) de l'objet `reg.s` avec :

```
> names(reg.s)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

On peut alors récupérer les coefficients avec :

```
> reg.s$coef
(Intercept)         T12
 -27.419636    5.468685
```

ou en utilisant la fonction **coef(reg.s)**.
Pour ajuster un modèle sans la constante, on procède de la manière suivante :

```
> reg.ss.constante <- lm(maxO3~T12-1, data=ozone)
```

4. Tracer la droite de régression
Nous pouvons simplement appliquer la commande **abline**`(reg.s)` mais nous préférons nous restreindre au domaine d'observation de la variable explicative. Nous créons donc une grille de points sur les abscisses et appliquons le modèle trouvé sur cette grille :

```
> plot(maxO3~T12,data=ozone,pch=15,cex=.5)
> grillex <- seq(min(ozone[,"T12"]),max(ozone[,"T12"]),length=100)
> grilley <- reg.s$coef[1]+reg.s$coef[2]*grillex
> lines(grillex,grilley,col=2)
```

Cela donne le graphique suivant :

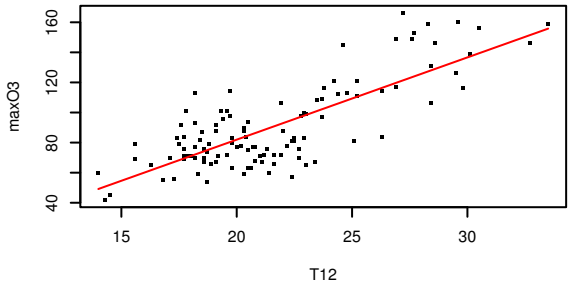


FIGURE 9.2 – Nuage de points et droite de régression.

5. Analyser les résidus
Les résidus sont obtenus par la fonction **residuals**, cependant les résidus obtenus ne sont pas de même variance (hétéroscédastiques). Nous utilisons donc des résidus studentisés, qui eux sont de même variance.

```
> res.simple<-rstudent(reg.s)
> plot(res.simple,pch=15,cex=.5,ylab="Résidus",ylim=c(-3,3))
> abline(h=c(-2,0,2),lty=c(2,1,2))
```

Cela donne le graphique suivant :

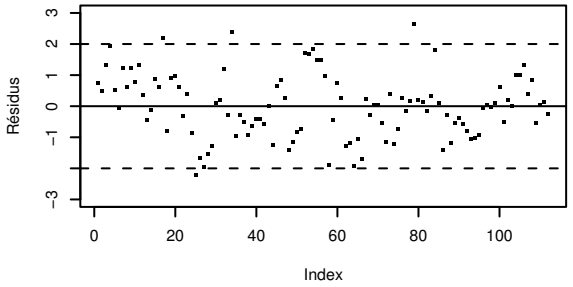


FIGURE 9.3 – Représentation des résidus.

En théorie 95 % des résidus studentisés se trouvent dans l'intervalle $[-2, 2]$. C'est le cas ici puisque 4 résidus seulement se trouvent à l'extérieur de cet intervalle.

D'autres diagnostics sur les résidus sont disponibles en appliquant la fonction **plot** sur l'objet `reg.s`.

6. Prévoir une nouvelle valeur
Ayant une nouvelle observation `xnew`, il suffit d'utiliser les estimations pour prévoir la valeur de Y correspondante. Cependant, la valeur prédite est de peu d'intérêt

sans l'intervalle de confiance associé. Voyons cela sur un exemple. Nous disposons d'une nouvelle observation de la température `T12` égale à 19 degrés pour le 1er octobre 2001.

```
> xnew <- 19
> xnew <- as.data.frame(xnew)
> colnames(xnew) <- "T12"
> predict(reg.s,xnew,interval="pred")
          fit      lwr      upr
[1,] 76.48538 41.4547 111.5161
```

Il faut noter que l'argument `xnew` de la fonction **predict** doit être un data-frame avec les mêmes noms de variables explicatives (ici `T12`). La valeur prévue est 76.5 et l'intervalle de prévision à 95 % est $[41.5, 111.5]$. Pour représenter sur un même graphique l'intervalle de confiance d'une valeur lissée et l'intervalle de confiance d'une prévision, nous calculons ces intervalles pour l'ensemble de points ayant servi à dessiner la droite de régression. Nous faisons figurer les deux sur le même graphique (Fig. 9.4).

```
> grillex.df <- data.frame(grillex)
> dimnames(grillex.df)[[2]]<-"T12"
> ICdte <- predict(reg.s, new=grillex.df, interval="conf", level=0.95)
> ICprev <- predict(reg.s, new=grillex.df, interval="pred", level=0.95)
> plot(maxO3~T12, data=ozone, pch=15, cex=.5)
> matlines(grillex,cbind(ICdte, ICprev[,-1]), lty=c(1,2,2,3,3),
   col=c("black","red","red","blue","blue"))
> legend("topleft", lty=3:2, col=c("blue","red"), c("prev","conf"))
```

On obtient le graphique suivant :

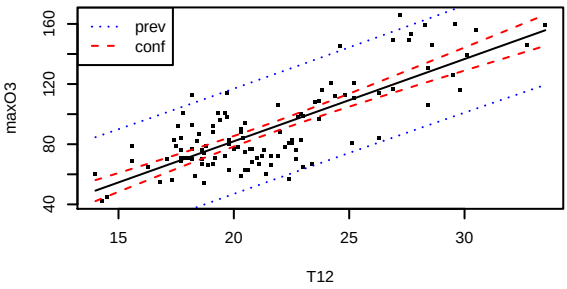


FIGURE 9.4 – Intervalles de confiance et de prévision.

Pour aller plus loin

Des rappels sur la régression sont disponibles dans beaucoup d'ouvrages comme Cornillon et Matzner-Løber (2010) ou Saporta (2011).

9.2 Régression multiple

Objet

La régression linéaire multiple consiste à expliquer et/ou prédire une variable quantitative Y par p variables quantitatives $X_1, \cdots, X_p$. Le modèle de régression multiple est une généralisation du modèle de régression simple (cf. fiche 9.1). Nous supposons donc que les n données collectées suivent le modèle suivant :

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i, \qquad i = 1, \cdots, n \tag{9.1}$$

où x_{ij} est la valeur prise par l'individu i pour la variable j, les valeurs x_{ij} étant rangées dans la matrice du plan d'expérience notée X. Les paramètres β_j du modèle sont inconnus et à estimer. Le paramètre β_0 (`intercept` dans les logiciels anglo-saxons) correspond à la constante du modèle. Les ε_i sont des variables aléatoires inconnues et représentent les erreurs de mesure.

En écrivant le modèle 9.1 sous forme matricielle, nous obtenons :

$$Y_{n\times 1} = X_{n\times(p+1)} \ \beta_{(p+1)\times 1} + \varepsilon_{n\times 1}, \tag{9.2}$$

où la première colonne de X est le vecteur uniquement constitué de 1. À partir des observations, nous estimons les paramètres inconnus du modèle en minimisant le critère des moindres carrés :

$$\hat{\beta} = \underset{\beta_0,\cdots,\beta_p}{\operatorname{argmin}} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 = \underset{\beta\in\mathbb{R}^{p+1}}{\operatorname{argmin}} (Y - X\beta)'(Y - X\beta).$$

Si la matrice X est de plein rang, l'estimateur des moindres carrés $\hat{\beta}$ de β est :

$$\hat{\beta} = (X'X)^{-1}X'Y.$$

Une fois les paramètres estimés, on peut calculer les valeurs lissées ou ajustées :

$$\hat{y}_i \quad = \quad \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip},$$

ou prévoir de nouvelles valeurs. La différence entre la valeur observée et la valeur lissée est par définition le résidu estimé :

$$\hat{\varepsilon}_i \quad = \quad y_i - \hat{y}_i.$$

L'analyse des résidus est primordiale car elle permet de vérifier l'ajustement individuel du modèle (point aberrant) et l'ajustement global en vérifiant, par exemple, qu'il n'y a pas de structure.

Exemple

Nous reprenons le jeu de données ozone présenté en détail dans la fiche 9.1 (p. 264). Nous souhaitons analyser ici la relation entre le maximum journalier de la concentration en ozone (en $\mu g/m^3$) et la température à différentes heures de la journée, la nébulosité à différentes heures de la journée, la projection du vent sur l'axe Est-Ouest à différentes heures de la journée et la concentration maximale de la veille du jour considéré. Nous disposons de 112 données relevées durant l'été 2001.

Étapes

1. Importer les données
2. Représenter les variables
3. Estimer les paramètres
4. Choix de variables
5. Analyser les résidus
6. Prévoir une nouvelle valeur

Traitement de l'exemple

1. Importer les données
Nous importons les données et sélectionnons les variables d'intérêt, ici la variable à expliquer `maxO3` et les variables explicatives quantitatives : température, nébulosité, projection du vent sur l'axe Est-Ouest à 9h, 12h et 15h, ainsi que le `maxO3v`. Seul le résumé des 4 premières variables est fourni.

```
> ozone <- read.table("ozone.txt",header=TRUE)
> ozone.m <- ozone[,1:11]
> summary(ozone.m)
     maxO3              T9              T12             T15
 Min.   : 42.00   Min.   :11.30   Min.   :14.00   Min.   :14.90
 1st Qu.: 70.75   1st Qu.:16.20   1st Qu.:18.60   1st Qu.:19.27
 Median : 81.50   Median :17.80   Median :20.55   Median :22.05
 Mean   : 90.30   Mean   :18.36   Mean   :21.53   Mean   :22.63
 3rd Qu.:106.00   3rd Qu.:19.93   3rd Qu.:23.55   3rd Qu.:25.40
 Max.   :166.00   Max.   :27.00   Max.   :33.50   Max.   :35.50
```

2. Représenter les variables
Il est toujours utile de résumer les variables et d'effectuer une analyse univariée de chaque variable (histogramme, par exemple). Il est possible aussi de représenter les variables deux à deux sur un même graphique grâce à la fonction **pairs**, ici (**pairs**`(ozone.m)`). L'ACP avec la variable à expliquer en illustratif (voir fiche 7.1) permet également d'explorer les données.

3. Estimer les paramètres

Pour estimer les paramètres, il faut écrire le modèle. Nous utilisons, comme pour la régression simple, **lm** (linear model) avec une formule (cf. § A.1, p. 401). Ici, il y a 10 variables explicatives et il serait donc fastidieux de toutes les écrire. Le logiciel R permet une écriture rapide où le point (`.`) indique que toutes les variables du jeu de données, exceptée celle qui correspond à Y, sont explicatives :

```
> reg.mul <- lm(maxO3~., data=ozone.m)
> summary(reg.mul)

Call:
lm(formula = maxO3 ~ ., data = ozone.m)

Residuals:
    Min      1Q  Median      3Q     Max
-53.566  -8.727  -0.403   7.599  39.458

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.24442   13.47190   0.909   0.3656
T9          -0.01901    1.12515  -0.017   0.9866
T12          2.22115    1.43294   1.550   0.1243
T15          0.55853    1.14464   0.488   0.6266
Ne9         -2.18909    0.93824  -2.333   0.0216 *
Ne12        -0.42102    1.36766  -0.308   0.7588
Ne15         0.18373    1.00279   0.183   0.8550
Vx9          0.94791    0.91228   1.039   0.3013
Vx12         0.03120    1.05523   0.030   0.9765
Vx15         0.41859    0.91568   0.457   0.6486
maxO3v       0.35198    0.06289   5.597 1.88e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14.36 on 101 degrees of freedom
Multiple R-Squared: 0.7638,     Adjusted R-squared: 0.7405
F-statistic: 32.67 on 10 and 101 DF,  p-value: < 2.2e-16
```

Comme en régression linéaire simple (cf. fiche 9.1), les sorties du logiciel donnent une matrice (sous le nom `Coefficients`) qui comporte quatre colonnes pour chaque paramètre : son estimation (colonne `Estimate`), son écart-type estimé (`Std. Error`), la valeur observée de la statistique de test d'hypothèse $H_0 : \beta_i = 0$ contre $H_1 : \beta_i \neq 0$ et la probabilité critique (`Pr(>|t|)`). Cette dernière donne, pour la statistique de test sous H_0, la probabilité de dépasser la valeur estimée.

Ici, lorsqu'elles sont testées une par une, les variables significatives sont `maxO3v` et `Ne9`. Cependant, la régression étant multiple et les variables explicatives non orthogonales, il est délicat d'utiliser ces tests. En effet, le test sur un coefficient

revient à tester la significativité d'une variable alors que les autres variables sont dans le modèle. Autrement dit, cela revient à tester que la variable n'apporte pas d'information supplémentaire sachant que toutes les autres variables sont dans le modèle. Il est donc important d'utiliser des procédures de choix de modèles que nous présentons ci-après.

Le résumé de l'étape d'estimation fait figurer l'estimation de l'écart-type résiduel σ, qui vaut ici 14.36, ainsi que le nombre de degrés de liberté associé $n - 11 = 101$.

La dernière ligne indique le résultat du test de comparaison entre le modèle utilisé et le modèle n'utilisant que la constante comme variable explicative. Il est évidemment significatif, car les variables explicatives apportent de l'information sur la variable à expliquer.

4. Choix de variables

Il est possible de faire un choix descendant (pas à pas) de variables à la main. On enlèverait la moins significative, soit `T9`, puis on recalculerait les estimations et ainsi de suite. Il existe dans R un package qui traite du choix de variables : le package leaps. La fonction **regsubsets** retourne, pour différents critères (bic, R^2 ajusté, Cp de Mallows, etc.), le meilleur modèle (si `nbest=1`) à 1 variable explicative, à 2 variables explicatives, ... à `nvmax` variables explicatives. La représentation graphique permet d'analyser ces résultats.

```
> library(leaps)
> choix <- regsubsets(maxO3~.,data=ozone.m,nbest=1,nvmax=11)
> plot(choix,scale="bic")
```

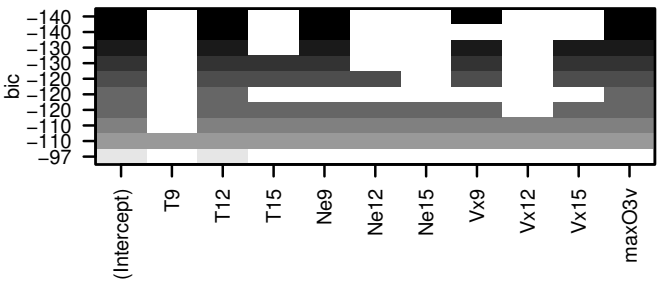


FIGURE 9.5 – Choix de variables avec BIC.

Le critère est optimum pour la ligne en haut du graphique. Nous conservons, pour le critère BIC, le modèle à 4 variables : `T12`, `Ne9`, `Vx9` et `maxO3v`. Nous ajustons ainsi le modèle avec les variables sélectionnées :

```
> reg.fin <- lm(maxO3~T12+Ne9+Vx9+maxO3v,data=ozone.m)
> summary(reg.fin)
Call:
lm(formula = maxO3 ~ T12 + Ne9 + Vx9 + maxO3v, data = ozone.m)

Residuals:
    Min      1Q  Median      3Q     Max
-52.396  -8.377  -1.086   7.951  40.933

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.63131   11.00088   1.148 0.253443
T12          2.76409    0.47450   5.825 6.07e-08 ***
Ne9         -2.51540    0.67585  -3.722 0.000317 ***
Vx9          1.29286    0.60218   2.147 0.034055 *
maxO3v       0.35483    0.05789   6.130 1.50e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14 on 107 degrees of freedom
Multiple R-Squared: 0.7622,     Adjusted R-squared: 0.7533
F-statistic: 85.75 on 4 and 107 DF,  p-value: < 2.2e-16
```

5. Analyser les résidus

Les résidus sont obtenus par la fonction **residuals**, cependant les résidus obtenus ne sont pas de même variance (hétéroscédastiques). Nous utilisons donc des résidus studentisés, qui eux sont de même variance.

```
> res.m <- rstudent(reg.fin)
> plot(res.m,pch=15,cex=.5,ylab="Résidus",main="",ylim=c(-3,3))
> abline(h=c(-2,0,2),lty=c(2,1,2))
```

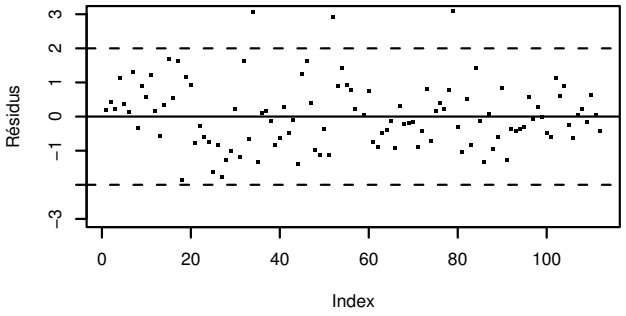


FIGURE 9.6 – Représentation des résidus.

En théorie 95 % des résidus studentisés se trouvent dans l'intervalle $[-2, 2]$. C'est le cas ici puisque trois résidus seulement se trouvent à l'extérieur de cet intervalle.

6. Prévoir une nouvelle valeur
Ayant une nouvelle observation, il suffit d'utiliser les estimateurs pour prévoir la valeur de Y correspondante. Cependant, la valeur prédite est de peu d'intérêt sans l'intervalle de confiance associé. Voyons cela sur un exemple. Nous obtenons pour le 1^{er} octobre 2001 les valeurs suivantes : `maxO3v=70`, `T12=19`, `Ne9=8` et `Vx9=2.05`. Il faut bien noter que l'argument `xnew` de la fonction **predict** doit être un data-frame avec les mêmes noms de variables explicatives.

```
> xnew <- matrix(c(19,8,2.05,70),nrow=1)
> colnames(xnew) <- c("T12","Ne9","Vx9","maxO3v")
> xnew <- as.data.frame(xnew)
> predict(reg.fin,xnew,interval="pred")
          fit      lwr      upr
[1,] 72.51437 43.80638 101.2224
```

La valeur prévue est 72.5 et l'intervalle de confiance à 95 % pour la prévision est $[43.8, 101.2]$.

Pour aller plus loin

Des rappels théoriques et des exercices sur la régression linéaire existent dans beaucoup d'ouvrages comme Cornillon et Matzner-Løber (2010), Husson et Pagès (2013b) ou Saporta (2011).

9.3 Analyse de la variance

Objet

L'analyse de la variance (ou ANOVA) est une méthode permettant de modéliser la relation entre une variable quantitative et une ou plusieurs variables qualitatives.

Quand il y a une seule variable explicative, on parle d'analyse de variance à 1 facteur. Cette méthode permet la comparaison de K moyennes et peut donc être vue comme une extension du test de comparaison de moyennes (vu à la fiche 6.3).

Nous allons nous intéresser ici au cas de deux variables explicatives, donc à l'analyse de variance à deux facteurs, mais la généralisation à un nombre quelconque de variables est immédiate. En fin de fiche, nous verrons également comment construire un modèle d'analyse de variance à 1 facteur.

Notons A et B deux variables explicatives qualitatives, et Y une variable à expliquer quantitative. Nous notons I le nombre de modalités de la variable A et J celui de la variable B.

En analyse de la variance, lorsque nous avons plusieurs mesures pour chaque croisement d'une modalité de A avec une modalité de B, il est possible et souvent intéressant de traiter le modèle complet avec interaction. Si l'interaction est significative, cela revient à dire que l'effet du facteur A dépend de la modalité du facteur B (et réciproquement).

Le modèle s'écrit classiquement :

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \varepsilon_{ijk}, \tag{9.3}$$

avec un effet moyen général μ, un effet de la modalité i du facteur A (α_i), un effet de la modalité j du facteur B (β_j), un terme d'interaction γ_{ij} et un résidu ε_{ijk} avec k l'indice de répétition pour le couple (i, j).

On teste la significativité des effets du facteur A, du facteur B et de l'interaction AB par des tests de Fisher qui comparent la variabilité expliquée par chaque facteur (ou par l'interaction) à la variabilité résiduelle. Nous préconisons de tester en premier la significativité de l'interaction. En effet, si l'interaction est significative, les deux facteurs sont influents via leur interaction, il n'est donc pas nécessaire de tester leur influence respective via l'effet principal. Les hypothèses du test de l'interaction sont :

$$(\mathrm{H}_0)_{AB} : \quad \forall (i,j) \quad \gamma_{ij} = 0 \qquad \text{contre} \qquad (\mathrm{H}_1)_{AB} : \exists (i,j) \quad \gamma_{ij} \neq 0.$$

Ces hypothèses reviennent à tester le sous-modèle (9.4) suivant contre le modèle complet (9.3) :

$$y_{ijk} \quad = \quad \mu + \alpha_i + \beta_j + \varepsilon_{ijk}, \quad \text{modèle sous } (\mathrm{H}_0)_{AB} \tag{9.4}$$

On procède ensuite de la même façon pour tester l'effet de chaque facteur. Par exemple, pour tester l'effet du facteur A, on teste le sous-modèle sans le facteur A

contre le modèle avec le facteur A mais sans l'interaction :

$$\begin{aligned} y_{ijk} &= \mu + \beta_j + \varepsilon_{ijk} \quad \text{modèle sous} \quad (\mathrm{H}_0)_A \\ y_{ijk} &= \mu + \alpha_i + \beta_j + \varepsilon_{ijk} \quad \text{modèle sous} \quad (\mathrm{H}_1)_A. \end{aligned}$$

Après avoir choisi le modèle final (dans lequel les facteurs ont tous des effets significatifs), on peut estimer les paramètres du modèle $(\mu, \alpha_i, \beta_j, \gamma_{ij})$. Cependant, le nombre de paramètres $1 + I + J + IJ$ à estimer est supérieur au nombre de paramètres qu'il est possible d'estimer à partir des données, on parle de paramètres indépendants, (IJ). Il faut donc imposer $1 + I + J$ contraintes afin de rendre le système inversible.

Les contraintes classiques sont :
- de type analyse par cellule : $\mu = 0,\ \forall i\ \alpha_i = 0,\ \forall j\ \beta_j = 0$;
- de type cellule de référence : $\alpha_1 = 0,\ \beta_1 = 0,\ \forall i \gamma_{i1} = 0,\ \forall j \gamma_{1j} = 0$;
- de type somme, qui impose que la somme des coefficients associés à chaque facteur soit égale à 0 : $\sum_i \alpha_i = 0,\ \sum_j \beta_j = 0,\ \forall i \sum_j \gamma_{ij} = 0,\ \forall j \sum_i \gamma_{ij} = 0$.

Exemple

Nous reprenons le jeu de données ozone présenté en détail dans la fiche 9.2. Nous souhaitons analyser ici la relation entre le maximum journalier de la concentration en ozone (en $\mu g/m^3$) avec la direction du `vent` classée en secteurs : `Nord`, `Sud`, `Est`, `Ouest` et la précipitation classée en deux modalités : `Sec` et `Pluie`. Nous disposons de 112 données relevées durant l'été 2001. La variable A admet $I = 4$ modalités et la variable B a $J = 2$ modalités.

Étapes

1. Importer les données
2. Représenter les données
3. Choisir le modèle
4. Estimer et interpréter les coefficients

Traitement de l'exemple

1. Importer les données

Importons le jeu de données et résumons les variables `maxO3`, `vent` et `pluie` :

```
> ozone <- read.table("ozone.txt",header=T)
> summary(ozone[,c("maxO3","vent","pluie")])
      maxO3           vent      pluie
 Min.   : 42.00   Est  :10   Pluie:43
 1st Qu.: 70.75   Nord :31   Sec  :69
```

```
Median : 81.50   Ouest:50
Mean   : 90.30   Sud  :21
3rd Qu.:106.00
Max.   :166.00
```

2. Représenter les données
Nous représentons une boîte à moustaches par cellule (croisement d'une modalité de `vent` avec une modalité de `pluie`), soit $IJ = 8$ boîtes à moustaches.

```
> boxplot(maxO3~vent*pluie, data = ozone,
    col=c(rep("Lightblue",4),rep("orange",4)))
```

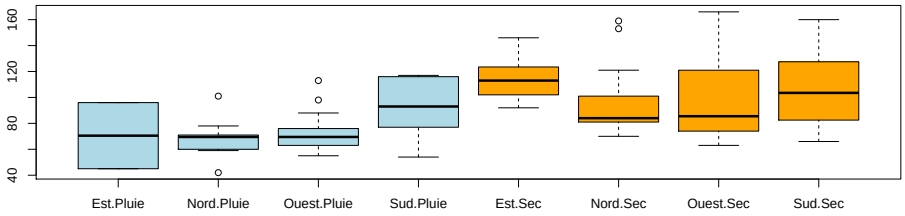


FIGURE 9.7 – Boîtes à moustaches du `maxO3` selon le croisement des modalités de la variable `vent` et de la variable `pluie`.

On peut remarquer (Fig. 9.7) que, généralement, le niveau d'ozone est supérieur par temps sec que par temps de pluie. Une autre façon d'analyser graphiquement l'interaction est la suivante :

```
> par(mfrow=c(1,2))
> with(ozone,interaction.plot(vent,pluie,maxO3,col=1:nlevels(pluie)))
> with(ozone,interaction.plot(pluie,vent,maxO3,col=1:nlevels(vent)))
```

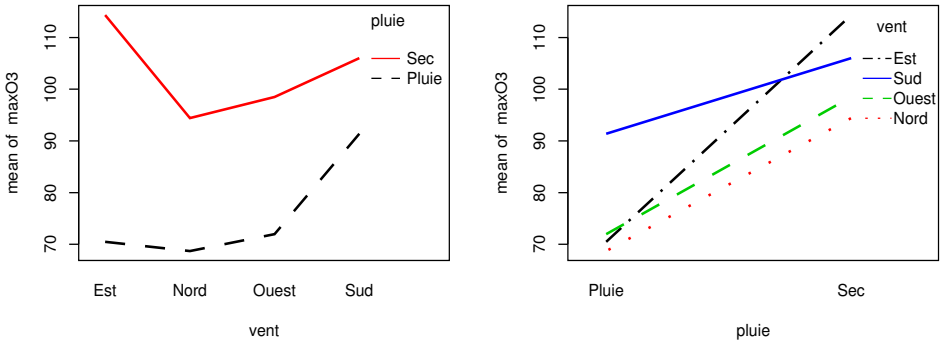


FIGURE 9.8 – Graphique de l'interaction avec en abscisse le vent (à gauche) et la pluie (à droite).

Les graphes (Fig. 9.8) permettent de visualiser l'interaction de deux façons différentes. Nous conseillons de construire les deux graphes et d'analyser le plus explicite. Les droites sont presque parallèles donc l'interaction n'est très marquée. Nous allons construire le test pour comparer les modèles (9.3) et (9.4).

3. Choisir le modèle

Dans notre exemple, le nombre d'observations diffère selon le croisement des modalités des facteurs `vent` et `pluie` :

```
> table(ozone$pluie, ozone$vent)
        Est Nord Ouest Sud
  Pluie   2   10    26   5
  Sec     8   21    24  16
```

On dit que les données sont « déséquilibrées ». Lors de la décomposition de la variabilité, ce déséquilibre a un impact sur le calcul des différentes sommes de carrés, et par suite sur les tests qui sont construits. Il est donc nécessaire de choisir quel type de sommes de carrés seront utilisés (type I, type II ou type III).

Pour construire le modèle, nous conseillons l'utilisation de la fonction `AovSum` de FactoMineR. Cette fonction utilise la contrainte que la somme des coefficients est nulle, et calcule des sommes de carrés de type III.

```
> library(FactoMineR)
> mod.interaction <- AovSum(maxO3~vent+pluie+vent:pluie,data=ozone)
> mod.interaction$Ftest
              SS  df      MS F value    Pr(>F)
vent        3227   3  1075.6  1.7633    0.1588
pluie      10996   1 10996.5 18.0271 4.749e-05 ***
vent:pluie  1006   3   335.5  0.5500    0.6493
Residuals  63440 104   610.0
Signif. codes:  0 "***" 0.001 "**" 0.01 "*" 0.05 "." 0.1 " " 1
```

La fonction retourne deux objets : un objet avec les résultats des tests de Fisher, i.e. la table d'analyse de variance, et un objet avec les résultats des tests de Student. Nous regardons dans un premier temps uniquement les résultats des tests Fisher (objet résultat `Ftest`), c'est-à-dire les tests globaux de chaque facteur ou de l'interaction.

La première colonne donne les sommes des carrés associées aux facteurs, la seconde les degrés de liberté et la troisième les carrés moyens (MS pour Mean Square). La quatrième colonne donne la valeur de la statistique de test et la cinquième colonne (`Pr(>F)`) contient la probabilité critique (ou « p-value ») qui est la probabilité, pour la statistique de test sous H_0, de dépasser la valeur estimée. La probabilité critique (0.65) est supérieure à 5 % donc on ne peut pas rejeter H_0 et on conclut à la non-significativité de l'interaction. Nous estimons donc le modèle sans interaction (9.4).

```
> modele.sans.int <- AovSum(maxO3~vent+pluie,data=ozone)
> modele.sans.int$Ftest
             SS  df      MS F value    Pr(>F)
vent       3791   3  1263.8  2.0982    0.1048
pluie     16159   1 16159.4 26.8295 1.052e-06 ***
Residuals 64446 107   602.3
```

La probabilité critique associée à l'effet `vent` (0.10) est supérieure à 5 % donc on conclut à la non-significativité de ce facteur : il n'y a pas d'effet de la direction du vent sur le maximum d'ozone journalier. Avant de conclure sur l'effet du facteur `pluie`, nous devons ajuster le modèle d'analyse de variance à un seul facteur :

```
> modele.anova.simple <- AovSum(maxO3~pluie,data=ozone)
> modele.anova.simple$Ftest
             SS  df      MS F value    Pr(>F)
pluie     19954   1 19954.2  32.166 1.157e-07 ***
Residuals 68238 110   620.3
```

La probabilité critique associée à l'effet `pluie` (1.157e-07) est très faible : on en conclut que le facteur `pluie` affecte le maximum d'ozone journalier. Pour interpréter cet effet, il faut estimer les coefficients.

4. Estimer et interpréter les coefficients

Les estimations des coefficients dépendent des contraintes utilisées. Ceci est toujours très important à avoir en tête car cela influe sur les résultats et les interprétations de ceux-ci. Tous les résultats sur les coefficients sont donnés dans l'objet `Ttest`.

```
> modele.anova.simple$Ttest
              Estimate Std. Error   t value     Pr(>|t|)
(Intercept)    87.11796   2.419555 36.005777 1.066479e-62
pluie - Pluie -13.72262   2.419555 -5.671545 1.156980e-07
pluie - Sec    13.72262   2.419555  5.671545 1.156980e-07
```

On obtient une matrice (très succincte dans cet exemple car le facteur significatif n'a que deux modalités, temps humide et temps sec) qui comporte, pour chaque coefficient (chaque ligne), 4 colonnes : son estimation (`Estimate`), son écart-type estimé (`Std. Error`), et enfin la valeur de la statistique de test (`t value`) et de la probabilité critique (`Pr(>|t|)`) associées au test de Student dont l'hypothèse nulle est $H_0 : \alpha_i = 0$.

L'estimation de μ (87.12), noté ici `Intercept`, correspond à l'effet moyen du maximum d'ozone journalier. L'effet du temps humide (α_1) est estimé à une baisse de -13.72 $\mu g/m^3$ du maximum d'ozone journalier. Et pour le temps sec, bien sûr 13.72 puisque la somme des coefficients est égale à 0 ! Cet effet s'explique par le fait que le rayonnement est un des catalyseurs de l'ozone.

Pour aller plus loin

La fonction **anova** de R utilise la contrainte que le premier coefficient est égal à 0. Avec cette contrainte, la constante s'interprète comme la valeur prise par la référence (dans le cas du dernier modèle à un facteur la référence serait la pluie). Et les coefficients s'interprètent comme l'écart à cette référence.

Les lignes de code suivantes montrent comment construire les tests de Fisher avec la fonction `anova` puis les tests de Student en utilisant la fonction **summary.lm** :

```
> mod1 <- lm(maxO3~pluie,data=ozone)
> anova(mod1)
Analysis of Variance Table

Response: maxO3
           Df Sum Sq Mean Sq F value    Pr(>F)
pluie       1  19954 19954.2  32.166 1.157e-07 ***
Residuals 110  68238   620.3
---
Signif. codes:  0 "***" 0.001 "**" 0.01 "*" 0.05 "." 0.1 " " 1
> summary(mod1)$coef
            Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 73.39535   3.798228 19.323575 3.757447e-37
pluieSec    27.44523   4.839110  5.671545 1.156980e-07
```

Les résultats sont ici parfaitement identiques aux précédents (le coefficient estimé pour la pluie 73.39 correspond bien à $87.11 - 13.72$ et pour le temps sec on a bien $87.11 + 13.72 = 73.39 + 27.44$).

Attention : lorsque les données sont déséquilibrées et qu'un modèle avec plusieurs facteurs et des interactions est construit, les tests de Fisher peuvent donner des résultats différents à cause des sommes de carrés utilisées, type III pour **AovSum** et type I pour **anova**. Aucun type de sommes de carrés n'est meilleur que l'autre, mais l'interprétation sera beaucoup plus facile si les résultats sont concordants et conduisent à une même interprétation quel que soit le type utilisé.

Lors de la construction du modèle, on suppose que les résidus suivent une loi normale de même variance. Il peut être intéressant de tester l'égalité de la variance des résidus pour les différentes combinaisons de l'interaction. Pour cela, on peut construire le test de Bartlett à l'aide de la fonction **bartlett.test**. Si on rejette l'égalité des variances, on peut construire le test de Friedman, qui est fondé sur les rangs, à l'aide de la fonction **friedman.test**.

9.4 Analyse de la covariance

Objet

L'analyse de la covariance est l'analyse d'une variable quantitative Y à expliquer par des variables quantitatives et qualitatives. Cette fiche traite le cas le plus simple : celui où il n'y a que deux variables explicatives, l'une quantitative et l'autre qualitative. Nous notons X la variable quantitative, Z la variable qualitative que l'on considère à I modalités. La relation entre la réponse Y et la variable quantitative X pouvant dépendre des modalités de la variable qualitative Z, la démarche naturelle consiste à effectuer I régressions, une pour chaque modalité i de Z. Cela donne en termes de modélisation :

$$y_{ij} = \alpha_i + \gamma_i x_{ij} + \varepsilon_{ij} \quad i = 1, \ldots, I, \quad j = 1, \ldots, n_i. \tag{9.5}$$

Cependant, on préfère écrire le modèle en faisant apparaître un effet moyen μ et une pente moyenne γ (comme en analyse de variance) :

$$y_{ij} = \mu + \alpha_i + (\gamma + \gamma_i) x_{ij} + \varepsilon_{ij} \quad i = 1, \ldots, I, \quad j = 1, \ldots, n_i. \tag{9.6}$$

avec n_i le nombre de répétitions pour le couple (i, j). Dans cette équation, γ est alors la pente moyenne des droites de régression quand γ_i est l'écart à cette pente moyenne pour la modalité i de la variable qualitative. De même, μ est l'ordonnée à l'origine moyenne et α_i est l'écart à cette ordonnée à l'origine pour la modalité i de la variable qualitative. Cependant, comme nous avons rajouté des paramètres par rapport à l'écriture (9.5), il est nécessaire, comme en analyse de variance, de poser des contraintes pour avoir un modèle identifiable. Classiquement, on pose $\sum_i \alpha_i = 0$ et $\sum_i \gamma_i = 0$, ou encore un des $\alpha_i = 0$ et un des $\gamma_i = 0$.
Selon les valeurs des coefficients μ_i, γ et γ_i, plusieurs situations sont envisageables (voir Fig. 9.9). Le modèle le plus général (9.6), appelé modèle complet, est illustré à gauche. Il correspond au cas où pentes et ordonnées à l'origine sont différentes dans chaque modalité de Z.

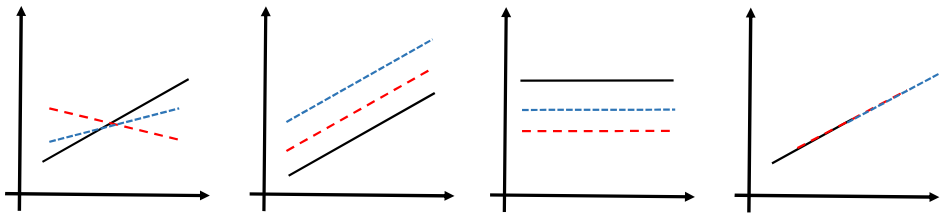


FIGURE 9.9 – Illustration des modèles d'analyse de covariance : représentation de Y en fonction de X, chaque modalité de Z est représentée par une couleur.

Une première simplification de ce modèle consiste à supposer que la variable X intervient de la même façon quelle que soit la modalité de la variable Z, cette moda-

lité fixant seulement le niveau. Autrement dit, on considère les pentes identiques mais pas les ordonnées à l'origine (cf. 2^{e} graphique Fig. 9.9). Ce modèle s'écrit :

$$y_{ij} = \mu + \alpha_i + \gamma x_{ij} + \varepsilon_{ij} \quad i = 1, \ldots, I, \quad j = 1, \ldots, n_i. \tag{9.7}$$

On dira qu'il n'y a pas d'interaction entre X et Z puisque les pentes sont identiques.

Une autre simplification du modèle complet (9.6) consiste à supposer qu'il n'y a pas d'effet de la variable quantitative (cf. 3^{e} graphique Fig. 9.9), on retrouve alors un modèle d'analyse de variance à 1 facteur. Le modèle s'écrit alors :

$$y_{ij} = \mu + \alpha_i + \varepsilon_{ij} \quad i = 1, \ldots, I, \quad j = 1, \ldots, n_i. \tag{9.8}$$

Enfin, une dernière simplification du modèle (9.6) consiste à supposer qu'il n'y a pas d'effet de la variable qualitative (cf. graphique de droite Fig. 9.9), on retrouve alors un modèle de régression simple. Le modèle s'écrit alors :

$$y_{ij} = \mu + \gamma x_{ij} + \varepsilon_{ij} \quad i = 1, \ldots, I, \quad j = 1, \ldots, n_i. \tag{9.9}$$

Pour choisir le modèle, nous préconisons de commencer par le modèle le plus général (9.6), puis, si les pentes sont les mêmes, on passe au modèle sans interaction (9.7). A partir de ce modèle sans interaction : si la pente est nulle on prend le modèle (9.8) ; si les ordonnées à l'origine sont identiques on choisit le modèle (9.9).

Exemple

Nous nous intéressons à l'équilibre des saveurs dans des cidres et plus précisément à la relation entre la saveur sucrée et la saveur amère en fonction du type de cidre (brut, demi-sec ou doux). Nous disposons de l'évaluation de la saveur sucrée et de la saveur amère fournie par un jury sensoriel (moyenne des notes, de 1 à 10, de 24 juges) pour chacun des 50 cidres bruts, 30 cidres demi-secs et 10 cidres doux.

Étapes

1. Importer les données
2. Représenter les données
3. Choix du modèle
4. Analyser les résidus

Traitement de l'exemple

1. Importer les données

Nous importons les données et résumons les trois variables d'intérêt :

```
> cidre <- read.table("cidre.csv",header=TRUE,sep=";")
> summary(cidre[,c(1,2,4)])
       Type       S.Sucree        S.Amere
 Brut     :50   Min.   :3.444   Min.   :2.143
 Demi-sec:30   1st Qu.:4.580   1st Qu.:3.286
 Doux     :10   Median :5.250   Median :3.964
               Mean   :5.169   Mean   :4.274
               3rd Qu.:5.670   3rd Qu.:5.268
               Max.   :7.036   Max.   :7.857
```

2. Représenter les données

Avant de réaliser l'analyse de la covariance, il est utile d'explorer les données en représentant, pour chaque modalité de la variable qualitative, le nuage de points croisant la variable à expliquer `S.Sucree` et la variable explicative quantitative `S.Amere`. Voici les lignes de code pour faire le graphique de la figure 9.10 avec la fonction **ggplot** :

```
> library(ggplot2)
> ggplot(cidre,aes(y=S.Sucree, x=S.Amere, col=Type)) +
    geom_point() + geom_smooth(method=lm, se=FALSE)
```

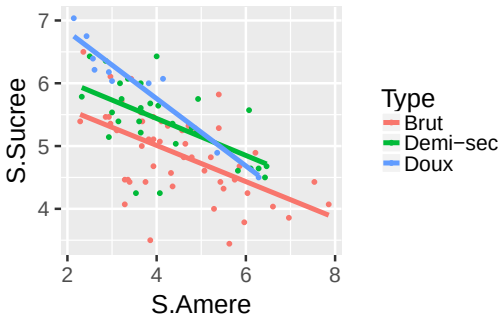


FIGURE 9.10 – Représentation du nuage de points et des droites de régression par type de cidre.

3. Choix du modèle

On ajuste le modèle complet (9.6) avec la fonction **AovSum** du package FactoMineR :

```
> complet <- AovSum(S.Sucree ~ Type + S.Amere + Type:S.Amere, data=cidre)
> complet$Ftest
                 SS df      MS F value    Pr(>F)
Type         2.7427  2  1.3714  4.9830  0.009015 **
```

```
S.Amere      13.2389  1 13.2389 48.1045 7.774e-10 ***
Type:S.Amere  0.9013  2  0.4506  1.6374  0.200634
Residuals    23.1178 84  0.2752
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La fonction **AovSum** retourne deux objets : un objet avec les résultats des tests de Fisher, i.e. la table d'analyse de variance, et un objet avec les résultats des tests de Student.

Nous regardons dans un premier temps uniquement les résultats des tests Fisher (objet `Ftest`), c'est-à-dire la table d'analyse de variance. Celle-ci décompose la variabilité totale de la saveur sucrée en plusieurs sources de variabilité : celle due au type de cidre (2.74), celle due à la saveur amère (13.24) et celle due à l'interaction type de cidre et saveur amère (0.90). La somme de ces trois variabilités est égale à la variabilité expliquée par le modèle. Il reste encore la variabilité résiduelle qui vaut 23.12. Les colonnes suivantes du tableau donnent les degrés de liberté associés à chaque source de variabilité (`df`), les carrés moyens (`MS` pour Mean Square), les statistiques des tests de Ficher et les probabilités critiques associées.

La probabilité critique (0.20) du test sur l'interaction étant supérieure à 5 %, nous concluons qu'il n'y a pas d'interaction et donc que les pentes peuvent être considérées égales. Cela signifie que quel que soit le type de cidre, l'effet de la perception de l'amertume sur la perception de la saveur sucrée reste le même.

Nous pouvons alors construire le modèle sans interaction (9.7) :

```
> mod.sans.int <- AovSum(S.Sucree ~ Type + S.Amere, data=cidre)
> mod.sans.int$Ftest
               SS df      MS F value    Pr(>F)
Type       6.9256  2  3.4628  12.399 1.857e-05 ***
S.Amere   14.5516  1 14.5516  52.102 1.960e-10 ***
Residuals 24.0191 86  0.2793
```

Les probabilités critiques associées au type de cidre et à la saveur sucrée sont inférieures à 5 %. On rejette donc l'hypothèse qu'il n'y a pas d'effet du type de cidre, et l'hypothèse qu'il n'y a pas d'effet de la saveur amère. Rejeter ces hypothèses revient donc à considérer que selon le type de cidre, la saveur sucrée n'est pas la même, et cela revient aussi à dire que la saveur sucrée évolue en fonction de la saveur amère.

Mais pour être plus précis, regardons les résultats sur les coefficients du modèle qui sont donnés dans l'objet `Ttest` :

```
> mod.sans.int$Ttest
                   Estimate Std. Error     t value     Pr(>|t|)
(Intercept)      6.718635980 0.19238203 34.92340710 1.415271e-52
Type - Brut     -0.416160940 0.08358068 -4.97915263 3.253757e-06
```

```
Type - Demi-sec  0.001066313 0.08877428  0.01201151 9.904443e-01
Type - Doux      0.415094627 0.12091691  3.43289157 9.205599e-04
S.Amere         -0.319275880 0.04423234 -7.21815542 1.959587e-10
```

L'interprétation des coefficients se fait comme en analyse de variance pour les variables qualitatives, et comme en régression pour variables quantitatives. Ici, la saveur amère à un effet négatif sur la saveur sucrée ($\hat{\gamma} = -0.319$). Pour les modalités, les coefficients s'interprètent en fonction de la contrainte utilisée, soit ici la somme des coefficients égale à 0 (c'est la contrainte utilisée par **AovSum**). On dira que les cidres bruts ont une saveur sucrée plus faible que la moyenne de tous les cidres (à une amertume donnée).

4. Analyser les résidus
L'analyse des résidus est similaire à celle effectuée en régression (fiche 9.1), mais il faut pour cela reconstruire le modèle avec la fonction **lm** (voir la section Pour aller plus loin).

Pour aller plus loin

Une autre façon de faire consiste à tester un sous-modèle contre un modèle. Par exemple, pour tester l'interaction, on compare le modèle complet (9.6) contre le modèle sans interaction (9.7). Pour ce faire, on construit chaque modèle et on compare les sommes de carré résiduelle par la fonction **anova** :

```
> complet <- lm(S.Sucree ~ Type + S.Amere + Type:S.Amere, data = cidre)
> ModSansInt <- lm(S.Sucree ~ Type + S.Amere, data = cidre)
> anova(complet,ModSansInt)
Analysis of Variance Table

Model 1: S.Sucree ~ Type + S.Amere
Model 2: S.Sucree ~ Type + S.Amere + Type:S.Amere
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     86 24.019
2     84 23.118  2   0.90126 1.6374 0.2006
```

On retrouve ici les sommes de carrés des résidus et les degrés de liberté des modèles précédemment construits. Et on retrouve une probabilité critique de 0.20 qui nous amène à accepter l'hypothèse que ces deux modèles sont équivalents. On choisira donc le modèle le plus parcimonieux, i.e. celui sans interaction. On peut alors continuer et construire un sous-modèle (un modèle d'analyse de variance à 1 facteur ou un modèle de régression simple) de ce modèle sans interaction.

La généralisation à un nombre quelconque de variables explicatives est immédiat d'un point de vue calculatoire. On peut même ajouter des interactions entre variables qualitatives. L'interprétation devient en revanche assez vite délicate en raison de la non-orthogonalité des variables explicatives.

9.5 Régression logistique

Objet

La régression logistique a pour objectif d'expliquer et de prédire les valeurs d'une variable qualitative Y, le plus souvent binaire, à partir de variables explicatives $X = (X_1, \ldots, X_p)$ qualitatives et quantitatives. Si on note 0 et 1 les modalités de Y, le modèle logistique s'écrit :

$$\log\left(\frac{p(x)}{1-p(x)}\right) \quad = \quad \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p,$$

où $p(x)$ désigne la probabilité $\mathbb{P}(Y = 1|X = x)$ et $x = (x_1, \ldots, x_p)$ est une réalisation de $X = (X_1, \ldots, X_p)$. La fonction $\log(u/(1+u))$ fait le lien entre la probabilité $p(x)$ et la combinaison linéaire des variables explicatives. Les coefficients $\beta_1, \ldots, \beta_p$ sont estimés par la méthode du maximum de vraisemblance à partir des observations. La figure 9.11 représente les valeurs de $p(x)$ en fonction de la combinaison linéaire $\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$, soit

$$p(x) = \frac{\exp(\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p)}{1 + \exp(\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p)}.$$

Cette probabilité est supérieure à 0.5 si et seulement si la combinaison linéaire est positive.

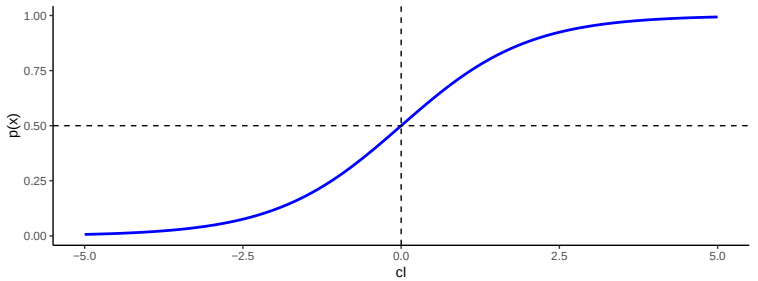


FIGURE 9.11 – Modèle logistique : probabilité $p(x)$ en fonction de la combinaison linéaire $\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$.

Exemple

Il s'agit d'expliquer et de prédire la réaction de clients suite à une campagne marketing effectuée par téléphone. On cherche plus précisément à détecter les clients intéressés par le produit proposé en fonction d'informations sur le client (âge, emploi, ...). Le but est alors d'expliquer la variable binaire `y` qui vaut `yes`

si le client a souscrit au produit, `no` sinon. Les données sont issues de l'`UCI Machine Learning Repository`, on trouvera un descriptif précis des variables à l'url `https://archive.ics.uci.edu/ml/datasets/Bank+Marketing`. On dispose de 4 119 individus (clients) sur lesquels on a observé la variable cible `y` ainsi que 20 autres variables potentiellement explicatives.

Étapes

1. Importer les données
2. Construire le modèle
3. Sélectionner un modèle
4. Faire de la prévision
5. Evaluer la performance du modèle

Traitement de l'exemple

1. Importer les données
Importons et résumons les données du fichier `bank-additional.csv` :

```
> bank <- read.csv("bank-additional.csv",sep=";")
> summary(bank)[,c(1:3,21)]
      age                  job           marital       y
 Min.   :18.00   admin.     :1012   divorced: 446   no :3668
 1st Qu.:32.00   blue-collar: 884   married :2509   yes: 451
 Median :38.00   technician : 691   single  :1153
 Mean   :40.11   services   : 393   unknown :  11
 3rd Qu.:47.00   management : 324
 Max.   :88.00   retired    : 166
                 (Other)    : 649
```

Par commodité, nous avons seulement affiché le résumé des 3 premières variables explicatives et de la variable à expliquer. On remarque que parmi les 4 119 clients, 451 ont souscrit au produit.

2. Construire le modèle
Nous séparons tout d'abord aléatoirement la base de données en :
– un échantillon d'apprentissage de taille 3000 qui sera utilisé pour estimer le (ou les) modèle(s) logistique(s) ;
– un échantillon test de taille 1119 qui sera utilisé pour mesurer la performance des modèles.

```
> set.seed(5678)
> perm <- sample(nrow(bank),3000)
```

```
> app <- bank[perm,]
> test <- bank[-perm,]
```

La régression logistique appartient à la famille des modèles linéaires généralisés. Un ajustement de ces modèles sur R est réalisé par la fonction **glm**. L'utilisation de cette fonction est similaire à celle de la fonction **lm**. Il faut écrire un modèle avec une `formule` du style (cf. annexe § A.1, p. 401) : `y~ X1+X2`. Il faut également spécifier une famille de lois de probabilité. Dans le cadre de la régression logistique, il s'agit de la famille binomiale. On estime le modèle logistique expliquant `y` par les autres variables du jeu de données `app` avec

```
> logit_complet <- glm(y~.,data=app,family=binomial)
```

La fonction **coef** permet d'obtenir les estimateurs des paramètres du modèle. Sur cet exemple, un paramètre n'a pas été estimé :

```
> coef(logit_complet)[is.na(coef(logit_complet))]
loanunknown
         NA
```

Il s'agit du coefficient associé à la modalité `unknown` de la variable `loan`. Il faut retourner dans les données pour expliquer ce problème. En croisant les variables explicatives entre elles, on s'aperçoit que la modalité `unknown` est prise par exactement les mêmes individus pour les variables `loan` et `housing` :

```
> table(app$loan,app$housing)
            no unknown  yes
  no      1169       0 1280
  unknown    0      82    0
  yes      188       0  281
```

L'information associée à ces deux modalités est redondante : on ne peut donc estimer qu'un paramètre pour ces deux modalités. Ce n'est pas un problème pour la qualité d'estimation mais ça en sera un pour l'interprétation des paramètres : il faudra en effet regarder le coefficient de la modalité `unknown` de la variable `housing` pour interpréter le paramètre associé à la modalité `unknown` de la variable `loan`.

La fonction **summary** permet de visualiser les tests de significativité des paramètres du modèle (on teste si chaque paramètre vaut 0) :

```
> summary(logit_complet)
```

Cependant, lorsque certaines variables explicatives sont qualitatives, il est souvent préférable de tester la significativité de chaque variable dans le modèle plutôt que de regarder coefficient par coefficient. On peut effectuer ces tests à l'aide de la fonction **Anova** du package `car`. On obtiendra pour chaque variable, les résultats du test du rapport de vraisemblance avec :

```
> library(car)
> Anova(logit_complet,type=3,test.statistic = "LR",singular.ok=TRUE)
Analysis of Deviance Table (Type III tests)

Response: y
               LR Chisq Df Pr(>Chisq)
age               0.65  1  0.4198705
job              11.47 11  0.4046378
marital           1.91  3  0.5918195
education         4.38  7  0.7357118
default           0.18  2  0.9158242
housing           0.59  1  0.4421206
loan              0.04  1  0.8465329
contact          10.95  1  0.0009373 ***
month            44.71  9  1.043e-06 ***
day_of_week       1.32  4  0.8587956
duration        486.70  1  < 2.2e-16 ***
campaign          2.47  1  0.1162119
pdays             0.17  1  0.6812723
previous          0.59  1  0.4411588
poutcome          7.64  2  0.0219673 *
emp.var.rate      1.77  1  0.1839836
cons.price.idx    1.04  1  0.3075302
cons.conf.idx     4.57  1  0.0324572 *
euribor3m         0.02  1  0.8951030
nr.employed       0.11  1  0.7393859
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

L'option `singular.ok=TRUE` est utilisée car un paramètre du modèle n'a pas pu être estimé. Cette option est inutile lorsque tous les paramètres ont bien été estimés. On pourra obtenir les tests de Wald avec l'option `test.statistic = "Wald"`.

3. Sélectionner un modèle

Dans notre exemple, plusieurs variables sont considérées non significatives. Il semble donc que l'on puisse retirer certaines variables explicatives du modèle complet. La fonction **step** permet de sélectionner un modèle à l'aide d'une procédure pas à pas basée sur la minimisation du critère `AIC` (Akaike Information Criterion) :

```
> logit_step <- step(logit_complet,direction="backward")
> Anova(logit_step,type=3,test.statistic = "LR")
Analysis of Deviance Table (Type III tests)

Response: y
               LR Chisq Df Pr(>Chisq)
contact          14.88  1  0.0001145 ***
month            51.09  9  6.713e-08 ***
```

```
duration          483.69  1  < 2.2e-16 ***
campaign            2.99  1  0.0839311 .
poutcome           37.24  2  8.186e-09 ***
emp.var.rate      109.95  1  < 2.2e-16 ***
cons.price.idx     47.58  1  5.273e-12 ***
cons.conf.idx      12.23  1  0.0004691 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La procédure de sélection pas à pas effectuée par la fonction **step** est ici une procédure descendante (`direction="backward"`) : à chaque étape, la variable dont le retrait du modèle conduit à la diminution la plus grande du critère `AIC` est enlevée. Le processus s'arrête lorsque toutes les variables sont retirées ou lorsque le retrait d'aucune variable ne permet de diminuer le critère. Pour utiliser cette méthode descendante, il faut vérifier que le modèle mis en argument de la fonction **step** contient suffisamment de variables. On aurait pu utiliser comme modèle initial de la procédure un modèle plus général comportant certaines interactions. La fonction **step** permet également d'effectuer des procédures pas à pas ascendante ou progressive (ascendante et descendante à la fois). Dans cet exemple, le modèle sélectionné ne comporte plus que 8 variables, 12 variables ont donc été supprimées. On remarque également que, au niveau $\alpha = 5$ %, les tests sont significatifs pour 7 des 8 variables restantes. On peut effectuer un test entre modèles emboîtés pour comparer le modèle sélectionné au modèle complet à l'aide de la fonction **anova** :

```
> anova(logit_step,logit_complet,test="LRT")
Analysis of Deviance Table

Model 1: y ~ contact + month + duration + campaign + poutcome
+ emp.var.rate + cons.price.idx + cons.conf.idx
Model 2: y ~ age + job + marital + education + default + housing +
  loan + contact + month + day_of_week + duration + campaign +
  pdays + previous + poutcome + emp.var.rate + cons.price.idx +
  cons.conf.idx + euribor3m + nr.employed
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1      2982     1192.2
2      2947     1171.9 35   20.213   0.9784
```

Le test accepte ici la nullité des paramètres du modèle `logit_complet` qui ne sont pas dans le modèle `logit_step`. On privilégiera donc le modèle `logit_step`.

4. Faire de la prévision

Les modèles logistiques construits précédemment peuvent être utilisés dans un contexte de prévision. On pourra utiliser la fonction **predict.glm** (appelée aussi par **predict**) pour obtenir les probabilités $p(x) = \mathbb{P}(Y = \texttt{yes}|X = x)$ prédites par le modèle `logit_step` des individus de l'échantillon test :

```
> prev_step <- predict(logit_step,newdata=test,type="response")
> prev_step[1:5]
          5          12          13          14          15
0.007852846 0.014818676 0.005213798 0.025447271 0.242988796
```

Si on décide de prédire `yes` pour les clients dont la probabilité de souscrire est supérieure à 0.5, alors ces cinq clients seront prédits `no`. Si on diminue le seuil, par exemple à 0.2, alors le cinquième client de la base test sera prédit `yes`.

5. Évaluer la performance du modèle

On souhaite comparer les performances des modèles `logit_complet` et `logit_step` en estimant les taux de mal classés et les courbes ROC obtenues sur l'échantillon test. On récupère d'abord les probabilités estimées d'être `yes` par les deux modèles pour les individus de l'échantillon test :

```
> prev_prob <- data.frame(complet=predict(logit_complet,newdata=test,
    type="response"),step=predict(logit_step,newdata=test,type="response"))
> head(round(prev_prob,3), n=3)
   complet  step
5    0.009 0.008
12   0.005 0.015
13   0.005 0.005
```

On peut ensuite en déduire une estimation de la classe en confrontant ces probabilités au seuil de 0.5 :

```
> prev_class <- apply(prev_prob>=0.5,2,factor,labels=c("no","yes"))
> head(prev_class, n=3)
   complet step
5  "no"    "no"
12 "no"    "no"
13 "no"    "no"
```

On obtient enfin les erreurs de classification estimées des 2 modèles en confrontant les valeurs prédites aux valeurs observées :

```
> library(tidyverse)
> prev_class <- data.frame(prev_class)
> prev_class %>% mutate(obs=test$y) %>%
   summarise_all(funs(err=mean(obs!=.))) %>% select(-obs_err) %>% round(3)
  complet_err step_err
1       0.074     0.08
```

Les taux d'erreur sont donc comparables. Ceci est confirmé par l'étude des courbes ROC, lesquelles s'obtiennent en `ggplot` à l'aide du package plotROC :

```
> library(plotROC)
> df_roc <- prev_prob %>% mutate(obs=test$y) %>%
      gather(key=methode,value=score,complet,step)
> ggplot(df_roc)+aes(d=obs,m=score,color=methode)+ geom_roc()+
      theme_classic()
```

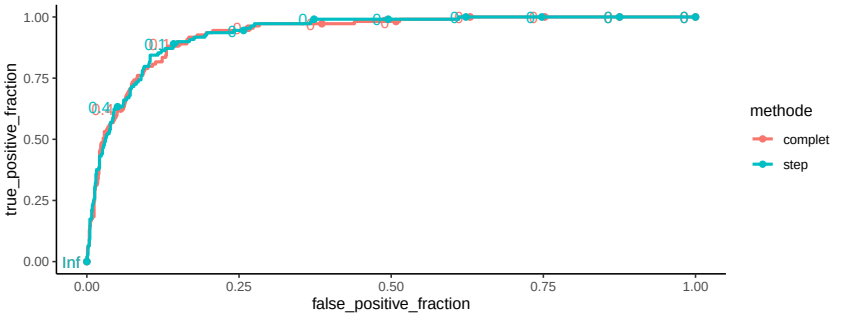


FIGURE 9.12 – Courbes ROC des modèles `logit_complet` et `logit_step`.

Les courbes ROC (Fig. 9.12) sont proches. Cela se confirme en calculant les AUC :

```
> df_roc %>% group_by(methode) %>% summarize(AUC=pROC::auc(obs,score))
# A tibble: 2 x 2
  methode   AUC
  <chr>   <dbl>
1 complet 0.933
2 step    0.935
```

Pour aller plus loin

La fonction **residuals** permet de calculer les différents types de résidus issus d'une régression logistique (résidus de Pearson ou de déviance par exemple). On peut également être amené à utiliser la fonction **logLik** pour obtenir la log-vraisemblance d'un modèle ou pour calculer la valeur de la statistique d'un test de rapport de vraisemblances. En présence d'une variable à expliquer ayant plus de deux modalités, il faut distinguer deux cas. Lorsque les modalités sont ordonnées, on construit, avec la fonction **polr** du package MASS, un modèle polytomique ordinal. Lorsqu'il n'y a pas de relation d'ordre entre les modalités de Y, on a recours à un modèle polytomique nominal aussi appelé modèle multinomial. Il faut alors utiliser la fonction **multinom** du package nnet. On peut aussi faire appel à la fonction **vglm** du package VGAM qui permet de construire à la fois des modèles polytomiques ordonnés et multinomiaux. Pour des éléments théoriques sur la régression logistique, on pourra se référer à Saporta (2011) et Collet (2003).

9.6 Analyse discriminante linéaire

Objet

Comme la régression logistique, l'analyse discriminante linéaire a pour principe d'expliquer et de prédire les valeurs d'une variable qualitative Y à partir de variables explicatives quantitatives et/ou qualitatives $X = (X_1, \ldots, X_p)$. Nous supposons que la variable Y est binaire, la généralisation à plus de deux modalités étant abordée en fin de fiche.

L'analyse discriminante linéaire peut être présentée selon deux aspects différents mais équivalents. Un premier point de vue consiste à modéliser la probabilité d'appartenance à un groupe à l'aide du théorème de Bayes. Pour simplifier, on désigne par 0 et 1 les modalités de la variable à expliquer Y. La règle bayésienne donne une estimation de la probabilité a posteriori d'affectation :

$$\mathbb{P}(Y = 1|X = x) = \frac{\pi_1 f_{X|Y=1}(x)}{\pi_0 f_{X|Y=0}(x) + \pi_1 f_{X|Y=1}(x)}, \tag{9.10}$$

où $\pi_0 = \mathbb{P}(Y = 0)$ et $\pi_1 = \mathbb{P}(Y = 1)$ désignent les probabilités a priori d'appartenance aux classes 0 et 1, et $f_{X|Y=j}$ la densité de la loi de X sachant $Y = j$. Les probabilités π_0 et π_1 doivent être fixées par l'utilisateur ou estimées à partir des données. Par ailleurs, l'analyse discriminante linéaire modélise les lois de X sachant $Y = j$ $(j = 0, 1)$ par des lois normales. Plus précisément, on fait l'hypothèse que :

$$X|Y = 0 \sim \mathcal{N}(\mu_0, \Sigma), \qquad X|Y = 1 \sim \mathcal{N}(\mu_1, \Sigma).$$

Les paramètres μ_0, μ_1 et Σ des lois normales sont estimés par la méthode du maximum de vraisemblance. Pour un nouvel individu, on déduit de (9.10) les estimations des probabilités a posteriori et on affecte ce nouvel individu au groupe pour lequel la probabilité a posteriori est la plus grande.

L'analyse discriminante linéaire peut également être envisagée comme une méthode de réduction de la dimension. Dans ce cas, le principe consiste, comme pour l'Analyse en Composantes Principales (voir fiche 7.1), à calculer une nouvelle variable, appelée variable canonique discriminante, $w'X = w_1X_1 + \ldots + w_pX_p$ comme combinaison linéaire des variables initiales. Elle est calculée de sorte que le rapport de la variance intergroupe à la variance intragroupe soit maximale (pour plus de détails voir Saporta (2011), chapitre 18). Pour un individu $x = (x_1, \ldots, x_p)$, la variable canonique définit une fonction de score $S(x) = w'x = w_1x_1 + \ldots + w_px_p$. L'affectation de l'individu x à un groupe s'effectue alors en comparant la valeur du score $S(x)$ à une valeur seuil s.

Remarque

Les deux manières de présenter l'analyse discriminante linéaire ne permettent pas de traiter le cas de variables explicatives qualitatives. Néanmoins, comme nous le verrons dans l'exemple, un codage disjonctif complet des variables qualitatives

permet de réaliser une analyse discriminante linéaire en présence de telles variables. Chaque modalité de la variable est alors traitée comme une variable quantitative prenant comme valeurs 0 ou 1. Afin d'éviter des problèmes de colinéarité entre les différentes modalités, la colonne associée à la première modalité du facteur n'est pas prise en compte dans le modèle. Lorsqu'une analyse discriminante est réalisée avec une ou plusieurs variables explicatives qualitatives, l'hypothèse de normalité effectuée n'est clairement pas vérifiée : il faut par conséquent être prudent dans l'interprétation des probabilités a posteriori.

Exemple

Dans le cadre d'une étude de la population angevine, le CHU d'Angers s'est intéressé à la propension à ronfler d'hommes et de femmes. Le fichier ronfle.txt contient un échantillon de 100 patients, les variables considérées sont :

- `age` : en années ;
- `poids` : en kg ;
- `taille` : en cm ;
- `alcool` : nombre de verres bus par jour (en équivalent verre de vin rouge) ;
- `sexe` : sexe de la personne (`F` = femme, `H` = homme) ;
- `ronfle` : diagnostic de ronflement (`O` = ronfle, `N` = ne ronfle pas) ;
- `taba` : comportement au niveau du tabac (`O` = fumeur, `N` = non fumeur).

Le but de cette étude est d'expliquer le ronflement (variable ronfle) par les six autres variables présentées ci-dessus. Le tableau suivant est un extrait du jeu de données :

	age	poids	taille	alcool	sexe	ronfle	taba
1	47	71	158	0	H	N	O
2	56	58	164	7	H	O	N
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
99	68	108	194	0	F	O	N
100	50	109	195	8	H	O	O

Étapes

1. Importer les données
2. Construire le modèle
3. Sélectionner des variables
4. Estimer le taux de mauvais classement
5. Faire de la prévision

Traitement de l'exemple

1. Importer les données

```
> donnees <- read.table("ronfle.txt",header=TRUE)
> summary(donnees)
      age            poids            taille          alcool
 Min.   :23.00   Min.   : 42.00   Min.   :158.0   Min.   : 0.00
 1st Qu.:43.00   1st Qu.: 77.00   1st Qu.:166.0   1st Qu.: 0.00
 Median :52.00   Median : 95.00   Median :186.0   Median : 2.00
 Mean   :52.27   Mean   : 90.41   Mean   :181.1   Mean   : 2.95
 3rd Qu.:62.25   3rd Qu.:107.00   3rd Qu.:194.0   3rd Qu.: 4.25
 Max.   :74.00   Max.   :120.00   Max.   :208.0   Max.   :15.00

sexe   ronfle    taba
F:25   N:65      N:36
H:75   O:35      O:64
```

2. Construire le modèle

La fonction **lda** de la librairie MASS permet de réaliser l'analyse discriminante linéaire. La librairie MASS est installée par défaut dans R, il suffit donc de la charger. La fonction **lda** utilise, comme la fonction **lm** (cf. annexe A.1, p. 401), une formule du type `Y ~ X1+X2`. Les probabilités a priori choisies pour l'analyse doivent également être spécifiées. Si on ne dispose d'aucun a priori sur ces deux quantités, deux stratégies peuvent être envisagées :

- les probabilités sont choisies égales, c'est-à-dire $\pi_0 = \pi_1 = 0.5$;
- les probabilités sont estimées par la proportion d'observations dans chaque groupe.

Le choix des probabilités a priori est spécifié dans l'argument `prior` de la fonction **lda**. Si rien n'est précisé, c'est la deuxième stratégie qui est utilisée par défaut. On écrit d'abord le modèle prenant en compte toutes les variables explicatives :

```
> library(MASS)
> mod.complet <- lda(ronfle~.,data=donnees)
> mod.complet
Call:
lda(ronfle ~ ., data = donnees)
Prior probabilities of groups:
   N    O
0.65 0.35

Group means:
       age    poids   taille   alcool     sexeH     tabaO
N 50.26154 90.47692 180.9538 2.369231 0.6923077 0.6769231
O 56.00000 90.28571 181.3714 4.028571 0.8571429 0.5714286

Coefficients of linear discriminants:
```

```
              LD1
age     0.05973655
poids  -0.01620579
taille  0.01590170
alcool  0.24058822
sexeH   0.55413371
tabaO  -1.14621434
```

On retrouve dans les sorties les probabilités a priori du modèle, le centre de gravité de chacun des deux groupes et les coefficients de la variable canonique. Le logiciel renvoie par convention les coefficients de la variable canonique de sorte que sa variance intraclasse soit égale à 1. Il est difficile de trouver les variables importantes pour la discrimination en comparant les centres de gravité ; il est souvent plus pertinent d'étudier l'influence des coefficients sur le score. La fonction **plot** permet de représenter la distribution des coordonnées de chaque individu sur la variable canonique pour chaque groupe :

```
> plot(mod.complet)
```

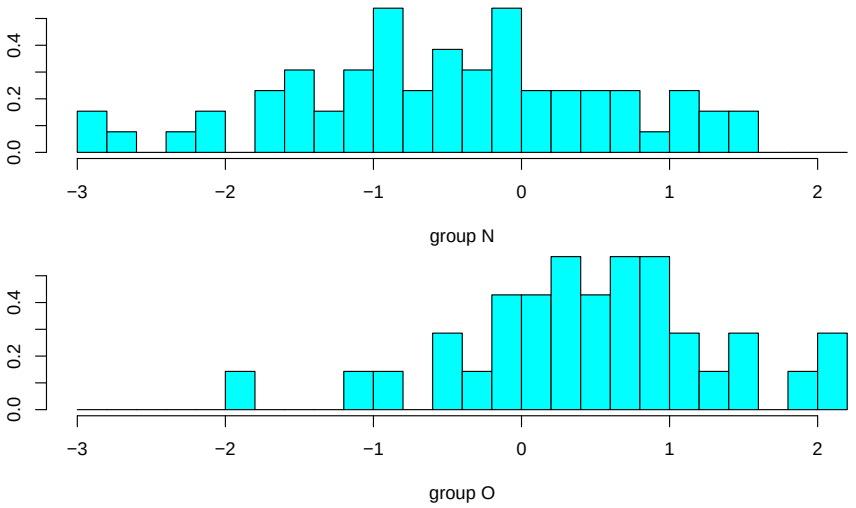


FIGURE 9.13 – Distribution des coordonnées des individus sur la variable discriminante.

On pourra remarquer que les ronfleurs ont tendance à posséder une coordonnée élevée sur la variable discriminante.

3. Sélectionner des variables

Tout comme pour les modèles linéaire et logistique, il est possible de faire de la sélection pour l'analyse discriminante linéaire. La fonction **stepclass** du package

klaR permet d'utiliser des procédures pas à pas pour choisir un modèle. À chaque étape, la procédure ajoute ou supprime la variable qui optimise un critère de performance estimé par validation croisée. Le critère utilisé par défaut est le taux de bon classement. Le processus s'arrête lorsqu'aucun ajout ou retrait de variable ne permet d'améliorer significativement le critère. Le niveau minimal d'amélioration pour ajouter ou supprimer une variable est défini par le paramètre `improvement`, ce paramètre étant par défaut fixé à 0.05.

La fonction **stepclass** exige que les variables qualitatives représentées par des facteurs soient codées en 0/1. On peut effectuer ce codage à l'aide de la fonction **model.matrix** :

```
> mat.X <- model.matrix(ronfle~.,data=donnees)[,-1]
> head(mat.X)
  age poids taille alcool sexeH tabaO
1  47    71    158      0     1     1
2  56    58    164      7     1     0
3  46   116    208      3     1     1
4  70    96    186      3     1     1
5  51    91    195      2     1     1
6  46    98    188      0     0     0
```

La première colonne, représentant la variable constante, est supprimée. On peut maintenant lancer la procédure de sélection :

```
> ronfle <- donnees$ronfle
> library(klaR)
> set.seed(1234)
> sel <- stepclass(mat.X,ronfle,method="lda",direction="both",maxvar=6)
> sel$model
  nr   name
1  1    age
2  2  poids
3  4 alcool
```

L'argument `direction` indique le type de procédure pas à pas utilisé : ascendante (`forward`), descendante (`backward`) ou les deux à la fois (`both`). L'argument `maxvar` renseigne sur le nombre maximal de variables souhaitées dans le modèle, ici on s'autorise un maximum de 6 variables (le modèle complet). La procédure a sélectionné 3 variables : `age`, `poids` et `alcool`. On obtient le modèle sélectionné comme suit :

```
> mod.sel <- lda(sel$formula,data=donnees)
> mod.sel
Call:
lda(sel$formula, data = donnees)
```

```
Prior probabilities of groups:
   N    O
0.65 0.35

Group means:
       age    poids   alcool
N 50.26154 90.47692 2.369231
O 56.00000 90.28571 4.028571

Coefficients of linear discriminants:
                LD1
age     0.072815712
poids  -0.004607728
alcool  0.245088681
```

4. Estimer le taux de mauvais classement

La fonction **lda** permet d'estimer le taux de mauvais classement par validation croisée. Il suffit pour cela d'ajouter l'argument `CV=TRUE` lors de l'appel à la fonction. On estime dans cette partie les taux de mauvais classement du modèle complet `mod.complet` et du modèle sélectionné dans la partie précédente `mod.sel`. On obtient les labels prédits par validation croisée de ces deux modèles avec :

```
> prev.complet <- lda(ronfle~.,data=donnees,CV=TRUE)$class
> prev.sel <- lda(sel$formula,data=donnees,CV=TRUE)$class
```

On peut ainsi confronter les prévisions aux valeurs observées :

```
> table(prev.complet,donnees$ronfle)

prev.complet  N  O
           N 53 22
           O 12 13
> table(prev.sel,donnees$ronfle)

prev.sel  N  O
       N 55 23
       O 10 12
```

et en déduire les taux de mauvais classement

```
> mean(prev.complet!=donnees$ronfle)
[1] 0.34
> mean(prev.sel!=donnees$ronfle)
[1] 0.33
```

Le modèle avec 3 variables a un taux d'erreur légèrement plus faible que le modèle complet.

5. Faire de la prévision
Les modèles construits précédemment peuvent être utilisés dans un contexte de prévision. Le tableau suivant contient les valeurs des six variables explicatives mesurées sur quatre nouveaux patients :

age	poids	taille	alcool	sexe	taba
42	55	169	0	F	N
58	94	185	4	H	O
35	70	180	6	H	O
67	63	166	3	F	N

Pour prédire le label de ces quatre nouveaux individus, on récolte d'abord les nouvelles données dans un data-frame qui possède la même structure que le tableau de données initial (notamment les mêmes noms de variables). On construit donc une matrice pour les variables quantitatives et une matrice pour les variables qualitatives avant de les regrouper dans un même data-frame :

```
> n_don1<-matrix(c(42,55,169,0,58,94,185,4,35,70,180,6,67,63,166,3),
    ncol = 4, byrow = TRUE)
> n_don2<-matrix(c("F","N","H","O","H","O","F","N"),
    ncol = 2, byrow = TRUE)
> n_donnees<-cbind.data.frame(n_don1,n_don2)
> names(n_donnees) <- names(donnees)[-6]
```

La fonction **predict.lda**, que l'on peut appeler plus simplement par le raccourci **predict**, permet d'affecter un groupe à chacun de ces quatre individus :

```
> predict(mod.sel,newdata=n_donnees)
$class
[1] N N N O
Levels: N O

$posterior
          N         O
1 0.8582230 0.1417770
2 0.5444034 0.4555966
3 0.7435646 0.2564354
4 0.4308341 0.5691659

$x
         LD1
1 -1.3076693
2  0.6580354
3 -0.4159631
4  1.2111277
```

La fonction **predict** renvoie une `liste` de longueur 3 :
- le premier élément (`class`) contient les groupes prédits ;
- le deuxième élément (`posterior`) est une matrice à 2 colonnes donnant les probabilités a posteriori d'appartenance aux groupes `N` et `O` pour chaque individu ;
- le troisième élément (`x`) désigne la valeur du score de chaque individu, il contient les coordonnées des 4 individus sur l'axe discriminant.

Le modèle attribue le groupe `N` aux trois premiers individus et le groupe `O` au quatrième. La règle d'attribution d'un groupe à une observation `x` s'effectue en comparant les probabilités a posteriori ou, de manière équivalente, en comparant le score à une valeur seuil. Modifier les probabilités a priori ne change pas la variable canonique ; seul le seuil à partir duquel on change l'affectation du groupe varie. Le logiciel ne renvoyant pas cette valeur seuil, on affecte en comparant les probabilités a posteriori de chaque groupe.

Pour aller plus loin

Pour réaliser une analyse discriminante linéaire, on suppose que les matrices de variance-covariance des variables explicatives sont les mêmes pour chaque groupe. L'analyse discriminante quadratique s'affranchit de cette hypothèse. Les probabilités a posteriori sont calculées en supposant que :

$$X|Y=0 \sim \mathcal{N}(\mu_0, \Sigma_0), \qquad X|Y=1 \sim \mathcal{N}(\mu_1, \Sigma_1).$$

La fonction **qda**, disponible dans la librairie MASS, permet alors de réaliser une analyse discriminante quadratique. Elle s'utilise de la même façon que **lda**.

Dans cette partie, nous nous sommes restreints à un problème de discrimination pour une variable à expliquer binaire. L'analyse discriminante linéaire se généralise aisément au cas où la variable à expliquer possède $J > 2$ modalités. Les probabilités a posteriori sont toujours calculées à l'aide du Théorème de Bayes en faisant pour chaque groupe l'hypothèse de normalité $X|Y = j \sim \mathcal{N}(\mu_j, \Sigma)$ pour $j = 1, \ldots, J$ (ou $X|Y = j \sim \mathcal{N}(\mu_j, \Sigma_j)$ pour l'analyse discriminante quadratique). La mise en œuvre sur R ne pose aucune difficulté et s'effectue comme précédemment : les modèles sont écrits de la même façon et on affecte l'individu au groupe pour lequel la probabilité a posteriori est la plus élevée.

Concernant la seconde manière d'envisager l'analyse discriminante linéaire (réduction de la dimension), on ne cherchera plus une variable canonique discriminante mais $(J-1)$ variables.

Des méthodes de régularisation ont été proposées pour répondre aux problèmes de la grande dimension ou de corrélations élevées entre les variables explicatives. On pourra mettre en œuvre ces techniques sur R en utilisant la fonction **rda** du package klaR ou encore la fonction **hdda** du package HDclassif .

9.7 Arbres

Objet

Les arbres sont des outils d'exploration des données et d'aide à la décision qui permettent d'expliquer et de prédire une variable quantitative (arbre de régression) ou qualitative (arbre de classification) à partir de variables explicatives quantitatives et/ou qualitatives. Il existe plusieurs façons de construire des arbres. Dans cette fiche, nous nous intéressons à l'algorithme CART (Breiman *et al.*, 1984) qui est la méthode de référence. Cet algorithme permet d'obtenir des classes d'individus construites à partir des variables explicatives de manière à ce que les individus d'une même classe soient le plus homogène possible du point de vue de la variable d'intérêt. Les principaux avantages de cette méthode sont sa simplicité et la facilité d'interprétation des résultats grâce à la représentation sous forme d'arbre. Cet arbre est composé de « nœuds » et de « feuilles » et sa lecture est très intuitive. En effet, il aide à la compréhension du phénomène par un étiquetage des nœuds en produisant des règles de décision. Deux packages, rpart et tree, proposent la construction d'arbres basés sur CART. Nous nous contentons ici d'illustrer sur un exemple l'utilisation du package rpart, intégré par défaut dans R.

Exemple

Nous reprenons l'exemple de marketing bancaire présenté dans la fiche sur la régression logistique (fiche 9.5). L'objectif reste le même : détecter les clients intéressés par le produit proposé à partir d'informations sur ceux-ci (âge, emploi, situation matrimoniale, ...). La variable à expliquer est qualitative : nous construisons donc un arbre de classification.

Étapes

1. Importer les données
2. Construire et analyser l'arbre de classification
3. Choisir la taille de l'arbre
4. Prédiction pour de pour de nouveaux individus
5. Estimer la performance de l'arbre
6. Interpréter l'arbre

Traitement de l'exemple

1. Importer les données

Importons et résumons les données contenues dans le fichier `bank-additional.csv` :

```
> bank <- read.csv("bank-additional.csv",sep=";")
> summary(bank)[,c(1:3,21)]
      age                  job           marital       y
 Min.   :18.00   admin.     :1012   divorced: 446   no :3668
 1st Qu.:32.00   blue-collar: 884   married :2509   yes: 451
 Median :38.00   technician : 691   single  :1153
 Mean   :40.11   services   : 393   unknown :  11
 3rd Qu.:47.00   management : 324
 Max.   :88.00   retired    : 166
                 (Other)    : 649
```

Pour ne pas surcharger la lecture, nous affichons uniquement le résumé des 3 premières variables explicatives et de la variable à expliquer. On remarque que parmi les 4119 clients, 451 ont souscrit au produit.

2. Construire et analyser l'arbre de classification

Nous séparons tout d'abord la base de données en :

- un échantillon d'apprentissage de taille 3000 qui sera utilisé pour construire le (ou les) arbre(s) ;
- un échantillon test de taille 1119 qui sera utilisé pour mesurer la performance des arbres.

```
> set.seed(5678)
> perm <- sample(nrow(bank),3000)
> app <- bank[perm,]
> test <- bank[-perm,]
```

Nous appliquons la fonction **rpart** du package rpart qui, à l'instar de la fonction **lm**, utilise une formule (cf. § A.1, p. 401) :

```
> library(rpart)
> arbre1 <- rpart(y~., data=app, cp=0.02)
> print(arbre1)
n= 3000

node), split, n, loss, yval, (yprob)
      * denotes terminal node

  1) root 3000 342 no (0.88600000 0.11400000)
    2) nr.employed>=5087.65 2634 177 no (0.93280182 0.06719818)
      4) duration< 627.5 2428  70 no (0.97116969 0.02883031) *
      5) duration>=627.5 206  99 yes (0.48058252 0.51941748)
       10) duration< 836.5 118  47 no (0.60169492 0.39830508) *
       11) duration>=836.5 88  28 yes (0.31818182 0.68181818) *
    3) nr.employed< 5087.65 366 165 no (0.54918033 0.45081967)
                    ⋮
```

Le paramètre cp permet de contrôler la profondeur de l'arbre : plus ce paramètre est petit, plus l'arbre est profond. Nous aborderons le problème du choix de ce paramètre dans la partie 3. À chaque nœud de l'arbre correspond un groupe d'individus, pour lequel R donne :

- le numéro du nœud : node, par exemple 4)
- la règle de coupure : split, par exemple duration<627.5
- le nombre d'individus dans le groupe : n, par exemple 2428
- le nombre d'individus mal classés : loss, par exemple 70
- la valeur prédite : yval, par exemple no
- la probabilité d'appartenance à chaque classe : yprob, par exemple (0.97116969 0.0288303)

Ainsi, dans le cas du premier nœud (la « racine » de l'arbre, notée « root »), nous constatons qu'il y a 3000 observations (la totalité des individus de l'échantillon d'apprentissage), et que 88.6 % des clients n'ont pas souscrit au produit (Y=no). Ce premier nœud est segmenté à l'aide de la variable nr.employed. Les individus sont répartis en deux sous-groupes : le premier (nœud 2, nr.employed>=5087.65) comporte 2634 individus, 93.3 % d'entre eux correspondant à Y=no et 6.7 % à Y=yes. Le second (nœud 3, voir la dernière ligne du listing présenté, nr.employed< 5087.65) comporte les 366 individus restants (55 % Y=no, 45 % Y=yes). Ces 2 nœuds sont ensuite eux-mêmes segmentés et ainsi de suite jusqu'à ce qu'un critère d'arrêt soit satisfait (nœud pur, nœud à effectif faible, etc.).

La sortie **print**(arbre1) n'est pas facile à analyser. Il est plus simple de visualiser l'arbre pour comprendre sa construction. La fonction **rpart.plot** du package rpart.plot permet d'obtenir la représentation graphique de cette segmentation (Fig. 9.14).

```
> library(rpart.plot)
> rpart.plot(arbre1,main="Représentation de l'arbre")
```

Ce graphique résume l'information contenue dans **print**(arbre1). Lorsque l'algorithme ne segmente plus de nœud, on aboutit à une feuille. Ainsi dans notre exemple nous obtenons 9 feuilles.

Il reste bien entendu à préciser la classe affectée à chaque feuille. Pour une feuille pure, la réponse est évidente : on affecte la classe correspondant à l'unique label observé dans la feuille. Si ce n'est pas le cas, une règle simple consiste à décider d'affecter la classe la plus représentée dans la feuille. Ainsi par exemple, dans la première feuille à gauche de l'arbre de la figure 9.14 nous avons 80 % des clients de l'échantillon d'apprentissage et, parmi ces 81 %, nous avons seulement 3 % de yes. Nous avons donc la règle d'affectation suivante : « si nr.employed >= 5088 et duration < 628 alors y=no ».

L'arbre peut également être analysé à l'aide de la fonction **summary**. L'intérêt de ce résumé réside surtout dans le listing indiquant les variables concurrentes à celles choisies. Ainsi, pour chaque nœud, le résumé retourne via Primary splits

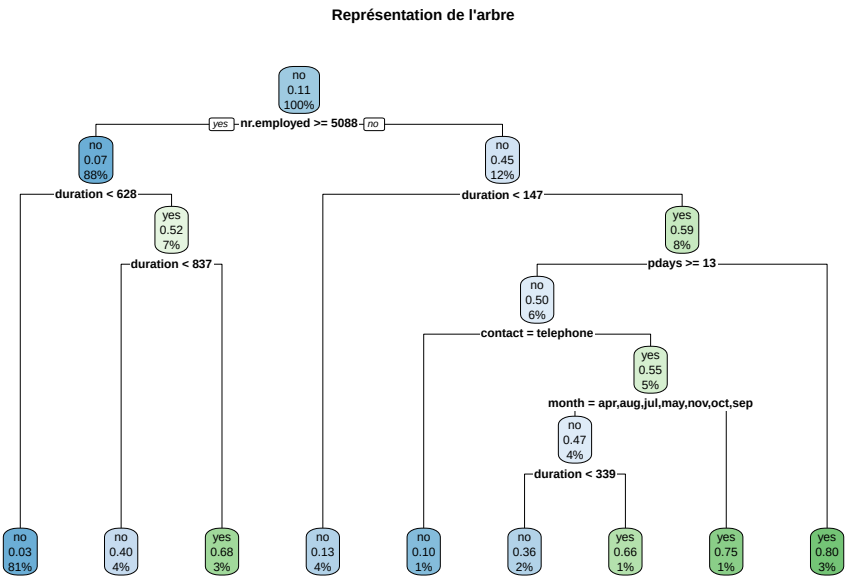


FIGURE 9.14 – Arbre de classification : prédire la variable `y`.

les variables qui auraient pu être choisies (en deuxième, troisième, quatrième choix, selon la valeur de l'argument `maxcompete`).

Les arbres permettent aussi de traiter les valeurs manquantes. Pour ce faire, l'arbre propose pour chaque variable sélectionnée (à chaque nœud), une variable suppléante (appelée `surrogate`) qui classe les individus quasiment de la même façon. L'argument `maxsurrogate` permet de préciser le nombre de variables suppléantes. On ne donne ici qu'un extrait du résumé :

```
> arbre2 <- rpart(y~., data=app, maxcompete=2, maxsurrogate=1)
> summary(arbre2)

Node number 1: 3000 observations,    complexity param=0.06871345
  predicted class=no   expected loss=0.114  P(node) =1
    class counts:  2658   342
   probabilities: 0.886 0.114
  left son=2 (2634 obs) right son=3 (366 obs)
  Primary splits:
      nr.employed < 5087.65 to the right, improve=94.58265, (0 missing)
      duration    < 606.5   to the left,  improve=89.55275, (0 missing)
      euribor3m   < 1.047   to the right, improve=79.61173, (0 missing)
  Surrogate splits:
      euribor3m < 1.1715  to the right, agree=0.984, adj=0.866, (0 split)
```

Au niveau du premier nœud, les deux variables compétitrices sont `duration` et `euribor3m` et la variable suppléante est `euribor3m`. Si on souhaite classer un nouvel individu pour lequel on n'a pas observé la variable `nr.employed`, on pourra ainsi utiliser la variable suppléante `euribor3m` pour découper le premier nœud.

3. Choisir la taille de l'arbre

L'objectif étant de produire des groupes homogènes, il paraîtrait naturel de choisir l'arbre ayant le maximum de feuilles pures. Cependant un arbre très profond (avec beaucoup de feuilles) risque d'avoir une mauvaise capacité de généralisation à de nouveaux individus : on parle de surapprentissage. Il est donc crucial de choisir la taille de l'arbre de manière convenable. L'approche classique consiste à construire un arbre maximal qui sera ensuite élagué. La fonction **rpart** permet de construire une suite d'arbres emboîtés qui optimise un critère prenant en compte à la fois l'ajustement et la complexité de l'arbre. La fonction **printcp** permet d'obtenir des informations sur les arbres de la suite :

```
> printcp(arbre1)
Classification tree:
rpart(formula = y ~ ., data = app, cp = 0.02)

Variables actually used in tree construction:
[1] contact     duration    month       nr.employed pdays

Root node error: 342/3000 = 0.114

n= 3000

        CP nsplit rel error  xerror     xstd
1 0.068713      0   1.00000 1.00000 0.050898
2 0.046784      2   0.86257 0.98538 0.050572
3 0.023392      4   0.76901 0.84795 0.047326
4 0.020000      8   0.66374 0.80117 0.046137
```

On a ici une suite de 4 arbres emboîtés. La colonne `CP` renvoie un paramètre associé à la complexité de l'arbre : plus ce paramètre est petit, plus l'arbre est profond. La colonne `nsplit` donne le nombre de coupures de l'arbre, `rel.error` contient l'erreur de classification calculée sur les données d'apprentissage tandis que la colonne `xerror` renvoie l'erreur de classification estimée par validation croisée sur 10 blocs. Ces erreurs sont normalisées par rapport à l'erreur de l'arbre racine qui vaut ici 0.114. Enfin, la dernière colonne donne un estimateur de l'écart-type de l'erreur estimée dans la colonne `xerror`.

La sélection de l'arbre final s'effectue généralement en choisissant l'arbre dont l'erreur de classification est minimale, on choisit donc l'arbre qui minimise la colonne `xerror`. Sur cet exemple le minimum est atteint pour l'arbre de la ligne 4. Néanmoins, l'erreur `xerror` ne cesse de décroître lorsque la profondeur augmente. Il est donc possible qu'un arbre plus profond puisse posséder une erreur plus petite et

donc être plus performant. Pour cet exemple, nous recommandons donc de considérer un arbre maximal plus profond. On peut le faire en diminuant les valeurs des paramètre `cp` et `minsplit` de la fonction **rpart** :

```
> set.seed(1234)
> arbre3 <- rpart(y~., data=app, cp=0.000001, minsplit=5)
> printcp(arbre3)
Classification tree:
rpart(formula = y ~ ., data = app, cp = 1e-06, minsplit = 5)

Variables actually used in tree construction:
 [1] age  campaign cons.price.idx contact day_of_week duration
          education euribor3m housing job
[11] loan           marital        month          nr.employed    pdays

Root node error: 342/3000 = 0.114

n= 3000
           CP nsplit rel error  xerror     xstd
1  0.06871345      0   1.00000 1.00000 0.050898
2  0.04678363      2   0.86257 0.96199 0.050044
3  0.02339181      4   0.76901 0.85673 0.047544
4  0.01169591      8   0.66374 0.83626 0.047033
5  0.01023392     16   0.56140 0.83626 0.047033
6  0.00877193     18   0.54094 0.80702 0.046288
7  0.00584795     19   0.53216 0.79825 0.046061
8  0.00487329     30   0.46784 0.80117 0.046137
9  0.00438596     33   0.45322 0.83918 0.047106
10 0.00389864     52   0.36842 0.84795 0.047326
11 0.00350877     71   0.28655 0.85965 0.047616
12 0.00292398     76   0.26901 0.88889 0.048329
13 0.00219298     95   0.21345 0.88889 0.048329
14 0.00194932     99   0.20468 0.91228 0.048888
15 0.00146199    102   0.19883 0.91520 0.048958
16 0.00097466    108   0.19006 0.93860 0.049505
17 0.00073099    114   0.18421 0.94152 0.049573
18 0.00058480    118   0.18129 0.94444 0.049641
19 0.00000100    123   0.17836 0.94444 0.049641
```

L'arbre maximal est ici beaucoup plus profond (123 coupures). L'arbre optimal se trouve à la ligne 7, on voit clairement que l'erreur `xerror` augmente pour les arbres en dessous de cette ligne. On peut visualiser graphiquement les erreurs des 19 arbres de la suite à l'aide de la commande **plotcp** :

```
> plotcp(arbre3)
```

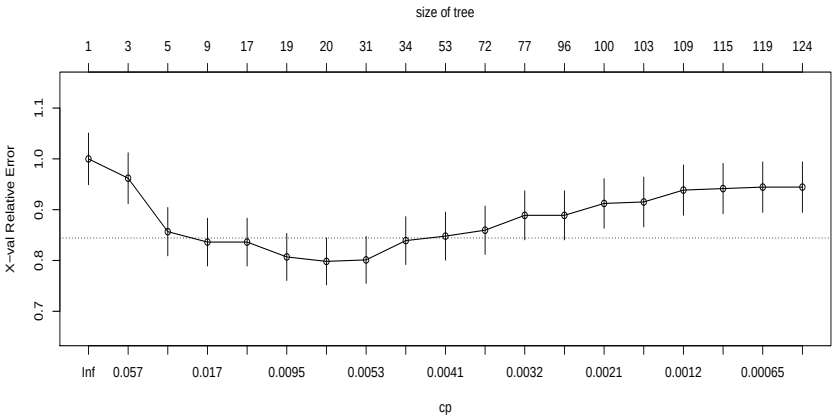


FIGURE 9.15 – Choix de la taille de l'arbre avec **plotcp**.

On choisit donc l'arbre à 19 coupures (ligne 7) dont la complexité vaut 0.00584795. Pour construire l'arbre final, on sélectionne dans `arbre3` le sous-arbre qui correspond à la valeur `cp=0.00584795` à l'aide de la fonction **prune** :

```
> library(tidyverse)
> cp_opt <- arbre3$cptable %>% as.data.frame() %>%
  filter(xerror==min(xerror)) %>% select(CP) %>% max() %>% as.numeric()
> arbre.fin <- prune(arbre3,cp=cp_opt)
> rpart.plot(arbre.fin)
```

La fonction **visTree** du package visNetwork propose une visualisation dynamique de l'arbre. On pourra obtenir une telle visualisation pour l'arbre sélectionné avec les commandes :

```
> library(visNetwork)
> visTree(arbre.fin)
```

4. Prédiction pour de pour de nouveaux individus
Les arbres construits précédemment peuvent être utilisés dans un contexte de prévision. La fonction **predict.rpart** (on peut utiliser simplement le raccourci **predict**) permet d'obtenir les probabilités $\mathbb{P}(Y = \texttt{yes}|X = x)$ et $\mathbb{P}(Y = \texttt{no}|X = x)$ prédites par l'arbre `arbre.fin` pour les individus de l'échantillon test :

```
> predict(arbre.fin,newdata=test) %>% head(n=3)
          no        yes
5  0.975478 0.02452203
12 0.975478 0.02452203
13 0.975478 0.02452203
```

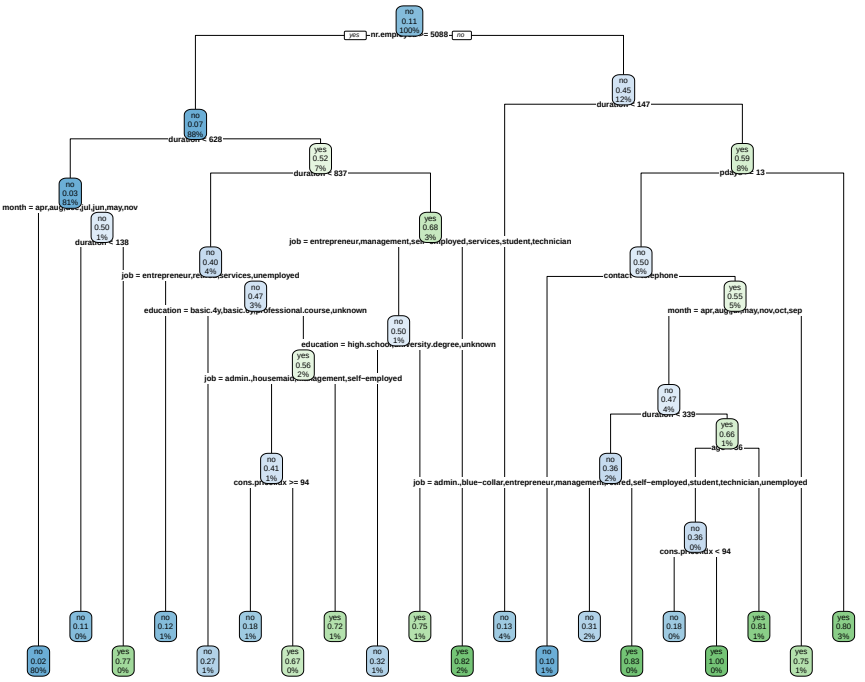


FIGURE 9.16 – Arbre final obtenu par validation croisée.

On obtient les classes prédites en ajoutant l'argument `type="class"` :

```
> predict(arbre.fin,newdata=test,type="class") %>% head(n=3)
 5 12 13
no no no
Levels: no yes
```

Les 3 premiers individus de l'échantillon test sont ici prédits dans le groupe `no`.

5. Estimer les performances de l'arbre

On souhaite évaluer les performances des arbres `arbre3` et `arbre.fin` en estimant les taux de mal classés et les courbes ROC de ces deux modèles sur l'échantillon test. On récupère d'abord les probabilités estimées d'être `yes` par les deux modèles pour les individus de l'échantillon test :

```
> prev.class<-data.frame(large=predict(arbre3,newdata=test,type="class"),
    fin=predict(arbre.fin,newdata=test,type="class"),obs=test$y)
```

```
> head(prev.class,n=3)
   large fin obs
5     no  no  no
12    no  no  no
13    no  no  no
```

On obtient les erreurs de classification estimées des 2 arbres en confrontant les valeurs prédites aux valeurs observées :

```
> prev.class %>% summarise_all(funs(err=mean(obs!=.))) %>%
        select(-obs_err) %>% round(3)
  large_err fin_err
1     0.118   0.095
```

L'arbre final possède une erreur plus petite. Les courbes ROC en `ggplot` s'obtiennent à l'aide du package plotROC :

```
> score<-data.frame(large=predict(arbre3,newdata=test)[,2],
   fin=predict(arbre.fin,newdata=test)[,2],obs=test$y)
> library(plotROC)
> df.roc <- score  %>% gather(key=methode,value=score,large,fin)
> ggplot(df.roc)+aes(d=obs,m=score,color=methode)+
   geom_roc()+theme_classic()
```

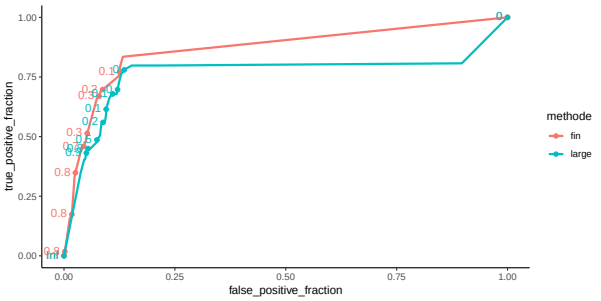


FIGURE 9.17 – Courbes ROC des 2 arbres considérés.

L'arbre final est également meilleur au sens de la courbe ROC (voir Fig. 9.17). Cela se confirme en calculant les AUC :

```
> df.roc %>% group_by(methode) %>% summarize(AUC=pROC::auc(obs,score))
# A tibble: 2 x 2
  methode   AUC
  <chr>   <dbl>
1 fin     0.867
2 large   0.766
```

6. Interpréter l'arbre
Le tracé de l'arbre permet une interprétation facile du modèle. Cependant, pour mesurer l'importance des variables, il ne faut pas se contenter de ce tracé. En effet, il est tout à fait possible que des variables n'apparaissant pas dans la construction de l'arbre soient importantes pour expliquer la variable d'intérêt. Il existe des scores d'importance qui permettent de mesurer la pertinence des variables explicatives. On pourra obtenir un de ces scores et le visualiser (Fig. 9.18) par :

```
> var.imp <- arbre.fin$variable.importance
> nom.var <- substr(names(var.imp),1,3)
> nom.var[c(4,5)] <- c("co.c","co.p") #éviter les noms identiques
> var.imp1 <- data.frame(var=nom.var,score=var.imp)
> var.imp1$var <- factor(var.imp1$var,levels=nom.var)
> ggplot(var.imp1)+aes(x=var,y=score)+geom_bar(stat="identity")+
      theme_classic()
```

FIGURE 9.18 – Scores d'importance des variables de l'arbre final.

Sur cet exemple, il semble que 7 variables se distinguent pour détecter les clients intéressés par le produit proposé.

Pour aller plus loin

Les arbres sont par essence instables. Pour cette raison, nous avons conseillé d'utiliser l'argument `maxcompete`, qui permet de voir les variables concurrentes. Par ailleurs, le principe des forêts aléatoires consiste à agréger un nombre important d'arbres : ceci peut par exemple se faire grâce aux packages randomForest ou ranger (voir fiche 10.2). L'algorithme CART souffre aussi de biais de sélection en faveur des covariables avec de nombreuses coupures possibles, comme par exemple des variables catégorielles avec de nombreuses catégories. Le package partykit implémente des « conditional inference tree » (CTree), pour pallier à ce problème et aux problèmes de surajustement. Enfin, pour plus d'informations sur les arbres de décision, on pourra par exemple se référer à Breiman *et al.* (1984) et Hastie *et al.* (2009).

9.8 Régression Partial Least Square (PLS)

Objet

La régression PLS consiste à prédire une variable quantitative Y par p variables quantitatives $X_1, \ldots, X_p$. En général, lorsque le nombre p de variables explicatives est très grand voire plus grand que le nombre n d'individus à analyser, il est difficile, voire impossible, d'utiliser la méthode des moindres carrés (cf. fiche 9.2). La régression PLS est particulièrement adaptée à ce cas de données en grande dimension comme les données de spectrométrie ou de génétique.

Dit simplement, la régression PLS recherche de façon itérative une suite de composantes (ou variables sous-jacentes) orthogonales entre elles. Ces composantes, appelées composantes PLS, sont choisies de façon à maximiser la covariance avec la variable explicative Y. Le choix du nombre de composantes est important car il influe fortement sur la qualité de la prévision. En effet, si le nombre choisi est insuffisant, des informations importantes peuvent être oubliées. Par contre si le nombre choisi est trop grand, on risque d'avoir un modèle surajusté. Ce choix est en général effectué par validation croisée ou apprentissage/validation. Ces deux procédures suivent le même principe : les données initiales sont séparées en deux parties distinctes, une partie d'apprentissage et une partie de validation, avec en général un nombre d'individus plus grand dans l'échantillon d'apprentissage que dans l'échantillon de validation. On construit le modèle de régression PLS à partir de l'échantillon d'apprentissage puis on effectue les prévisions sur l'échantillon de validation. On construit tout d'abord le modèle à une composante PLS puis on répète cette procédure pour $2, 3, \ldots, k$ composantes PLS. Le nombre de composantes retenu correspond au modèle qui conduit à l'erreur de prévision minimale. Cette procédure est implémentée dans le package pls.

Exemple

Nous nous intéressons à la prédiction de la teneur en carbone organique dans le sol en fonction de mesures spectroscopiques. Comme mesurer la teneur en carbone organique du sol est plus coûteux qu'effectuer des mesures spectroscopiques, nous souhaitons pouvoir prédire la teneur en carbone organique en fonction des mesures spectroscopiques.

Afin de construire un modèle de prévision, nous disposons d'une base de données de 177 sols (individus) avec à la fois la teneur en carbone organique (CO, variable à expliquer) et les spectres acquis dans le domaine du visible et du proche infrarouge (400 nm-2500 nm), ce qui donne 2101 variables explicatives. On veut ensuite prédire la teneur en carbone organique de trois nouveaux sols. Nous remercions Y. Fouad et C. Walter qui ont mis ces données [1] à notre disposition.

1. Aïchi H., Fouad Y., Walter C., Viscarra Rossel R.A., Lili Chabaane Z. & Sanaa M. (2009). Regional predictions of soil organic carbon content from spectral reflectance measurments. *Biosystems Engineering*, **104**, p. 442-446.

Étapes

1. Importer les données
2. Représenter les données
3. Effectuer une régression PLS après avoir choisi le nombre de composantes PLS
4. Analyser les résidus
5. Prévoir une nouvelle valeur

Traitement de l'exemple

1. Importer les données

Lors de l'importation, un « `X` » est ajouté par défaut aux noms des variables correspondant aux longueurs d'onde. Les données de spectres nécessitent souvent un prétraitement qui consiste à centrer-réduire les données d'un spectre. Ici, nous calculons la moyenne et l'écart-type de chaque spectre (i.e. de chaque ligne de la matrice, en oubliant la première colonne qui correspond à Y) puis, à toutes les données d'une ligne, nous soustrayons sa moyenne et divisons par son écart-type.

```
> don <- read.table("spe_bretagne.txt", sep=";", header=TRUE, row.names=1)
> moy.ligne <- apply(don[,-1], 1, mean)
> don[,-1] <- sweep(don[,-1], 1, moy.ligne, FUN="-")
> et.ligne <- apply(don[,-1], 1, sd)
> don[,-1] <- sweep(don[,-1], 1, et.ligne, FUN="/")
```

2. Représenter les données

On peut dans un premier temps visualiser la variable à expliquer Y, qui correspond à la teneur en carbone organique `CO`, via la représentation de son histogramme :

```
> hist(don[,"CO"], prob=TRUE, main="", xlab="Teneur en carbone organique")
> lines(density(don[,"CO"]))
> rug(don[,"CO"], col="red")
```

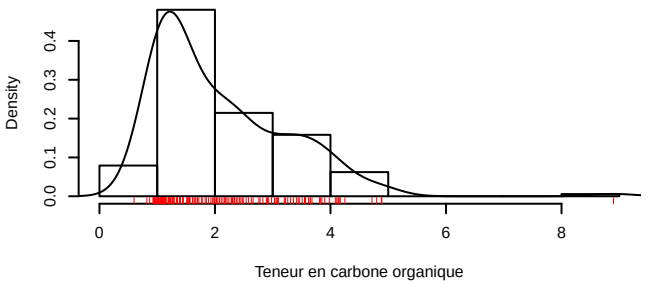


FIGURE 9.19 – Représentation de la teneur en carbone organique.

La fonction **rug** permet d'ajouter des traits verticaux sur l'axe des abscisses pour les observations de la variable Y. Nous voyons sur la figure 9.19 qu'une observation est extrême : elle est supérieure à 8 alors que les autres sont toutes en deçà de 5. Les lignes suivantes permettent de montrer que l'individu 79 prend une valeur très élevée (8.9). Les autres valeurs n'excèdent pas 4.89.

```
> which(don[,1]>8)
[1] 79
> max(don[-79,1])
[1] 4.89
```

Cet individu est particulier et peut être enlevé pour la construction du modèle :

```
> don <- don[-79,]
```

Les variables explicatives sont des spectres et peuvent être représentées par des courbes. Il est possible de représenter chaque spectre par un trait différent (trait plein, tiret, etc.) et/ou d'une couleur différente en fonction de la valeur de la variable `CO`. Pour réaliser ce graphique, nous allons découper la variable `CO` en 7 classes d'effectifs quasiment égaux par la fonction **cut**, les points de coupure étant déterminés par les quantiles (**quantile**). Le facteur ainsi obtenu est transformé en code couleur grâce à la fonction **as.numeric** et à l'argument `col`. La palette de couleurs choisie est celle fournie par **terrain.colors** pour 7 couleurs (avec une transparence de 0.5) et elle est mise en fonction grâce à **palette**. En abscisse, on utilise les longueurs d'onde :

```
> coul <- as.numeric(cut(don[,1], quantile(don[,1], prob=seq(0,1,by=1/7)),
    include.lowest = TRUE))
> palette(terrain.colors(7,alpha=0.5))
> lity <- coul
> matplot(x=400:2500, y=t(as.matrix(don[,-1])), type="l", lty=1,
    col=coul, xlab="Longueur d'onde", ylab="Réflectance")
```

Dans le graphe 9.20 où chaque individu est une courbe,

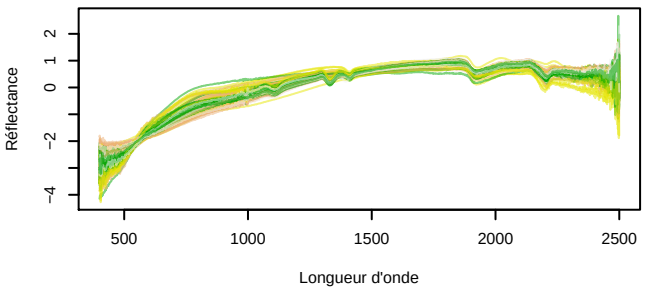


FIGURE 9.20 – Représentation des individus (après centrage-réduction en ligne).

Il semble possible de discerner des groupes de courbes dessinées avec le même code couleur, ce qui signifie que des courbes assez proches admettent des valeurs proches pour la variable `CO`.

3. Effectuer une régression PLS après avoir choisi le nombre de composantes PLS Pour effectuer cette régression, nous utilisons le package `pls`. Par défaut ce package centre toutes les variables mais ne les réduit pas. Nous réduisons les variables explicatives grâce à l'argument `scale`. Il faut également fixer un nombre maximum de composantes PLS (plus ce nombre est élevé plus le temps d'exécution du programme est long). Par sécurité, nous choisissons un nombre élevé, ici 100.

```
> library(pls)
> modele.pls <- plsr(CO~., ncomp=100, data=don,scale=TRUE,validation="CV")
```

Par défaut, le nombre de composantes PLS est déterminé par validation croisée. Nous calculons l'erreur d'ajustement et l'erreur de prévision obtenues avec 1, 2, ..., 100 composantes PLS et nous ensuite les deux types d'erreur (Fig. 9.21) :

```
> msepcv.pls <- MSEP(modele.pls, estimate=c("train","CV"))
> palette("default") ## retour aux couleurs habituelles
> plot(msepcv.pls, lty=1, type="l", legendpos="topright", main="")
```

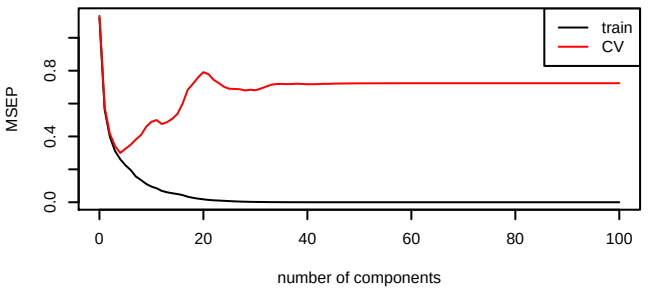


FIGURE 9.21 – Evolution des deux erreurs en fonction du nombre de composantes PLS.

Les deux courbes d'évolution des erreurs sont typiques : une décroissance continue pour l'erreur d'ajustement en fonction du nombre de composantes et une décroissance suivie d'une croissance pour l'erreur de prévision. Le nombre optimal de composantes pour la prévision correspond à la valeur pour laquelle l'erreur de prévision est minimum :

```
> ncomp.pls <- which.min(msepcv.pls$val["CV",,])-1
> ncomp.pls
4 comps
      4
```

La première erreur est donnée pour 0 composante, c'est pourquoi on soustrait 1 pour obtenir directement le nombre de composantes PLS, ici 4. Nous obtenons alors le modèle final avec les quatre composantes PLS :

```
> reg.pls <- plsr(CO~., ncomp=ncomp.pls, data=don, scale=TRUE)
```

4. Analyser les résidus
Les résidus sont obtenus par la fonction **residuals**. Attention, les résidus sont fournis pour le modèle PLS à 1 composante, puis pour le modèle à 2 composantes, ..., jusqu'au modèle avec le nombre de composantes choisi (ici 4). On dessine alors uniquement les résidus du dernier modèle.

```
> res.pls <- residuals(reg.pls)
> plot(res.pls[,,ncomp.pls],pch=15,cex=.5,ylab="Résidus",main="")
> abline(h=c(-2,0,2), lty=c(2,1,2))
```

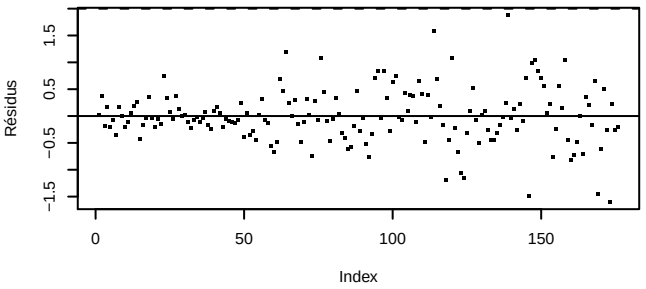


FIGURE 9.22 – Représentation des résidus.

Aucun individu n'est extrême au point de devoir être enlevé de l'analyse. Notons que si l'individu 79 n'avait pas été préalablement supprimé, celui-ci aurait eu un résidu très grand et aurait été supprimé à cette étape. La variabilité des résidus est plus grande pour les derniers individus du jeu de données (individus codés par `rmqs`) et plus faible pour les premiers (individus codés par `Bt`). L'échantillon de parcelles analysées ne semble pas très homogène.

5. Prévoir une nouvelle valeur
Nous avons obtenu un jeu de trois courbes complémentaires pour lesquelles nous allons comparer les résultats de la teneur en `CO` prévue par la régression PLS avec les valeurs effectivement mesurées.

```
> donN <- read.table("spe_nouveau.txt", sep=";", header=TRUE, row.names=1)
> moy.ligneN <- apply(donN[,-1], 1, mean)
> donN[,-1] <- sweep(donN[,-1], 1, moy.ligneN, FUN="-")
```

```
> et.ligneN <- apply(donN[,-1], 1, sd)
> donN[,-1] <- sweep(donN[,-1], 1, et.ligneN, FUN="/")
```

Les valeurs prédites sont obtenues en utilisant la fonction **predict** :

```
> pred <- predict(reg.pls, ncomp=ncomp.pls, newdata=donN[,-1])
> pred
, , 4 comps
                CO
3236     0.4065104
rmqs_726 2.6930854
rmqs_549 2.6035296
```

Ces valeurs prédites peuvent être comparées aux valeurs mesurées :

```
> donN[,1]
[1] 1.08 1.60 1.85
```

Pour aller plus loin

La régression PLS est souvent associée à des graphiques spécifiques permettant d'analyser plus finement le modèle construit et de décrire l'échantillon analysé.

Pour mieux connaître le rôle de chacune des variables dans le modèle de régression, il est possible de tracer les coefficients (figure 9.23) de chacune des longueurs d'onde par la commande suivante :

```
> plot(reg.pls, plottype="coef", comps=1:ncomp.pls, main="",
    legendpos="topleft", xlab="Longueur d'onde", lty=1,labels=400:2500)
```

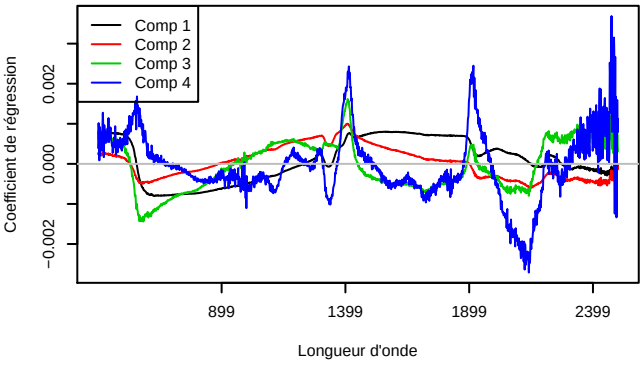


FIGURE 9.23 – Coefficients de la régression PLS pour des modèles de 1 à 4 composantes.

L'interprétation de ce type de graphe dépasse les objectifs de ce livre et nous nous contenterons de rappeler qu'il s'agit de coefficients pour chaque réflectance d'une longueur d'onde donnée. Ils sont donc interprétables comme ceux d'une régression multiple, à ceci près qu'ils sont ici tous sur la même échelle puisque les variables initiales (les réflectances) sont centrées et réduites.

Il est possible de représenter le nuage des individus sur les composantes PLS (appelées « PLS scores ») pour visualiser les ressemblances entre individus comme en ACP (cf. fiche 7.1). Par exemple, nous traçons les composantes 1 et 2 sur un repère orthonormé (`asp=1`) en colorant en rouge tous les échantillons du groupe `rmqs` grâce aux commandes suivantes :

```
> colo <- rep(1,nrow(don))
> colo[substr(rownames(don),0,4)=="rmqs"] <- 2
> plot(reg.pls, plottype="scores", comps=c(1,2), col=colo, asp=1)
> abline(h=0,lty=2)
> abline(v=0,lty=2)
```

Nous retrouvons (Fig. 9.24) encore plus clairement la séparation des individus observés en deux populations.

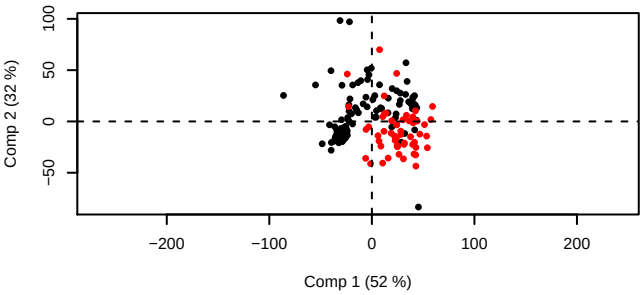


FIGURE 9.24 – Analyse du modèle par composante : composantes 1 et 2.

Une autre analyse classique consiste à représenter les corrélations entre les loadings et les variables comme en ACP (cf. graphe des variables, fiche 7.1). On visualise ainsi les variables les plus liées à chacune des composantes. On peut colorier les longueurs d'onde grâce à un dégradé de gris (du noir pour les premières variables au gris clair pour les dernières).

```
> coul <- gray(seq(0,.9,len=ncol(don)))
> plot(modele.pls,plottype="correlation",comps=1:2,col=coul,pch=20)
```

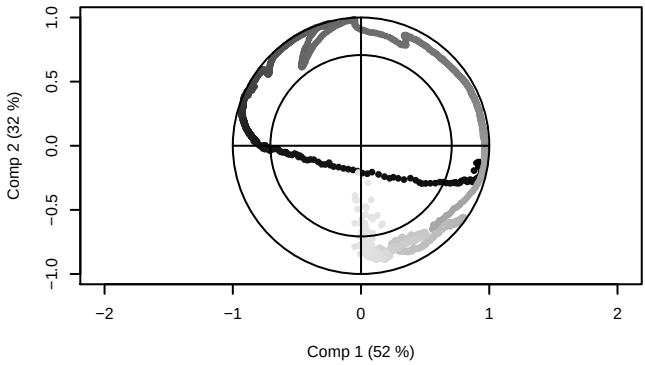


FIGURE 9.25 – Représentation des corrélations avec les loadings.

Quand les variables sont ordonnées (cas des longueurs d'onde), il est possible de représenter l'évolution de la liaison entre les composantes PLS et les variables. Ce type de graphique est obtenu avec l'argument `plottype="loadings"`.

```
> plot(modele.pls,plottype="loadings", labels=400:2500, comps=1:ncomp.pls,
    legendpos="topright",lty=1, xlab="Longueur d'onde", ylab="Loadings")
> abline(h=0,lty=2)
```

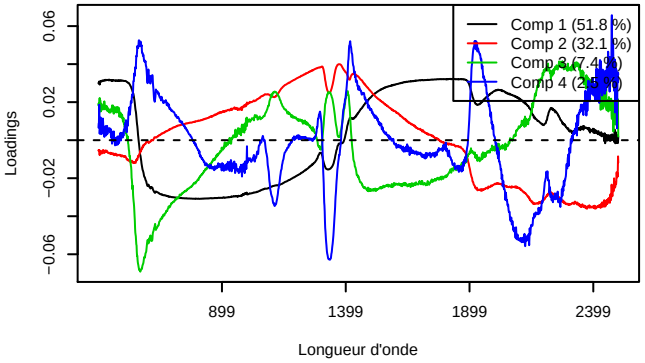


FIGURE 9.26 – Représentation des loadings.

Des rappels théoriques et des exercices sur la régression PLS existent dans beaucoup d'ouvrages comme Cornillon et Matzner-Løber (2010) ou Saporta (2011).

Chapitre 10

Machine learning

Ce chapitre expose plusieurs méthodes de *machine learning* qui ont toutes un même objectif de prédiction. Ces techniques requièrent typiquement le calibrage d'un ou plusieurs paramètres à partir des données. Aussi, la première fiche (10.1) de ce chapitre présente une stratégie pour calibrer un jeu de paramètres pour n'importe quelle méthode. Les fiches suivantes présentent diverses méthodes comme les forêts aléatoires (fiche 10.2), la régression sous contraintes (fiche 10.3), le gradient boosting (fiche 10.4), les SVM (fiche 10.5) ou encore le deep learning (fiche 10.6). Enfin, pour clore ce chapitre, nous proposons une stratégie permettant de comparer les qualités prédictives de différentes méthodes (fiche 10.7).

La grande majorité des méthodes de machine learning nécessite d'optimiser de nombreux paramètres. Ainsi, les packages associés à une méthode proposent le plus souvent des fonctions d'optimisation pour choisir les paramètres par validation croisée par exemple (cf. fiche suivante). Il est donc possible de traiter toutes les données et de choisir les paramètres avec le jeu de données initial. La méthode est alors calibrée pour prédire de nouveaux individus en utilisant une fonction **predict** comme nous l'avons fait en régression par exemple.

Dans chaque fiche de ce chapitre, nous présentons les fonctions qui permettent de calibrer une méthode puis de prédire de nouvelles données. Mais nous avons choisi d'insister sur l'évaluation de la qualité de prévision du modèle. Pour ce faire, nous avons décidé de découper initialement les données en une partie d'apprentissage et une partie de validation. Cette dernière permet d'évaluer les qualités prédictives du modèle optimisé sur l'échantillon d'apprentissage.

Nous détaillerons comment mettre en œuvre chaque méthode via l'enchaînement des différentes instructions R avant de commenter brièvement les résultats numériques et graphiques. Pour ce faire, nous utiliserons un seul et même jeu de données sur la détection automatique de spams. Sur 4 601 courriels, on a mesuré 57 indicateurs qui correspondent à la fréquence d'apparition dans le courriel de certains mots ou de certains caractères de ponctuation. On a également identifié parmi ces

courriels ceux qui sont des spams. Le but est de proposer une règle pour prédire si un nouveau message est un spam ou non à partir des mots utilisés dans le courriel. Le jeu de données est disponible dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

On présente seulement le résumé des cinq dernières variables :

```
> summary(spam[,54:58])
    charHash          capitalAve        capitalLong
 Min.   : 0.00000   Min.   :   1.000   Min.   :   1.00
 1st Qu.: 0.00000   1st Qu.:   1.588   1st Qu.:   6.00
 Median : 0.00000   Median :   2.276   Median :  15.00
 Mean   : 0.04424   Mean   :   5.191   Mean   :  52.17
 3rd Qu.: 0.00000   3rd Qu.:   3.706   3rd Qu.:  43.00
 Max.   :19.82900   Max.   :1102.500   Max.   :9989.00
  capitalTotal          type
 Min.   :    1.0   nonspam:2788
 1st Qu.:   35.0   spam   :1813
 Median :   95.0
 Mean   :  283.3
 3rd Qu.:  266.0
 Max.   :15841.0
```

Les 48 premières variables contiennent la fréquence dans le courriel du nom de la variable (par exemple `money`). La variable 54 indique la fréquence du caractère # dans le message. Les variables 55 à 57 contiennent la longueur moyenne, la longueur la plus grande et la longueur totale des lettres majuscules. Sur les 4 601 courriels, 1 813 ont été identifiés comme des spams.

10.1 Calibration d'un algorithme avec **caret**

Objet

Un des problèmes les plus courants de l'apprentissage supervisé consiste à prédire une sortie $y \in \mathcal{Y}$ à partir d'entrées $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$. Il s'agit donc de chercher un algorithme de prévision $f : \mathbb{R}^d \to \mathcal{Y}$ qui pour une entrée x renverra la prévision $y = f(x)$. Pour trouver la « meilleure » fonction f, le statisticien dispose d'un échantillon $(X_1, Y_1), \ldots, (X_n, Y_n)$ composé de n couples indépendants et identiquement distribués à valeurs dans $\mathbb{R}^d \times \mathcal{Y}$. La plupart des méthodes statistiques fournissent à l'utilisateur une famille d'algorithmes $\{f_\theta, \theta \in \Theta\}$, où θ désigne le paramètre à calibrer et Θ l'ensemble des valeurs possibles de ce(s) paramètre(s). Le paramètre θ peut par exemple représenter le nombre de plus proches voisins pour l'algorithme des plus proches voisins, le nombre d'itérations en boosting, la profondeur de l'arbre pour les arbres CART, etc. Ce paramètre permet le plus souvent de mesurer la complexité du modèle. Une trop faible complexité ne permettra pas d'ajuster suffisamment bien les données et engendrera un mauvais pouvoir prédictif. À l'inverse, un modèle trop complexe ajustera presque parfaitement les données d'apprentissage mais aura du mal à bien prédire de nouveaux individus : on parle alors de sur-ajustement ou sur-apprentissage (Fig. 10.1).

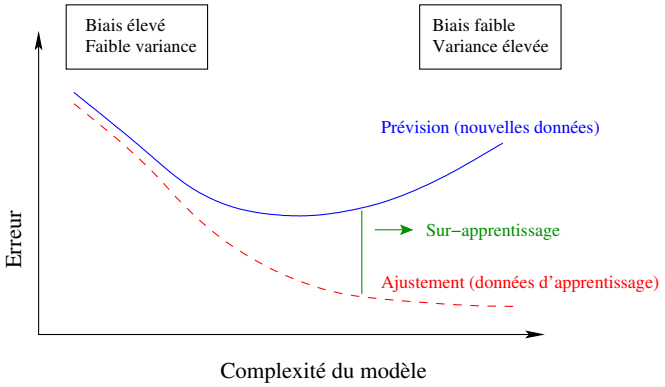


FIGURE 10.1 – Le problème du sur-apprentissage.

Ainsi, le choix de θ se révèle crucial pour la performance de l'algorithme de prévision et l'utilisateur doit trouver une procédure permettant de sélectionner automatiquement ce paramètre. Ce processus de sélection dépend de plusieurs points :
- l'algorithme d'apprentissage (arbres, forêts, boosting, etc.) ;
- la grille des paramètres candidats Θ ;
- la fonction de perte (AUC, taux de mal classés, erreur quadratique moyenne, etc.) ;
- la méthode d'estimation du critère (apprentissage/validation, validation croisée, bootstrap, estimation out of bag, etc.).

Le package caret (Classification And REgression Training) propose une démarche générale permettant de calibrer les paramètres d'un très grand nombre d'algorithmes statistiques (plus de 230 algorithmes en 2018). Un descriptif complet du package est disponible à l'url `http://topepo.github.io/caret/index.html`. Ce package propose un très grand nombre de fonctionnalités. Nous considérerons ici uniquement la sélection automatique de paramètres à l'aide de la fonction **train**.

Exemple

Nous prenons l'exemple de la détection automatique de spams (voir page 321).

Étapes

1. Importer les données
2. L'algorithme des plus proches voisins
3. Calibration des paramètres
4. Compléments

Traitement de l'exemple

1. Importer les données
Le jeu de données se trouve dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

2. L'algorithme des plus proches voisins
Nous illustrons le choix de paramètres avec caret pour l'algorithme des k plus proches voisins. Etant donné un entier $k \leq n$, cette méthode consiste à prédire le label d'un nouvel individu x en faisant un vote à la majorité parmi les labels des k plus proches voisins de x. Cette méthode dépend bien entendu du choix de k :
- si k est trop grand, on aura toujours tendance à prédire le groupe majoritaire de l'échantillon, l'estimateur aura un biais élevé ;
- si k est petit, très peu d'individus seront utilisés pour faire la prévision, l'estimateur possédera une grande variance et sur-ajustera les données.

On désigne par f_k la règle des k plus proches voisins et on considère l'erreur de classification ou taux de mal classés comme risque. Nous proposons tout d'abord d'estimer ce risque par apprentissage/validation. Cette méthode consiste à couper l'échantillon en 2 sous-échantillons indépendants. Le premier sous-échantillon, appelé échantillon d'apprentissage, est utilisé pour construire la règle f_k. Son erreur de classification est ensuite estimée par la proportion d'erreur sur le deuxième échantillon, appelé échantillon de validation.

À titre d'exemple, nous illustrons cette technique pour la règle des 3 plus proches voisins. On coupe d'abord l'échantillon en 2 : un échantillon d'apprentissage de taille 3 000 et un échantillon de validation de taille 1 601 :

```
> set.seed(1234)
> spam1 <- spam[sample(nrow(spam)),]
> app <- spam1[1:3000,]
> valid <- spam1[-(1:3000),]
```

On ajuste la règle des 3 plus proches voisins sur les données d'apprentissage et on prédit le groupe des individus de l'échantillon de validation à l'aide de la fonction **knn** du package class :

```
> library(class)
> reg3ppv <- knn(app[,-58],valid[,-58],cl=app$type,k=3)
```

On déduit l'erreur de classification en confrontant les prévisions aux observations :

```
> mean(reg3ppv!=valid$type)
[1] 0.1923798
```

On pourra conclure qu'environ 19 % des courriels sont mal classés par cette règle. Un moyen naturel de choisir automatiquement le nombre de plus proches voisins est de répéter ce protocole pour chaque valeur de k dans $\{1, \ldots, k_{\max}\}$ et de choisir la valeur pour laquelle l'erreur de classification est minimale. Il suffirait de faire une boucle sur les valeurs de k. Le package caret permet d'effectuer cette tâche.

3. Calibration des paramètres

La fonction **train** du package caret permet d'estimer les risques classiques d'un très grand nombre d'algorithmes sur des grilles de paramètres fixées par l'utilisateur. Il faut tout d'abord spécifier le type d'algorithme souhaité et la grille de valeurs pour les paramètres à calibrer. Pour trouver ces éléments, nous recommandons de consulter la page située à l'url `http://topepo.github.io/caret/index.html`. Il faut ensuite indiquer la méthode souhaitée dans la section `Available model`. Par exemple, pour les plus proches voisins on pourra indiquer `knn` comme sur la copie d'écran de la figure 10.2.

Ainsi, pour faire des plus proches voisins avec caret, il faudra indiquer `knn` dans l'argument `method` de la fonction **train**. La grille de valeurs du nombre de plus proches voisins devra être un data-frame avec k comme nom de colonne (comme indiqué en dernière colonne de la figure 10.2). Si l'on décide de sélectionner k entre 1 et 100, la grille sera définie par :

```
> grille.K <- data.frame(k=seq(1,100,by=1))
```

Une fois ces paramètres renseignés, il reste à préciser dans **train** le risque utilisé et la méthode d'estimation de ce risque. Par défaut, le risque utilisé est le taux

6 Available Models

The models below are available in `train`. The code behind these protocols can be obtained using the function `getModelInfo` or by going to the github repository.

Show 237 entries

Search: knn

Model	*method* Value	Type	Libraries	Tuning Parameters
k-Nearest Neighbors	kknn	Classification, Regression	kknn	kmax, distance, kernel
k-Nearest Neighbors	knn	Classification, Regression		k

Showing 1 to 2 of 2 entries (filtered from 237 total entries)

FIGURE 10.2 – Informations pour les paramètres des k plus proches voisins.

de bien classés (un moins l'erreur de classification) pour la classification supervisée et l'erreur quadratique moyenne en régression. La méthode d'estimation du risque est à renseigner dans la fonction **trainControl**. Si l'on souhaite passer par apprentissage/validation (ou validation hold out) comme en partie précédente, on effectuera :

```
> library(caret)
> ctrl1 <- trainControl(method="LGOCV",number=1,index=list(1:3000))
```

`LGOCV` indique que l'on veut utiliser la technique validation hold out, l'argument `number` contrôle le nombre de fois où l'on souhaite répéter le processus : si `number=1`, la procédure est effectuée une seule fois comme ci-dessus, si ce n'est que l'on va balayer toutes les valeurs de `grille.K`. On indique dans `index` les individus que l'on souhaite avoir dans l'échantillon d'apprentissage : ici les 3 000 premières observations de `spam1` seront dans l'échantillon d'apprentissage, les 1 601 autres dans l'échantillon de validation, exactement comme en section précédente.

Si `number=10`, la procédure est répétée 10 fois et la fonction retourne la moyenne sur les 10 répétitions. Les résultats seront plus stables mais plus longs à obtenir. Dans ce cas, si on ne renseigne pas l'argument `index`, le découpage est fait aléatoirement. On peut maintenant lancer la procédure de sélection de k :

```
> sel.k1 <- train(type~.,data=spam1,method="knn",trControl=ctrl1,
    tuneGrid=grille.K)
> sel.k1
```

```
k-Nearest Neighbors
4601 samples
  57 predictor
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling: Repeated Train/Test Splits Estimated (1 reps, 75%)
Summary of sample sizes: 3000
Resampling results across tuning parameters:

  k    Accuracy   Kappa
    1  0.7926296  0.5601299
    2  0.7695191  0.5133254
    3  0.8076202  0.5894339
    4  0.7920050  0.5551572
    5  0.8082448  0.5918892
    6  0.7995003  0.5730181
    7  0.7938788  0.5606447
           ⋮
Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 5.
```

Pour chaque valeur de k dans `grille.K` le taux de bien classés (`Accuracy`) et le Kappa de Cohen (`Kappa`) sont calculés. On retrouve au passage l'erreur de classification précédente pour la règle des 3 plus proches puisque $1 - 0.8076202 = 0.19238$. L'accuracy est maximale pour `k=5`, avec cette technique nous choisirons donc la règle des 5 plus proches voisins, comme indiqué à la fin de la sortie. On peut récupérer cette valeur avec :

```
> sel.k1$bestTune
  k
5 5
```

Un **plot** appliqué à l'objet construit permet de visualiser la précision (`Accuracy`) en fonction de k (voir Fig. 10.3) :

```
> plot(sel.k1)
```

4. Compléments

Le package caret permet de sélectionner automatiquement et avec efficacité des paramètres en très peu de lignes de codes. L'utilisateur a néanmoins toujours des choix à effectuer et il importe de maîtriser les techniques utilisées. Un des principaux risques avec ce genre d'outils est de laisser le package dominer et de perdre le contrôle de l'analyse. Parmi les choix à faire, il y a en premier lieu la méthode et la grille de paramètres. Pour trouver une grille pertinente, il est

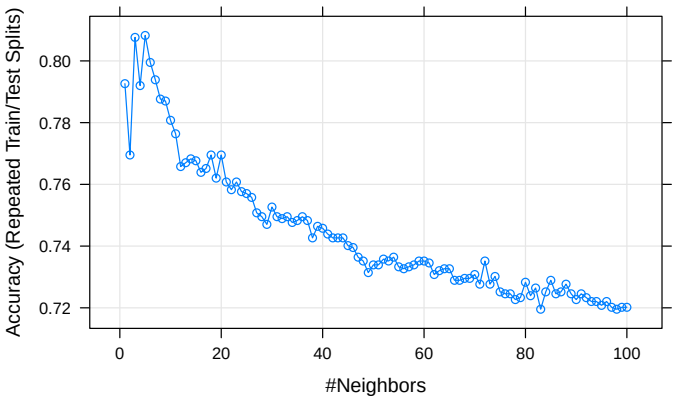


FIGURE 10.3 – Précision en fonction du nombre de plus proches voisins.

souvent nécessaire de tester au préalable l'algorithme pour quelques valeurs de paramètres. La technique d'estimation du critère doit également être choisie par l'utilisateur. Ici, nous avons appliqué la validation hold out, mais il existe bien d'autres méthodes : validation croisée, bootstrap, out of bag, etc. La méthode d'estimation du critère est à renseigner dans **trainControl**. Par exemple, pour faire de la validation croisée 10 blocs, on utilisera :

```
> ctrl2 <- trainControl(method="cv",number=10)
> set.seed(123)
> sel.k2 <- train(type~.,data=spam1,method="knn",trControl=ctrl2,
    tuneGrid=grille.K)
> sel.k2
k-Nearest Neighbors
4601 samples
  57 predictor
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 4140, 4141, 4142, 4141, 4140, 4141, ...
Resampling results across tuning parameters:
  k    Accuracy   Kappa
    1  0.8219977  0.6275191
    2  0.7878747  0.5555259
    3  0.8046163  0.5909410
    4  0.7976588  0.5756348
    5  0.8083077  0.5972499
           ⋮
Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 1.
```

On sélectionne cette fois `k=1`. La validation croisée peut se révéler coûteuse en temps de calcul. Si l'on dispose d'une machine avec plusieurs cœurs, il est facile de lancer les calculs en parallèle en utilisant le package doParallel. Nous comparons ici les temps de calculs pour la validation croisée lancée sur 1 cœur et sur 4 cœurs :

```
> set.seed(123)
> system.time(sel.k3 <- train(type~.,data=spam1,method="knn",
    trControl=ctrl2,tuneGrid=grille.K))
utilisateur     système      écoulé
    125.045       1.173     126.354
> library(doParallel)  ## pour paralléliser
> cl <- makePSOCKcluster(4)
> registerDoParallel(cl)   ## les clusters seront fermés à la fin
> set.seed(123)
> system.time(sel.k4 <- train(type~.,data=spam1,method="knn",
    trControl=ctrl2,tuneGrid=grille.K))
utilisateur     système      écoulé
    197.401       3.106      52.071
```

Sur cet exemple, le temps de calcul est divisé par 2.5 en utilisant 4 cœurs. Enfin, un dernier choix à effectuer par l'utilisateur concerne le critère de performance. Par défaut, **train** utilise le taux de bien classés (`Accuracy`). D'autres critères peuvent bien entendu être envisagés. Si l'on souhaite par exemple considérer l'aire sous la courbe ROC (AUC), il faudra dans un premier temps ajouter les arguments `classProbs=TRUE` et `summary=twoClassSummary` dans la fonction **trainControl**. Pour de la validation hold out, on tapera donc :

```
> ctrl3 <- trainControl(method="LGOCV",number=1,index=list(1:3000),
    classProbs=TRUE,summary=twoClassSummary)
```

Il reste ensuite à ajouter `metric=ROC` dans la fonction **train** :

```
> sel.k5 <- train(type~.,data=spam1,method="knn",trControl=ctrl3,
    metric="ROC",tuneGrid=grille.K)
> sel.k5
k-Nearest Neighbors

4601 samples
  57 predictor
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling: Repeated Train/Test Splits Estimated (1 reps, 75%)
Summary of sample sizes: 3000
Resampling results across tuning parameters:
```

```
  k    ROC        Sens       Spec
    1  0.7787106  0.8394309  0.7179903
    2  0.8338225  0.8313008  0.6985413
    3  0.8533736  0.8628049  0.7196110
    4  0.8678845  0.8384146  0.7131280
    5  0.8717494  0.8587398  0.7277147
    6  0.8669753  0.8465447  0.7131280
    7  0.8660571  0.8516260  0.7066451
           ⋮
ROC was used to select the optimal model using the largest value.
The final value used for the model was k = 5.
```

Les critères `Accuracy` et `Kappa` ont été remplacés par `ROC` pour l'AUC, `Sens` pour la sensibilité et `Spec` pour la spécificité. La valeur de k sélectionnée est ici celle qui maximise l'AUC : `k=5`. On récupère les performances de cette règle avec :

```
> getTrainPerf(sel.k5)
   TrainROC TrainSens TrainSpec method
1 0.8717494 0.8587398 0.7277147    knn
```

Pour aller plus loin

Dans cette fiche, nous avons vu que le package caret permet de calibrer de nombreuses méthodes. Il est également possible de calibrer son propre algorithme de prévision. Par ailleurs, caret propose aussi de traiter un très grand nombre d'autres étapes d'une étude statistique : pré-traitement des données (centrage-réduction, gestion des données manquantes, etc.), sélection de variables, etc. Là encore, il faut être prudent dans l'utilisation de cet outil et s'assurer qu'on a une maîtrise suffisante de ce qui est fait par le package. Pour plus d'informations sur la sélection de paramètres et le package caret, on pourra se référer à Kuhn et Johnson (2013).

La méthodologie présentée dans cette fiche est susceptible de dépendre trop fortement des découpages initiaux des données. Une solution pour pallier à ce problème consiste à répéter un certain nombre de fois les algorithmes de validation hold out ou de validation croisée en modifiant simplement le découpage initial. L'argument `repeats` de la fonction **trainControl** permet d'effectuer ces répétitions facilement. On pourra par exemple répéter 20 fois la validation croisée 10 blocs pour sélectionner le nombre de plus proches voisins avec

```
> ctrl3 <- trainControl(method="repeatedcv",number=10,repeats=20)
> train(type~.,data=spam1,method="knn",trControl=ctrl3,tuneGrid=grille.K)
> stopCluster(cl)  # fermeture des clusters utilisés pour calcul parallèle
```

10.2 Forêts aléatoires

Objet

Tout comme les arbres, les forêts aléatoires permettent de prédire une variable quantitative ou qualitative à partir de variables explicatives quantitatives et qualitatives. Cette famille d'algorithmes a été introduite par Breiman (2001) et permet de pallier, dans une certaine mesure, le manque de stabilité des arbres. La méthode est simple à mettre en œuvre et possède souvent de bonnes performances en terme de qualité de prédiction sur des jeux de données complexes, notamment en présence d'un grand nombre de variables explicatives.

Comme son nom l'indique, une forêt aléatoire consiste à agréger un grand nombre d'arbres de classification ou de régression (selon la nature de la variable à expliquer). Le caractère aléatoire du processus vient tout d'abord du fait que les arbres sont construits sur des échantillons bootstrap. Les échantillons bootstrap s'obtiennent généralement par tirage de n observations avec remise parmi n dans l'échantillon initial et les arbres sont construits selon une légère variante de l'algorithme CART présenté en fiche 9.7. En particulier, un second niveau d'aléatoire est introduit à chaque étape de la construction des arbres : à chaque nœud, seul un sous-ensemble de variables est sélectionné aléatoirement pour choisir la coupure. L'estimateur des forêts aléatoires en régression s'écrit

$$\hat{F}(x) = \frac{1}{B} \sum_{k=1}^{B} T_k(x),$$

où $T_k(x)$ représente l'arbre de décision construit sur le k-ème échantillon bootstrap.

Exemple

Nous prenons l'exemple de la détection automatique de spams présenté en introduction de ce chapitre (page 321).

Étapes

1. Importer les données
2. Construire et analyser une forêt aléatoire
3. Sélectionner les paramètres de la forêt
4. Faire de la prévision
5. Estimer les performances de la forêt
6. Interpréter la forêt aléatoire

Traitement de l'exemple

1. Importer les données
Le jeu de données se trouve dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

On sépare l'échantillon en un échantillon d'apprentissage de taille 3 000 et un échantillon de validation de taille 1 601. L'échantillon d'apprentissage sera utilisé pour construire la forêt et sélectionner ses paramètres, celui de validation pour estimer les performances de la forêt et vérifier que cette dernière ne fait pas de sur-ajustement.

```
> set.seed(5678)
> perm <- sample(4601,3000)
> app <- spam[perm,]
> valid <- spam[-perm,]
```

2. Construire et analyser une forêt aléatoire
Plusieurs packages permettent de construire des forêts aléatoires, nous utiliserons le package randomForest et la fonction **randomForest**. La syntaxe de cette fonction est similaire aux autres fonctions de régression ou de discrimination, une formule indique la variable à expliquer et les variables explicatives.

```
> library(randomForest)
> set.seed(1234)
> foret <- randomForest(type~.,data=app)
> foret

Call:
 randomForest(formula = type ~ ., data = app)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 7

        OOB estimate of  error rate: 4.73%
Confusion matrix:
        nonspam spam class.error
nonspam    1764   54  0.02970297
spam         88 1094  0.07445008
```

La sortie renseigne sur :
– le type de forêt aléatoire ajustée : `classification`, puisque la variable à expliquer est ici qualitative ;
– le nombre d'arbres agrégés : 500, la valeur par défaut ;

– le paramètre `mtry`, qui correspond au nombre de variables sélectionnées à chaque étape de construction des arbres : 7 (par défaut ce paramètre est égal à la partie entière de la racine carrée du nombre de variables explicatives en classification et au nombre de variables divisé par 3 en régression) ;
– le taux d'erreur et la matrice de confusion estimés par la méthode Out Of Bag.

La méthode Out Of Bag consiste à utiliser les observations qui ne sont pas dans les échantillons bootstrap comme un échantillon de validation pour estimer la performance de la forêt. Cette astuce permet d'éviter de faire de la validation croisée pour avoir une idée précise de la performance de la forêt aléatoire en terme de prévision. Le taux d'erreur estimé par cette technique est ici de 4.73 %.

3. Sélectionner les paramètres de la forêt

Plusieurs paramètres peuvent être calibrés pour construire une forêt aléatoire, notamment le nombre d'arbres `ntree` et le nombre de variables `mtry` sélectionnées aléatoirement à chaque étape de construction des arbres. Un des avantages des forêts aléatoires est la stabilité des performances en fonction de ces paramètres. Il faut néanmoins s'assurer de les choisir convenablement. Concernant le nombre d'arbres, il est recommandé de le prendre assez grand. La théorie garantit que la forêt aléatoire est convergente lorsque le nombre d'arbres tend vers l'infini. On choisit généralement le nombre d'arbres de manière à se situer aussi près que possible de la limite. Le nombre d'itérations (arbres) pour atteindre le régime limite dépend bien entendu des données, le plus souvent on utilise la fonction **plot** sur la forêt et on regarde si les taux d'erreur représentés dans la figure construite sont stables :

```
> plot(foret)
```

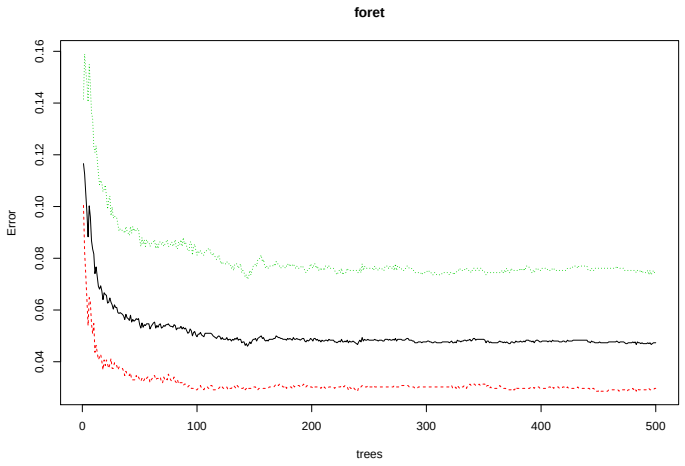


FIGURE 10.4 – Estimation des taux d'erreur en fonction du nombre d'arbres.

La courbe noire correspond à l'erreur de classification totale, les courbes verte et rouge aux erreurs de classification pour les spams et les non-spams. On peut retrouver les valeurs dans `foret$err.rate` :

```
 > tail(foret$err.rate)
              OOB    nonspam       spam
[495,] 0.04733333 0.02915292 0.07529611
[496,] 0.04666667 0.02915292 0.07360406
[497,] 0.04700000 0.02915292 0.07445008
[498,] 0.04733333 0.02915292 0.07529611
[499,] 0.04733333 0.02970297 0.07445008
[500,] 0.04733333 0.02970297 0.07445008
```

Sur cet exemple, les taux d'erreur sont stables, on pourra donc conserver une forêt avec 500 arbres. Si le nombre d'arbres n'est pas suffisant, il suffit de modifier le paramètre `ntree` dans la fonction **randomForest**.

Concernant le paramètre `mtry`, nous proposons ici de le sélectionner à l'aide du package caret (voir fiche 10.1). On considère la grille suivante de candidats :

```
> grille.mtry <- data.frame(mtry=seq(1,57,by=3))
```

On estime ensuite le taux de bien classés (`Accuracy`) en utilisant la technique Out Of Bag. Nous privilégions cette technique car elle est bien adaptée aux forêts aléatoires, elle est notamment plus rapide qu'une validation croisée.

```
> library(caret)
> ctrl <- trainControl(method="oob")
> library(doParallel)  ## pour paralléliser
> cl <- makePSOCKcluster(4)
> registerDoParallel(cl)
> set.seed(12345)
> sel.mtry <- train(type~.,data=app,method="rf",trControl=ctrl,
         tuneGrid=grille.mtry)
> stopCluster(cl)      # fermeture des clusters
> sel.mtry
Random Forest

3000 samples
  57 predictor
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
   1    0.9233333  0.8356570
```

```
   4    0.9506667  0.8960131
   7    0.9536667  0.9025254
  10    0.9540000  0.9032122
  13    0.9536667  0.9025543
  16    0.9540000  0.9033846
  19    0.9503333  0.8955749
          ⋮
Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 10.
```

Avec cette méthode de sélection, nous choisirons `mtry=10`.

4. Faire de la prévision

La fonction **predict.randomForest**, que l'on peut appeler en utilisant le raccourci **predict**, permet de prédire le label d'un nouveau courriel. Pour la forêt construite avec le paramètre `mtry` sélectionné dans la partie précédente, on obtient les prévisions pour les individus de l'échantillon de validation avec :

```
> set.seed(5432)
> foret1 <- randomForest(type~.,data=app,mtry=10)
> prev.valid <- predict(foret1,newdata=valid)
> prev.valid[1:10]
   2      6      9     11     15     16     18     19     21     22
spam   spam   spam   spam   spam   spam   spam   spam nonspam   spam
Levels: nonspam spam
```

Parmi les dix premiers individus de l'échantillon de validation, 9 sont considérés comme `spam`. On peut également estimer la probabilité qu'un nouveau courriel soit un spam en ajoutant l'argument `type="prob"` :

```
> prob.valid <- predict(foret1,newdata=valid,type="prob")
> prob.valid[1:10,]
   nonspam  spam
2    0.000 1.000
6    0.306 0.694
..   ..... .....
21   0.542 0.458
22   0.026 0.974
```

La probabilité d'être un spam pour le premier courriel de l'échantillon de validation est estimée à 1.000, celle pour le second à 0.694, etc. Par défaut, quand les probabilités sont supérieures à 0.5, le courriel est prévu comme spam.

5. Estimer les performances de la forêt

On souhaite estimer la performance de la forêt sélectionnée (avec `mtry=10`) en estimant son taux de mal classés et sa courbe ROC. Nous allons également comparer ces performances à la forêt utilisant la valeur par défaut de `mtry`, c'est-à-dire

mtry=7. Les taux de mal classés peuvent être estimés en utilisant l'échantillon de validation ou par la méthode Out Of Bag. On les obtient directement avec la fonction **randomForest** :

```
> set.seed(5432)
> foret2 <- randomForest(type~.,data=app,
         xtest=valid[,-58],ytest=valid[,58],keep.forest=TRUE)
> set.seed(891)
> foret3 <- randomForest(type~.,data=app,mtry=10,
         xtest=valid[,-58],ytest=valid[,58],keep.forest=TRUE)
> foret2

Call:
 randomForest(formula = type ~ ., data = app, xtest = valid[,-58],
         ytest = valid[, 58], keep.forest = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 7

        OOB estimate of  error rate: 4.63%
Confusion matrix:
        nonspam spam class.error
nonspam    1760   58  0.03190319
spam         81 1101  0.06852792
                Test set error rate: 5.5%
Confusion matrix:
        nonspam spam class.error
nonspam     941   29  0.02989691
spam         59  572  0.09350238
> foret3
Call:
 randomForest(formula = type ~ ., data = app, mtry = 10,
         xtest = valid[,-58],ytest = valid[, 58], keep.forest = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 10

        OOB estimate of  error rate: 4.5%
Confusion matrix:
        nonspam spam class.error
nonspam    1765   53  0.02915292
spam         82 1100  0.06937394
                Test set error rate: 5.68%
Confusion matrix:
        nonspam spam class.error
nonspam     939   31  0.03195876
spam         60  571  0.09508716
```

On lit dans les sorties les taux d'erreurs estimés par les deux approches. On reporte ces taux dans le tableau 10.1.

	OOB	Ech. test
`mtry`=7	4.63 %	5.5 %
`mtry`=10	4.5 %	5.68 %

TABLE 10.1 – Taux d'erreur en fonction du paramètre `mtry`.

Les taux d'erreur sont relativement proches, ce qui confirme bien que la méthode est relativement peu sensible aux choix des paramètres. De plus, ces taux d'erreur sont du même ordre que ceux obtenus sur les données d'apprentissage dans la section 3. On peut donc considérer qu'il n'y a pas de sur-ajustement pour ces deux forêts.
Le package pROC permet de tracer rapidement les courbes ROC et de calculer les aires sous les courbes ROC (AUC) :

```
> library(pROC)
> prev2 <- predict(foret2,newdata=valid,type="prob")[,2]
> roc2 <- roc(valid$type,prev2)
> prev3 <- predict(foret3,newdata=valid,type="prob")[,2]
> roc3 <- roc(valid$type,prev3)
> plot(roc2,print.auc=TRUE,print.auc.x=0.4,print.auc.y=0.3)
> plot(roc3,add=TRUE,col="red",print.auc=TRUE,print.auc.col="red",
        print.auc.x=0.4,print.auc.y=0.2)
```

Nous comparons les courbes ROC et les AUC (voir Fig. 10.5) de ces deux forêts aléatoires à celles d'un arbre de classification (voir fiche 9.7) :

```
> library(rpart)
> set.seed(12345)
> arbre <- rpart(type~.,data=app,cp=0.0001)
> library(tidyverse)
> cp_opt <- arbre$cptable %>% as.data.frame() %>%
   filter(xerror==min(xerror)) %>% select(CP) %>% max() %>% as.numeric()
> arbre_sel <- prune(arbre,cp=cp_opt)
> prev.arbre <- predict(arbre_sel,newdata=valid,type="prob")[,2]
> roc.arbre <- roc(valid$type,prev.arbre)
> plot(roc.arbre,add=TRUE,col="blue",print.auc=TRUE,
        print.auc.col="blue",print.auc.x=0.4,print.auc.y=0.1)
```

Nous remarquons que les performances des deux forêts sont très proches et, comme souvent, nettement meilleures que celles de l'arbre de classification.

6. Interpréter la forêt aléatoire
On reproche souvent aux forêts aléatoires l'aspect « boîte noire » et le manque d'interprétabilité. Il existe néanmoins des indicateurs qui permettent de mesurer

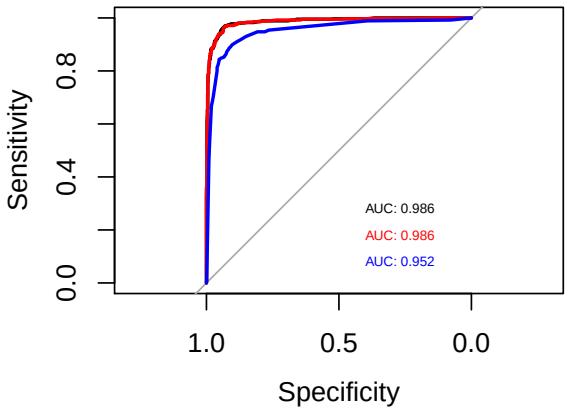


FIGURE 10.5 – Courbes ROC et AUC des forêts aléatoires avec `mtry=7` (noir), `mtry=10` (rouge) et d'un arbre de classification (bleu).

l'importance des variables explicatives dans la construction de la forêt. La sortie `importance` de la fonction **randomForest** renvoie une note (ou un score) pour chaque variable : plus la note est élevée, plus la variable est importante dans la forêt. Cette note est construite à partir de l'erreur OOB de la forêt. Pour chaque variable, on compare l'erreur OOB de la forêt à l'erreur calculée sur l'échantillon OOB où les valeurs de la variable en question ont été permutées aléatoirement. Si l'écart entre ces deux erreurs est petit, cela signifie que les valeurs de la variable n'ont pas grande importance vis-à-vis de la performance de la forêt. A contrario, plus l'écart est grand, plus la variable sera importante. Les commandes suivantes permettent de représenter, à l'aide d'un diagramme en bâtons, les 10 variables les plus importantes de la forêt `foret2` selon ce critère (voir figure 10.6) :

```
> var.imp <- foret2$importance
> ord <- order(var.imp,decreasing=TRUE)
> barplot(sort(var.imp,decreasing = TRUE)[1:10],
    names.arg=rownames(var.imp)[ord][1:10],cex.names=0.6)
```

Le point d'exclamation est le caractère le plus influent pour identifier les spam. La valeur de la mesure d'importance n'a pas vraiment d'interprétation, on l'utilisera généralement pour ordonner les variables par ordre décroissant d'importance. Pour un grand nombre de variables, on pourra aussi utiliser cette mesure d'importance pour faire de la sélection de variables à l'aide d'une procédure en 2 étapes :

- on construit la forêt et on calcule l'importance de chaque variable ;
- on reconstruit une forêt en ne conservant que les K variables les plus importantes de la forêt précédente, le paramètre K devant être choisi par l'utilisateur.

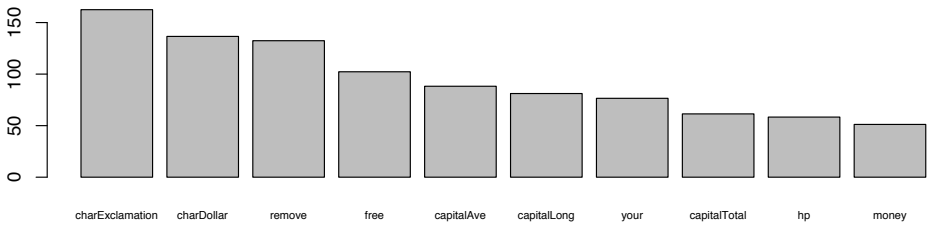


FIGURE 10.6 – Représentation des 10 variables les plus importantes pour `foret2`.

Pour aller plus loin

Pour de gros volumes de données, le package ranger est très efficace en temps de calcul et d'utilisation similaire à celle de randomForest. La sélection de variables est aussi possible avec le package `VSURF`. Le package grf pour generalized random forests propose de nombreuses généralisations des forêts pour effectuer par exemple des régressions quantiles et de l'inférence causale. Par ailleurs, il utilise des « forêts honnêtes » où un échantillon est utilisé pour choisir les coupures et un autre pour faire les prédictions. Il fournit aussi des intervalles de confiances pour les valeurs prévues. Pour trouver des éléments complémentaires sur les forêts aléatoires, on pourra se référer à Hastie *et al.* (2009).

10.3 Régression sous contraintes

Objet

Les méthodes de régression (linéaire, logistique, etc.) sont très utilisées en pratique. Cependant, lorsque le nombre de variables explicatives est élevé voire plus grand que le nombre d'individus, ou lorsqu'il existe de fortes corrélations entre ces variables explicatives, des problèmes d'estimation apparaissent. Une idée consiste alors à forcer les solutions à vivre dans un espace plus petit afin de diminuer la variance des estimateurs. Cet espace plus petit, on dit aussi contraint, est obtenu par minimisation du problème initial sous contrainte de norme. La contrainte d'appartenance à l'espace est donnée par une fonction de régularisation pénalisant les solutions ayant de grandes normes.

On considère que cette fonction de régularisation $J(.)$ est convexe à valeurs positives et que λ est un réel positif. Pour un problème de régression (Y quantitative), la régression sous contraintes consiste à chercher le vecteur de paramètres β qui minimise le critère des moindres carrés pénalisés comme suit :

$$\frac{1}{2n}\sum_{i=1}^{n}(y_i - x_i^t\beta)^2 + \lambda J(\beta).$$

Lorsque la variable à expliquer Y est binaire à valeurs dans $\{0,1\}$, on remplace le critère des moindres carrés par l'opposé de la log-vraisemblance du modèle logistique :

$$-\frac{1}{n}\sum_{i=1}^{n}(y_i x_i^t\beta - \log(1+\exp(x_i^t\beta))) + \lambda J(\beta). \tag{10.1}$$

L'utilisateur doit choisir la fonction de régularisation $J(\beta)$ et le réel positif ou nul λ. On utilise souvent comme fonction de régularisation

- la norme euclidienne au carré (à un facteur 0.5 près) : $J(\beta) = \frac{1}{2}\|\beta\|_2^2 = \sum_{j=1}^{p}\beta_j^2$ qui correspond à la régression ridge ;
- la norme 1 : $J(\beta) = \|\beta\|_1 = \sum_{j=1}^{p}|\beta_j|$ qui correspond à la régression lasso.

En général, le coefficient constant (intercept) n'est pas inclu dans la contrainte.

Le paramètre λ assure la balance entre la régularité de la solution (faible valeur de $J(\beta)$) et son adéquation au problème initial posé (faible valeur du critère des moindres carrés ou forte valeur de vraisemblance). Le choix de ce paramètre se révèle crucial sur la performance de la méthode. Pour $\lambda = 0$, on retrouve les estimateurs classiques des moindres carrés du modèle de régression linéaire et du maximum de vraisemblance du modèle logistique. Lorsque λ augmente, le poids de la pénalité (et donc de la contrainte) augmente et on obtiendra une solution qui possèdera une plus petite norme que ces estimateurs traditionnels. Ainsi dans le cas limite où $\lambda = \infty$, la solution pour $\hat{\beta}$ est le vecteur nul. Pour résumer, quand λ augmente la norme de $\hat{\beta}$ diminue et on dit que les coefficients « rétrécissent ».

Il est intéressant de dessiner ce rétrécissement en fonction de λ, on parle alors de « chemin de régularisation ». L'approche classique pour sélectionner λ consiste à estimer un critère de performance pour différentes valeurs de λ et à choisir la valeur qui optimise ce critère.

Les solutions des problèmes d'optimisation s'obtiennent de manière numérique à l'aide d'algorithmes de type programmation non linéaires. Le package glmnet utilise par exemple une méthode de descente par coordonnées pour calculer les estimateurs ridge et lasso. C'est ce package que nous allons utiliser dans cette fiche, en nous focalisant sur la régression logistique sous contraintes.

Remarquons enfin que la valeur des paramètres dépend de l'unité de mesure de la variable associée : il est donc d'usage de réduire les variables pour qu'elles aient la même norme. Cette stratégie est effectuée par défaut dans glmnet.

Exemple

Nous prenons l'exemple de la détection automatique de spams (cf. page 321).

Étapes

1. Importer les données
2. Construire les modèles lasso et ridge
3. Sélectionner le paramètre λ
4. Faire de la prévision
5. Estimer les performances d'une régression sous contraintes

Traitement de l'exemple

1. Importer les données

Le jeu de données se trouve dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

On sépare l'échantillon en un échantillon d'apprentissage de taille 3000 et un échantillon de validation de taille 1601. L'échantillon d'apprentissage sera utilisé pour construire les algorithmes de régression sous contraintes et sélectionner leurs paramètres, celui de validation pour estimer leurs performances.

```
> set.seed(5678)
> perm <- sample(4601,3000)
> app <- spam[perm,]
> valid <- spam[-perm,]
```

2. Construire les modèles lasso et ridge
La fonction **glmnet** du package `glmnet` a une syntaxe légèrement différente des autres fonctions de régression ou de discrimination. Il faut préciser la matrice des variables explicatives et le vecteur à expliquer. Comme pour la régression logistique, il faut spécifier la famille : ici `binomial`. Un paramètre α précise la norme considérée : si $\alpha = 1$ c'est la norme 1, donc une régression lasso ; si $\alpha = 0$ c'est la norme euclidienne (ou norme 2), donc une régression ridge. Il est possible de choisir α entre 0 et 1, ce qui conduit à une combinaison des normes 1 et 2 : cette méthode est appelée **elastic net** (voir section « Pour aller plus loin »). Considérons tous les arguments par défaut ($\alpha = 1$, régression lasso, variables X réduites, intercept ajouté et non contraint) :

```
> library(glmnet)
> lasso <- glmnet(as.matrix(app[,1:57]),app[,58],family="binomial")
```

Le premier argument de **glmnet** correspond aux variables explicatives et doit être un objet de classe `matrix`. Lorsque certaines variables explicatives sont qualitatives, il est nécessaire de les recoder en numérique 0/1. On pourra utiliser la fonction **model.matrix** pour effectuer ce codage rapidement.

Pour la régression ridge, il suffira d'ajouter `alpha=0` :

```
> ridge <- glmnet(as.matrix(app[,1:57]),app[,58],family="binomial",alpha=0)
```

La fonction **glmnet** choisit automatiquement une grille de λ, grille qu'il est possible de spécifier par l'argument `lambda`, et retourne en sortie les coefficients $\hat{\beta}_\lambda$ pour chaque valeur de λ. La fonction **plot.glmnet**, que l'on peut appeler par le raccourci **plot**, dessine le chemin de régularisation des estimateurs lasso et ridge (voir Fig. 10.7) :

```
> par(mfrow=c(1,2))
> plot(lasso)
> plot(ridge)
```

Par défaut, **plot.glmnet** met en abscisses la norme des $\hat{\beta}_\lambda$ pour chaque λ, et non les valeurs de λ. En ordonnées, elle donne les valeurs des coefficients de la régression. Ainsi si la contrainte est grande (λ grand), le critère (10.1) est minimal pour une norme de $\hat{\beta}_\lambda$ proche de zéro, i.e. tous les coefficients valent zéro (point gauche des graphiques). Par contre quand la contrainte est petite, le terme dominant du problème est le terme d'ajustement et la norme de $\hat{\beta}_\lambda$ peut être grande. Si on souhaite visualiser la valeur des coefficients $\hat{\beta}_\lambda$ en fonction de λ, il faudra ajouter l'argument `xvar="lambda"` dans la fonction **plot**.

Le choix de λ est donc crucial. Il est intéressant de voir, que pour le lasso, les coefficients « quittent la valeur 0 » les uns après les autres au fur et à mesure que la norme de $\hat{\beta}_\lambda$ augmente. Ce n'est pas le cas pour ridge où tous les coefficients

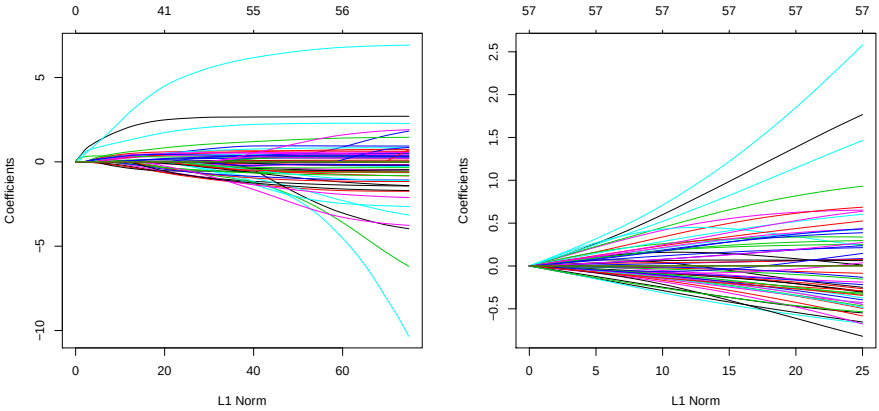


FIGURE 10.7 – Chemin de régularisation pour la régression lasso (à gauche) et la régression ridge (à droite).

deviennent très rapidement non nuls. Ainsi, le lasso aura tendance à conserver un certain nombre de coefficients nuls lorsque λ n'est pas trop petit. On dit que la régression lasso permet de sélectionner des variables.

3. Sélectionner le paramètre λ

Une fois que l'utilisateur a choisi sa méthode (ridge, lasso, elasticnet), il doit déterminer le paramètre λ. Ce paramètre est souvent choisi en estimant un critère par validation croisée. Le package glmnet propose une fonction effectuant par défaut une procédure de validation croisée en 10 blocs. Pour le lasso, cela donne

```
> set.seed(1234)
> Llasso <- cv.glmnet(as.matrix(app[,1:57]),app[,58],family="binomial")
```

La fonction retourne, pour chaque valeur de λ testée :
- une erreur estimée par validation croisée (`cvm`) ainsi que l'écart-type (`cvsd`) et un intervalle de confiance (`cvlo` et `cvup`) associés à cette erreur ;
- le nombre de coefficients non nuls (`nzero`) ;
- la valeur de λ qui minimise l'erreur (`lambda.min`).

La valeur `lambda.min` minimise donc une erreur estimée. Pour prendre en compte la précision d'estimation de l'erreur, la fonction renvoie également une autre valeur `lambda.1se` qui correspond à la plus grande valeur de λ pour laquelle l'erreur se situe à plus un écart type de l'erreur en `lambda.min`. En pratique, cela signifie que l'utilisateur peut choisir n'importe quelle valeur entre `lambda.min` et `lambda.1se`. Si on privilégie la parcimonie du modèle (lorsqu'on fait du lasso par exemple), on choisira `lambda.1se`. C'est le choix qui est fait par défaut pour prédire (voir section suivante). On obtient ces deux valeurs de λ avec :

```
> Llasso$lambda.min
[1] 0.000430318
> Llasso$lambda.1se
[1] 0.003331796
```

Il est possible de visualiser toutes ces sorties à l'aide de la fonction **plot.cv.glmnet** (voir Fig. 10.8) :

```
> plot(Llasso)
```

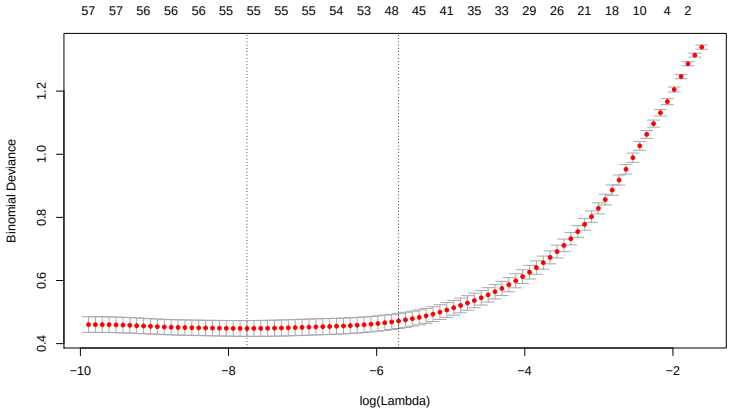


FIGURE 10.8 – Évolution du critère d'erreur en fonction de λ pour le lasso.

On remarque que deux lignes verticales sont représentées sur le graphe. Celle de gauche correspond à la valeur `lambda.min`, celle de droite à `lambda.1se`. Par défaut, l'erreur utilisée par **cv.glmnet** est la déviance. Il est possible de changer ce critère avec l'argument `type.measure`. On pourra également utilisé le taux de mal classés (`type.measure="class"`) ou l'AUC (`type.measure="auc"`). On effectue le même travail de sélection pour la régression ridge (voir Fig. 10.9 à gauche) :

```
> set.seed(1234)
> Lridge <- cv.glmnet(as.matrix(app[,1:57]),app[,58],family="binomial",
          alpha=0)
> plot(Lridge)
```

On voit sur la figure de gauche que l'erreur minimale est obtenue pour la plus petite valeur de la grille. Il est donc possible que la valeur optimale soit en dehors de la grille choisie par défaut par **cv.glmnet**. Il est donc nécessaire de refaire la validation croisée en spécifiant des valeurs de lambda plus petites que celles prises par défaut :

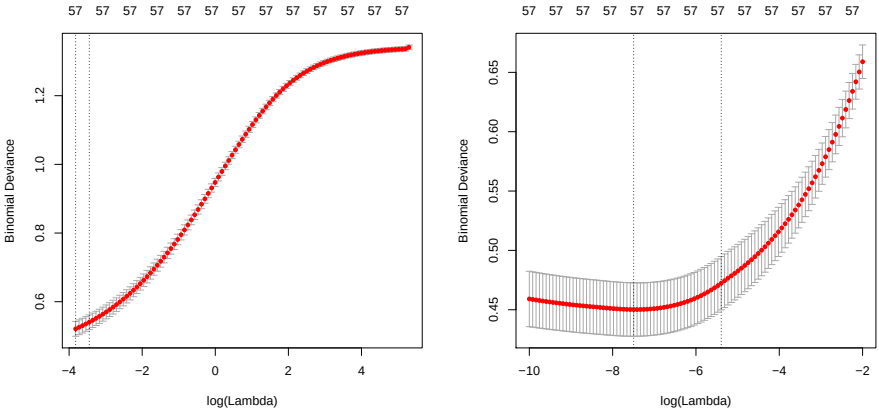


FIGURE 10.9 – Évolution du critère d'erreur en fonction de λ pour le ridge.

```
> set.seed(1234)
> Lridge1 <- cv.glmnet(as.matrix(app[,1:57]),app[,58],family="binomial",
        alpha=0,lambda=exp(seq(-10,-2,length=100)))
> plot(Lridge1)
```

Cette fois, le minimum a bien été atteint à l'intérieur de la grille (voir le graphe de droite de la Fig. 10.9) et nous pourrons donc utiliser cette procédure en choisissant `lambda.min` ou `lambda1.se`.

4. Faire de la prévision

La fonction **predict.glmnet**, que l'on peut appeler en utilisant le raccourci **predict**, permet de prédire le label d'un nouveau courriel. Il est également possible d'utiliser la fonction **predict** directement sur l'objet `cv.glmnet`, cela évite de recalculer un objet `glmnet` avec le paramètre optimal. Par défaut, cette fonction utilise la valeur `lambda.1se`, mais on peut aussi choisir le `lambda.min` via l'argument `s="lambda.min"`. Pour les estimateurs lasso et ridge sélectionnés dans la section précédente, on obtient ainsi les prévisions pour les individus de l'échantillon de validation avec :

```
> prev.class.lasso <- predict(Llasso,newx=as.matrix(valid[,1:57]),
        type="class")
> prev.class.ridge <- predict(Lridge1,newx=as.matrix(valid[,1:57]),
        type="class")
> prev.class <- data.frame(Lasso=as.character(prev.class.lasso),
        Ridge=as.character(prev.class.ridge),obs=valid$type)
> head(prev.class)
  Lasso Ridge  obs
1  spam  spam spam
```

```
2  spam  spam spam
3  spam  spam spam
4  spam  spam spam
5  spam  spam spam
6  spam  spam spam
```

On observe que les 6 premiers individus de l'échantillon de validation sont prédits comme des spams par les deux méthodes. On peut également estimer la probabilité qu'un nouveau courriel soit un spam en ajoutant l'argument `type="response"` :

```
> prev.lasso <- predict(Llasso,newx=as.matrix(valid[,1:57]),type="response")
> prev.ridge <- predict(Lridge1,newx=as.matrix(valid[,1:57]),type="response")
> prev.prob <- data.frame(Lasso=as.numeric(prev.lasso),
        Ridge=as.numeric(prev.ridge),obs=valid$type)
> head(prev.prob)
      Lasso     Ridge  obs
1 0.9504035 0.9510156 spam
2 0.5909894 0.6392713 spam
3 0.9960009 0.9974515 spam
4 0.9020829 0.8843358 spam
5 0.9936782 0.9859536 spam
6 0.8985610 0.8686000 spam
```

La probabilité d'être un spam pour le premier courriel de l'échantillon de validation est estimée par le lasso à 0.950, celle pour le second à 0.591, etc.

5. Estimer les performances d'une régression sous contraintes

On souhaite comparer les performances des deux algorithmes en estimant leur taux de mal classés et leur courbe ROC sur les données de validation.

L'erreur de classification s'obtient en confrontant les valeurs prédites aux valeurs observées à partir du data-frame `prev.class` :

```
> library(tidyverse)
> prev.class %>% summarise_all(funs(err=mean(obs!=.)))
           %>% select(-obs_err) %>% round(3)
  Lasso_err Ridge_err
1     0.095     0.093
```

Les taux d'erreur sont relativement proches avec une légère préférence pour lasso. La courbe ROC peut s'obtenir avec le package pROC comme dans la fiche 10.2 sur les forêts aléatoires. Nous la traçons (Fig. 10.10) en `ggplot` à l'aide du package plotROC :

```
> library(plotROC)
> df.roc <- prev.prob %>% gather(key=Methode,value=score,ada,logit)
> ggplot(df.roc)+aes(d=obs,m=score,color=Methode)+
      geom_roc()+theme_classic()
```

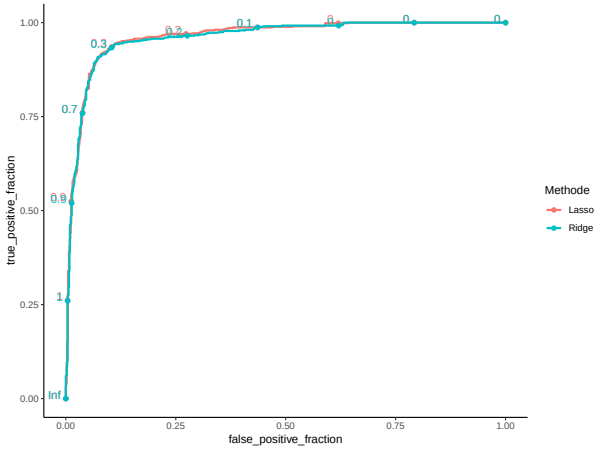


FIGURE 10.10 – Courbes ROC des algorithmes lasso et ridge.

On peut également obtenir les AUC avec

```
> library(pROC)
> df.roc %>% group_by(Methode) %>% summarize(AUC=pROC::auc(obs,score))
# A tibble: 2 x 2
  Methode   AUC
  <chr>   <dbl>
1 Lasso   0.962
2 Ridge   0.960
```

Les courbes ROC et les AUC sont très proches. Nous pouvons donc conclure que, pour ces critères, les deux méthodes sont équivalentes. Ce n'est pas surprenant puisque nous remarquons que les valeurs de λ sélectionnées sont très petites : par conséquent, les 2 modèles sont proches du modèle logistique classique.

Pour aller plus loin

Le package glmnet propose de mixer les pénalités ridge et lasso. Cela s'effectue en pondérant par α la pénalité lasso et $1-\alpha$ la pénalité ridge où $\alpha \in [0,1]$. Cette approche, appelée elastic net, est plus générale que les méthodes ridge et lasso. Le prix à payer est l'ajout d'un nouveau paramètre α qu'il faut calibrer. glmnet ne propose pas de fonction pour choisir ce paramètre, on pourra utiliser caret (voir fiche 10.1) pour sélectionner simultanément les paramètres α et λ.

Pour plus d'informations sur les régressions sous contraintes, on pourra se référer par exemple à Hastie *et al.* (2009) et Giraud (2015).

10.4 Gradient boosting

Objet

Les algorithmes de gradient boosting permettent de répondre à des problèmes de régression et de classification supervisée. On désigne par Y la variable à expliquer et par $X = (X_1, \ldots, X_p)$ le vecteur des variables explicatives. L'approche consiste à utiliser une descente de gradient pour minimiser le risque empirique

$$\mathcal{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f(X_i)),$$

où $\ell(y, f(x))$ mesure l'erreur entre la prévision $f(x)$ et l'observation y. La fonction $u \mapsto \ell(y, u)$ doit être différentiable et convexe. Les algorithmes de gradient boosting classiques sont :

- adaboost : il s'applique dans le cas de la classification binaire et correspond à la fonction de perte $\ell(y, f(x)) = \exp(-yf(x))$ avec $y \in \{-1, 1\}$;
- logitboost : il s'applique dans le cas de la classification binaire et correspond à la fonction de perte $\ell(y, f(x)) = \log(1 + \exp(-yf(x)))$ avec $y \in \{-1, 1\}$;
- L_2-boosting : il s'applique dans le cas de la régression et correspond à la fonction de perte $\ell(y, f(x)) = (y - f(x))^2$ avec $y \in \mathbb{R}$.

L'algorithme renvoie une suite récursive d'estimateurs $(f_m)_m$ telle que

$$f_m(x) = f_{m-1}(x) + \lambda h_m(x), \tag{10.2}$$

où $\lambda \in]0, 1[$ et h_m une règle faible. Le plus souvent, ces règles faibles sont des arbres avec très peu de coupures. L'utilisateur doit enfin sélectionner un estimateur (ou règle d'estimation) dans la suite $(f_m)_m$ en estimant la performance de chaque f_m par des méthodes de type validation croisée. En pratique, la suite est évaluée de $m = 1$ jusqu'à une valeur M à choisir.

Exemple

Nous prenons l'exemple de la détection automatique de spams présenté en introduction de ce chapitre (page 321).

Étapes

1. Importer les données
2. Construire et analyser un algorithme de gradient boosting
3. Sélectionner le nombre d'itérations
4. Faire de la prévision
5. Estimer les performances d'un algorithme de gradient boosting
6. Interpréter un algorithme de gradient boosting

Traitement de l'exemple

1. Importer les données
Le jeu de données se trouve dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

Pour la classification binaire, les algorithmes boosting présentés dans cette fiche requièrent que la variable à expliquer soit à valeurs dans $\{0, 1\}$. On recode donc d'abord les valeurs de la variables `type` :

```
> spam$type <- as.numeric(spam$type)-1
```

On sépare l'échantillon en un échantillon d'apprentissage de taille 3000 et un échantillon de validation de taille 1601. L'échantillon d'apprentissage sera utilisé pour construire les algorithmes boosting et sélectionner leurs paramètres, celui de validation pour estimer leurs performances.

```
> set.seed(5678)
> perm <- sample(4601,3000)
> app <- spam[perm,]
> valid <- spam[-perm,]
```

2. Construire et analyser un algorithme de gradient boosting
On souhaite ici utiliser la descente de gradient pour l'algorithme adaboost. Nous utilisons la fonction **gbm** du package gbm. La syntaxe de cette fonction est similaire aux autres fonctions de régression ou de discrimination : on utilise une formule pour indiquer la variable à expliquer et les variables explicatives.

```
> set.seed(1234)
> library(gbm)
> gbm(type~., data=app, distribution="adaboost", shrinkage=0.01,
    n.trees=3000)
gbm(formula = type ~ ., distribution = "adaboost", data = app,
    n.trees = 3000, shrinkage = 0.01)
A gradient boosted model with adaboost loss function.
3000 iterations were performed.
There were 57 predictors of which 36 had non-zero influence.
```

Nous fixons une graine à l'aide de **set.seed** avant d'appeler la fonction **gbm** car les arbres sont par défaut construits sur des échantillons bootstrap. On peut les construire sur l'échantillon `app` entier en ajoutant l'option `bag.fraction=1` dans la fonction **gbm**. L'argument `distribution="adaboost"` indique qu'on utilise la fonction de perte $\ell(y, f(x)) = \exp(-yf(x))$, `shrinkage` correspond au paramètre

λ dans (10.2) et `n.trees` désigne le nombre maximal M d'itérations dans la descente de gradient. Ces deux derniers paramètres sont généralement liés, le nombre d'itérations optimal augmente lorsque le paramètre `shrinkage` diminue. Il est souvent conseillé de prendre `shrinkage` plus petit que 0.1 et de choisir le nombre d'itérations comme nous allons le faire dans la partie suivante. On remarque que la sortie renseigne sur le nombre de variables explicatives pertinentes (36 sur 57) pour construire la suite de règles (voir section 6 pour obtenir ces variables).

3. Sélectionner le nombre d'itérations
L'objet `mod.ada` contient ainsi 3 000 règles de prévision $f_1, \ldots, f_{3000}$. Plusieurs méthodes peuvent être envisagées pour sélectionner la règle de prévision optimale. La fonction **gbm.perf** permet d'estimer l'erreur associée à la fonction de perte considérée par apprentissage/validation, validation croisée ou Out Of Bag. Nous proposons ici d'utiliser la validation croisée 5 blocs pour les algorithmes adaboost et logitboost dont les fonctions de perte sont présentées dans l'objet de la fiche. Pour ce faire, on calcule d'abord les suites d'estimateurs pour ces deux algorithmes en ajoutant l'argument `cv.folds=5` dans **gbm** :

```
> set.seed(1234)
> mod.ada <- gbm(type~.,data=app,distribution="adaboost",cv.folds=5,
         shrinkage=0.01,n.trees=3000)
> set.seed(567)
> mod.logit <- gbm(type~.,data=app,distribution="bernoulli",cv.folds=5,
         shrinkage=0.05,n.trees=3000)
```

Remarquons qu'on a utilisé `distribution="bernoulli"` dans la fonction **gbm** pour obtenir les estimateurs logitboost. On obtient les nombres d'itérations sélectionnés ainsi que les erreurs calculées en fonction du nombre d'itérations (voir Fig. 10.11) à l'aide de **gbm.perf** :

```
> Mopt.ada <- gbm.perf(mod.ada,method="cv")
> Mopt.ada
[1] 1740
> Mopt.logit <- gbm.perf(mod.logit,method="cv")
> Mopt.logit
[1] 1007
```

Le trait vertical bleu sur la figure 10.11 représente le nombre d'itérations qui minimise l'erreur calculée par validation croisée. Ici on choisira 1 740 itérations pour adaboost et 1 007 pour logitboost.

4. Faire de la prévision
La fonction **predict.gbm**, que l'on peut appeler en utilisant le raccourci **predict**, permet d'estimer la probabilité qu'un nouveau courriel soit un spam. On calcule ici ces estimations pour les courriels de l'échantillon de validation par les 2 algorithmes sélectionnés dans la section précédente.

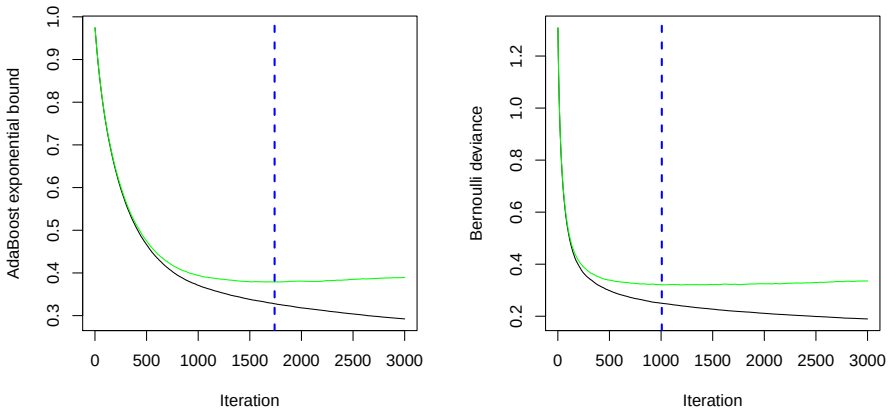


FIGURE 10.11 – Erreurs calculées sur l' échantillon d'apprentissage (noir) et par validation croisée (vert) pour adaboost (gauche) et logitboost (droite).

```
> prev.ada <- predict(mod.ada,newdata=valid,type="response",
        n.trees=Mopt.ada)
> head(round(prev.ada,3))
[1] 0.998 0.421 0.996 0.746 0.963 0.998
> prev.logit <- predict(mod.logit,newdata=valid,type="response",
        n.trees=Mopt.ada)
> head(round(prev.logit,3))
[1] 0.997 0.812 0.999 0.953 0.997 0.999
```

Adaboost prédit des probabilités élevées d'être un spam pour 4 des 5 premiers individus de l'échantillon de validation tandis que logitboost renvoie des probabilités élevées pour les 5.

5. Estimer les performances d'un algorithme de gradient boosting

On souhaite comparer les performances des deux algorithmes en estimant leur taux de mal classés et leur courbe ROC sur les données de validation. On collecte les probabilités estimées d'être un spam par les deux algorithmes pour les individus de l'échantillon de validation dans un même `data.frame` :

```
> prev.prob <- data.frame(ada=prev.ada,logit=prev.logit,obs=valid$type)
> head(round(prev.prob,3))
    ada logit obs
1 0.998 0.997   1
2 0.421 0.812   1
3 0.996 0.999   1
4 0.746 0.953   1
5 0.963 0.997   1
6 0.998 0.999   1
```

On peut en déduire une estimation de la classe en confrontant ces probabilités au seuil de 0.5 :

```
> prev.class <- round(prev.prob)
> head(prev.class)
  ada logit obs
1   1     1   1
2   0     1   1
3   1     1   1
4   1     1   1
5   1     1   1
6   1     1   1
```

On obtient l'erreur de classification estimée pour les 2 algorithmes en confrontant les valeurs prédites aux valeurs observées :

```
> library(tidyverse)
> prev.class %>% summarise_all(funs(err=mean(obs!=.)))
           %>% select(-obs_err) %>% round(3)
  ada_err logit_err
1   0.069     0.062
```

Les taux d'erreur sont relativement proches avec une légère préférence pour logitboost. La courbe ROC peut s'obtenir avec le package pROC comme dans la fiche 10.2 sur les forêts aléatoires. Nous proposons de la tracer ici en ggplot à l'aide du package plotROC.

```
> library(plotROC)
> df.roc <- prev.prob %>% gather(key=Methode,value=score,ada,logit)
> ggplot(df.roc)+aes(d=obs,m=score,color=Methode)+
      geom_roc()+theme_classic()
```

Les courbes ROC (voir Fig. 10.12) sont proches. On retrouve un léger mieux pour logitboost qui se confirme en calculant les AUC :

```
> library(pROC)
> df.roc %>% group_by(Methode) %>% summarize(AUC=pROC::auc(obs,score))
# A tibble: 2 x 2
  methode   AUC
  <chr>   <dbl>
1 ada     0.978
2 logit   0.981
```

6. Interpréter un algorithme de gradient boosting

On reproche souvent aux algorithmes de gradient boosting leur aspect « boîte noire » et leur manque d'interprétabilité. Il existe néanmoins des indicateurs qui

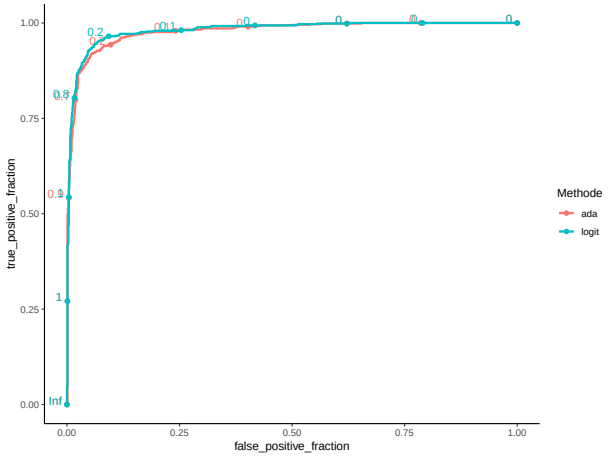


FIGURE 10.12 – Courbes ROC des algorithmes adaboost et logitboost.

permettent de mesurer l'influence des variables explicatives dans la construction de l'algorithme. On peut par exemple obtenir les 10 variables les plus influentes de l'algorithme logitboost avec la fonction **summary** :

```
> summary(mod.logit)[1:10,]
                            var   rel.inf
charExclamation charExclamation 22.107888
charDollar           charDollar 18.328872
remove                   remove 12.903802
free                       free  7.229789
your                       your  6.039477
hp                           hp  4.981167
capitalAve           capitalAve  4.909642
capitalLong         capitalLong  4.014214
report                   report  2.208803
our                         our  2.091343
```

Le point d'exclamation est le caractère le plus influent pour identifier les spam.

Pour aller plus loin

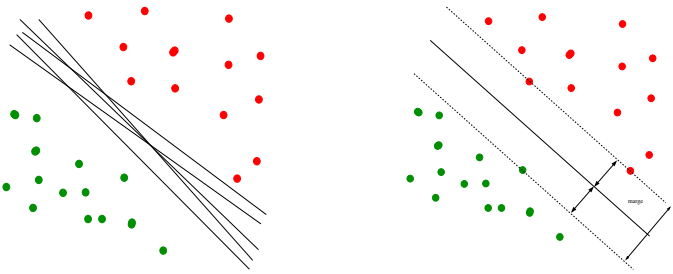
Pour plus d'information sur les algorithmes de gradient boosting, on pourra se référer par exemple à Hastie *et al.* (2009).

10.5 SVM

Objet

Les Support Vector Machines (SVM) sont une famille d'algorithmes dédiés aux problèmes de régression et de classification supervisée. Nous présenterons la démarche dans le cas de la classification supervisée avec une variable à expliquer Y binaire. On désigne par $X = (X_1, \ldots, X_p)$ le vecteur des variables explicatives.

Dans ce cadre, l'approche SVM consiste à trouver un hyperplan de l'espace des variables explicatives qui séparent « au mieux » les observations. Une illustration est proposée sur un exemple jouet sur les graphiques suivants (2 variables explicatives, le label Y est représenté par la couleur).



Les données sont ici linéairement séparables, on voit en effet sur le graphique de gauche qu'il existe une infinité d'hyperplans séparateurs. Les SVM consistent à trouver l'hyperplan qui maximise la distance de l'observation la plus proche de l'hyperplan (figure de droite). Cette distance minimale est appelée marge et les points qui se trouvent sur la marge sont appelés vecteurs supports.

Bien entendu, ce dessin représente un cas très particulier où les données sont séparables, ce qui n'est pas la situation générale. Dans le cas non séparable, on autorise des points à être dans la marge ou à être mal classés, c'est-à-dire du mauvais côté de l'hyperplan. Grosso modo, on affecte à ces points un coût lié à la distance à l'hyperplan séparateur multiplié par un paramètre d'échelle à calibrer. Le choix de ce paramètre, souvent noté C et appelé paramètre de coût, est crucial pour la performance de l'algorithme. Il existe des méthodes permettant de calibrer automatiquement ce paramètre.

Le succès des SVM dans les années 90 est lié d'une part à la facilité d'implémentation et d'autre part à la possibilité de transporter les données dans des espaces de grandes dimensions où on espère qu'elles seront plus séparables sans perte d'efficacité. Ce second point repose sur le fait que l'algorithme permettant d'obtenir l'hyperplan de marge maximale n'a besoin que des produits scalaires entre données pour fonctionner. Plus précisément, il est possible que les données ne soient pas linéairement séparables mais qu'elles le deviennent après leur avoir appliqué une transformation pertinente. Une telle transformation revient à plonger les données

dans un nouvel espace, souvent de plus grande dimension que l'espace original. Il s'avère qu'appliquer la SVM linéaire sur les données transformées revient à appliquer une SVM sur les données initiales en utilisant un noyau à la place du produit scalaire usuel. Cette astuce, dite astuce du noyau, incite à définir directement le noyau (le produit scalaire après transformation) et non la transformation explicite. Plusieurs noyaux existent et ce choix est également primordial. Nous utiliserons les noyaux suivants dans cette fiche :
– linéaire : $k(x, x') = \langle x, x' \rangle$;
– polynomial : $k(x, x') = (\texttt{scale}\ \langle x, x' \rangle + \texttt{offset})^{\texttt{degree}}$;
– gaussien ou radial : $k(x, x') = \exp(-\sigma \|x - x'\|)^2$.
Ici x et x' sont dans $\mathbb{R}^p$, $\langle x, x' \rangle$ est le produit scalaire usuel entre x et x', tandis que `scale`, `offset`, `degree` et σ sont des paramètres à calibrer.

Exemple

Nous prenons l'exemple de la détection automatique de spams présenté en introduction de ce chapitre (page 321).

Étapes

1. Importer les données
2. Construire et analyser un algorithme de SVM
3. Sélectionner les paramètres d'un SVM
4. Faire de la prévision
5. Estimer les performances de l'algorithme

Traitement de l'exemple

1. Importer les données

Le jeu de données se trouve dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

On sépare l'échantillon en un échantillon d'apprentissage de taille 3000 et un échantillon de validation de taille 1601. L'échantillon d'apprentissage sera utilisé pour construire les algorithmes de SVM et sélectionner leurs paramètres, celui de validation pour estimer les performances de ces algorithmes et vérifier qu'ils ne font pas de sur-ajustement.

```
> set.seed(5678)
> perm <- sample(4601,3000)
> app <- spam[perm,]
> valid <- spam[-perm,]
```

2. Construire et analyser un algorithme de SVM
Il existe plusieurs fonctions sur R permettant de faire des SVM. Les deux plus utilisées sont certainement la fonction **svm** du package e1071 et la fonction **ksvm** du package kernlab. Nous présentons dans cette fiche la fonction **ksvm** qui présente l'avantage d'être prise en compte dans le package caret (voir la fiche 10.1) pour calibrer les paramètres d'un SVM.
La syntaxe est similaire aux autres fonctions de régression ou de discrimination : on utilise une formule pour indiquer la variable à expliquer et les variables explicatives. Il faut faire attention aux paramètres : par défaut, la fonction **ksvm** transforme les données avec le noyau gaussien, et le paramètre de coût C vaut 1. Pour traiter les données initiales, c'est-à-dire non transformées ou non transportées dans un espace plus grand, il suffit de spécifier `kernel="vanilladot"` :

```
> mod.svm <- ksvm(type~.,data=app,kernel="vanilladot",C=1)
 Setting default kernel parameters
> mod.svm
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 1

Linear (vanilla) kernel function.

Number of Support Vectors : 599

Objective Function Value : -546.7826
Training error : 0.067
```

La sortie renseigne sur :
- le type de SVM ajusté : `C-svc (classification)`, puisque la variable à expliquer est ici qualitative ;
- le paramètre `cost`, qui correspond au paramètre de coût C, vaut 1 ;
- la transformation des données effectuées : ici le noyau est linéaire, les variables sont conservées en l'état ;
- le nombre de vecteurs supports (`Support Vectors`), qui est le nombre de données à conserver pour calculer la fonction de classification : ici il vaut 599, ce qui signifie qu'il suffit de mémoriser 599 exemples pour calculer la solution et non pas les 3000 exemples de la base d'apprentissage ;
- la valeur de la fonction optimisée pour obtenir les vecteurs supports à l'optimum (cette valeur est rarement utile) ;
- l'erreur de classification calculée sur les données d'apprentissage, ici 0.067.

3. Sélectionner les paramètres d'un SVM
Le SVM dépend de plusieurs paramètres, notamment le paramètre de coût C et le noyau, dont le choix se révèle crucial pour la performance de la méthode. La sélection de ces paramètres est la partie difficile de la mise en œuvre des SVM.

L'approche classique consiste à essayer plusieurs noyaux et à se donner une grille de valeurs pour les paramètres quantitatifs. Pour chaque jeu de paramètres et pour chaque noyau, on estime ensuite un critère (par exemple par validation croisée) et on choisit le jeu de paramètres qui optimise ce critère estimé. L'utilisateur doit choisir les valeurs de la grille, il faut en général faire quelques essais sur les données pour se donner les valeurs minimales et maximales de chaque grille de paramètres. Pour notre exemple, nous proposons de confronter 2 noyaux : le noyau polynomial et le noyau radial (le noyau linéaire étant un cas particulier du noyau polynomial). Les grilles pour chaque paramètre sont définies comme suit :

```
> C <- c(0.1,1,10,100)
> degree <- c(1,2,3)
> scale <- 1
> sigma <- c(0.0001,0.001,0.01,0.1,1)
```

Comme nous l'avons dit, l'utilisateur doit faire des choix. Nous choisissons ici de ne pas optimiser le paramètre `scale` du noyau polynomial et de le laisser à sa valeur par défaut 1. Ce paramètre n'est généralement pas le plus important. Pour chaque jeu de paramètres, nous allons estimer le taux de bon classement par validation croisée 3 blocs à l'aide du package `caret` (voir fiche 10.1). Nous commençons par le noyau polynomial :

```
> library(caret)
> gr.poly <- expand.grid(C=C,degree=degree,scale=scale)
> ctrl <- trainControl(method="cv",number=3)
> set.seed(123)
> sel.poly <- train(type~.,data=app,method="svmPoly",trControl=ctrl,
    tuneGrid=gr.poly)
> sel.poly
Support Vector Machines with Polynomial Kernel

3000 samples
  57 predictor
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling: Cross-Validated (3 fold)
Summary of sample sizes: 2000, 2000, 2000
Resampling results across tuning parameters:

  C      degree  Accuracy   Kappa
    0.1  1       0.9270000  0.8461040
    0.1  2       0.9100000  0.8105364
    0.1  3       0.9083333  0.8077676
    1.0  1       0.9280000  0.8479742
    1.0  2       0.8936667  0.7774023
```

```
    1.0  3       0.8920000  0.7745405
   10.0  1       0.9270000  0.8460247
   10.0  2       0.8853333  0.7608912
   10.0  3       0.8830000  0.7553775
  100.0  1       0.9240000  0.8395595
  100.0  2       0.8806667  0.7518021
  100.0  3       0.8763333  0.7413648

Tuning parameter 'scale' was held constant at a value of 1
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were degree = 1, scale = 1 and C = 1.
```

Les taux de bien classés estimés se trouvent dans la colonne `Accuracy`. On observe que ce taux est maximal pour `degree=1` et `C=1`. Pour le noyau radial on a :

```
> gr.radial <- expand.grid(C=C,sigma=sigma)
> set.seed(345)
> sel.radial <- train(type~.,data=app,method="svmRadial",
    trControl=ctrl,tuneGrid=gr.radial)
> sel.radial
Support Vector Machines with Radial Basis Function Kernel

3000 samples
  57 predictor
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling: Cross-Validated (3 fold)
Summary of sample sizes: 2000, 2000, 2000
Resampling results across tuning parameters:

  C      sigma  Accuracy   Kappa
    0.1  1e-04  0.6060000  0.0000000
    0.1  1e-03  0.7663333  0.4578780
    0.1  1e-02  0.8990000  0.7825103
    0.1  1e-01  0.7610000  0.4447726
    0.1  1e+00  0.6060000  0.0000000
    1.0  1e-04  0.7723333  0.4734765
    1.0  1e-03  0.9000000  0.7855698
    1.0  1e-02  0.9276667  0.8470048
    1.0  1e-01  0.9050000  0.7958078
    1.0  1e+00  0.7576667  0.4321881
   10.0  1e-04  0.8970000  0.7790195
   10.0  1e-03  0.9280000  0.8477454
   10.0  1e-02  0.9353333  0.8636455
   10.0  1e-01  0.9040000  0.7948708
   10.0  1e+00  0.7626667  0.4475051
```

```
  100.0  1e-04  0.9203333  0.8315819
  100.0  1e-03  0.9333333  0.8594359
  100.0  1e-02  0.9283333  0.8486409
  100.0  1e-01  0.8966667  0.7797074
  100.0  1e+00  0.7603333  0.4430619

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.01 and C = 10.
```

Ici les paramètres sélectionnés sont `sigma=0.01` et `C=10`. Le tableau 10.2 présente une synthèse de la procédure de sélection.

Noyau	Param. noyau	C	Accuracy
Polynomial	`degree=1`	1	0.928
Radial	`sigma=0.01`	10	0.935

TABLE 10.2 – Résumé de la procédure de sélection.

On remarque que les taux de bien classés sont proches pour les deux SVM sélectionnés. Si on devait en choisir un sur la base de ces résultats, nous privilégierions celui avec le noyau radial. On peut maintenant ajuster les SVM avec les paramètres optimaux pour les deux noyaux concurrents :

```
> mod.poly <- ksvm(type~.,data=app,kernel="polydot",
    kpar=list(degree=1,scale=1,offset=1),C=1,prob.model = TRUE)
> mod.radial <- ksvm(type~.,data=app,kernel="rbfdot",
    kpar=list(sigma=0.01),C=10,prob.model = TRUE)
```

On ajoute l'argument `prob.model = TRUE` afin de pouvoir obtenir les probabilités prédites qu'un courriel soit un spam pour de nouvelles observations.

4. Faire de la prévision

La fonction **predict.ksvm**, que l'on peut appeler avec le raccourci **predict**, permet de prédire le label d'un nouveau courriel. On peut obtenir les labels prédits pour les individus de l'échantillon de validation avec :

```
> prev.class.poly <- predict(mod.poly,newdata=valid)
> prev.class.radial <- predict(mod.radial,newdata=valid)
> prev.class.poly[1:10]
[1] spam   spam   spam   spam   spam   spam   spam   spam nonspam spam
Levels: nonspam spam
> prev.class.radial[1:10]
[1] spam   spam   spam   spam   spam   spam   spam   spam nonspam spam
Levels: nonspam spam
```

Parmi les 10 premiers messages de l'échantillon de validation, 9 sont considérés comme des spams par les deux SVM. La fonction **predict** permet d'estimer la

probabilité qu'un nouveau courriel soit un spam. On peut par exemple obtenir ces estimations pour les courriels de l'échantillon de validation avec les commandes :

```
> prev.prob.poly <- predict(mod.poly,newdata=valid, type="probabilities")
> prev.prob.radial <- predict(mod.radial,newdata=valid,
    type="probabilities")
> round(head(prev.prob.poly),3)
     nonspam  spam
[1,]   0.056 0.944
[2,]   0.412 0.588
[3,]   0.000 1.000
[4,]   0.172 0.828
[5,]   0.005 0.995
[6,]   0.066 0.934
```

La probabilité que le premier courriel de l'échantillon de validation soit un spam est estimée par le SVM polynomial à 0.944, celle pour le second à 0.588, etc. Par défaut, quand les probabilités sont supérieures à 0.5, le courriel est prédit comme spam.

5. Estimer les performances de l'algorithme
À partir des prévisions faites sur les données de validation dans la partie précédente, on peut obtenir une estimation de l'erreur de classification des deux SVM avec :

```
> library(tidyverse)
> prev.class <- data.frame(poly=prev.class.poly,
    radial=prev.class.radial,obs=valid$type)
> prev.class %>% summarise_all(funs(err=mean(obs!=.))) %>%
    select(-obs_err) %>% round(3)
  poly_err radial_err
1    0.082      0.077
```

Les taux d'erreur sont relativement proches avec une légère préférence pour le noyau radial. De plus, ces taux d'erreur sont du même ordre que ceux obtenus par validation croisée sur les données d'apprentissage (voir tableau 10.2). On peut donc considérer qu'il n'y a pas de sur-ajustement pour ces deux SVM.
D'autres indicateurs peuvent bien entendu être utilisés pour mesurer la performance, on peut par exemple comparer les courbes ROC. Cela peut s'effectuer avec le package pROC comme dans la fiche sur les forêts aléatoires 10.2. Nous proposons de le faire ici en `ggplot` à l'aide du package plotROC.

```
> library(plotROC)
> prev.prob <- data.frame(poly=prev.prob.poly[,2],
    radial=prev.prob.radial[,2],obs=valid$type)
> df.roc <- prev.prob %>% gather(key=Methode,value=score,poly,radial)
> ggplot(df.roc)+aes(d=obs,m=score,color=Methode)+geom_roc()+theme_classic()
```

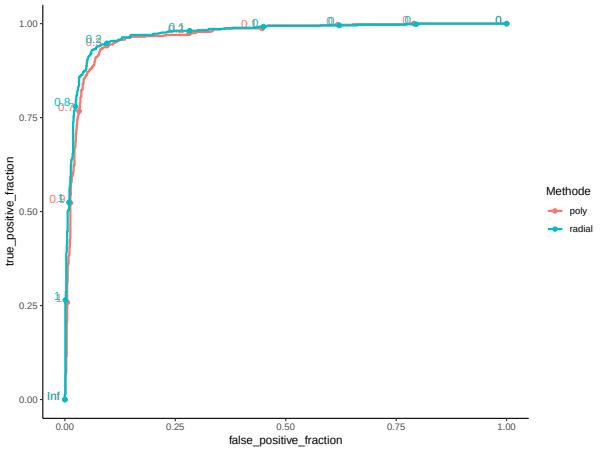


FIGURE 10.13 – Courbes ROC des SVM polynomial et radial.

Les courbes ROC (voir Fig. 10.13) sont proches. On retrouve un léger mieux pour le noyau radial, ce qui se confirme en calculant les AUC :

```
> library(pROC)
> df.roc %>% group_by(Methode) %>% summarize(AUC=pROC::auc(obs,score))
# A tibble: 2 x 2
  Methode   AUC
  <chr>   <dbl>
1 poly    0.966
2 radial  0.973
```

Pour aller plus loin

Les SVM ont été présentés dans un contexte de classification binaire. La méthode se généralise bien entendu aux cas de la régression (Y quantitative) et de la classification multiclasses (Y qualitative avec plus de deux modalités). La mise en œuvre se fait toujours à l'aide de la fonction **ksvm**.

Pour plus d'informations sur les SVM, on pourra se référer par exemple à Hastie *et al.* (2009).

10.6 Réseaux de neurones et deep learning

Objet

Les réseaux de neurones sont souvent considérés comme des boîtes noires aux résultats impressionnants, notamment en traitement d'images. Il est vrai que ces réseaux peuvent être vus comme des machines à prédire des valeurs ou des probabilités dont les paramètres sont appris grâce à des algorithmes efficaces d'optimisation de la vraisemblance (comme dans la plupart des méthodes d'apprentissage automatique). Ils s'appliquent donc à des problèmes de régression comme à des problèmes de classification supervisée.

L'objectif de cette fiche est de décrire les réseaux multicouches et de montrer comment ces réseaux peuvent être mis en œuvre dans R. Nous considérerons ici le cas de la prédiction d'une variable binaire.

Pour expliquer ce qu'est un réseau de neurones, commençons par définir un unique neurone. Un neurone prend en entrée une observation $x \in \mathbb{R}^p$, calcule le produit scalaire entre x et un vecteur (de dimension p) de poids w et ajoute un biais b, enfin cette réponse (un réel) est transformée par une fonction d'activation Φ :

$$x \mapsto \Phi(< x, w > +b).$$

Si la fonction Φ est l'identité, on retrouve un modèle linéaire. Si la fonction Φ est la sigmoïde, $\phi(z) = \frac{e^z}{1+e^z}$, on retrouve un modèle logistique. Il faut bien sûr choisir le poids et le biais, i.e. $p+1$ paramètres. Ceci se fait classiquement en utilisant un algorithme d'optimisation de type descente de gradient qui cherche à minimiser une erreur au carré ou un terme de log-vraisemblance négative.

Les réseaux de neurones sont obtenus en combinant plusieurs neurones similaires à celui du paragraphe précédent. Pour débuter, dans le cas de la classification supervisée, on associe un neurone à chacune des classes. Dans notre cas de classification binaire, on a ainsi deux neurones en sortie, on parle d'une couche de sortie à deux neurones :

$$x \mapsto \Phi(< x, w_1 > +b_1, < x, w_2 > +b_2)$$

Pour que le vecteur de sortie corresponde à des probabilités pour chacune des classes, la fonction d'activation utilisée n'est ni l'identité (qui renvoie des valeurs dans $\mathbb{R}$) ni la sigmoïde (qui renvoie des valeurs dans $[0,1]$) mais une fonction appelée softmax combinant simultanément les deux sorties par la formule :

$$\Phi(z_1, z_2) = \big(\exp z_1/(\exp z_1 + \exp z_2), \exp z_2/(\exp z_1 + \exp z_2)\big),$$

qui renvoie des valeurs dans $[0,1] \times [0,1]$ sommant à 1 donc des probabilités pour chaque classe. Il faut bien sûr apprendre ces poids w_1 et w_2 et ces biais b_1 et b_2, i.e. $2p+2$ paramètres, par un algorithme d'optimisation de la vraisemblance. Ce nombre de paramètres représente déjà le double du nombre de paramètres de

la régression logistique. Il s'avère que les solutions obtenues par les algorithmes d'optimisation utilisés dans ce cadre donnent malgré tout de très bonnes prévisions.

Afin d'avoir plus de choix sur la relation entre la variable d'entrée x et les probabilités en sortie, on peut rendre la fonction plus complexe en interprétant les valeurs sur la couche de sortie comme une nouvelle représentation de la variable d'entrée et en utilisant ce vecteur en entrée d'un autre algorithme d'apprentissage, par exemple un neurone ou une couche de neurones. Dans ce cas, la couche précédemment en sortie devient une couche interne, dite couche cachée, et on obtient une seconde couche en sortie. On commence alors à pouvoir faire preuve de créativité en augmentant le nombre de neurones dans la couche cachée ou en changeant la fonction d'activation de la couche interne, par exemple en utilisant l'identité $\Phi(z) = z$, la sigmoïde $\Phi(z) = \exp z/(1 + \exp z)$, la tangente hyperbolique, historiquement la plus utilisée, $\phi(z) = (\exp z - \exp -z)/(\exp z + \exp -z)$ ou la relu (rectified linear unit), actuellement la plus utilisée, $\Phi(z) = \max\{z, 0\}$. On obtient alors un modèle paramétrique avec un poids et un biais pour chacun des neurones utilisés. Il reste ensuite à apprendre les paramètres de toutes ces couches avec un algorithme de descente de gradient dans lequel le gradient est calculé de manière efficace pour cette structure en couches.

Dans cette fiche, nous nous contenterons des réseaux de neurones comportant au plus une couche cachée. Rien n'empêche bien sûr d'en rajouter d'autres pour obtenir une architecture profonde, *deep architecture* en anglais, d'où le nom de deep learning caractérisant les réseaux de neurones. Lorsqu'on augmente le nombre de neurones dans les couches ou le nombre de couches, i.e. la profondeur du réseau, la relation entre l'entrée et la sortie peut devenir très complexe. Dans le même temps, comme toujours en apprentissage automatique, il existe un fort risque de sur-apprentissage lorsque les modèles sont trop complexes par rapport au volume de données. Il s'agira donc de trouver une bonne architecture (nombre de neurones et profondeur) donnant de bonne garantie de performance, souvent en choisissant un réseau minimisant une erreur de type validation croisée.

Les succès actuels des réseaux profonds ou plus généralement du *deep learning* résident dans l'utilisation de nombreuses couches avec bien souvent une surparamétrisation très forte ainsi que dans le partage de poids à l'intérieur de celles-ci (par exemple dans les réseaux de convolution). Bien que cela soit contre intuitif, car le problème d'optimisation semble impossible et le risque de sur-apprentissage évident, les algorithmes d'optimisation utilisés dans ce cadre fournissent, si l'on fait bien le travail de validation croisée, des poids donnant des réseaux très efficaces en pratique.

En ce qui concerne les différents packages nécessaires à l'implémentation de la méthode, l'installation dépend de votre plate-forme. Il vous faut tout d'abord installer le logiciel python. Ensuite, vous devrez installer les modules Python de `tensorflow` et `keras`. Ceci peut se faire facilement à l'aide du package keras. Ce package permet en effet d'installer les deux modules à l'aide d'une commande :

```
> library(keras)
> install_keras()
```

Ces installations ne sont à faire qu'une seule fois sur votre ordinateur.

Enfin, keras n'a pas été pensé pour produire des résultats reproductibles. Si vous souhaitez qu'ils le soient, nous proposons de fixer une graine mais, dans ce cas, keras déconnecte les GPU et vous ne travaillez plus que sur un seul cœur.

Exemple

Nous prenons l'exemple de la détection automatique de spams présenté en introduction du chapitre 10 (page 321).

Étapes

1. Importer les données
2. Construire un réseau
3. Faire de la prévision
4. Sélectionner les caractéristiques d'un réseau
5. Estimer les performances du réseau

Traitement de l'exemple en une couche

1. Importer les données

Le jeu de données se trouve dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

Pour être compatible avec keras, il faut coder la variable qualitative sous forme disjonctive et mettre X sous forme matricielle

```
> library(keras)
> spamX <- as.matrix(spam[,-58])
> spamY <- to_categorical(as.numeric(spam$type)-1, 2)
```

Enfin, nous allons séparer les données en ensemble d'apprentissage et en ensemble de validation

```
> set.seed(5678)
> perm <- sample(4601,3000)
> appX <- spamX[perm,]
> appY <- spamY[perm,]
```

```
> validX <- spamX[-perm,]
> validY <- spamY[-perm,]
```

2. Construire un réseau et optimiser les paramètres
Les réseaux séquentiels se construisent facilement avec keras en spécifiant les couches les unes après les autres. On peut ainsi spécifier un réseau à une couche en précisant le nombre de neurones dans la couche grâce au paramètre `unit` :

```
> use_session_with_seed(42)
> mod.1couche <- keras_model_sequential() %>%
    layer_dense(units=2, activation ="softmax")
```

Pour optimiser les paramètres, il faut spécifier la méthode puis entraîner sur un jeu de données

```
> mod.1couche %>% compile(loss = "categorical_crossentropy",
             optimizer=optimizer_rmsprop(),metrics=c("accuracy"))
> hist.1couche <- mod.1couche %>%
          fit(appX,appY,epochs=30,validation_split=0.2)
```

L'algorithme optimise ses paramètres par descente de gradient à partir d'un échantillon d'apprentissage de 80 % des données, laissant de côté un échantillon de validation de 20 %, ce qui est indiqué par `validation_split`. Le nombre d'itérations sur l'ensemble des données est défini par `epochs` : plus ce nombre est élevé plus le temps de calcul est long et plus vous risquez de sur-apprendre. keras est implémenté avec une logique objet : ceux-ci sont donc modifiés par les commandes et il n'y a pas besoin d'assigner les résultats des fonctions pour que les modifications soient enregistrées. C'est le cas ici avec les fonctions **layer_dense**, **compile** ou encore **fit** qui modifie le modèle stocké dans l'objet `mod.1couche`. Notons que **fit** retourne une valeur : l'historique des performances du modèle en fonction du nombre d'`epochs`, le nombre de passages sur toutes les données, dans l'algorithme de gradient. On peut visualiser cette évolution avec **plot**.

```
> plot(hist.1couche)
```

On observe que la performance du modèle s'améliore au début puis stagne, voir se dégrade légèrement. Le paramètre `epochs` devrait en toute rigueur être choisi par validation croisée. Ici, par souci de simplicité, nous l'avons fixé à 30.

On peut alors réestimer les paramètres du réseau à l'aide de toutes les données d'apprentissage :

```
> hist.1couche <- mod.1couche %>%
       fit(appX,appY,epochs=30,validation_split=0)
```

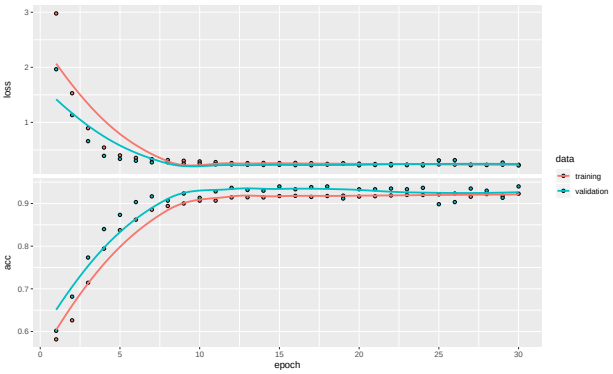


FIGURE 10.14 – Evolution des erreurs en fonction de l'epochs.

Pour construire un réseau à deux couches, il suffit d'insérer une couche supplémentaire

```
> mod.2couches <- keras_model_sequential() %>%
          layer_dense(units=30, activation ="relu") %>%
          layer_dense(units=2, activation ="softmax")
```

puis de l'optimiser

```
> mod.2couches %>% compile(loss = "categorical_crossentropy",
       optimizer=optimizer_rmsprop(),metrics=c("accuracy"))
> hist.2couches <- mod.2couches %>%
       fit(appX,appY,epochs=30,validation_split=0)
```

3. Faire de la prévision
On peut bien sûr utiliser ces modèles pour faire des prédictions avec la fonction **predict**.

```
> predict(mod.1couche, validX)[1:3,]
              [,1]      [,2]
 [1,] 1.236067e-02 0.9876394
 [2,] 1.891615e-01 0.8108385
 [3,] 4.253551e-06 0.9999957
> apply(predict(mod.1couche, validX), 1, which.max)[1:3]
 [1] 2 2 2
> predict(mod.2couches, validX)[1:3,]
            [,1]      [,2]
 [1,] 0.01000948 0.9899905
 [2,] 0.06321210 0.9367879
 [3,] 0.02073319 0.9792668
```

```
> apply(predict(mod.2couches, validX), 1, which.max)[1:3]
 [1] 2 2 2
```

On prédit ici que les trois premiers courriels de l'échantillon de validation sont des spams avec les deux réseaux.

4. Sélectionner les caractéristiques d'un réseau

keras permet d'obtenir des réseaux très complexes en combinant des couches de manière quasi arbitraire. La méthodologie usuelle pour sélectionner la meilleure *architecture* de réseau est la validation croisée sur une famille d'architectures (nombre de couches et nombre de neurones par couche) choisie manuellement. On considérera ici le cas des réseaux à deux couches dont le principal paramètre est le nombre de neurones dans la première couche dite couche cachée.

caret propose un modèle, `mlpKerasDecay`, permettant d'optimiser par validation croisée le choix des paramètres dans ce cas :

```
> library(caret)
> app <- spam[perm, ]; valid <- spam[-perm, ]
> param_grid <-  expand.grid(size = c(15, 30, 45),
                             lambda = 0, batch_size = 32, lr = 0.001,
                             rho = 0.9, decay = 0,
                             activation = c("relu", "tanh"))
> caret_mlp <- train(type ~ . , data = app, method = "mlpKerasDecay",
                     tuneGrid = param_grid, epoch = 30, verbose = 0,
                     trControl = trainControl(method = "cv", number = 5))
```

Notons qu'on a choisi de n'optimiser que deux paramètres : `size` le nombre de neurones sur la couche cachée et `activation` la fonction d'activation utilisée dans cette même couche. Les autres paramètres, `lambda` un facteur de régularisation, `batch_size` le nombre d'exemples utilisés à chaque étape de gradient, `lr` le taux d'apprentissage dans le même algorithme ainsi que `rho` et `decay`, deux paramètres de ce même algorithme, sont maintenus à leurs valeurs par défaut.

On peut alors obtenir le meilleur choix de paramètres

```
> caret_mlp
Multilayer Perceptron Network with Weight Decay

3000 samples
  57 predictor
   2 classes: 'nonspam', 'spam'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 2400, 2399, 2401, 2400, 2400
Resampling results across tuning parameters:
```

```
  size  activation  Accuracy   Kappa
  15    relu        0.9119899  0.8148062
  15    tanh        0.9366755  0.8673520
  30    relu        0.9233382  0.8406110
  30    tanh        0.9219816  0.8396988
  45    relu        0.9106543  0.8152101
  45    tanh        0.9356733  0.8656601

Tuning parameter 'lambda' was held constant at a value of 0
Tuning parameter 'rho' was held constant at a value of 0.9
Tuning parameter 'decay' was held constant at a value of 0
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were size = 15, lambda = 0,
 batch_size = 32, lr = 0.001, rho = 0.9, decay = 0 and activation = tanh.
```

La solution choisie comporte 15 neurones sur la couche cachée et utilise la fonction d'activation `tanh`. Elle peut désormais être utilisée pour faire des prédictions à partir d'un data-frame à l'aide de la commande **predict** :

```
> predict(caret_mlp, newdata = valid[1:3,])
[1] spam spam spam
Levels: nonspam spam
```

5. Estimer les performances du réseau
Le réseau a été choisi comme le meilleur réseau parmi ceux testés au sens de la plus petite erreur de validation croisée. Comme il s'agit d'un choix, il est commun que les performances soient légèrement surestimées. Pour corriger cet optimisme, il faut tester l'algorithme sur un jeu de données non encore utilisé, le jeu de données de validation :

```
> mean(predict(caret_mlp, newdata = valid)==valid[["type"]])
 [1] 0.9300437
```

Comme attendu, le taux de bien classés est légèrement plus faible que celui donné par la validation croisée. Il reste cependant bien du même ordre, ce qui conforte le choix.

Pour aller plus loin

Les réseaux proposés ici sont de la forme la plus classique celle des réseaux multi-couches entièrement connectés pour la prédiction. Il existe de nombreuses extensions de ces réseaux (réseaux convolutifs, réseaux récurrents...) et d'autres utilisations (apprentissage de représentation, synthèse...). On pourra consulter le site de keras pour R ou l'ouvrage *Deep Learning with R* (Chollet et Allaire, 2018).

Enfin, notons qu'il existe d'autres packages pour les réseaux de neurones dans R, par exemple mxnet ou tensorflow.

10.7 Comparaison de méthodes

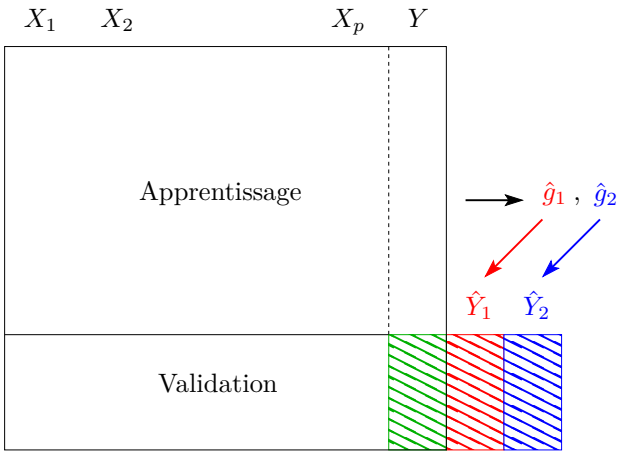
Objet

En classification supervisée comme en régression, l'utilisateur dispose en général d'une kyrielle de méthodes pour arriver à ses fins. Ainsi, dans les fiches précédentes, nous avons vu qu'il existe un grand nombre de techniques pour répondre au même problème de détection de spams. Chacune a ses spécificités et il est fort possible qu'une méthode fonctionne bien pour un jeu de données et moins bien pour un autre. Pour les utiliser à bon escient, le data scientist doit donc aussi être en mesure de comparer ces algorithmes.

Les techniques utilisées pour sélectionner une méthode sont proches de celles vues en fiche 10.1 pour calibrer un paramètre. L'idée est d'estimer un risque pour chaque stratégie et de choisir celle qui minimise ce risque estimé. Il faut cependant veiller à bien dissocier le problème du choix de paramètres de celui du choix de la méthode. Différentes approches peuvent être envisagées. Nous proposons ici la stratégie qui consiste à découper l'échantillon initial en :

- un échantillon d'apprentissage utilisé pour calibrer les paramètres de chaque méthode via les techniques vues dans les fiches précédentes : on déduira ainsi une règle de classification pour chaque méthode ;
- un échantillon de validation qui sera utilisé pour choisir la meilleure règle parmi celles construites à l'étape précédente.

Cette stratégie est illustrée ci-dessous dans le cas de deux méthodes :



Deux règles de prévision $\hat{g}_1, \hat{g}_2$ sont calculées à l'aide uniquement des données d'apprentissage. Les labels des individus de l'échantillon de validation sont ensuite prédits par chacune de ces règles. Les risques respectifs sont alors calculés en confrontant les valeurs prédites (rouge et bleu) aux valeurs observées (vert) : il ne reste plus qu'à choisir la règle ayant le risque minimal.

Le choix de la répartition apprentissage-validation est important, nous préconisons un découpage de l'ordre de 2/3 - 1/3 ou 80 % - 20 %, mais nous verrons dans la partie « Pour aller plus loin » que ce choix peut être évité. Par ailleurs, on peut utiliser n'importe quelle technique présentée dans les fiches précédentes pour calibrer chaque méthode sur les données d'apprentissage. Dans cette fiche, nous utiliserons la validation croisée 10 blocs. Ceci donne la procédure suivante :

1. Importer les données
2. Découper en une partie apprentissage et une partie validation
3. Optimiser les paramètres de l'algorithme avec les fonctions prévues du package ou avec le package caret
4. Comparer les méthodes en calculant le critère sur les données de validation
5. Proposer un modèle final

Enfin, nous remarquons que deux risques doivent être proposés : un pour calibrer les paramètres sur les données d'apprentissage (étape 3(a)), un autre pour comparer les méthodes sur les données de validation (étape 4). Nous conseillons d'utiliser le même risque. Il doit être déterminé par l'utilisateur en fonction du problème auquel il est confronté.

Exemple

Nous reprenons l'exemple de la détection automatique de spams présenté page 321. Nous allons comparer la méthode lasso (fiche 10.3) à celle des forêts aléatoires (10.2) en utilisant comme risque le taux de mal classés.

Traitement de l'exemple

1. Importer les données
Le jeu de données se trouve dans le package kernlab, on peut le charger par :

```
> library(kernlab)
> data(spam)
```

2. Découper les données
On coupe d'abord l'échantillon en deux : un échantillon d'apprentissage et un échantillon de validation :

```
> set.seed(1234)
> perm <- sample(4601,round(4601*.8))
> app <- spam[perm,]
> valid <- spam[-perm,]
```

3. Optimiser les paramètres de l'algorithme avec les fonctions prévues du package ou avec le package caret
Nous nous intéressons tout d'abord au lasso. Nous avons vu en fiche 10.3 que la fonction **cv.glmnet** permet de choisir le λ optimal par validation croisée 10-blocs (option par défaut de **cv.glmnet**). Le choix par défaut de la fonction de perte est la déviance mais nous choisissons le taux de mal classés en le précisant avec l'argument `type.measure="class"`. Nous obtenons ainsi le paramètre optimal comme suit :

```
> library(glmnet)
> set.seed(123)
> optlasso <- cv.glmnet(as.matrix(app[,-58]),app[,58],
    family="binomial", nfold=10, type.measure="class")
> optlasso$lambda.min
[1] 0.000302071
```

La fonction **cv.glmnet** retourne le modèle ré-estimé sur toutes les données de app en utilisant le paramètre optimal. On récupère ensuite les prévisions pour les courriels de l'échantillon de validation avec :

```
> prevlasso <- predict(optlasso,newx=as.matrix(valid[,-58]),
        type="class",s=c("lambda.min"))
```

Nous pouvons faire de même avec l'algorithme des forêts aléatoires en sélectionnant le nombre de variables `mtry` grâce au package caret de la fiche 10.1 (on parallélise avec doParallel pour accélérer les temps de calcul) :

```
> library(caret)
> ctrl <- trainControl(method="cv",number=10,classProbs=TRUE)
> library(doParallel)
> registerDoParallel(4)  # parallélisation sur 4 coeurs
> set.seed(123)
> sel.mtry <- train(type~.,data=app,method="rf",trControl=ctrl,
        tuneGrid=data.frame(mtry=seq(1,51,by=10)), type.measure="class")
> sel.mtry$bestTune
  mtry
2   11
```

Tout comme **cv.glmnet**, la fonction **train** ré-estime le modèle sur toutes les données de app en utilisant le paramètre optimal. On déduit les prévisions sur l'échantillon test :

```
> prevforet <- predict(sel.mtry, valid)
```

4. Comparer les méthodes en calculant le critère sur les données de validation
Nous pouvons maintenant calculer les erreurs de classement de chaque méthode sur les données de validation :

```
> prev.methode <- data.frame(lasso=as.vector(prevlasso),foret=prevforet,
        obs=valid$type)
> library(tidyverse)
> prev.methode %>% summarise_all(funs(err=mean(obs!=.))) %>%
        select(-obs_err) %>% round(3)
  lasso_err foret_err
1     0.051     0.041
```

Ainsi, pour cette étude, les forêts aléatoires donnent de meilleures prédictions (au sens du taux de mal classés) que la régression lasso.

5. Proposer un modèle final

En optimisant les paramètres de la forêt sur l'échantillon d'apprentissage, nous avons trouvé une erreur d'environ 4 %. La méthode étant choisie, il faut maintenant définir un modèle final. Pour ce faire, nous répliquons cette stratégie d'optimisation des paramètres (ici la validation croisée 10 blocs) sur l'échantillon entier et avons maintenant un modèle estimé sur les n données initiales qui est prêt à être appliqué sur de nouvelles données.

```
> ctrl <- trainControl(method="cv",number=10,classProbs=TRUE)
> set.seed(123)
> model_final <- train(type~.,data=spam,method="rf",trControl=ctrl,
        tuneGrid=data.frame(mtry=seq(1,51,by=10)), type.measure="class")
> stopCluster(cl)  # fermeture des clusters utilisés pour calcul parallèle
```

Pour aller plus loin

La méthodologie présentée ci-dessus est susceptible de dépendre trop fortement du découpage initial. Une variante consiste à répéter un certain nombre de fois, par exemple 20 fois, les étapes 2 et 3 en modifiant simplement le découpage initial, c'est-à-dire en supprimant la ligne `set.seed(1234)` en début d'étape 2. On effectue ensuite la moyenne des 20 taux d'erreurs ainsi obtenus pour en déduire un taux d'erreur global et on choisit la méthode (étape 4) ayant le plus faible taux d'erreur global. L'étape 5 reste inchangée.

Chapitre 11

Divers

Ce dernier chapitre regroupe plusieurs fiches sur des sujets variés mais importants et/ou en expansion. La première fiche (11.1) propose différentes méthodes pour imputer un tableau de données incomplet. Ces stratégies d'imputation permettent d'obtenir un tableau complet, lequel peut alors servir de base à toute autre méthode d'analyse statistique.

La fiche d'analyse textuelle (11.2), *text mining* en anglais, présente à la fois l'importation de données textuelles et leur mise en forme en un tableau quantitatif sur lequel des méthodes classiques de traitement de données permettront une analyse textuelle.

Enfin, la dernière fiche (11.3) présente la fouille de graphe, ou *graph mining* en anglais, c'est-à-dire l'ensemble des méthodes permettant de caractériser un graphe et d'en extraire des informations.

Dans chacune des fiches, on présente succinctement la méthode, puis un jeu de données et une problématique avant d'énumérer les principales étapes de l'analyse. L'exemple est alors traité via l'enchaînement des différentes instructions R et les résultats numériques et graphiques sont brièvement commentés.

Le site `https://r-stat-sc-donnees.github.io/` permet de retrouver les jeux de données. On peut les sauvegarder sur sa machine avant de les importer en R ou les importer en utilisant directement l'adresse URL.

11.1 Gestion de données manquantes

Objet

La gestion des données manquantes est un problème récurrent et crucial dans la pratique statistique. Pour savoir quelle stratégie adopter, l'analyse d'un tableau de données incomplet commence par l'étude du dispositif et du mécanisme des données manquantes.

Un certain nombre de fonctions, comme **lm** ou **glm**, suppriment par défaut les observations ayant au moins une valeur manquante avec l'argument par défaut `na.action=na.omit`. Cette pratique a l'avantage d'être simple et semble acceptable s'il y a très peu de valeurs manquantes et que celles-ci sont distribuées au hasard dans le tableau de données. Cependant, de meilleures alternatives sont toujours possibles.

On peut distinguer deux types d'approches lorsque le mécanisme de données manquantes est (complètement) aléatoire (Little et Rubin, 1987, 2002). Une première stratégie revient à estimer directement les paramètres d'un modèle en présence de données manquantes grâce à des algorithmes de type EM (Dempster *et al.*, 1977). Même si cette stratégie peut être parfaitement adaptée pour un problème donné, elle nécessite de définir un algorithme spécifique pour chaque modèle. Pour cette raison, on lui préfère souvent une seconde approche, à savoir l'imputation, qui consiste à compléter le jeu de données puis à appliquer n'importe quelle méthode sur ce jeu complété. Si l'objectif est de prévoir au mieux les valeurs manquantes, l'imputation par une seule valeur, ou imputation simple, peut être suffisante. En revanche, pour faire de l'inférence avec données manquantes, c'est-à-dire pour estimer des paramètres et leurs variances, il est nécessaire de pouvoir refléter l'incertitude sur les valeurs prévues. C'est ce que permet l'imputation multiple, qui consiste à générer M valeurs possibles pour chaque valeur manquante. On obtient ainsi M tableaux imputés sur lesquels on estime les paramètres θ_m du modèle d'analyse (régression par exemple) ainsi que leurs variances $V(\hat{\theta}_m)$. Les résultats sont ensuite combinés :

$$\hat{\theta} = \frac{1}{M}\sum_{m=1}^{M}\hat{\theta}_m$$

$$\widehat{V}(\hat{\theta}) = \frac{1}{M}\sum_{m}\widehat{V}(\hat{\theta}_m) + \left(1+\frac{1}{M}\right)\frac{1}{M-1}\sum_{m}\left(\hat{\theta}_m - \hat{\theta}\right)^2. \qquad (11.1)$$

Exemple

Nous considérons un jeu de données ozone incomplet qui a la même structure que celui de la fiche 9.2. Nous souhaitons analyser la relation entre le maximum journalier de la concentration en ozone (en $\mu g/m^3$) et la température à différentes heures de la journée, la nébulosité à différentes heures de la journée, la projection

du vent sur l'axe Est-Ouest à différentes heures de la journée et la concentration maximale de la veille du jour considéré. Nous disposons de 112 jours de mesures et 11 variables, 29 % des données étant manquantes (ce jeu de données a été obtenu en mettant des valeurs manquantes dans le jeu initialement complet).

L'objectif est ici de faire une régression avec données manquantes. Pour ce faire, après quelques analyses descriptives, nous appliquerons l'imputation multiple.

Étapes

1. Visualiser le dispositif de données manquantes
2. Imputation simple
3. Imputation multiple
4. Modélisation avec données manquantes

Traitement de l'exemple

1. Visualiser le dispositif de données manquantes

Supprimer des observations ayant des données manquantes est possible si le nombre d'observations complètes est suffisant et que les valeurs manquantes sont de type MCAR (missing completely at random), de sorte que le nouvel échantillon est un sous-échantillon des données d'origine. Dans ce cas, les estimateurs obtenus seront sans biais, mais avec une plus grande variance. Les méthodes qui admettent l'argument `na.action = na.omit`, comme la régression (fonction **lm**), se restreindraient ici à l'analyse de seulement 13 individus et la perte d'information est clairement trop importante :

```
> don <- read.table("ozoneNA.csv",header=TRUE,sep=";",row.names=1)
> nrow(na.omit(don))
[1] 13
```

Le package VIM permet de visualiser le dispositif de données manquantes. La fonction **aggr** représente le pourcentage de valeurs manquantes pour chaque variable (graphe de gauche Fig. 11.1) et donne les combinaisons de variables qui ont des données manquantes simultanément (graphe de droite Fig. 11.1).

```
> library(VIM)
> res<-summary(aggr(don,sortVar=TRUE))$combinations
 Variables sorted by number of missings:
 Variable      Count
     Ne12 0.37500000
       T9 0.33035714
      T15 0.33035714
      ... ...
```

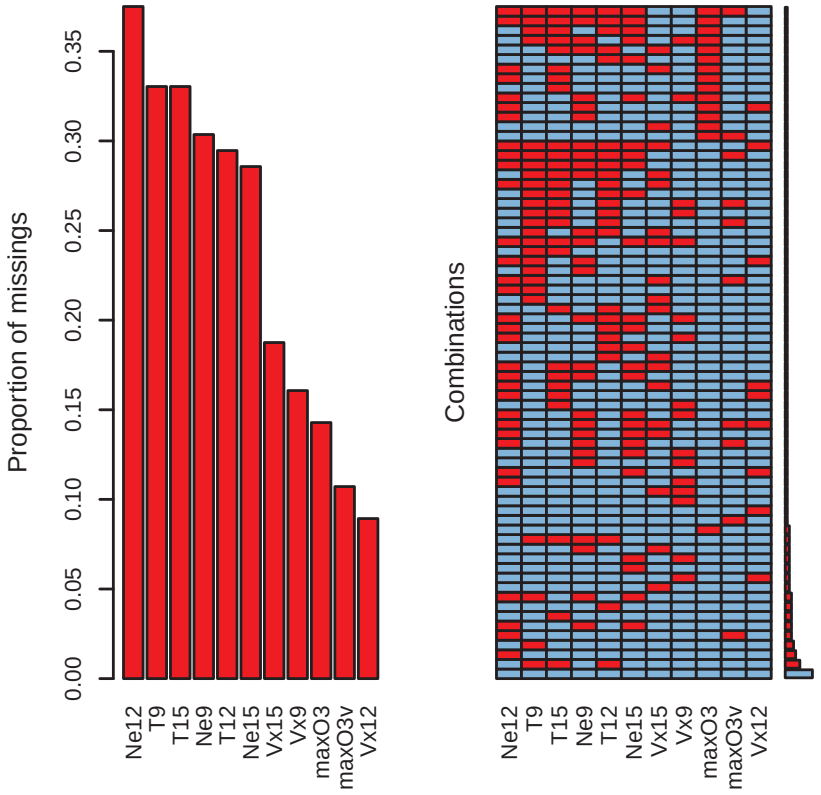


FIGURE 11.1 – Comptage des données manquantes avec le package VIM.

Le graphique de droite (Fig. 11.1) montre que la combinaison la plus fréquente est celle où toutes les variables sont observées : toutes les cases sont bleues, ligne du bas, ce qui comme on l'a vu ne concerne cependant que 13 jours. Ensuite, la deuxième combinaison la plus fréquente est celle où T9, T12 et T15 sont manquantes simultanément (cases en rouge), et ainsi de suite.

Le graphique de gauche de la figure 11.2, obtenu grâce à la fonction **matrixplot**, colorie les valeurs observées avec un niveau de gris : de blanc pour les faibles valeurs à noir pour les plus fortes, les valeurs manquantes étant indiquées en rouge. Ceci permet de voir si l'absence de données pour une variable dépend des valeurs d'une autre variable.

```
> matrixplot(don,sortby = 2)
> marginplot(don[,c("T9","maxO3")])
```

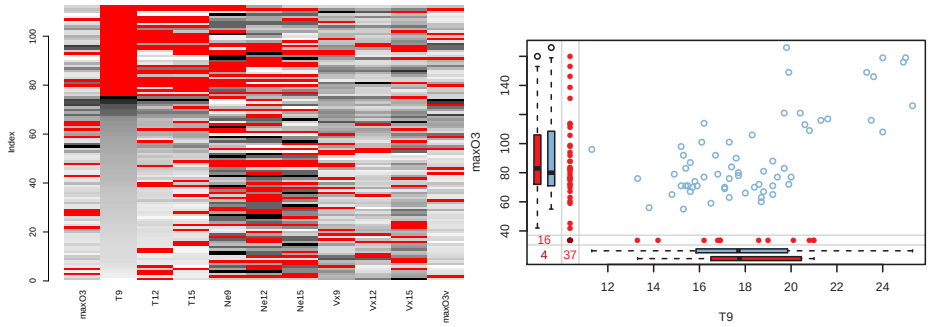


FIGURE 11.2 – À gauche : relation entre les valeurs de T9 et les valeurs des autres variables ; à droite : nuage de points et distribution de maxO3 en fonction de T9 avec et sans données manquantes.

Comme vu ci-dessus, les variables de température T9, T12 et T15 ont tendance à manquer simultanément : il y a plus de cases rouges pour T12 et T15 quand T9 est manquant. Cependant, quand T9 est manquant, il n'y a aucune variable qui a une surreprésentation de valeurs faibles (ou resp. de valeurs élevées), ce qui aurait suggéré des valeurs manquantes MAR : missing at random, dénomination trompeuse puisqu'elle correspond en fait au cas où la probabilité qu'une valeur soit manquante dépend des valeurs d'une autre variable. De même, pour les faibles valeurs de T9 en gris clair ou les fortes valeurs de T9 en gris foncé, il ne semble pas qu'il y ait une surreprésentation de données manquantes pour d'autres variables. Bref, tout indique ici des valeurs manquantes de type MCAR.

Le graphique de droite de la figure 11.2, obtenu avec la fonction **marginplot**, représente en bleu le nuage de points des données observées entre deux variables x et y. Les observations pour lesquelles il manque x (resp. y) sont représentées en rouge le long de l'axe des ordonnées (resp. des abscisses). De plus, deux boîtes à moustaches sont représentées pour la variables x sous l'axe des abscisses : l'une pour les valeurs de x quand y est manquant (en rouge) et l'autre pour les valeurs de x quand y est observée (en bleu). Ici, la distribution de T9 est à peu près la même que maxO3 soit observée ou manquante. La conclusion est la même pour l'axe des ordonnées. Ceci suggère à nouveau un mécanisme MCAR.

Les associations entre les valeurs manquantes (et les valeurs observées) de toutes les variables peuvent être étudiées globalement en utilisant l'analyse des correspondances multiples (cf. fiche 7.3). On crée un tableau de présence-absence avec « p » lorsque la valeur de la cellule est présente et « a » lorsqu'elle est absente. Ensuite, on effectue l'analyse des correspondances multiples avec la fonction **MCA** du package FactoMineR :

```
> tabNA <- matrix("p",nrow=nrow(don),ncol=ncol(don))
> tabNA[is.na(don)] <- "a"
> dimnames(tabNA) <- dimnames(don)
> library(FactoMineR)
> res.mca <- MCA(tabNA, graph=FALSE)
> plot(res.mca, invisible="ind")
```

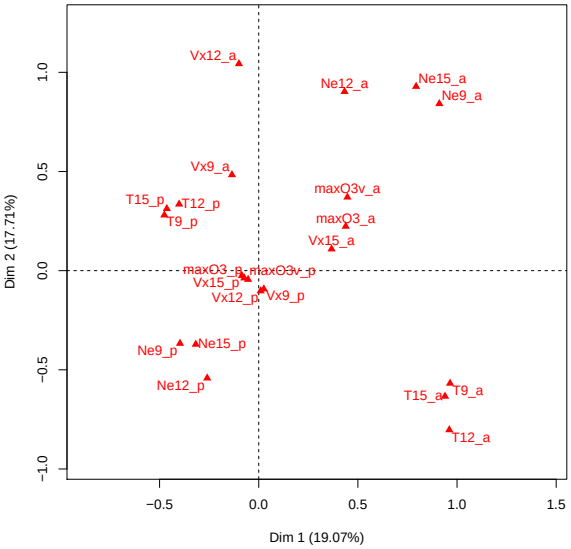


FIGURE 11.3 – Visualisation du dispositif de données manquantes par ACM.

L'ACM est un outil simple pour visualiser les associations de données manquantes même si le nombre de variables est important. Ici, par exemple, la proximité entre les modalités `Ne9_a`, `Ne12_a` et `Ne15_a` indique que des valeurs sont souvent manquantes sur toutes les données de nébulosité d'une même journée.

2. Imputation simple

Pour imputer par forêts aléatoires, on utilise la fonction **missForest** du package éponyme, laquelle a des arguments (`mtree`, `mtry`) proches de ceux de la fonction **randomForest**.

```
> library(missForest)
> missForest(don)
```

Pour imputer par ACP, on utilise le package missMDA et la fonction **imputePCA**. Celle-ci prend en argument le nombre de dimensions nécessaires pour imputer les données, nombre estimable par validation croisée via la fonction **estim_ncpPCA**. La fonction **imputePCA** retourne un tableau complété dans l'objet `completeObs`.

```
> library(missMDA)
> nb <- estim_ncpPCA(don)
> imp <- imputePCA(don,ncp=nb$ncp)
> imp$completeObs[1:3,1:8] # jeu imputé
         maxO3      T9      T12      T15 Ne9      Ne12     Ne15     Vx9
20010601    87 15.6000 18.50000 20.47146   4 4.000000 8.000000  0.6946
20010602    82 18.5047 20.86986 21.79932   5 5.000000 7.000000 -4.3301
20010603    92 15.3000 17.60000 19.50000   2 3.984066 3.812104  2.9544
```

Il est important de noter que les propriétés de l'imputation dépendent des qualités de la méthode. Ainsi, on peut espérer une meilleure imputation des données par ACP quand les liaisons entre variables sont linéaires. Les forêts aléatoires prendront mieux en compte des liaisons non linéaires, mais ne pourront pas extrapoler contrairement à une méthode linéaire.

Le package missMDA permet d'imputer des tableaux quantitatifs avec la fonction `imputePCA`, qualitatifs avec la fonction `imputeMCA`, mixtes avec la fonction `imputeFAMD` ou encore avec des groupes de variables avec la fonction **imputeMFA**. La fonction **missForest** permet d'imputer des tableaux de tout type.

3. Imputation multiple

L'imputation multiple consiste à générer plusieurs tableaux de données imputées pour refléter la variance de prédiction des valeurs manquantes. Les valeurs observées sont les mêmes dans tous les tableaux, mais les valeurs imputées sont différentes, ce qui permet d'appréhender l'incertitude liée à l'imputation.

Citons trois packages qui permettent de faire de l'imputation multiple : mice, Amelia et missMDA. Le nombre de tableaux générés est `m` pour les fonctions **mice** et **amelia** et `nboot` pour la fonction **MIPCA** du package missMDA.

```
> library(mice)
> imp.mice <- mice(don,m=100,defaultMethod="norm.boot")

> library(Amelia)
> res.amelia <- amelia(don,m=100)

> library(missMDA)
> nb <- estim_ncpPCA(don)
> impMI <- MIPCA(don,ncp=nb$ncp,nboot=100)
```

Nous pouvons alors inspecter les valeurs imputées pour savoir si la méthode d'imputation doit faire l'objet d'un examen plus approfondi ou si nous pouvons poursuivre et analyser les données. Une pratique courante consiste à comparer, pour chaque variable, la distribution des valeurs observées à celle des moyennes des valeurs imputées. Voir la figure 11.4 obtenue avec la fonction **compare.density**.

```
> compare.density(res.amelia, var = "T12")
```

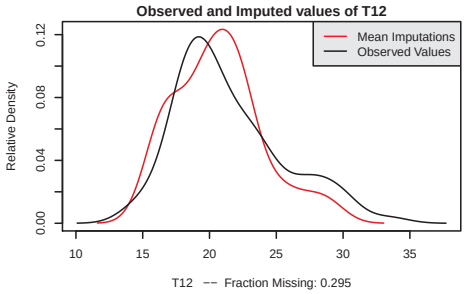


FIGURE 11.4 – Comparaison de la densité des valeurs observées et de celle des moyennes (sur m tableaux) des valeurs imputées.

Le package missMDA permet également d'examiner l'incertitude liée aux données manquantes, i.e. la variabilité des valeurs prédites, en projetant en supplémentaire les m tableaux de données imputées sur la configuration de l'ACP. Les graphiques de la figure 11.5 représentent la projection des individus (à gauche) et des variables (à droite) de chaque tableau imputé. Pour les individus, une zone de confiance est construite, et s'il n'y a pas de valeurs manquantes, il n'y a pas de zone de confiance. Les deux graphiques montrent que la variabilité entre les différentes imputations est faible et qu'on peut donc poursuivre l'analyse avec les tableaux imputés.

```
> plot(impMI, choice= "ind.supp")
> plot(impMI, choice= "var")
```

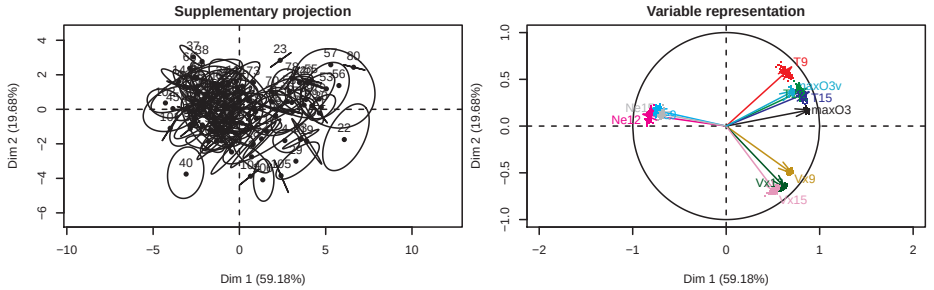


FIGURE 11.5 – À gauche : visualisation par ACP de l'incertitude autour de la position des individus ; à droite : visualisation par ACP de l'incertitude autour de la projection des variables.

L'imputation multiple est également possible pour les jeux de données qualitatifs ainsi que pour les jeux de données mixtes (i.e. contenant à la fois des variables quantitatives et qualitatives). Les fonctions **mice** et **amelia** gèrent directement la nature des variables, le package missMDA a des fonctions dédiées à chaque type de variables : **MIPCA** pour les variables quantitatives ou **MIMCA** pour les variables qualitatives.

4. Modélisation avec données manquantes

Le package mice permet d'appliquer un modèle statistique sur chacun des m tableaux imputés et de combiner automatiquement les résultats selon les règles de Rubin. Cette procédure est directement implémentée en lançant les lignes de code suivantes :

```
> set.seed(1234)
> imp <- mice(don, m=100, defaultMethod="norm.boot", printFlag=FALSE)
> res.lm.mice <- with(imp,
     lm(maxO3 ~ T9+T12+T15+Ne9+Ne12+Ne15+Vx9+Vx12+Vx15+maxO3v))
```

Si les tableaux imputés ont été obtenus par missMDA, il est nécessaire de transformer légèrement l'objet résultat :

```
> res.mi <- MIPCA(don, nboot=100)
> imp <- prelim(res.mi, don)
> res.lm <- with(imp,
     lm(maxO3 ~ T9+T12+T15+Ne9+Ne12+Ne15+Vx9+Vx12+Vx15+maxO3v))
```

Si les tableaux imputés ont été obtenus par Amelia, on utilisera :

```
> resDF <- lapply(res.amelia$imputations, as.data.frame)
> res.lm <- lapply(resDF, lm,
    formula="maxO3~ T9+T12+T15+Ne9+Ne12+Ne15+Vx9+Vx12+Vx15+maxO3v")
> res.lm <- as.mira(res.lm)
```

Il reste alors à concaténer les résultats avec la fonction **pool** de mice :

```
> pool.res <- pool(res.lm.mice)
> summary(pool.res, conf.int=TRUE)
             estimate std.error statistic    df p.value  2.5 % 97.5 %
(Intercept)    24.54     18.99      1.29 39.71    0.20 -13.86  62.93
T9             -0.23      2.56     -0.09 25.66    0.93  -5.49   5.03
T12             3.25      2.85      1.14 24.13    0.26  -2.62   9.12
...              ....      ....      .... .....    ....  .....   ....
maxO3v          0.32      0.11      3.06 46.19    0.00   0.11   0.53
> pool.res
             estimate   ubar      b      t dfcom    df  riv lambda  fmi
(Intercept)    24.54 164.14 194.61 360.69   101 39.71 1.20   0.54 0.57
```

```
T9              -0.23   1.94   4.55   6.54   101 25.66 2.37   0.70 0.72
T12              3.25   2.26   5.78   8.10   101 24.13 2.58   0.72 0.74
...              ....   ....   ....   ....   ... ..... ....   .... ....
maxO3v           0.32   0.01   0.01   0.01   101 46.19 0.92   0.48 0.50
```

Les cinq premières colonnes donnent les résultats classiques de la régression pour chaque paramètre : `estimate` pour l'estimation, `std.error` l'écart-type estimé, `statistic`, `df` et `p.value` donnent la statistique, les degrés de liberté et la probabilité critique associés au test de nullité du coefficient. Les bornes inférieure et supérieure de l'intervalle de confiance de niveau 95 % sont données quand `conf.int=TRUE`.
Le deuxième tableau détaille les calculs de l'imputation multiple : `ubar` correspond à la variance intra-imputation (premier terme de la somme dans l'équation (11.1)), `b` correspond à la variance inter-imputations (deuxième terme de la somme) et `t` correspond à la variance totale (somme des variances inter et intra), et est égale au carré de l'écart-type du précédent tableau. `lambda` est la proportion de la variance totale due aux données manquantes, i.e. la variabilité supplémentaire due aux données manquantes, c'est-à-dire le rapport entre la variance inter-imputations et la variance totale. Enfin `fmi` donne la fraction d'information manquante et `riv` donne l'augmentation relative de la variance (selon la terminologie de Little et Rubin).
L'imputation multiple avec des données qualitatives ou des données mixtes fonctionne mutatis mutandis de la même façon. On utilisera par exemple des modèles `glm` au lieu d'un modèle de régression.

Pour aller plus loin

Comme indiqué en introduction, il existe des alternatives à l'imputation. En effet, il est possible d'estimer directement les paramètres des modèles à partir d'un tableau incomplet en utilisant un algorithme EM. La fonction **em.norm** du package norm permet d'estimer un vecteur de moyennes et une matrice de variance-covariance à partir d'un jeu de données incomplet sous l'hypothèse que les données sont issues d'une distribution normale multivariée :

```
> library(norm)
> pre <- prelim.norm(as.matrix(don)) # manipulations préliminaires
> thetahat <- em.norm(pre) # estimation par MV
> getparam.norm(pre,thetahat)
```

L'inconvénient ici est que la variance des paramètres n'est pas calculée.

11.2 Analyse de texte

Objet

Les données se présentant sous la forme de texte sont très fréquentes en linguistique, mais également pour le traitement de questions ouvertes dans les questionnaires ou encore pour les données récupérées d'internet via du web scraping (cf. section 5.4), ou des packages dédiés comme twitteR, Rfacebook, etc.

Les données textuelles sont complexes et non structurées et, de fait, ne peuvent être utilisées directement. Une approche courante consiste à coder d'abord les textes en vecteurs, ce qui permet ensuite d'appliquer des méthodes de statistique et de machine learning comme la classification supervisée ou non supervisée, etc.

Le codage que nous allons utiliser, le sac de mots (bag of words en anglais), repose sur une idée simple : décrire un document par le vecteur des occurrences des mots utilisés dans celui-ci. On perd l'ordre des mots (d'où l'expression sac de mots) mais on obtient une représentation vectorielle de taille fixe si l'on fixe l'ensemble des mots considérés. On parle de vocabulaire ou de dictionnaire de mots. Nous verrons qu'il existe des outils simples en R pour faire cette transformation.

Remarque
Cette méthodologie de codage est la plus classique mais il existe d'autres approches passant par le codage des mots par un vecteur (word2vec, glove...).

Une fois ce codage effectué, on peut utiliser ce vecteur (soit directement, soit en le transformant en proportions pour s'abstraire de la taille) pour caractériser les textes et appliquer des méthodes classiques sur la matrice de données documents $\times$ mots.

L'objectif de cette fiche est de voir comment créer la matrice documents $\times$ mots à partir de textes puis d'appliquer des techniques de classification non supervisée.

Exemple

On se propose d'étudier deux œuvres de Victor Hugo (*Notre-Dame de Paris* et les 5 tomes des *Misérables*) et de voir si le style diffère suffisamment pour que l'on puisse distinguer de manière automatique l'origine de blocs de 100 lignes.

Étapes

1. Importation des textes et nettoyage
2. Codage en terme de sac de mots
3. Analyse des correspondances et visualisation
4. Classification par l'algorithme des k-means

Traitement de l'exemple

On utilisera les packages tidyverse, tidytext, gutenbergr, wordcloud, FactomineR et janitor. N'oubliez pas de les installer.

1. Importation des textes et nettoyage
Les textes "NotreDameDeParis", "LesMiserables1", "LesMiserables2", "LesMiserables3", "LesMiserables4", "LesMiserables5" sont disponibles sur le site web du livre sous la forme de fichiers textes, au format UTF-8, suivi de la licence du site Gutenberg (`www.gutenberg.org`) où ils ont été chargés. Classiquement, la lecture d'un texte se fait à l'aide de la commande **ReadLines**. Cependant, celle-ci est lente et il est préférable d'utiliser une implémentation plus rapide, par exemple **read_lines** du tidyverse. L'une des difficultés vient du fait qu'il n'existe pas une unique manière d'encoder un texte et que le choix par défaut dépend du système d'exploitation. Il est nécessaire de spécifier ce format pour bien lire le document, ici il s'agit d'UTF-8.

```
> library(tidyverse)
> ND <- read_lines("NotreDameDeParis.txt",
    locale = locale(encoding = "UTF-8"))
> head(ND)
```

Lorsque le format d'un texte n'est pas connu, il est possible d'utiliser la fonction **stri_enc_detect** du package stringi, qui détermine automatiquement une liste ordonnée de codages possibles pour un texte donné :

```
> library(stringi)
> stringi::stri_enc_detect(paste(read_lines_raw("LesMiserables1.txt",
    n_max = 30), collapse = " "))
[[1]]'Encoding'
 [1] "UTF-8"
```

Un affichage des premières lignes permet de se rendre compte d'un problème : la présence d'un entête (et en fait également d'un pied-de-page) propre aux textes du projet Gutenberg, qu'il faut enlever pour obtenir le texte brut. Pour cela, une fonction, **gutenberg_strip**, existe dans le package gutenbergr dédié aux textes téléchargés depuis ce site :

```
> library(gutenbergr)
> ND <- gutenbergr::gutenberg_strip(ND)
> ND %>% head
```

Le travail de nettoyage est presque terminé : le projet Gutenberg utilise deux caractères non classiques `_` et `--`, qui poseraient un souci par la suite. La fonction **str_replace_all** du tidyverse permet d'utiliser des expressions régulières pour remplacer ces caractères par des espaces. Il s'avère également utile de remplacer

l'apostrophe par un espace qui est lui compris comme séparant des mots. L'utilisation de cette fonction est extrêmement simple : on indique en premier argument le texte, puis en deuxième argument tous les caractères que l'on souhaite remplacer séparés par `|`, et enfin le caractère utilisé comme remplacement :

```
> ND <- str_replace_all(ND, "_|--|’", " ")
```

Le vecteur de chaîne de caractères ainsi obtenu est une version « propre » de *Notre-Dame de Paris*. Il s'agit maintenant de répéter la lecture et le nettoyage pour tous les textes considérés et de les stocker dans un unique tibble. On définit pour cela une fonction qui lit et nettoie un document. Notons que la fonction **str_c** permet, comme la fonction **paste**, de concaténer un vecteur de caractères. La fonction **str_sub** permet, comme la fonction **subst**, de sélectionner une chaîne de caractères : ici on spécifie que l'on récupère le dernier caractère de la chaîne (`1L` représente le premier et `-1L` le dernier) puis on applique la fonction **lecture** à chaque texte à l'aide de la fonction **map** du tidyverse :

```
> noms <- c("NotreDameDeParis", "LesMiserables1", "LesMiserables2",
    "LesMiserables3", "LesMiserables4", "LesMiserables5")
> lecture <- function(nom) {
    read_lines(str_c(nom, ".txt"),
        locale = locale(encoding = "UTF-8")) %>%
     gutenbergr::gutenberg_strip() %>% str_replace_all("_|--|’", " ")
    }
> hugo <- tibble(nom = noms) %>%
    mutate(id = if_else(nom == "NotreDameDeParis",
        "ND", str_c("M", str_sub(nom, start = -1L)))) %>%
    mutate(txt = map(nom, lecture))
```

Il reste alors à modifier cette table à l'aide de la fonction **unnest** pour obtenir non pas un tableau de cinq lignes dont une colonne contient des vecteurs de texte, mais un tableau contenant une ligne par ligne de texte. On ajoute de plus des numéros de lignes. La fonction **row_number** donne le rang des observations.

```
> hugo <- hugo %>% unnest() %>% group_by(id) %>%
    mutate(ligne = row_number()) %>% ungroup()
> hugo %>% slice(75000:75003)
# A tibble: 4 x 4
  nom            id    txt                       ligne
  <chr>          <chr> <chr>                     <int>
1 LesMiserables5 M5    valait la peine, et cet     241
2 LesMiserables5 M5    même où la Bastille ava     242
3 LesMiserables5 M5    ainsi qu'il les bâtirai     243
4 LesMiserables5 M5    encombrement difforme.      244
```

2. Codage en terme de sac de mots

Comme indiqué plus haut, le principe du codage en sac de mots est de compter les occurrences des mots dans des documents. Une première étape naturelle est d'établir la liste des mots apparaissant (dans l'ordre et avec répétition) dans les documents. Le package tidytext permet de faire ce travail d'extraction des mots en générant à partir d'une table dont une des colonnes correspond à un texte une nouvelle table dans laquelle on retrouve les lignes de la table initiale, privée de la colonne de texte, mais avec une colonne de mots répétés autant de fois qu'il y a de mots dans le texte correspondant. La fonction **unnest_tokens** permet d'extraire d'autres *jetons* (token en anglais) que des mots : des couples de mots (bigramme), des lignes ou des paragraphes par exemple.

```
> library(tidytext)
> hugo_tokens <- hugo %>% unnest_tokens(mots, txt)
> hugo_tokens %>% slice(1:5)
# A tibble: 5 x 4
  nom              id    ligne mots
  <chr>            <chr> <int> <chr>
1 NotreDameDeParis ND        1 victor
2 NotreDameDeParis ND        1 hugo
3 NotreDameDeParis ND        3 notre
4 NotreDameDeParis ND        3 dame
5 NotreDameDeParis ND        3 de
```

Il est alors facile de compter les mots sur l'ensemble des documents et de déterminer les mots les plus fréquents. La fonction **top_n** permet de récupérer les 20 occurrences les plus importances triées en fonction de la variable `n`.

```
> hugo_mots <- hugo_tokens %>% group_by(mots) %>% summarize(n = n())
> hugo_mots %>% slice(1000:1005)
# A tibble: 6 x 2
  mots         n
  <chr>    <int>
1 agrafes      2
2 agraire      1
3 agrandi      3
4 agrandir     1
5 agrandis     1
6 agréable    15
> hugo_mots %>% top_n(20, n) %>% arrange(desc(n)) %>% pull(mots)
 [1] "de"    "la"    "et"    "le"    "il"    "l"
 [7] "à"     "un"    "les"   "d"     "une"   "en"
[13] "que"   "qui"   "est"   "dans"  "était" "des"
[19] "qu"    "ce"
```

On remarque qu'il s'agit essentiellement de mots vides (stop words) que l'on peut souhaiter exclure de l'analyse, de même que les mots n'apparaissant pas assez

souvent. En effet, ces mots ne sont pas nécessairement utiles pour caractériser des documents. Ces stop words dépendent bien évidemment de la langue utilisée et il faut donc récupérer cette liste de mots avec la fonction **get_stopwords** en précisant la langue. Dans cette version du package, il y en a 164 mais il est possible d'en ajouter manuellement. La liste est consultable comme suit :

```
> library(stopwords)
> stopwords(language = "fr")
```

On filtre à la fois les stop words et les mots n'apparaissant pas plus de 10 fois :

```
> hugo_mots <- hugo_mots %>%
    anti_join(get_stopwords("fr"), by = c("mots" = "word")) %>% filter(n>=10)
```

On obtient alors :

```
> hugo_mots %>% top_n(20, n) %>% arrange(desc(n)) %>% pull(mots)
 [1] "plus"    "a"       "comme"   "tout"    "dit"
 [6] "deux"    "là"      "où"      "homme"   "si"
[11] "bien"    "fait"    "être"    "marius"  "jean"
[16] "valjean" "sous"    "rue"     "autre"   "cosette"
```

Une dernière étape du prétraitement consiste à faire de la lemmatisation ou de la stemmatisation. La lemmatisation regroupe les singuliers-pluriels, masculins-féminins, les formes verbales (chanteras, chantait), etc. La stemmatisation regroupe les formes graphiques de même racine (frais, fraîcheur). Pour la stemmatisation, on peut par exemple utiliser la fonction **wordStem** du package SnowballC. Dans cet exemple, le corpus de texte étant important, nous ne ferons ni lemmatisation ni stemmatisation.

Ensuite, on peut construire un nuage de mots qui met en évidence les mots les plus fréquents puisque ceux-ci sont écrits avec une police d'autant plus grande qu'ils ont une fréquence élevée (voir Fig. 11.6) :

```
> library(wordcloud)
> hugo_mots %>% top_n(100, n) %>% {
    wordcloud(.[["mots"]], .[["n"]], min.freq = 10, max.words = 100,
      color = brewer.pal(8,"Dark2"), random.order=FALSE, scale=c(3,.5))
    }
```

Bien que visuellement plaisante, cette représentation en nuage de mots a généralement un intérêt très limité. Elle permet difficilement d'analyser des textes.

Notre objectif est de classer automatiquement les textes ou plutôt des groupes de lignes de texte. Ces groupes de lignes doivent être suffisamment grands pour que le style soit détectable mais pas trop tout de même, de façon à ce que le travail soit intéressant. Nous considérons ici des documents correspondant à des groupes de 100 lignes consécutives, ce qui semble être une taille raisonnable.

FIGURE 11.6 – Nuage des mots les plus utilisés dans les œuvres étudiées.

```
> hugo_tokens <- hugo_tokens %>%
    mutate(doc_id = str_c(id, sprintf("_%03i",
    (ligne-1) %/% 100 + 1)))
```

La fonction **sprintf** est une fonction qui existe dans de nombreux langages avec une syntaxe similaire : elle permet de générer une chaîne de caractères à partir de variables. Ici, la suite de caractères `%03i` est remplacée par la valeur de `(ligne-1) %/% 100 + 1`, un entier, écrit avec 3 chiffres dont éventuellement des 0 au début.

Le codage en sac de mots peut alors se faire en trois temps : on ne conserve que les mots de la liste précédente, on les compte et on transforme le tableau en une table avec une ligne par document et une colonne par mots.

```
> hugo_mots_doc <- hugo_tokens %>% semi_join(hugo_mots) %>%
    group_by(doc_id, mots) %>% summarize(n = n())
> hugo_dtm <- hugo_mots_doc %>% spread(mots, n, fill = 0)
```

3. Analyse des correspondances et visualisation
À partir de la matrice documents-mots, on calcule des distances entre documents et des distances entre mots. Ici, on utilise la métrique du χ^2, ce qui implique que deux documents sont proches s'ils ont les mêmes profils de mots, i.e. utilisent les mots dans des proportions similaires. On peut ensuite représenter graphiquement les documents et les mots dans un espace de dimension inférieure. Ces deux étapes reviennent à effectuer une analyse des correspondances sur la matrice documents-mots (voir fiche 7.2).

Une alternative classique consiste à coder la matrice documents-mots par le codage tf-idf (term frequency and inverse document frequency) qui multiplie l'effectif d'un mot par l'opposé du logarithme de la proportion de documents contenant ce mot.

Ce codage s'obtient dans tidytext à l'aide de la commande **bind_tf_idf**. Puis, on projette dans un espace de dimension inférieure en effectuant une décomposition en valeurs singulières de la matrice tf-idf. Cette méthode s'appelle analyse sémantique latente.
Pour effectuer l'analyse des correspondances, on écrit :

```
> library(FactoMineR)
> hugo_ca <- CA(hugo_dtm %>% as.data.frame() %>%
                column_to_rownames("doc_id"), ncp = 1000, graph = FALSE)
```

Les deux premières coordonnées de cette représentation permettent de visualiser les textes dans un espace naturel.

```
> hugo_ca_coord <- hugo_ca$row$coord
> hugo_ca_df <- tibble(doc_id = row.names(hugo_ca_coord),
                       Dim1 = hugo_ca_coord[,1], Dim2 = hugo_ca_coord[,2],
                       livre = str_sub(doc_id, 1L, 2L))
> ggplot(hugo_ca_df,
    aes(x = Dim1, y = Dim2, color = livre, group = livre)) +geom_path() +
    geom_point() + coord_equal(xlim = c(-1, 1), ylim = c(-1, 1))
```

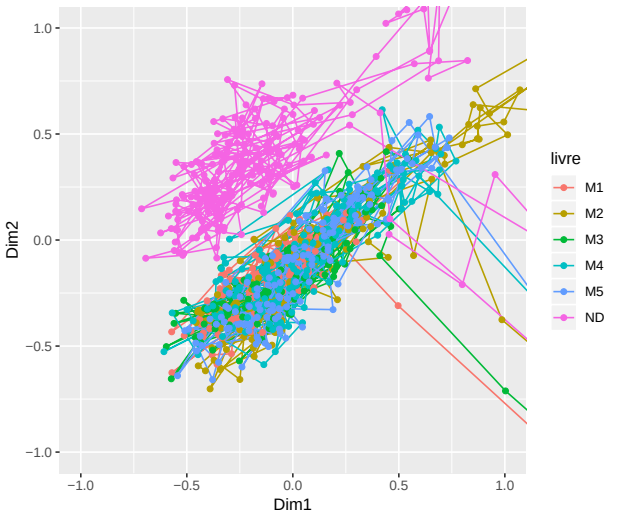


FIGURE 11.7 – Positions des blocs de 100 lignes reliés dans l'ordre dans la représentation de l'analyse des correspondances.

Cette représentation montre clairement une différence entre *Notre-Dame de Paris* et *Les Misérables*.

4. Classification par l'algorithme des k-means
On souhaite classer les groupes de lignes de manière automatique. L'algorithme le plus classique de classification est l'algorithme des k-means (voir fiche 8.2). Son unique paramètre est le nombre de groupes souhaité. Ce choix est difficile. On prendra ici 4 groupes, un choix que l'on expliquera un peu plus tard.

```
> set.seed(42)
> hugo_ca_kmeans <- kmeans(hugo_ca_coord, 4, nstart = 100)
> hugo_ca_df <- hugo_ca_df %>%
                mutate(classe = as.factor(hugo_ca_kmeans$cluster))
> ggplot(hugo_ca_df, aes(x = doc_id, y = 1, fill = classe)) +
    geom_raster() + scale_x_discrete(breaks=NULL) +
    geom_vline(xintercept =669)
```

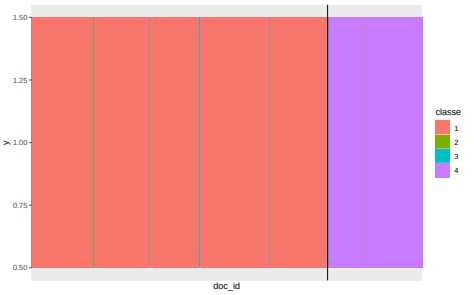


FIGURE 11.8 – Classification obtenue pour les blocs de 100 lignes ordonnés sur l'axe horizontal dans l'ordre des textes avec *Les Misérables* suivi de *Notre-Dame de Paris*.

Cette représentation peut être utile pour voir une évolution des mots utilisés dans les textes. Ici, ce n'est pas le cas car les classes se contentent de dissocier les deux œuvres.

Le choix de quatre groupes alors qu'il n'y a que deux livres peut sembler étrange. En regardant bien la taille des groupes (et la répartition des blocs de 100 lignes dans ces groupes), on peut se rendre compte à l'aide de la fonction **table** qu'ici, bien que l'on ait quatre groupes, seuls deux sont de taille raisonnable.

```
> hugo_ca_df %>% select(classe, livre) %>% table()
      livre
classe  M1  M2  M3  M4  M5  ND
     1 139 124 111 157 131   1
     2   1   0   0   0   0   0
     3   1   1   1   2   1   1
     4   0   0   0   0   0 211
```

Pour terminer, visualisons le vocabulaire associé à chacune des classes en choisissant les mots en fonction d'un critère combinant fréquence et fréquence relative :

```
> hugo_crit_classe <- hugo_mots_doc %>%
    left_join(hugo_ca_df %>% select(doc_id, classe)) %>%
    group_by(classe, mots) %>% summarize(n = sum(n)) %>%
    mutate(prop = n / sum(n))%>%
    group_by(mots) %>% mutate(n_tot = sum(n)) %>%
    ungroup() %>% mutate(prop_tot = n_tot / sum(n),
                         crit = prop * log(prop / prop_tot))

> hugo_top_classe <- hugo_crit_classe %>%
    group_by(classe) %>% top_n(20, crit) %>%
    arrange(desc(crit)) %>% mutate(rank = row_number()) %>%
    filter(rank <= 20) %>% ungroup()

> ggplot(hugo_top_classe %>%
           unite(mots_, mots, classe, remove = FALSE) %>%
           mutate(mots = fct_reorder(mots_,
                  crit)),
         aes(x = mots, y = crit, fill = classe)) +
    geom_col() +
    facet_wrap(~classe, scales = "free") +
    scale_x_discrete(labels = function(x) {str_match(x, "([^_]*)")}) +
    coord_flip()
```

En figure 11.9, on reconnaît bien *Les Misérables* dans la classe 1 et *Notre-Dame de Paris* dans la classe 4. La classe 3 correspond aux tables des matières tandis que la classe 2 ne concerne qu'un bloc de 100 lignes.

Pour aller plus loin

Un plugin Rcmdr est disponible pour l'analyse de texte : RcmdrPlugin.temis (Graphical Integrated Text Mining Solution).

L'algorithme de classification présenté ici est un algorithme très simple. L'algorithme le plus classique est en fait une méthode bayésienne reposant sur des modèles de mélanges (voir fiche 8.3) de multinomiales appelée Latent Dirichlet Allocation. Une implémentation est disponible en R dans le package topicmodels.

L'extraction de jetons peut être effectuée à l'aide d'outils de traitement automatique du langage (NLP) qui donnent des informations complémentaires sur les jetons : par exemple, en détectant la fonction des mots dans les phrases ou en extrayant la forme neutre canonique (lemmatisation). Ceci permet d'améliorer les analyses. Le package cleaNLP propose ainsi une interface avec trois librairies différentes : CoreNLP, spaCy et udpipe.

Soulignons également l'existence du package tm et des ses extensions qui permettent eux aussi une analyse de texte. Enfin, le package Xplortext permet d'aller

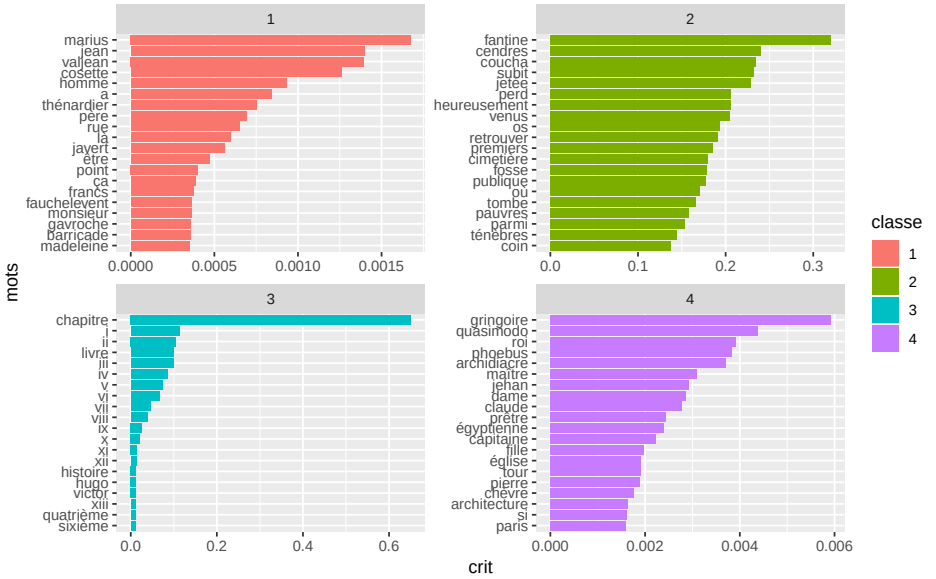


FIGURE 11.9 – Vocabulaire associé aux quatre classes obtenues par l'algorithme des k-means sur la représentation de l'analyse factorielle.

plus loin dans l'analyse textuelle, et notamment de faire de la classification avec une contrainte de contiguïté, ce qui est utile si l'on veut par exemple découper un discours en parties : voir par exemple le livre *Analyse textuelle avec R* (Bécue-Bertaut, 2018).

11.3 Fouille de graphe

Objet

La fouille de graphe (Graph Mining) est la version de la fouille de données spécifique aux graphes. Un graphe est défini par des nœuds reliés par des arêtes pondérées ou non et orientées ou non. Cette structure apparaît naturellement lorsque l'on travaille sur des entités ayant des relations entre elles. Les entités correspondent alors aux nœuds et les relations aux arêtes. Il existe une multitude d'exemples de ce type de données : les réseaux sociaux où les nœuds sont des personnes avec une arête entre deux personnes si elles sont amies ; des données d'échanges commerciaux où les nœuds sont des pays et où les arêtes peuvent indiquer la présence d'échanges entre les pays ou encoder le montant des transactions entre les pays ; les réseaux écologiques où les nœuds représentent des espèces et les relations sont les liens trophiques ; les graphes d'interactions entre gènes, etc.

L'objectif de la fouille de graphe est de caractériser ces graphes et de les utiliser pour extraire des informations. Il s'agit d'un sujet en expansion. Nous nous contenterons de regarder comment caractériser l'importance des nœuds à l'aide d'indices de centralité et d'utiliser des algorithmes de détection de communautés dans ces graphes. L'idée est d'établir des classes où les nœuds d'une même classe sont fortement connectés et où les nœuds de deux communautés sont très peu connectés.

Exemple

Dans son ouvrage *The Stanford GraphBase : A Platform for Combinatorial Computing*, Donald Knuth a décrit les relations entre les personnages des *Misérables* en comptant le nombre d'apparitions de toutes les paires de personnages dans le même chapitre. Le graphe correspondant a pour nœuds les personnages des *Misérables* et les arêtes relient deux personnages apparaissant simultanément dans les mêmes chapitres avec un poids égal au nombres de chapitres communs.

L'objectif est de visualiser ce graphe, de calculer et visualiser des indices de centralité pour repérer les personnages importants et de déterminer et visualiser des groupes de personnages. Nous utiliserons pour cela le package `igraph`.

Étapes

1. Importer le graphe
2. Visualiser le graphe
3. Calculer des indices de centralités
4. Déterminer des classes

Traitement de l'exemple

1. Importer le graphe
Le package igraph permet d'importer des graphes dans des formats variés à l'aide de la commande **read_graph**. Le graphe des *Misérables* est disponible au format gml sur le site du livre.

```
> library(igraph)
> mis_graph <- read_graph("lesmiserables.gml", format = "gml")
> mis_graph
IGRAPH 830c84c U--- 77 254 --
+ attr: id (v/n), label (v/c), group
| (v/n), fill (v/c), border (v/c),
| value (e/n)
+ edges from 830c84c:
 [1]  1-- 2  1-- 3  1-- 4  3-- 4  1-- 5
 [6]  1-- 6  1-- 7  1-- 8  1-- 9  1--10
[11] 11--12  4--12  3--12  1--12 12--13
[16] 12--14 12--15 12--16 17--18 17--19
[21] 18--19 17--20 18--20 19--20 17--21
[26] 18--21 19--21 20--21 17--22 18--22
+ ... omitted several edges
```

L'objet résultant est un objet de classe igraph qui contient l'équivalent de deux tables : une table des nœuds et de leurs propriétés, et une table des arêtes et de leurs propriétés. Il y a 4 caractères, la lettre `U` signifie que le graphe n'est pas dirigé (Undirected). Le trait d'union signifie que le graphe n'est pas nommé (sinon on aurait `N`), puis le troisième caractère, ici un trait d'union, signifie que le graphe n'est pas pondéré (sinon on aurait `W`, pour weighted). Enfin, le dernier trait d'union indique que le graphe est unipartite, (sinon on aurait `B`, pour bipartite). On a ensuite deux nombres, 77 pour le nombre de nœuds (vertices) et 254 pour le nombre d'arêtes (edges), ce qu'on peut retrouver en utilisant les fonctions :

```
> vcount(mis_graph)
[1] 77
> ecount(mis_graph)
[1] 254
```

Le détail des nœuds et arêtes est donné dans :

```
> V(mis_graph)
> E(mis_graph)
```

La matrice d'adjacence du graphe peut directement se lire de la façon suivante (on affiche ici seulement les 5 premiers nœuds) :

```
> mis_graph[1:5, 1:5]
5 x 5 sparse Matrix of class "dgCMatrix"
[1,] . 1 1 1 1
[2,] 1 . . . .
[3,] 1 . . 1 .
[4,] 1 . 1 . .
[5,] 1 . . . .
```

Dans cette représentation, les zeros sont affichés comme des points. On voit ici que le deuxième nœud est connecté au nœud 1 et à aucun des autres nœuds compris entre 1 et 5.

Notons qu'il est possible de transformer une matrice d'adjacence en un graphe à l'aide de la commande **graph_from_adjacency_matrix** :

```
> graph_from_adjacency_matrix(mis_graph[], mode = "undirected")
IGRAPH bd01441 U--- 77 254 --
+ edges from bd01441:
 [1]  1-- 2  1-- 3  1-- 4  1-- 5  1-- 6  1-- 7  1-- 8
 [8]  1-- 9  1--10  1--12  3-- 4  3--12  4--12 11--12
[15] 12--13 12--14 12--15 12--16 12--24 12--25 12--26
[22] 12--27 12--28 12--29 12--30 12--32 12--33 12--34
[29] 12--35 12--36 12--37 12--38 12--39 12--44 12--45
[36] 12--49 12--50 12--52 12--56 12--59 12--65 12--69
[43] 12--70 12--71 12--72 12--73 13--24 17--18 17--19
[50] 17--20 17--21 17--22 17--23 17--24 17--27 17--56
[57] 18--19 18--20 18--21 18--22 18--23 18--24 19--20
+ ... omitted several edges
```

On peut également utiliser une représentation sous la forme d'un data-frame :

```
> as_long_data_frame(mis_graph)[1:5,]
  from to value from_id     from_label from_group from_fill from_border
1    1  2     1       0         Myriel          1   #BEBADA     #aba7c4
2    1  3     8       0         Myriel          1   #BEBADA     #aba7c4
3    1  4    10       0         Myriel          1   #BEBADA     #aba7c4
4    3  4     6       2 MlleBaptistine          1   #BEBADA     #aba7c4
5    1  5     1       0         Myriel          1   #BEBADA     #aba7c4
  to_id       to_label to_group to_fill to_border
1     1       Napoleon        1 #BEBADA   #aba7c4
2     2 MlleBaptistine        1 #BEBADA   #aba7c4
3     3    MmeMagloire        1 #BEBADA   #aba7c4
4     3    MmeMagloire        1 #BEBADA   #aba7c4
5     4   CountessDeLo        1 #BEBADA   #aba7c4
> graph_from_data_frame(as_long_data_frame(mis_graph), directed = FALSE)
IGRAPH 9c4fc18 UN-- 77 254 --
+ attr: name (v/c), value (e/n), from_id (e/n),
```

```
| from_label (e/c), from_group (e/n), from_fill
| (e/c), from_border (e/c), to_id (e/n), to_label
| (e/c), to_group (e/n), to_fill (e/c), to_border
| (e/c)
+ edges from 9c4fc18 (vertex names):
 [1] 1 --2  1 --3  1 --4  3 --4  1 --5  1 --6  1 --7
 [8] 1 --8  1 --9  1 --10 11--12 4 --12 3 --12 1 --12
[15] 12--13 12--14 12--15 12--16 17--18 17--19 18--19
[22] 17--20 18--20 19--20 17--21 18--21 19--21 20--21
+ ... omitted several edges
```

Enfin, on peut également importer un jeu de données en précisant les extrémités de toutes les arêtes dans une matrice. Voici les lignes de code pour importer les données ayant permis de dessiner le graphe de la couverture :

```
> don <- read.table("https://r-stat-sc-donnees.github.io/couverture.csv",
    header=TRUE,sep=";",fileEncoding="UTF-8")
> don[1:6,]
  chapitre     section
1        R       tests
2    tests    int_conf
3    tests comparaison
4    tests        chi2
5    tests   test prop
> graph_couv <- graph_from_edgelist(as.matrix(don))
```

2. Visualiser le graphe

Pour visualiser un graphe, il faut assigner une position à chaque nœud et tracer les arêtes existant entre ces nœuds. Cela paraît simple mais les nœuds n'ont pas de position naturelle : il faut les déduire à partir des arêtes existantes. Il n'existe pas une manière unique de faire cela et, bien souvent, le résultat dépend d'une initialisation aléatoire. Par défaut, igraph propose une méthode choisie automatiquement parmi un catalogue, mais l'on peut bien sûr en choisir une explicitement en utilisant l'argument `layout` :

```
> plot(mis_graph)
> plot(mis_graph, layout = layout.kamada.kawai)
```

3. Calculer des indices de centralité

Il existe de nombreux indicateurs pour décrire un graphe. Parmi les indicateurs globaux, on peut vérifier que le graphe est connecté, ou encore calculer la densité du graphe (nombre d'arêtes sur le nombre de paires de nœuds) :

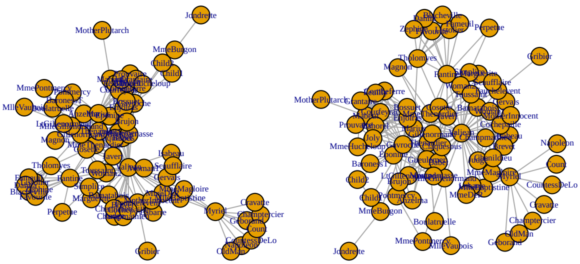


FIGURE 11.10 – Deux visualisations différentes du graphe des relations entre les personnages des *Misérables*.

```
> is_connected(mis_graph)
> graph.density(mis_graph)
```

Il existe aussi de nombreux indicateurs locaux pour caractériser l'importance des nœuds. Le plus simple est le degré, ou degré de centralité, qui compte le nombre d'arêtes associées au nœud. On peut visualiser le résultat (Fig. 11.11) à l'aide de l'option `vertex.size`. On spécifie ici que le rayon du nœud doit être égal à 4 fois la racine du degré : la surface est donc proportionnelle au degré, le facteur 4 ayant été choisi arbitrairement.

```
> centrality <- degree(mis_graph)
> plot(mis_graph, layout = layout.kamada.kawai,
    vertex.size = 4*sqrt(centrality))
```

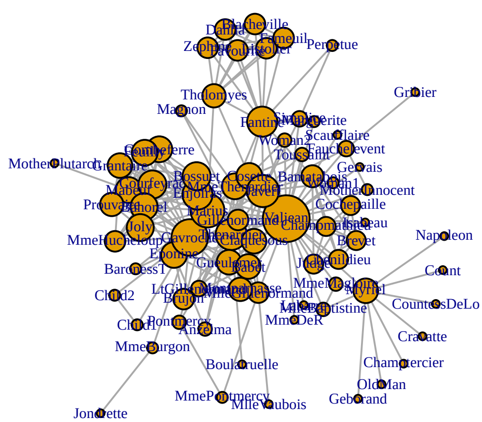


FIGURE 11.11 – Centralité des personnages des *Misérables* calculée comme le degré des nœuds dans le graphe.

On peut également définir la centralité par un indice dit d'intermédiarité. Il repose sur le concept de chemin minimal entre deux nœuds, c'est-à-dire le trajet le plus court pour aller d'un nœud à un autre. L'importance d'un point est alors donnée par le nombre de chemins minimaux passant par ce point (Fig. 11.12).

```
> centrality2 <- betweenness(mis_graph, directed = FALSE)
> plot(mis_graph, layout = layout.kamada.kawai,
     vertex.size = sqrt(centrality2))
```

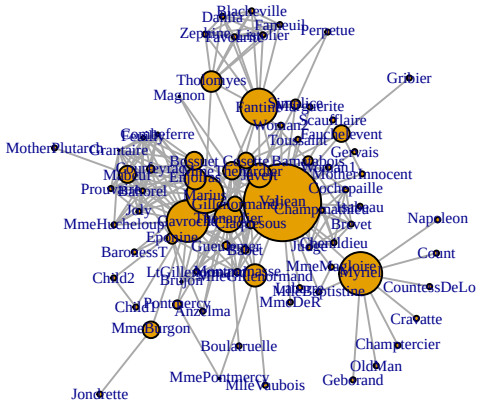


FIGURE 11.12 – Centralité des personnages des *Misérables* calculée via l'intermédiarité des nœuds dans le graphe.

4. Déterminer des classes

Il existe de nombreux algorithmes permettant de déterminer des classes de nœuds d'un graphe. Ceux-ci diffèrent par la notion de groupe utilisé ou par l'algorithme d'optimisation. Par exemple, la fonction **cluster_egdge_betweeness** construit un arbre en retirant de manière séquentielle les arêtes par ordre décroissant d'indice d'intermédiarité. L'intuition de cette méthode est que les arêtes à fort indice d'intermédiarité sont des arêtes reliant des groupes plutôt que des arêtes internes à ceux-ci (Fig. 11.13).

```
> group <- membership(cluster_edge_betweenness(mis_graph))
> plot(mis_graph, layout = layout.kamada.kawai,
     vertex.color = group)
```

Enfin, on peut afficher la centralité et le groupe sur le même graphe (Fig. 11.14).

```
> plot(mis_graph, layout = layout.kamada.kawai,
     vertex.size = 4*sqrt(centrality),
     vertex.color = group)
```

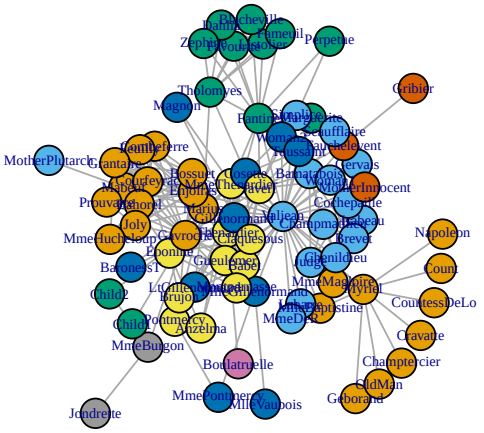


FIGURE 11.13 – Groupes des personnages des *Misérables* obtenus à partir de la notion d'intermédiarité.

Les méthodes utilisant le spectre de la matrice d'adjacence ou du Laplacien (spectral clustering) sont aussi extrêmement populaires pour faire des groupes de nœuds. Dans sa version la plus simple, la méthode part de la matrice d'adjacence, calcule le Laplacien du graphe, diagonalise cette matrice, puis effectue un algorithme de type k-means sur les vecteurs propres du Laplacien. Cette approche nécessite de spécifier le nombre de groupes souhaités, ici 6. L'argument `which = "sa"` indique qu'on retient les plus petites valeurs propres et l'argument `scaled = FALSE` que l'on ne souhaite pas renormaliser ses valeurs propres. Le Laplacien se calcule par $L = D - A$ avec A la matrice d'adjacence et D la matrice diagonale des degrés des nœuds correspondant au vecteur somme des lignes de A.

```
> Lap <- embed_laplacian_matrix(mis_graph, no = 6, which = "sa",
scaled = FALSE)
> resspectral <- kmeans(Lap$X[ ,-1], centers = 6, nstart = 1)
```

On peut représenter le résultat de cette classification :

```
> plot(mis_graph, vertex.size = 5, vertex.label = NA,
     vertex.color = resspectral$cluster,
     vertex.frame.color = resspectral$cluster)
```

Notons que la classification par spectral clustering peut se faire manuellement avec les lignes suivantes :

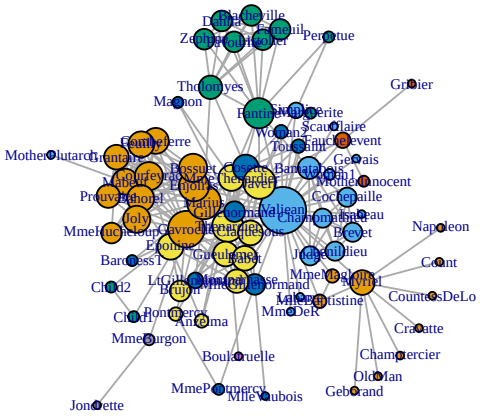


FIGURE 11.14 – Groupes des personnages des *Misérables* obtenus via l'intermédiarité et la centralité.

```
> A <- as.matrix(mis_graph[])
> D <- diag(rowSums(A))
> L <- D - A
> LR <- svd(L)$u[,76:71]
> resspectral <- kmeans(LR, centers = 6, nstart = 1)
```

Pour aller plus loin

Pour aller plus loin sur l'analyse de graphe, on pourra se référer au site de igraph : `http://igraph.org/r/` et au livre *Statistical Network Analysis with igraph*.Les packages tidygraph et ggraph permettent une approche de type tidyverse en étendant la syntaxe de dplyr et ggplot2 aux graphes. Le package visnetwork permet une visualisation interactive des graphes : il repose sur la bibliothèque javascript vis.js.

Dans une optique de test, il est courant de se comparer au graphe d'Erdös-Renyi (où la probabilité d'observer une arête entre deux nœuds est la même pour tous les nœuds) avec le même nombre de nœuds et d'arêtes. Il peut être généré par :

```
>  sample_gnm(vcount(mis_graph), ecount(mis_graph))
```

Des procédures de tests qui intègrent la distribution du degré des nœuds sont aussi disponibles. Pour la partie classification, il faut mentionner les techniques de classification de type modèle de blocs latents, stochastic bloc models (avec mixer et blockcluster par exemple) qui sont des modèles de mélange pour les réseaux. Le modèle génératif sous-jacent suppose que chaque nœud appartient à une classe avec une certaine probabilité puis, sachant la classe, la probabilité d'avoir une arête entre deux nœuds est donnée par une distribution de Bernoulli.

Annexes

A.1 Écriture d'une formule pour les modèles

De nombreuses méthodes telles la régression linéaire, l'analyse de variance, la régression logistique, etc., nécessitent l'écriture d'un modèle. Pour toutes ces fonctions, les modèles s'écrivent de façon analogue. Nous montrons dans cette section comment y parvenir à partir de plusieurs exemples :

- `Y ~ .` : modèle avec la variable `Y` expliquée en fonction de toutes les autres variables du jeu de données ;
- `Y ~ x1+x2` : modèle avec la variable `Y` expliquée en fonction des variables `x1`, `x2` et de la constante ; équivalent à `Y ~ 1+x1+x2` ;
- `Y ~ -1+x1+x2` : modèle avec la variable `Y` expliquée en fonction des variables `x1` et `x2` sans la constante (`-1` élimine la constante) ;
- `Y ~ x1+x2 + x1:x2` : modèle avec la variable `Y` expliquée en fonction des variables `x1`, `x2` et de l'interaction entre `x1` et `x2` ;
- `Y ~ x1*x2` : équivaut au modèle précédent ;
- `Y ~ x1*x2*x3` : équivaut au modèle avec tous les effets principaux et les interactions entre `x1`, `x2` et `x3`, donc `Y ~ x1+x2+x3 + x1:x2 + x1:x3 + x2:x3 + x1:x2:x3` ;
- `Y ~ (x1+x2):x3` : équivaut au modèle `Y ~ x1:x3+x2:x3` ;
- `Y ~ x1*x2*x3 - x1:x2:x3` : équivaut au modèle `Y ~ x1+x2+x3 + x1:x2 + x1:x3 + x2:x3` ;
- `Y ~ x1+x2 %in% x1` : modèle avec l'effet de `x1` et l'effet de `x2` hiérarchisé (ou emboîté) dans `x1` ;
- `Y ~ sin(x1)+sin(x2)` : modèle de `Y` en fonction de sin(`x1`) et sin(`x2`) ;
- `Y ~ x1 | fac` : modèle où l'effet de `x1` est envisagé pour chaque niveau de `fac` ;
- `Y ~ I(x1^ 2)` : modèle avec la variable `Y` expliquée par $\texttt{x1}^2$; `I(.)` protège l'expression `x1^2`, sinon `x1^2` est interprété comme `x1*x1 = x1+x1+x1:x1 = x1` ;
- `Y ~ I(x1+x2)` : modèle où la variable `Y` est expliquée par la constante et la variable résultant de la somme (individu par individu) des variables `x1` et `x2` ; `I(.)` protège l'expression `x1+x2`, sinon `x1+x2` est interprété comme deux variables explicatives.

A.2 Environnement RStudio

RStudio est un environnement de développement intégré (IDE) permettant d'utiliser le logiciel R dans une interface conviviale et personnalisable. Son installation se fait sans difficulté (`https://www.rstudio.com/`), mais suppose d'avoir préalablement installé R.

L'environnement RStudio se présente sous la forme d'une fenêtre globale scindée en 4 sous-fenêtres distinctes :
- la fenêtre de scripts (en haut à gauche) ;
- la console (en bas à gauche) ;
- la fenêtre d'environnement et d'historique (en haut à droite) ;
- la fenêtre des fichiers, graphes, packages et d'aide (en bas à droite).

Dans la barre d'outils de la fenêtre globale, on retrouve :
- la possibilité d'ouvrir des fichiers existants ou d'en créer de nouveaux (icône `File`) ;
- les options de fermeture et de sauvegarde de la session ou de sélection du répertoire de travail (icône `Session`), etc.

RStudio permet également de :
- créer des documents dynamiques via l'outil RMarkdown (cf. § 1.3) ;
- appeler et développer des fonctions écrites en `C++` (menu `File` puis `New File` puis `C++ File`) ;
- créer des applications interactives grâce au package `shiny`, installé automatiquement avec RStudio (menu `File` puis `New File`) ;
- faciliter l'appréhension des fonctionnalités proposées (menu `Help`), en particulier grâce à des aide-mémoires (icône `Help` puis `Cheatsheets`) ;
- proposer une aide à la création de packages (menu `File` puis `New Project` puis `New Directory` et `R Package`).

Nous donnons maintenant une description sommaire des sous-fenêtres mentionnées plus haut. La fenêtre de scripts se présente sous la forme habituelle. Plusieurs scripts peuvent être ouverts simultanément. L'onglet symbolisé par une baguette magique permet de retrouver facilement certains éléments de code, par exemple la définition d'une fonction. Par ailleurs, plusieurs options pour exécuter le code dans la console sont possibles : ensemble du script, partie située avant pointeur par exemple. Via l'icône symbolisée par un carnet de notes, on peut exporter au format `HTML`, `PDF` ou `word` et ainsi visualiser l'ensemble du code et des sorties.

Dans la fenêtre `Workspace/History`, le menu `Workspace` (ou `Environment`) recense tous les objets créés ou importés. Il offre également un menu facilitant l'importation de données. L'onglet `History` recense l'ensemble des commandes qui ont été exécutées dans la console.

La dernière sous-fenêtre (`Files`, `Plots`, etc.) propose bon nombre de fonctionnalités. L'onglet `Files` agit comme un explorateur de fichiers en affichant tous les éléments d'un dossier. En cliquant sur un fichier en particulier, on l'ouvre dans R lorsque cela est possible, comme pour des fichiers de scripts ou des fichiers au for-

mat `.txt` ou `.tex` par exemple : un onglet spécifique apparaît alors dans la fenêtre de scripts. Sinon, le fichier s'ouvre dans une fenêtre à part en lançant l'application nécessaire à son ouverture.

L'onglet `Plots` permet de travailler sur les graphiques qui ont été créés dans la session. On peut passer de l'un à l'autre grâce aux flèches de défilement, agrandir le graphe en l'ouvrant dans une nouvelle fenêtre (icône `Zoom`) ou bien l'exporter au format souhaité. L'onglet `Packages` liste l'ensemble des packages venant avec l'installation de R et RStudio, ainsi que ceux installés par l'utilisateur. Cocher un package conduit directement à son chargement dans l'espace de travail (fonction **library**). Par ailleurs, un onglet est dédié à l'installation de packages. Si l'on rencontre des difficultés lors du téléchargement, il peut être utile de modifier certains paramètres, typiquement le choix du site miroir. Pour cela, il suffit de recourir au menu `Tools` de la fenêtre principale et de sélectionner `Global Options` et `Packages` pour procéder aux modifications voulues.

A.3 Le package Rcmdr

L'interface graphique Rcmdr, prononcée R Commandeur (Fox, 2016), est disponible dans le package Rcmdr. Cette interface permet d'utiliser R à l'aide d'un menu déroulant de façon conviviale. L'intérêt de ce package est aussi pédagogique puisqu'il fournit les lignes de code correspondant aux analyses effectuées : on se familiarise ainsi avec la programmation en voyant quelles sont les fonctions employées. L'interface Rcmdr ne contient pas toutes les fonctions disponibles sous R, ni toutes les options des différentes fonctions, mais les fonctions les plus courantes sont programmées et les options les plus classiques disponibles.

Le package doit être installé une seule fois (voir § 1.7, p. 28). Ensuite, on charge l'interface par :

```
> library(Rcmdr)
```

L'interface (Fig. A.15) s'ouvre automatiquement. Cette interface possède un menu déroulant et une fenêtre de script. Si Rcmdr est ouvert dans la console R et non dans RStudio alors il y a aussi une fenêtre de sortie. Lorsque le menu déroulant est utilisé, l'analyse est lancée et les lignes de code qui ont servi à générer l'analyse sont écrites dans la fenêtre de script.
Pour importer des données avec Rcmdr, le plus simple est d'avoir un fichier Excel :
`Données` → `Importer des données` → `Depuis un fichier Excel`
Avec un fichier au format txt ou csv :
`Données` → `Importer des données` → `Depuis un fichier texte ou le presse papier`
Il faut ensuite préciser le séparateur de colonnes (séparateur de champs) et le séparateur de décimales (un `"."` ou une `","`).

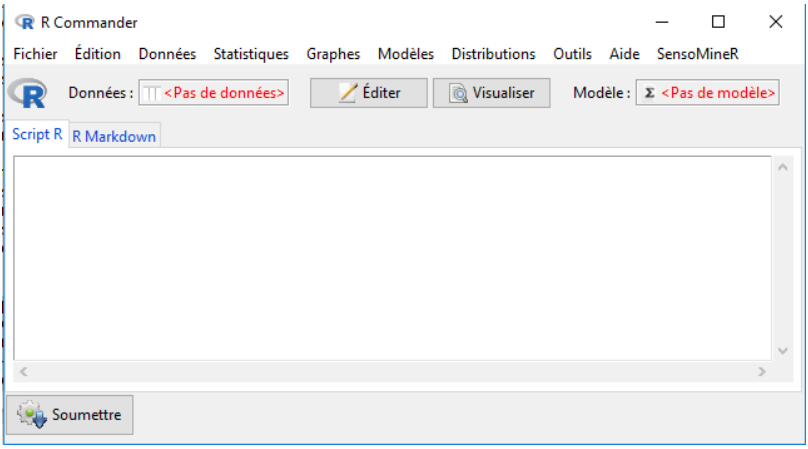


FIGURE A.15 – Fenêtre principale de Rcmdr.

Pour vérifier que le jeu de données a bien été importé :
`Statistiques` → `Résumés` → `Jeu de données actif`

Si on veut importer un jeu de données au format `csv` qui contient l'identifiant des individus, il n'est pas possible de préciser dans le menu déroulant de Rcmdr que la première colonne contient l'identifiant. On peut alors importer le jeu de données en considérant l'identifiant comme une variable puis modifier la ligne de code écrite dans la fenêtre script en rajoutant l'argument `row.names=1` puis cliquer sur `Soumettre`.

Pour changer de jeu de données actif, il suffit de cliquer sur l'encadré `Données`. Si on modifie le jeu de données actif (par exemple en convertissant une variable), il est nécessaire de rafraîchir le jeu de données par :
`Données` → `Jeu de données actif` → `Rafraîchir le jeu de données actif`

Les résultats des analyses, numériques ou graphes, se retrouvent dans la fenêtre qui convient de RStudio. Il est possible de récupérer le code en RMarkdown, ce qui permet d'avoir un travail reproductible (cf. § 1.3). À l'issue d'une session Rcmdr, il est possible de sauvegarder la fenêtre de script, c'est-à-dire toutes les instructions, ainsi que le fichier de sortie, c'est-à-dire tous les résultats. On peut fermer à la fois R et Rmcdr en faisant `Fichier` → `Sortir` → `Fermer Commander et R`.

Remarques

- Si vous avez fermé la fenêtre Rcmdr et que vous souhaitez en rouvrir une dans une même session R, taper la commande **`Commander()`**.
- Écrire dans la fenêtre de script de Rcmdr ou dans RStudio est totalement équivalent. Si une instruction est lancée depuis Rcmdr, elle est reconnue également dans RStudio et inversement. Les objets créés par Rcmdr peuvent donc être utilisés dans Rstudio.

Bibliographie

Bécue-Bertaut M. (2018). *Analyse textuelle avec R.* Presses universitaires de Rennes, Rennes.

Bergé L., Bouveyron C. et Girard S. (2012). Hdclassif : An R package for model-based clustering and discriminant analysis of high-dimensional data. *Journal of Statistical Software, Articles*, **46**(6), 1–29.

Breiman L. (2001). Random forests. *Machine Learning*, **45**, 5–32.

Breiman L., Friedman J., Olshen R. et Stone C. (1984). *Classification and Regression Trees.* Monterey.

Chollet F. et Allaire J. (2018). *Deep Learning with R.* Manning.

Collet D. (2003). *Modelling Binary Data.* Chapman & Hall.

Cornillon P.A. et Matzner-Løber E. (2010). *Régression avec R.* Springer, Paris.

Dagnelie P. (2007). *Statistique théorique et appliquée : Vol. 2. Inférence statistique à une et à deux dimensions.* De Boeck Université, Bruxelles.

Dempster A.P., Laird N.M. et Rubin D.B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society B*, **39**(1), 1–38.

Escofier B. et Pagès J. (2016). *Analyses factorielles simples et multiples. Objectifs méthodes et interprétation.* Dunod, Paris, 5 ed.

Fox J. (2016). *Using the R Commander. A Point-and-Click Interface for R.* Chapman & Hall/CRC The R Series.

Giraud C. (2015). *Introduction to High-Dimensional Statistics.* CRC Press, Boca Raton.

Hastie T., Tibshirani R. et Friedman J. (2009). *The Elements of Statistical Learning : Data Mining, Inference, and Prediction.* Springer, 2 ed.

Husson F., Lê S. et Pagès J. (2016). *Analyse de données avec R.* Presses universitaires de Rennes, Rennes, 2 ed.

Husson F. et Pagès J. (2013a). *Statistiques générales pour utilisateurs – tome 2, exercices et corrigés.* Presses universitaires de Rennes, Rennes, 2 ed.

Husson F. et Pagès J. (2013b). *Statistiques générales pour utilisateurs – tome 2, exercices et corrigés.* Presses universitaires de Rennes, Rennes, 2 ed.

Kaufman L. et Rousseeuw P.J. (1990). *Finding Groups in Data : An Introduction to Cluster Analysis.* Wiley, New York.

Kuhn M. et Johnson K. (2013). *Applied Predictive Modeling.* Springer.

Lebart L., Prion M. et Morineau A. (2006). *Statistique exploratoire multidimensionnelle.* Dunod, Paris.

Little R.J.A. et Rubin D.B. (1987, 2002). *Statistical Analysis with Missing Data.* John Wiley & Sons series in probability and statistics, New York.

Miele V. et Louvet V. (2016). *Calcul parallèle avec R.* Springer, Paris.

Murell P. (2011). *R Graphics.* Chapman & Hall.

Pagès J. (2010). *Statistiques générales pour utilisateurs – tome 1, méthodologie.* Presses universitaires de Rennes, Rennes, 2 ed.

Saporta G. (2011). *Probabilités, analyse des données et statistiques.* Technip, Paris, 3 ed.

Welch B.L. (1951). On the comparison of several mean values : an alternative approach. *Biometrika*, **38**, 330–336.

Wickham H. (2009). *Ggplot2 : Elegant Graphics for Data Analysis.* Springer.

Index des fonctions

`!=` 14, 113
`->` 9
`:` 12
`<-` 9
`<=` 14
`<` 14
`==` 14, 16
`=` 9
`>=` 14
`>` 14
`?` 8
`Inf` 11
`NA` 11
`NaN` 11
`[ ]` 15, 18
`[[ ]]` 24
`#` 9
`$` 28
`%*%` 20
`|` 16
`&` 15
abline 65
abs 16
addCircles 94
addLayersControl 101
addTiles 93
aggregate 117, 205
agnes 245
all.equal 45
all 15
amelia 379
anova 281
any 15, 47
AovSum 279, 284
apply 21, 47, 114
args 28
array 21, 115
as.character 10, 23
as.complex 10
as.data.frame 25
as.factor 22
as.logical 10
as.matrix 18
as.numeric 10, 23
attributes 11
axis 60, 72
barplot 200
bartlett.test 208, 281
boxplot 48, 205, 278
by 118, 119
catdes 247, 254, 259
cbind.data.frame 52
cbind 21, 51
chisq.test 199, 201
chol 20
class 11
cloud 67
colMeans 115
colnames 49
colorNumeric 94, 101
colors 71
Commander 404
contour 66
coplot 73
crossprod 20
cut 41, 103
cv.glmnet 343
c 11, 13, 24

data.frame 25
density 63
det 20
dev.off 68
diag 20
dimdesc 220, 242
dimnames 11, 17, 25
dim 11, 17, 21
duplicated 49
eigen 20
facet_wrap 99
factor 22, 40, 44
format 13
fread 140
friedman.test 281
fwrite 39
gather 98
gbm 349
geocode 92
geom_sf 99
get_map 91
getwd 5
glmnet 342
glm 289
gray 71
grep 14
gsub 14
HCPC 249
hddc 262
heat.colors 66
help 8
hist 62, 73
image 66, 73
imputePCA 378
install.packages 29
interaction.plot 74, 278
is.character 10
is.complex 10
is.logical 10
is.na 11, 16, 45
is.null 10
is.numeric 10
kmeans 253
kruskal.test 207
ksvm 356
lapply 24, 116
layout 69
lda 296
leaflet 93
legend 72, 269, 284, 315
length 11, 17, 22, 23
levels 22, 42
library 29
lines 65
lm 266, 272
load 6
ls 9
make_bbox 92
matplot 74
matrix 17
MCA 230
mclustBIC 258
Mclust 258
merge 51, 52
MFA 237
mice 379
missForest 379
mode 10, 17, 22, 23
names 11, 23
ncol 21
nlevels 22
nrow 21
object.size 12
objects 9
ordered 22
outer 66, 120
pairs 74
palette 71, 314
par 69, 71, 72
paste 13
PCA 214
pdf 68
persp 66, 74
pie 64, 74
plot3d 68
plot 58–62, 65, 71–72, 74
power.t.test 209
princomp 214

print.default 27
print 9
prop.table 200
qqnorm 74
qqplot 74
qr 20
quantile 42
randomForest 332
rbind 21, 49
read.table 36
read_sf 98
regsubsets 273
relevel 43
replicate 120
rep 12, 13
require 29
residuals 274
return 112
rm 9
rowMeans 115
rownames 49
rpart 303
rstudent 274
rug 314
sample 114
save.image 5
scale_color_gradient 91
scale 118
scan 12
seq 12, 110
set.seed 28
setView 93
setwd 5
shapiro.test 206
solve 20
spineplot 74
st_transform 101
step 290
stream_in 178
substr 14
summary 37, 40
svd 20
svg 69
sweep 21, 118
t.test 207
table 23, 53
tapply 116
terrain.colors 314
theme_void 91
train 324
typeof 12
unique 49
unlist 24
updata.packages 30
var.test 204, 207
visTree 308
which.min 16
which 16, 45, 47, 48
wilcox.test 207
write.infile 24, 39
write.table 39
X11 71
xfig 69
xtabs 53, 203

Index

A

ACM 228–236
  contribution d'une modalité 234
  contribution d'une variable . 234
  coordonnée d'une modalité . 234
  description d'une variable . . 236
  description des axes 235
  données manquantes 236
  graphique des individus 231
  graphique des modalités 232
  inertie 230
  qualité de représentation d'une modalité 234
  regroupement de modalités . 230
  valeurs propres 230
  var. qualitative supp. 230
  var. quantitative supp. 230
  ventilation 230
ACP 214–222
  axes 3, 4, etc. 219
  contribution d'un individu . . 219
  coordonnée d'un individu . . . 219
  description des axes 220
  données manquantes 222
  habillage des points 219
  inertie 216
  non normée 216
  qualité de représentation d'un individu 219
  valeurs propres 216
  var. qualitative supp. 216
  var. quantitative supp. 216
AFC 223
  colonne supplémentaire 224
  inertie 225
  valeurs propres 225
AFCM . . . . . *voir* ACM
AFM 237–242
  description des axes 242
  données manquantes 242
Ajout à un graphique
  cercles, carrés, etc. 64
  droite 65
  flèches 64
  points 64
  polygone 64
  segments 64
Algèbre linéaire 19
Analyse de la covariance 282
Analyse de texte 383
Analyse de variance
  à deux facteurs 276
  avec interaction 276
  à un facteur 276
Analyse des Correspondances Multiples *voir* ACM
Analyse discriminante linéaire . . . 294
Analyse en Composantes Principales *voir* ACP
Analyse Factorielle des Correspondances *voir* AFC
Analyse Factorielle Multiple . . . . *voir* AFM
Analyse sémantique latente 389
Application shiny . . . . . *voir* Shiny
Arbres 302
Argument
  d'une fonction 27

valeur par défaut 28
Attribut 11
Axe
taille 72

B

Bases de données 162
Boucle
for 109
repeat 111
while 110
Boxplot . . . . *voir* Boîte à moustaches
Boîte à moustaches . 48, 58, 205, 278

C

CAH 244
Calcul parallèle 121
Camembert 64
Carte 90
Centralité 396
Centrer réduire 118
Classification
arbre hiérarchique 245
Ascendante Hiérarchique . . . 244
coupure de l'arbre 245
dendrogramme 245
description des classes . 247, 254
K-means 252
sur variables qualitatives . . . 249, 255
Colonne
modifier le nom 25
nombre de 21
Commander (package) 403
Comparaison de K moyennes . . . *voir* Analyse de variance à un facteur
Concaténation 21
par colonnes 49
par lignes 49
Contraintes
identifiantes 277
Conversion
logique en numérique 14
qualitative à quantitative 41
quantitative à qualitative 40
Convertir un objet 10
Couleur 95
arrière-plan 71
choix 71
dégradé chaleur 66
dégradé de gris 318
nom 71
numérotation 71
palette 71, 314
définition 71
par défaut 315
rgb 71
CRAN 3
Cube de données 21

D

Data-frame 25
conversion en matrice 26
création 25
extraction d'un 26
Date 55, 103
Décomposition
de Choleski 20
en valeurs singulières 20
Découpage en classes 41, 314
Deep learning 363
Device
exportation 68
Diagonalisation d'une matrice 20
Diagramme
en bandes 59
en barres 63
en bâtons 61
en secteurs 64
Dimension 21
Donnée(s)
aberrante (repérer) 47
incomplète 374
manquante 11, 374
résumé 37
Droite
ajout 65

pointillés 64, 65, 73
régression 267
tracé 65
épaisseur 73

E

Écart-type
estimation 197
Échantillonnage 114
Encodage 37, 384
détection 384
Entier 13
Estimation
coefficients
ANOVA 280
régression 266
densité
histogramme 62
noyaux 63
Exportation
de graphiques 68
de résultats 38

F

Facteur
conversion en numérique . 23, 41
création 41
définition 22
fusionner des niveaux 43
modalité . . . . . *voir* niveau
niveau
définition 22
fusion 43
nombre 22
ordre (changement) 43
renommer 42
supprimer 44
ordonnés 22
ventilation 44, 55
Fonction
appel 27, 112
argument 27
création 112
graphique 73, 74
résultat 28, 112
Fonction R 27
Fond de carte 91
Forêts aléatoires 331
Fouille de graphe 393
Fusionner . . . . . *voir* concaténer
selon une clef 51

G

GMM 256
Gradient
boosting 348
Graphe 393
Graphique(s)
3D
fonction 66
image 66
lignes de niveau 66
nuage de points 67
axe
création 60, 72
légendes 58, 60
bandes 59
barres 63
bâtons 61
carte shapefile 97, 100
conditionnels 73, 74
droite 65
fonction 65
fond de carte 90, 93
légende 72, 269, 284, 315
lignes 61, 65
limites 65, 73
multiples
conditionnels 74
sur une même page 69
nuage de points 58
orthonormé 72
paramètres 69, 71, 72
symboles 61

H

Histogramme 62

I

Identifiant 35
If then else 111
Image
    charger une 6
    sauver une 5
Importation 140
    données 35
    données volumineuses 140
    problème 37
Imputation
    données manquantes 374
    multiple 379
Individu
    aberrant (repérer) 47
    importer le nom 35
Installation
    de RStudio 3
    de R 3
    package 29
Interaction
    représentation graphique . . . 278
    variables qualitatives 276
Intervalle de confiance
    d'une moyenne 196
    d'une proportion 211
Inversion d'une matrice 20

J

Juxtaposer des tableaux
    en colonne 49
    en ligne 49

K

K-means 252

L

L 13, 146
Lasso 340
LDA 294
Légende 72, 269, 284, 315
Librairie de fonctions . . *voir* Package
Lignes (d'un objet)
    modifier les noms 25
    nombre de 21
Lignes (traits)
    épaisseur 73
    pointillés 65, 73
    type 61
Liste 23
    création 23
    extraction d'une 24

M

Manquantes (données) 374
Markdown 6
Matrice 17
    conversion en data-frame 26
    création 17
    décomposition de Choleski . . . 20
    décomposition en valeurs singulières 20
    déterminant 20
    diagonale 20
    diagonalisation 20
    identité 20
    inversion 20
    produit 20
    sélection dans 18
    transposée 20
Menu déroulant 403
Méthode de partitionnement . . . . *voir* K-means
Mise à jour
    package 30
    R 30
Modalité . . . . . *voir* Niveau
Mode d'un objet 10
Modèle de mélanges 256
Moyenne
    estimation 197
    intervalle de confiance 196
    test d'égalité 204

N

Niveau
    définition 22
    fusion 43

nombre 22
ordre (changement) 43
renommer 42
supprimer 44
Nom 11
colonne 49
des individus 36
ligne 49
liste 23
matrice 17, 25
Nuage de points 58, 265

O

Objet 8
de type null 10
affichage 9
atomique 11
changer de mode 10
créer 9
de type booléen 10
de type chaîne de caractères . 10
de type nombre complexe 10
de type nombre réel 10
mode 10
suppression 9

P

Package 28
caret 323
cluster 245
data.table 139
doParallel 125
FactoMineR . . 216, 224, 230, 238
Factoshiny 216
foreach 124
gbm 349
lattice 67
MASS 296
mclust 258
missMDA 378
parallel 122
Rcmdr 403
rpart 302
shiny 128
tidyverse 161
tree 302
chargement 29
installation 29
mise à jour 30
utilisation 29
Palette . . . . . *voir* Couleur
Partition . . . . . *voir* K-means
Pipe 154
PLS 312
Points
ajout 64
nuage 58
tracé 61
Produit matriciel 20
Programmation
boucle 109
condition 111
Proportion
intervalle de confiance 211
test d'égalité 210
test de conformité 210

R

R Commander (package) 403
Réel 13
Régression
elastic net 347
lasso 340
logistique 287
prévision 291
sélection de variables 290
PLS 312
prévision 317
résidus 316
ridge 340
Régression linéaire
choix de variables 273
coefficient R^2 267
Cp de Mallows 273
critère BIC 273
droite de régression 267
estimation des paramètres . . 266

intervalle de confiance d'une prévision 269
modèle sans constante 267
multiple 270
prévision 268, 275
résidus 268, 274
résidus studentisés 268, 274
simple 264
variance des résidus 267
Remplacer un motif 14
Réseau de neurones 362
Ridge 340
RMarkdown 6
RStudio 402
aide 8
environnement 3
importation de données 38
installation 3
installation de packages 29
mise à jour 5
mise à jour de packages 31
suppression d'objets 9

S

Sac de mots 383
Sauvegarde d'objets dans une image 5
Sélection dans un vecteur 15
Sélectionner sous chaîne 14
Session 5
Shiny 128
Studio . . . . . *voir* RStudio
SVM 354
Symbole graphique 61
Système d'équations 20

T

Tableau croisé 52
Tableau à 3 dimensions 21
Taille 72
axes 72
fontes 64, 72
points 61, 72
titre 72
Test
d'égalité de moyennes 204
d'égalité de proportions 210
d'égalité de variances . . 204, 207
de Bartlett 208, 281
de Friedman 281
de normalité 206
de Welch 207
du χ^2 d'indépendance 199
Texte 383
tf-idf 389
Tibble 151
Tidyverse 151
Titre 62
Tracé de
cercles, carrés, etc. 64, 94
flèches 64
fonction 65
lignes 61
points 58, 61, 265
polygone 64, 97
segments 64
Transposer une matrice 20
Type
d'un objet 12
double 13
entier 13

V

Valeur manquante
définition 11
importation 36
imputation 374
rechercher/remplacer 45
Variance
test d'égalité 207
Vecteur 11
sélection 15
Ventilation
facteur 44, 55, 56
Visualisation interactive 400

W

Web scrapping 180

PRATIQUE DE LA STATISTIQUE

R pour la statistique et la science des données

Le logiciel R est un outil incontournable de statistique, de visualisation de données et de science des données tant dans le monde universitaire que dans celui de l'entreprise. Ceci s'explique par ses trois principales qualités : il est gratuit, très complet et en essor permanent. Récemment, il a su s'adapter pour entrer dans l'ère du big-data et permettre de recueillir et traiter des données hétérogènes et de très grandes dimensions (issues du Web, données textuelles, etc.).

Ce livre s'articule en deux grandes parties : la première est centrée sur le fonctionnement du logiciel R tandis que la seconde met en oeuvre une trentaine de méthodes statistiques au travers de fiches. Ces fiches sont chacune basées sur un exemple concret et balayent un large spectre de techniques pour traiter des données.

Ce livre s'adresse aux débutants comme aux utilisateurs réguliers de R. Il leur permettra de réaliser rapidement des graphiques et des traitements statistiques simples ou élaborés.

Les auteurs

Pierre-André Cornillon,
Université Rennes 2

Arnaud Guyader,
Sorbonne Université

François Husson,
Agrocampus-Ouest

Nicolas Jégou,
Université Rennes 2

Julie Josse,
École polytechnique

Nicolas Klutchnikoff,
Université Rennes 2

Erwan Le Pennec,
École polytechnique

Eric Matzner-Løber,
Cepe ENSAE

Laurent Rouvière,
Université Rennes 2

Benoît Thieurmel,
Datastorm

Presses Universitaires de Rennes
www.pur-editions.fr

ISBN 978-2-7535-7573-8
25 €