

Ewen Harrison and Riinu Ots

HealthyR: R for health data analysis

Never trust a data scientist - they are always plotting.

Contents

List of Tables	xiii
List of Figures	xvii
Preface	xxiii
About the Authors	xxvii
I Data wrangling and visualisation	1
1 Why we love R	3
1.1 Help, what's a script?	4
1.2 What is RStudio?	6
1.3 Getting started	7
2 R Basics	9
2.1 Getting help	9
2.2 Objects and functions	9
2.3 Working with Objects	14
2.4 Pipe - %>%	16
2.4.1 When pipe sends data to the wrong place: use , data = . to direct it	17
2.5 Reading data into R	18
2.5.1 Reading in the Global Burden of Disease example dataset (short version)	21
2.6 Operators for filtering data	24
2.6.1 Worked examples	27
2.7 The combine function: c()	28
2.8 Missing values (NAs) and filters	29
2.9 Variable types and why we care	33
2.10 Numeric variables (continuous)	35

2.11	Character variables (categorical, IDs, free text)	37
2.12	Factor variables (categorical)	39
2.13	Date/time variables	40
2.14	Creating new columns - <code>mutate()</code>	44
2.14.1	Worked example/exercise	47
2.15	Conditional calculations - <code>if_else()</code>	48
2.16	Create labels - <code>paste()</code>	49
2.17	Joining multiple datasets	50
2.17.1	Further notes about the joins	52
3	Summarising data	55
3.0.1	Dataset: Global Burden of Disease (year, cause, sex, income, deaths)	55
3.1	Aggregating: <code>group_by()</code> , <code>summarise()</code>	58
3.2	Add new columns: <code>mutate()</code>	59
3.2.1	percentages formatting: <code>percent()</code>	60
3.3	<code>summarise()</code> VS <code>mutate()</code>	61
3.4	Common arithmetic functions - <code>sum()</code> , <code>mean()</code> , <code>median()</code> , etc.	64
3.5	Reshaping data - long vs wide format	65
3.5.1	<code>spread()</code> values from rows into columns	67
3.5.2	<code>gather()</code> values from columns to rows	69
3.6	<code>select()</code> columns	72
3.6.1	using select helpers in other functions	74
3.7	<code>arrange()</code> rows	75
3.7.1	factor levels	76
3.8	Exercise - <code>spread()</code>	78
3.9	Exercise - <code>group_by()</code> , <code>summarise()</code>	78
3.10	Exercise - <code>full_join()</code> , <code>percent()</code>	81
3.11	Exercise - <code>mutate()</code> , <code>summarise()</code>	82
3.12	Exercise - <code>filter()</code> , <code>summarise()</code> , <code>spread()</code>	83
4	Different types of plots	85
4.1	Data - gapminder	87
4.2	Anatomy of ggplot explained	88
4.3	Set your theme - grey vs white	94
4.4	Scatter plots/bubble plots	95

<i>Contents</i>	v
4.5 Line plots/time series plots	96
4.5.1 Exercise	97
4.6 Bar plots	98
4.6.1 colour vs fill	101
4.6.2 Proportions	101
4.6.3 Exercise	102
4.7 Histograms	103
4.8 Box plots	104
4.9 Multiple geoms, multiple <code>aes()</code> ?	105
4.9.1 Worked example - three geoms together . .	107
4.10 All other types of plots	109
4.11 Solutions	110
4.12 Extra: Advanced examples	110
5 Fine tuning plots	113
5.1 Data and initial plot	113
5.2 Scales	114
5.2.1 Logarithmic	114
5.2.2 Expand limits	114
5.2.3 Zoom in	115
5.2.4 Exercise	116
5.2.5 Axis ticks	116
5.2.6 Swap the axes	117
5.3 Colours	117
5.3.1 Using the Brewer palettes:	117
5.3.2 Legend title	117
5.3.3 Choosing colours manually	118
5.4 Titles and labels	120
5.4.1 Annotation	120
5.4.2 Annotation with a superscript and a variable	121
5.5 Text size	122
5.5.1 Legend position	123
5.6 Saving your plot	124
II Data analysis	125
6 Working with continuous outcome variables	129

6.1	Continuous data	129
6.2	The Question	130
6.3	Get the data	130
6.4	Check the data	130
6.5	Plot the data	132
6.5.1	Histogram	133
6.5.2	Q-Q plot	133
6.5.3	Boxplot	135
6.6	Compare the means of two groups	137
6.6.1	T-test	137
6.6.2	Two-sample <i>t</i> -tests	137
6.6.3	When pipe sends data to the wrong place: use , data = . to direct it	139
6.6.4	Paired <i>t</i> -tests	139
6.7	Compare the mean of one group	142
6.7.1	One sample <i>t</i> -tests	142
6.8	Compare the means of more than two groups	143
6.8.1	Plot the data	144
6.8.2	ANOVA	144
6.8.3	Assumptions	145
6.8.4	Pairwise testing and multiple comparisons	146
6.9	Non-parametric data	149
6.9.1	Transforming data	149
6.9.2	Non-parametric test for comparing two groups	151
6.9.3	Non-parametric test for comparing more than two groups	152
6.10	Finalfit approach	153
6.11	Conclusions	153
6.12	Exercises	154
6.12.1	Exercise 1	154
6.12.2	Exercise 2	154
6.12.3	Exercise 3	154
6.12.4	Exercise 4	154
6.13	Exercise solutions	154

7 Linear regression**159**

<i>Contents</i>	vii
7.1 Regression	159
7.1.1 The Question (1)	160
7.1.2 Fitting a regression line	160
7.1.3 When the line fits well	162
7.1.4 The fitted line and the linear equation . .	164
7.1.5 Effect modification	166
7.1.6 R-squared and model fit	171
7.1.7 Confounding	171
7.1.8 Summary	172
7.2 Fitting simple models	173
7.2.1 The Question (2)	173
7.2.2 Get the data	174
7.2.3 Check the data	174
7.2.4 Plot the data	174
7.2.5 Simple linear regression	175
7.2.6 Multivariable linear regression	179
7.2.7 Check assumptions	184
7.3 Fitting more complex models	185
7.3.1 The Question (3)	185
7.3.2 Model fitting principles	185
7.3.3 AIC	187
7.3.4 Get the data	187
7.3.5 Check the data	187
7.3.6 Plot the data	188
7.3.7 Linear regression with finalfit	189
7.3.8 Summary	196
8 Working with categorical outcome variables	197
8.1 Factors	197
8.2 The Question	198
8.3 Get the data	198
8.4 Check the data	198
8.5 Recode the data	200
8.6 Should I convert a continuous variable to a categorical variable?	201
8.6.1 Equal intervals vs quantiles	203
8.7 Plot the data	205

8.8	Group factor levels together - <code>fct_collapse()</code>	208
8.9	Change the order of values within a factor - <code>fct_relevel()</code>	208
8.10	Summarising factors with Finalfit	209
8.11	Pearson's chi-squared and Fisher's exact tests	210
8.11.1	Base R	211
8.12	Fisher's exact test	213
8.13	Chi-squared / Fisher's exact test using Finalfit	214
8.14	Exercise	216
8.15	Exercise	217
8.16	Exercise	217
8.17	Exercise	218
9	Logistic regression	219
9.1	Generalised linear modelling	219
9.2	Binary logistic regression	219
9.2.1	The Question (1)	220
9.2.2	Odds and probabilities	221
9.2.3	Odds ratios	222
9.2.4	Fitting a regression line	223
9.2.5	The fitted line and the logistic regression equation	224
9.2.6	Effect modification and confounding	226
9.3	Data preparation and exploratory analysis	229
9.3.1	The Question (2)	229
9.3.2	Get the data	230
9.3.3	Check the data	230
9.3.4	Recode the data	230
9.3.5	Plot the data	233
9.3.6	Tabulate data	235
9.4	Model assumptions	236
9.4.1	Linearity of continuous variables to the re- sponse	236
9.4.2	Multicollinearity	237
9.5	Fitting logistic regression models in base R	241
9.6	Modelling strategy for binary outcomes	244
9.7	Fitting logistic regression models with Finalfit	244

<i>Contents</i>	ix
9.7.1 Criterion-based model fitting	245
9.8 Model fitting	246
9.8.1 Odds ratio plot	252
9.9 Correlated groups of observations	254
9.9.1 Simulate data	254
9.9.2 Plot the data	255
9.9.3 Mixed effects models in base R	256
9.9.4 Exercise	260
10 Time-to-event data and survival	261
10.1 The Question	261
10.2 Get and check data	262
10.3 Death status	262
10.4 Time and censoring	263
10.5 Recode the data	263
10.6 Kaplan Meier survival estimator	264
10.6.1 KM analysis for whole cohort	264
10.6.2 Model	264
10.6.3 Life table	265
10.7 Kaplan Meier plot	265
10.8 Cox-proportional hazards regression	266
10.8.1 Univariable and multivariable models . . .	267
10.8.2 Reduced model	268
10.8.3 Testing for proportional hazards	268
10.8.4 Stratified models	270
10.8.5 Correlated groups of observations	270
10.8.6 Hazard ratio plot	272
10.9 Competing risks regression	273
10.10 Summary	274
10.11 Exercise	275
10.11.1 Exercise	276
10.12 Dates in R	277
10.12.1 Converting dates to survival time	277
10.13 Solutions	278
III Workflow	281
11 Notebooks and Markdown	285

11.1 What is a Notebook?	285
11.2 What is Markdown?	286
11.3 What is the difference between a Notebook and a Markdown file?	287
11.4 Notebook vs HTML vs PDF vs Word	288
11.5 The anatomy of a Notebook / R Markdown file . .	288
11.5.1 YAML header	289
11.5.2 R code chunks	289
11.5.3 Setting default chunk options	291
11.5.4 Setting default figure options	292
11.5.5 Markdown elements	292
11.6 Output	292
11.6.1 Running code and chunks	292
11.6.2 Knitting	293
11.7 File structure and workflow	295
11.7.1 Data cleaning	295
12 Exporting and reporting	299
12.1 Working in a <code>.r</code> file	299
12.2 Demographics table	300
12.3 Logistic regression table	301
12.4 Odds ratio plot	302
12.5 MS Word via knitr/R Markdown	303
12.6 Create Word template file	304
12.7 PDF via knitr/R Markdown	306
12.8 Working in a <code>.Rmd</code> file	309
13 Version control	311
13.1 Setup Git on RStudio and Associate with GitHub	311
13.2 Create an SSH RSA key and add to your GitHub account	311
13.3 Create a project in RStudio and commit a file .	312
13.4 Create a new repository on GitHub and link to RStudio project	314
13.5 Clone an existing GitHub project to new RStudio project	314
14 Missing data	319

<i>Contents</i>	xi
14.1 The problem of missing data	319
14.2 Some confusing terminology	320
14.2.1 Missing completely at random (MCAR) .	320
14.2.2 Missing at random (MAR)	320
14.2.3 Missing not at random (MNAR)	320
14.3 Ensure your data are coded correctly: ff_glimpse	321
14.4 The Question	321
14.5 2. Identify missing values in each variable: missing_plot	323
14.6 3. Look for patterns of missingness: missing_pattern	324
14.6.1 Make sure you include missing data in demographics tables	326
14.7 4. Check for associations between missing and observed data: missing_pairs missing_compare .	326
14.8 For those who like an omnibus test	330
14.9 5. Decide how to handle missing data	331
14.10 MCAR, MAR, or MNAR	332
14.10.1 MCAR vs MAR	332
14.10.2 MCAR vs MAR	333
14.10.3 MNAR vs MAR	337
15 Encryption	339
15.1 Safe practice	339
15.2 Encryptr package	340
15.3 Get the package	340
15.4 Get the data	341
15.5 Generate private/public keys	341
15.6 Encrypt columns of data	342
15.7 Decrypt specific information only	343
15.8 Using a lookup table	343
15.9 Encrypting a file	344
15.10 Ciphertexts are no matchable	345
15.11 Providing a public key	345
15.12 Use in clinical trials	346
15.13 Caution	346

16 RStudio settings, good practise	347
16.1 Installation	347
16.1.1 R	347
16.1.2 RStudio	347
16.1.3 R packages	347
16.1.4 Troubleshooting 3.3	348
16.2 Script vs Console	349
16.3 Starting with a blank canvas	349
Bibliography	351
Index	353

List of Tables

2.1	Example of a table (=tibble once read into R), including missing values denoted NA (Not applicable/Not available)	10
2.2	Deaths per year from three broad disease categories (short version of the Global Burden of Disease example dataset)	22
2.3	Filtering operators	27
3.1	Deaths per year (2017) from three broad disease categories, sex, and World Bank county-level income groups	56
3.2	Global Burden of Disease data in human-readable wide format. This is not tidy data.	65
3.3	Global Burden of Disease data in analysis-friendly long format. This is tidy data.	66
3.4	Exercise: putting the cause variable into the wide format using spread.	78
6.1	Gapminder dataset, ff_glimpse: continuous	132
6.2	Gapminder dataset, ff_glimpse: categorical	132
6.3	Transformations that can be applied to skewed data	149
6.4	Life expectancy, population and GDPperCap in Africa 1982 v 2007	153
7.1	WCGS data, ff_glimpse: continuous	188
7.2	WCGS data, ff_glimpse: categorical	188
7.3	Linear regression: Systolic blood pressure by personality type.	190
7.4	Model metrics: Systolic blood pressure by personality type.	190

7.5	Multivariable linear regression: Systolic blood pressure by personality type and weight.	191
7.6	Multivariable linear regression metrics: Systolic blood pressure by personality type and weight.	191
7.7	Multivariable linear regression: Systolic blood pressure by available explanatory variables.	192
7.8	Model metrics: Systolic blood pressure by available explanatory variables.	192
7.9	Multivariable linear regression: Systolic blood pressure using BMI.	193
7.10	Model metrics: Systolic blood pressure using BMI.	194
7.11	Multivariable linear regression: Systolic blood pressure by available explanatory variables and reduced model.	194
7.12	Model metrics: Systolic blood pressure by available explanatory variables (top) with reduced model (bottom).	194
8.1	Two-by-two table with Finalfit: Died with melanoma by tumour ulceration status.	209
8.2	Multiple variables by outcome: Outcome after surgeryr for melanoma by patient and disease factors.	210
8.3	Two-by-two table with chi-squared test using final fit: Outcome after surgery for melanoma by tumour ulceration status.	214
8.4	Multiple variables by outcome with hypothesis tests: Outcome after surgery for melanoma by patient and disease factors (chi-squared test).	215
8.5	Multiple variables by outcome with hypothesis tests: Outcome after surgery for melanoma by patient and disease factors (Fisher's exact test).	216
8.6	Multiple variables by outcome with hypothesis tests: options including missing data, rounding, and labels.	216

9.1	Multiple variables by explanatory variable of interest: Malignant melanoma ulceration by patient and disease variables.	235
9.2	Univariable logistic regression: 5-year survival from malignant melanoma by tumour ulceration (fit 1).	245
9.3	Model metrics: 5-year survival from malignant melanoma by tumour ulceration (fit 1).	245
9.4	Multivariable logistic regression: 5-year survival from malignant melanoma (fit 2).	247
9.5	Model metrics: 5-year survival from malignant melanoma (fit 2).	247
9.6	Multivariable logistic regression: using ‘cut’ to convert a continuous variable as a factor (fit 3).	248
9.7	Model metrics: using ‘cut’ to convert a continuous variable as a factor (fit 3).	248
9.8	Multivariable logistic regression: including a quadratic term (fit 4).	249
9.9	Model metrics: including a quadratic term (fit 4).	249
9.10	Multivariable logistic regression model: comparing a reduced model in one table (fit 5).	250
9.11	Model metrics: comparing a reduced model in one table (fit 5).	250
9.12	Multivariable logistic regression: further reducing the model (fit 6).	251
9.13	Model metrics: further reducing the model (fit 6).	251
9.14	Multivariable logistic regression: including an interaction term (fit 7).	252
9.15	Multivariable logistic regression: final model (fit 8).	253
9.16	Model metrics: final model (fit 8).	253
9.17	Multilevel (mixed effects) logistic regression.	258
9.18	Model metrics: multilevel (mixed effects) logistic regression.	258
10.1	Univariable and multivariable Cox Proportional Hazards: Overall survival following surgery for melanoma by patient and tumour variables.	267

10.2	Univariable and multivariable Cox Proportional Hazards: Overall survival following surgery for melanoma by patient and tumour variables (tidied).	268
10.3	Cox Proportional Hazards: Overall survival following surgery for melanoma with reduced model.	268
10.4	Cox Proportional Hazards: Overall survival following surgery for melanoma stratified by year of surgery.	270
10.5	Cox Proportional Hazards: Overall survival following surgery for melanoma (frailty model)	272
10.6	Cox Proportional Hazards and competing risks regression combined	274
11.1	Chunk output options when knitting an R Markdown file.	291
12.1	Exporting 'table 1': Tumour differentiation by patient and disease factors.	300
12.2	Exporting 'table 1': adjusting labels and output.	301
12.3	Exporting a regression results table.	302
14.1	Simulated missing completely at random (MCAR) and missing at random (MAR) dataset.	327
14.2	Missing data comparison: Smoking (MCAR).	330
14.3	Missing data comparison: Smoking (MAR).	330
14.4	Regression analysis with missing data: listwise deletion.	332
14.5	Regression analysis with missing data: multiple imputation using 'mice()'.	336
14.6	Regression analysis with missing data: explicitly modelling missing data.	337

List of Figures

1.1	R logo, © 2016 The R Foundation.	3
1.2	An example R script.	4
1.3	We use RStudio to work with R.	6
2.1	This is not a pipe. René Magritte inspired artwork by Stefan Milton Bache (creator of <code>%>%</code> in R). Image source: https://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html	17
2.2	View or import a data file.	18
2.3	Import: Some of the special settings your data file might have.	19
2.4	After using the Import Dataset window, copy-paste the resulting code into your script.	19
2.5	Causes of death from the Global Burden of Disease dataset (Table 2.2). Data on (B) is the same as (A) but stacked to show the total (sum) of all causes.	23
3.1	Global Burden of Disease data with subgroups: cause, sex, World Bank income group.	57
3.2	Same data in the long ('tidy', necessary for efficient analysis) and wide (easier for human-readability/presentation/manual data entry) formats. TODO: replace with updated data.	66
4.1	Example steps for building and modifying a ggplot. (1) initialising the canvas, defining variables, (2) adding points, (3) colouring points by continent, (4) changing point type, (5) facetting, (6) changing the plot theme and the scale of the x variable.	86
4.2	A stip plot using <code>geom_point()</code>	90

4.3	A selection of shapes for plotting. Shapes 0, 1, and 2 are hollow, whereas for shapes 21, 22, and 23 we can define both a colour and a fill (for these shapes, colour is the border around the fill).	91
4.4	Using a filtering condition (e.g., population > 50 million) directly inside a <code>facet_wrap()</code>	92
4.5	Some of the built-in ggplot themes (1) default (2) <code>theme_bw()</code> , (3) <code>theme_dark()</code> , (4) <code>theme_classic()</code>	93
4.6	Turn the scatter plot from Figure 4.1:(2) to a bubble plot by (1) adding <code>size = pop</code> inside the <code>aes()</code> , (2) make the points hollow and transparent.	95
4.7	The ‘zig-zag plot’ is a common mistake: using <code>geom_line()</code> (1) without a <code>group = aesthetic</code> , (2) after adding <code>group = country</code>	97
4.8	Bar plots using <code>geom_col()</code> : (1) using the code example, (2) same plot but with <code>+ coord_cartesian(ylim=c(79, 81))</code> to manipulate the scale into something a lot more dramatic.	99
4.9	<code>geom_bar()</code> counts up the number of observations for each group. (1) <code>gapminder2007 %>% ggplot(aes(x = continent)) + geom_bar()</code> , (2) same + a little bit of magic to reveal the underlying data.	100
4.10	Barplot Exercise. Life expectancies in European countries in year 2007 from the Gapminder dataset.	103
4.11	Multiple geoms together. (1) <code>geom_boxplot() + geom_point()</code> , (2) <code>geom_boxplot() + geom_jitter()</code> , (3) colour aesthetic inside <code>ggplot(aes())</code> , (4) colour aesthetic inside <code>geom_jitter(aes())</code>	106
6.1	Histogram: country life expectancy by continent and year	133
6.2	Q-Q plot: country life expectancy by continent and year	134
6.3	Boxplot: country life expectancy by continent and year	136

6.4	Boxplot with jitter points: country life expectancy by continent and year	136
6.5	Line plot: Change in life expectancy in Asian countries from 2002 to 2007	140
6.6	Boxplot: Life expectancy in selected continents for 2007	144
6.7	Diagnostic plots: ANOVA model of life expectancy by continent for 2007	146
6.8	Histogram: Log transformation of life expectancy for countries in Africa 2002	150
6.9	Panels plots: histogram, Q-Q, boxplot for life expectancy in Africa 1992 v 2007	152
7.1	The anatomy of a regression plot.	161
7.2	How a regression line is fitted.	163
7.3	Regression diagnostics. Does this also appear in the contents. What about this?	165
7.4	Linking the fitted line, regression equation and R output.	167
7.5	Causal pathways, effect modification and confounding.	168
7.6	Multivariable linear regression with additive and multiplicative effect modification.	170
7.7	Multivariable linear regression with confounding of coffee drinking by smoking.	173
7.8	Scatterplot with fitted line plot: Life expectancy by year in European countries	175
7.9	Scatterplot: Life expectancy by year Turkey and Europe.	176
7.10	Scatter and line plot. Life expectancy in Turkey and the UK - univariable fit.	181
7.11	Scatter and line plot. Life expectancy in Turkey and the UK - multivariable additive fit.	182
7.12	Scatter and line plot. Life expectancy in Turkey and the UK - multivariable multiplicative fit.	183
7.13	Scatter and line plot. Systolic blood pressure by weight and personality type.	189

7.14 Diagnostic plots: linear regression model of systolic blood pressure.	195
8.1 cut a continuous variable into a categorical variable.	203
8.2 Bar chart: outcome after surgery for patients with ulcerated melanoma.	205
8.3 Bar chart: outcome after surgery for patients with ulcerated melanoma, reversed levels.	206
8.4 Facetted bar plot: outcome after surgery for patients with ulcerated melanoma aggregated by sex and age.	207
9.1 Probability vs odds.	221
9.2 Odds ratios.	222
9.3 A logistic regression model of life-time cardiovascular event occurrence by coffee consumption stratified by smoking (simulated data). Fitted lines plotted on the log-odds scale (A) and probability scale (B). *lines are straight when no polynomials or splines are included in regression.	225
9.4 Linking the logistic regression fitted line, equation and R output.	227
9.5 Multivariable logistic regression with additive and multiplicative effect modification.	228
9.6 Multivariable logistic regression with interaction term. The exponential of the interaction term is a ratio-of-odds ratios (ROR).	229
9.7 CAPTION	234
9.8 Odds ratio plot	253
10.1 Hazard ratio plot	272
11.1 Traditional versus literate programming using Notebooks.	286
11.2 RStudio Markdown quick reference guide.	287
11.3 The Anatomy of a Notebook/Markdown file. Input (left) and output (right)	290

11.4 Chunk and document options in Notebook/Markdown files.	294
12.1 Odds ratio plot.	302
12.2 Knitting to Microsoft Word from R Markdown. Before (A) and after (B) adjustment.	305
12.3 Knitting to Microsoft Word from R Markdown. Before (A) and after (B) adjustment.	307
12.4 Writing a final report in a Markdown document.	310
13.1 SSH.	312
13.2 CAPTION	313
13.3 CAPTION	315
13.4 CAPTION	316
13.5 CAPTION	317
13.6 CAPTION	318
14.1 Missing data matrix with <code>missing_pairs()</code>	328
14.2 Missing data matrix with <code>missing_pairs(position = 'fill')</code>	329



Preface

Version 0.3.1

Contributors: Riinu Ots, Ewen Harrison, Tom Drake, Peter Hall, Kenneth McLean.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/us/>

Why read this book

We are drowning in information but starved for knowledge.
John Naisbitt

In this age of information, the manipulation, analysis and interpretation of data has become paramount. Nowhere more so than in the delivery of healthcare. From the understanding of disease and the development of new treatments, to the diagnosis and management of individual patients, the use of data and technology is now an integral part of the business of healthcare.

Those working in healthcare interact daily with data, often without realising it. The conversion of this avalanche of information to

useful knowledge is essential for high quality patient care. An important part of this information revolution is the opportunity for everybody to become involved in data analysis. This democratisation of data analysis is driven in part by the open source software movement – no longer do we require expensive specialised software to do this.

The statistical programming language, R, is firmly at the heart of this!

This book will take an individual with little or no experience in data analysis all the way through to performing sophisticated analyses. We emphasise the importance of understanding the underlying data with liberal use of plotting, rather than relying on opaque and possibly poorly understand statistical tests. There are numerous examples included that can be adapted for your own data, together with our own R packages with easy-to-use functions.

We have a lot of fun teaching this course and focus on making the material as accessible as possible. We banish equations in favour of code and use examples rather than lengthy explanations. We are grateful to the many individuals and students who have helped refine these and welcome suggestions and bug reports via <https://github.com/SurgicalInformatics>.

Ewen Harrison and Riinu Ots

August 2019

Structure of the book

Chapter 2 introduces a new topic, and ...

Acknowledgments

A lot of people helped us when we were writing the book.

This is how to Write on the right



About the Authors

Ewen is a surgeon and Riinu is a physicist. And they're both data scientists too. They dabble with a few programming languages and are generally all over technology. But they are most enthusiastic about the R statistical programming language, and know that you will be too. They work at The University of Edinburgh and have taught R to hundreds of healthcare professionals and researchers.

They believe that the first introduction to R and statistical programming should be relatively jargon free and outcome oriented (get those pretty plots out). The understanding of complicated concepts will come over time with practise and experience, not by us defining the history of computing bit-by-byte, or by presenting the equations of every statistical test (although Ewen has sneaked a few equations in).



Part I

Data wrangling and visualisation



1

Why we love R

We are extremely pleased that you have picked up this book to learn R for health data analysis. Even if you're already familiar with the R language, you will probably find new approaches here as we make the most of the latest R packages and tools including some we've developed ourselves. Those already familiar with R are encouraged to still skim through the first two chapters to familiarise with the style of R we recommend.

Here are the main reasons we love R are:

- R is versatile and powerful - use it for
 - graphics
 - all the statistics you can dream of
 - machine learning, deep learning
 - automated reports
 - websites
 - books (yes, R can be used to make whole websites or books, this one is written in R)
- R scripts can be reused - gives you efficiency and reproducibility
- It is free to use by anyone, anywhere



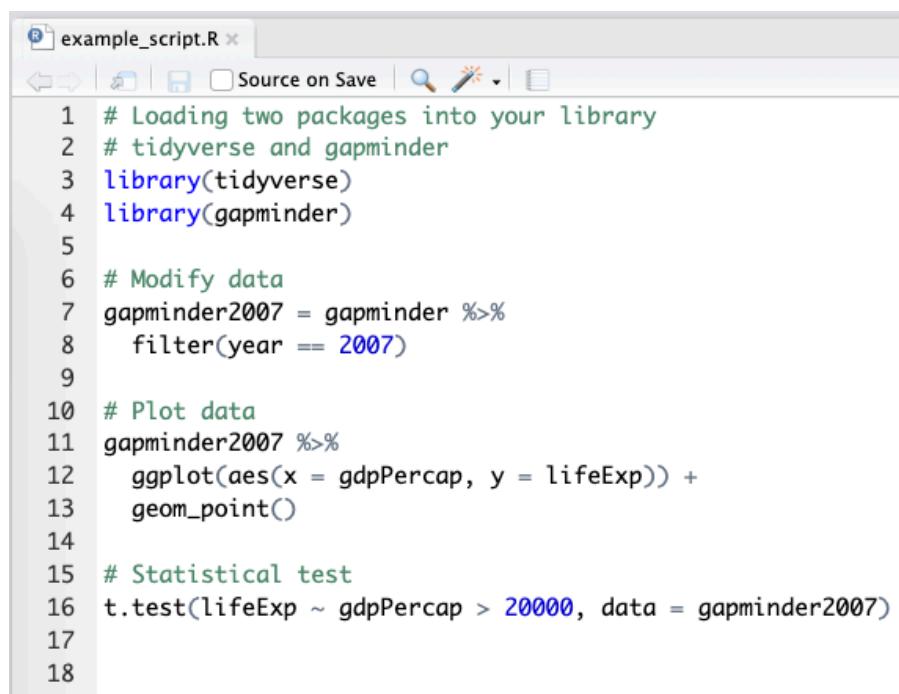
FIGURE 1.1: R logo, © 2016 The R Foundation.

1.1 Help, what's a script?

A script is a list of instructions. It is just a text file and no special software is required to view one. An example R script is shown in Figure 1.2.

Don't panic! The only thing you need to understand at this point is that what you're looking at is a list of instructions written in the R language.

You should also notice that some parts of the script looks like normal English. These are the lines that start with a `#` and they are called “comments”. We can (and should) include these comments in everything we do. These are notes of what we were doing, both for colleagues as well as our future self.



The screenshot shows a code editor window titled "example_script.R". The window has a standard OS X-style interface with a toolbar at the top featuring icons for back, forward, save, and search. The main area contains 18 numbered lines of R code:

```
1 # Loading two packages into your library
2 # tidyverse and gapminder
3 library(tidyverse)
4 library(gapminder)
5
6 # Modify data
7 gapminder2007 = gapminder %>%
8   filter(year == 2007)
9
10 # Plot data
11 gapminder2007 %>%
12   ggplot(aes(x = gdpPerCap, y = lifeExp)) +
13   geom_point()
14
15 # Statistical test
16 t.test(lifeExp ~ gdpPerCap > 20000, data = gapminder2007)
17
18
```

FIGURE 1.2: An example R script.

Lines that do not start with a `#` are R code. This is where the number crunching really happens. We will cover the details of this R code in the next few chapters, the purpose of this chapter is to describe some of the terminology as well as the interface and tools we use.

For the impatient:

- We interface R using RStudio;
- We use the **tidyverse** packages that are a substantial extension to base R functionality (we repeat: extension, not replacement).

Even though R is a language, don't think that after reading this book you should be able to open a blank file and just start typing in R code like an evil computer genius from a movie. This is not what real world programming looks like.

Firstly, you should be copy-pasting and adapting existing R code examples - whether from this book or later from your own previous work. Re-writing everything from scratch is not efficient. Yes, you will understand and eventually remember a lot of it. But to spend time memorising very specific things that can easily be looked up and copied is simply not necessary.

Secondly, R is an interactive language. Meaning that we "run" R code line by line and get immediate feedback. We would never write a whole script without trying everything out as we go along.

Thirdly, do not worry about making mistakes. Celebrate them! The whole point of R and reproducibility is that manipulations are not applied directly on a dataset but a copy of it. And that everything is in a script - so if we do make a wrong move (e.g. accidentally overwrite or remove some data) we can always reload it, rerun the steps that worked well and continue figuring our where we went wrong at the end. And since all of these steps are written down in a script, R will redo everything with a single push of a button. You do not have to redo anything, that's what R is for.

1.2 What is RStudio?

RStudio is a free program that makes working with R easier. An example screen shot of RStudio is shown in Figure 1.3. We have already introduced the top-left pane - the **Script**.

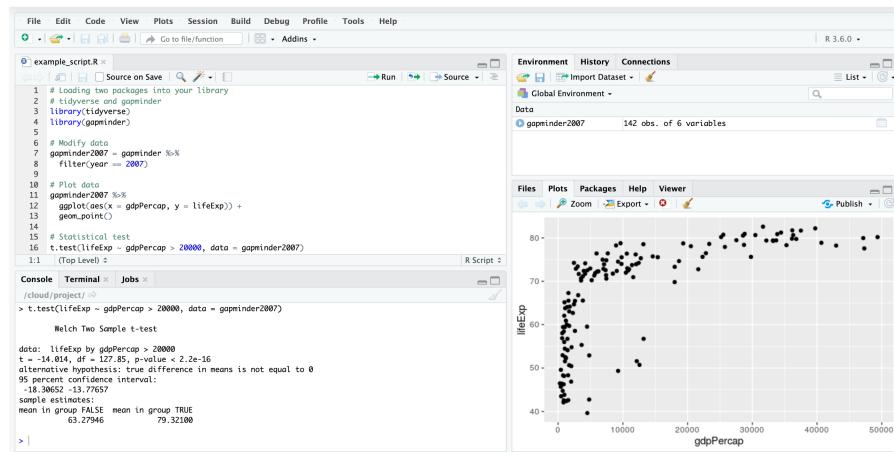


FIGURE 1.3: We use RStudio to work with R.

Now, look at the little **Run** and **Source** buttons at the top-right corner of the script pane. Clicking **Run** executes a line of R code. Clicking **Source** executes all lines of R code in the script (it is essentially ‘Run all lines’). When you run R code, it gets sent to the **Console** which is the bottom-left panel. This is where R really lives.

Keyboard Shortcuts!

Run line: Ctrl+Enter

Run all lines (Source): Ctrl+Shift+Enter

(On a Mac, both Ctrl or Command work)

The Console is also where R speaks to us. When we're lucky, we get results in there - in this example the results of a *t*-test (last line of the script). When we're less lucky, this where also where Errors or Warnings appear.

R Errors are a lot less scary than they seem! Yes, if you're using using a regular computer program where all you do is click on some buttons, then getting a proper red error that stops everything is quite unusual. But in programming, Errors are just a way for R to communicate with us. We see Errors in our own work every single day, they are very normal and do not mean that everything is wrong or that you should give up. Try to reframe the word Error to mean "feedback", as in "Hello, this is R. I can't continue, this is the feedback I am giving you." The most common Errors you'll see are along the lines of "Error: something not found". This almost always means there's a typo or you've misspelled something. Furthermore, R is case sensitive so capitalisation matters (variable name `lifeExp` is not the same as `liffeexp`).

The Console can only print text, so any plots your create in your script appear in the **Plots** pane (bottom-right).

Similarly, datasets that you've loaded or created appear in the **Environment** tab. When you click on a dataset, it pops up in a really nice and fast viewer. This means you can have a look and scroll through your rows and columns the same way you would in a spreadsheet.

1.3 Getting started

To start using R, you should do these two things:

- Install R (from <https://www.r-project.org/>)
- Install RStudio Desktop (from <https://www.rstudio.com/>)

When you first open up RStudio, you'll also want to install some extra packages to extend the base R functionality. You can do this

in the **Packages** tab (next to the Plots tab in the bottom-right in Figure 1.3).

A lot of the functionality introduced in this book comes from the **tidyverse** family of R packages (<http://tidyverse.org>). So when you go to **Packages**, click **Install**, type in **tidyverse**, a whole collection of useful and modern packages will be installed.

Even though you've installed the **tidyverse** packages, you'll still need to tell R when you're about to use them. We usually include `library(tidyverse)` at the top of every script:

```
library(tidyverse)

## -- Attaching packages -----
## v ggplot2 3.2.1     v purrr    0.3.2
## v tibble  2.1.3     v dplyr    0.8.3
## v tidyverse 0.8.3    v stringr  1.4.0
## v readr   1.3.1     vforcats  0.4.0

## -- Conflicts -----
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
```

We can see that it has loaded 8 packages (**ggplot2**, **tibble**, **tidyr**, **readr**, **purrr**, **dplyr**, **stringr**, **forcats**), the number behind a package name is its version.

The “Conflicts” message is expected and can safely be ignored.¹

There are a few other R packages that we use and are not part of the tidyverse, but we will introduce them as we go along. If you are incredibly curious, take a peak at Chapter 16 for a full list.

¹It just means that when we use `filter` or `lag`, they will come from the `dplyr` package, rather than the `stats` package. We've never needed to use `filter` and `lag` from `stats`, but if you do, then use the double colon, i.e., `stats::filter()` or `stats::lag()`, as just `filter()` will use the `dplyr` one.

2

R Basics

The aim of this chapter is to familiarise you with how R works. We will read in data and start basic manipulations.

2.1 Getting help

RStudio has a built in **Help** tab. To use the Help tab, click your cursor on something in your code (e.g. `read_csv()`) and press F1. This will show you the definition and some examples.

However, the **Help** tab is only useful if you already know what you are looking for but can't remember exactly how it works. For finding help on things you have not used before, it is best to Google it. R has about 2 million users so someone somewhere has had the same question or problem.

2.2 Objects and functions

There are two fundamental concepts in statistical programming that it is important to get straight from the outset - objects and functions. As throughout this book, we will introduce new concepts using specific examples first.

The most common data object you will be working with is a table.

TABLE 2.1: Example of a table (=tibble once read into R), including missing values denoted NA (Not applicable/Not available).

id	sex	var1	var2	var3
1	Male	4	NA	2
2	Female	1	4	1
3	Female	2	5	NA
4	Male	3	NA	NA

This is something with rows and columns much like the example in Table 2.1.

A table can live anywhere: on paper, in a Spreadsheet, in an SQL database, or it can live in your R Session’s Environment. And yes, R sessions are as fun as they sound, almost as fun as music sessions.

We usually initiate and interface R using RStudio, but everything we talk about here (objects, functions, sessions, environment) also work when RStudio is not available, but R is. This can be the case if you are working on a supercomputer that can only serve the R Console, and not an RStudio IDE (reminder from first chapter: Integrated Development Environment).

So, regularly shaped data in rows and columns is called a table when it lives outside R, but once you read it into R (import it), we call it a tibble.¹

When you are in one of your very cool R sessions and read in some data, it goes into this session’s Environment. Everything in your Environment needs to have a name. You will likely have many objects such as tibbles going on at the same time. Note that `tibble` is what the thing is, rather than its name. This is the `class` of an object.

To keep our code examples easy to follow, we call our example tibble `mydata`. In a real analysis, you should give your tibbles meaningful names, e.g., `patient_data`, `lab_results`, `annual_totals`, etc.

¹There is also the original version of tables in R - they are called data frames. In most cases, `data frames` and `tibbles` work interchangeably, but `tibbles` often work better. Another great alternative to base R `data frames` are `data tables`. In this book, and for most of our day-to-day work these days, we will use `tibbles`.

So, the tibble named `mydata` is example of an object that can be in the Environment of your R Session:

```
mydata

## # A tibble: 4 × 5
##   id sex     var1  var2  var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     1 Male      4     NA      2
## 2     2 Female    1      4      1
## 3     3 Female    2      5     NA
## 4     4 Male      3     NA     NA
```

So what is a function?

A function is a procedure which takes some information (input), does something to it, and passes back the modified information (output).

A simple function that can be applied to numeric data for instance is `mean()`.

R functions always have round brackets after their name. This is for two reasons. First, it easily differentiates them as functions - you will get used to reading them like this.

Second, and more important, we can put **arguments** in these brackets.

Arguments can also be thought of as input. In data analysis, the most common input for a function is data - we need to give `mean()` some data to average over. It does not make sense (nor will it work) to feed it the whole tibble with multiple columns, including patient IDs and a categorical variable (`sex`).

To quickly extract a single column, we use the `$` symbol like this:

```
mydata$var1
```

```
## [1] 4 1 2 3
```

You can ignore the `## [1]` at the beginning of the extracted values - this is something that becomes more useful when printing multiple

lines of data as the number in the square brackets keeps count on how many values we are seeing.

We can then use `mydata$var1` as the first argument of `mean()` by putting it inside its brackets:

```
mean(mydata$var1)
```

```
## [1] 2.5
```

Which tells us that the mean of `var1` (4, 1, 2, 3) is 2.5. In this example, `mydata$var1` is the first and only argument to `mean()`. But what happens if we try to calculate the average value of `var2` (NA, 4, 5, NA)?

```
mean(mydata$var2)
```

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

We get an `NA` (“Not applicable”). We would expect to see an `NA` if we tried to, for example, calculate the average of `sex`:

```
mean(mydata$sex)
```

```
## Warning in mean.default(mydata$sex): argument is not numeric or logical:
##   returning NA
```

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

In fact, in this case, R also gives us a pretty clear warning suggesting it can't compute the mean of an argument that is not numeric or logical. The sentence actually reads pretty fun, as if R was saying it was not logical to calculate the mean of something that is not numeric.

But what R is actually saying that it is happy to calculate the mean of two types of variables: numerics or logicals, but what you have passed is neither.²

²Logical is a data type with two potential values: TRUE or FALSE. We will come back to data types.

So why does `mean(mydata$var2)` return `NA` rather than the mean of the numeric values included in this column? That is because the column itself includes missing values (`NAs`), and R does not want to average over NAs implicitly. It is being cautious - what if you didn't know there were missing values for some patients? If you wanted to compare the means of `var1` and `var2` without any further filtering, you would be comparing samples of different sizes.

If you decide to ignore the NAs and want to calculate the mean anyway, you can do so by adding another argument to `mean()`:

```
mean(mydata$var2, na.rm = TRUE)
```

```
## [1] 4.5
```

Adding `na.rm = TRUE` tells R that you are happy for it to calculate the mean of any existing values (but to remove - `rm` - the `NA` values). This ‘removal’ excludes the NAs from the calculation, it does not affect the actual tibble (`mydata`) holding the dataset.

R is case sensitive, so `na.rm`, not `NA.rm` etc. There is, however, no need to memorize how the arguments of functions are exactly spelled - this is what the Help tab is for (press `F1` when the cursor is on the name of the function). Help pages are built into R, so an internet connection is not required for this.

Make sure to separate multiple arguments with commas or R will give you an error of `Error: unexpected symbol`.

Finally, some functions do not need any arguments to work. A good example is the `sys.time()` which returns the current time and date. This is very useful when using R to generate and update reports automatically. Including this means you can always be clear on when the results were last updated.

```
Sys.time()
```

```
## [1] "2019-08-29 18:01:36 BST"
```

To summarise, objects and functions work hand in hand. Objects are both an input as well as the output of a function (what the function returns).

When passing data to a function it is usually its first argument, with further arguments used to specify behaviour.

When we say “the function returns”, we are referring to its output (or an Error if it’s one of those days).

The returned object can be different to its input object. In our `mean()` example above, the input object was a column (`mydata$var1: 4, 1, 2, 3`), whereas the output was a single value: 2.5.

2.3 Working with Objects

To create a new object in our Environment we use the equals sign:

```
a = 103
```

This reads: the object `a` is assigned value 103. You know that the assignment worked when it shows up in the Environment tab. If we now run `a` just on its own, it gets printed back to us:

```
a
```

```
## [1] 103
```

Similarly, if we run a function without assignment to an object, it gets printed but not saved in your Environment:

```
seq(15, 30)
```

```
## [1] 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

`seq()` is a function that creates a sequence of numbers (+1 by default) between the two arguments you pass to it in its brackets. We can assign the result of `seq(15, 30)` into an object, let's call it `example_sequence`:

```
example_sequence = seq(15, 30)
```

Doing this creates `example_sequence` in our Environment, but it does not print it.

If you save the results of an R function in an object, it does not get printed. If you run a function without the assignment (=), its results get printed, but not saved in an object.

You can call your object pretty much anything you want, as long as it starts with a letter. It can then include numbers as well, for example, we could have named the new object `sequence_15_to_30`. Spaces in object names are not easy to work with, we tend to use underscores in their place, but you could also use capitalization, e.g. `exampleSequence = seq(15, 30)`.

Finally, R doesn't mind overwriting an existing object, for example (notice how we then include the variable on a new line to get it printed as well as overwritten):

```
example_sequence = example_sequence/2
```

```
example_sequence
```

```
## [1] 7.5 8.0 8.5 9.0 9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0
## [15] 14.5 15.0
```

Note that many people use `<-` instead of `=`. They mean the same thing in R: both `=` and `<-` save what is on the right into an object with the name on the left. There is also a left-to-right operator: `->`.

2.4 Pipe - `%>%`

The pipe - denoted `%>%` - is probably the oddest looking thing you'll see in this book. But please bear with, it is not as scary as it looks! Furthermore, it is super useful. We use the pipe to send objects into functions.

In the above examples, we calculated the mean of column `var1` from `mydata` by `mean(mydata$var1)`. With the pipe, we can rewrite this as:

```
library(tidyverse)
mydata$var1 %>% mean()
```

```
## [1] 2.5
```

Which reads: “Working with `mydata`, we select a single column called `var1` (with the `$`) **and then** calculate the `mean()`.” The pipe becomes especially useful once the analysis includes multiple steps applied one after another. A good way to read and think of the pipe is **“and then”**.

This piping business is not standard R functionality and before using it in a script, you need to tell R this is what you will be doing. The pipe comes from the **magrittr** package (Figure 2.1), but loading the **tidyverse** will also load the pipe. So `library(tidyverse)` initialises everything you need (no need to include `library(magrittr)` explicitly).

To insert a pipe `%>%`, use the keyboard shortcut `ctrl+shift+M`.

With or without the pipe, the general rule “if the result gets printed it doesn’t get saved” still applies. To save the result of the function into a new object (so it shows up in the Environment), you need to add the name of the new object with the assignment operator (`=`):

```
mean_result = mydata$var1 %>% mean()
```



FIGURE 2.1: This is not a pipe. René Magritte inspired artwork by Stefan Milton Bache (creator of `%>%` in R). Image source: <https://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html>

2.4.1 When pipe sends data to the wrong place: use `, data = .` to direct it

The pipe usually sends data to the beginning of function brackets (as most of the functions we use expect a tibble as the first argument). So `mydata %>% lm(dependent~explanatory)` is equivalent to `lm(mydata, dependent~explanatory)` (`lm()` stands for linear model introduced in detail in Chapter ??).

However, the `lm()` function does not expect data as its first argument. `lm()` wants us to specify the variables first

(`dependent~explanatory`), and then wants the tibble these columns are in. So we have to use the `.` to tell the pipe to send the data to the second argument of `lm()`, not the first, e.g.

```
mydata %>%
  lm(var1~var2, data = .)
```

2.5 Reading data into R

We mentioned before that once a table (e.g. from spreadsheet or database) gets read into R we start calling it a `tibble`. The most common format data comes to us in is CSV (comma separated values). CSV is basically an uncomplicated spreadsheet with no formatting. It is just a single table with rows and columns (no worksheets or formulas). Furthermore, you don't need special software to quickly view a CSV file - a text editor will do, and that includes RStudio.

For example, look at “example_data.csv” in the `healthyr` project’s folder in Figure 2.2 (this is the **Files** pane at the bottom-right corner of your RStudio).

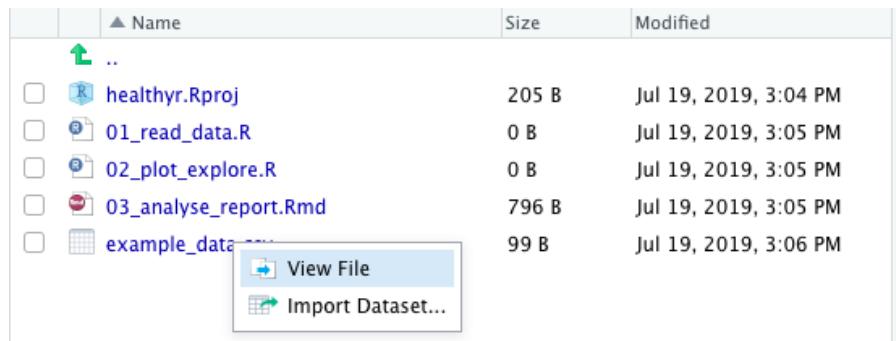


FIGURE 2.2: View or import a data file.

Clicking on a data file gives us two options: `View File` or `Import`

`Dataset`. For standard CSV files, we don't usually bother with the Import interface and just type in (or copy from a previous script):

```
library(tidyverse)
example_data = read_csv("example_data.csv")
```

Without further arguments, `read_csv()` defaults to:

- values are delimited by commas (e.g., `id`, `var1`, `var2`, ...);
- numbers use decimal point (e.g., `4.12`), rather than decimal comma (e.g., `4,12`);
- the first line has column names (it is a “header”);
- and missing values are empty or denoted NA.

If your file, however, is different to these, then the `Import Dataset` interface (Figure 2.2) is very useful and will give you the relevant `read_()` syntax with extra arguments completed for you.

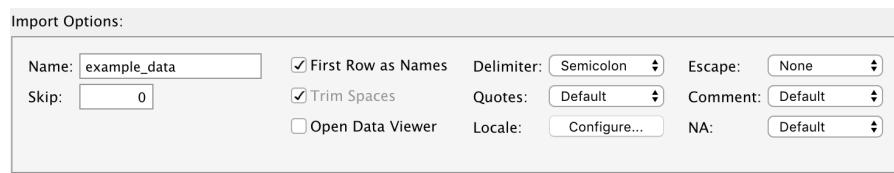


FIGURE 2.3: Import: Some of the special settings your data file might have.

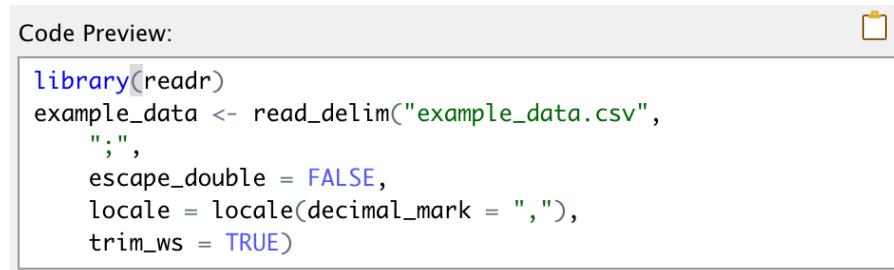


FIGURE 2.4: After using the Import Dataset window, copy-paste the resulting code into your script.

After selecting the specific options to import file, a friendly preview

window will show whether R understands the format of your data. DO NOT BE tempted to press the `Import` button.

Yes, this will read in your dataset once, but means you have to reselect the options every time you come back to RStudio. Instead, copy-paste the code (e.g., Figure 2.4) into your R script - this way you can use it over and over again.

Ensuring all steps are recorded in scripts make your workflow reproducible by your future self, colleagues, supervisors, and extraterrestrials.

The `Import Dataset` button can also help you to read in Excel, SPSS, Stata, or SAS files (instead of `read_csv()`, it will give you `read_excel()`, `read_sav()`, `read_stata()`, or `read_sas()`).

If you've used R before or are using older scripts passed by colleagues, you might see `read.csv()` rather than `read_csv()`.

In short, `read_csv()` is faster and more predictable and in all new scripts this is what you should use. In existing scripts that work and are tested, do not just start replacing `read.csv()` with `read_csv()`. `read_csv()` handles categorical variables differently ³. An R script written using the `read.csv()` might not work as expected any more if just replaced with `read_csv()`.

Do not start updating and possibly breaking existing R scripts by replacing base R functions with the tidyverse ones we show here. Do use the modern functions in any new code you write.

³It does not silently convert strings to factors, i.e., it defaults to `stringsAsFactors = FALSE`. For those not familiar with the terminology here - don't worry, we will cover this in just a few sections.

2.5.1 Reading in the Global Burden of Disease example dataset (short version)

In the next few chapters of this book, we will be using the Global Burden of Disease datasets. The Global Burden of Disease Study (GBD) is the most comprehensive worldwide observational epidemiological study to date. It describes mortality and morbidity from major diseases, injuries and risk factors to health at global, national and regional levels.⁴

GBD data are publicly available from the website. Table 2.2 and Figure 2.5 show a high level version of the project data with just 3 variables: `cause`, `year`, `deaths_millions` (number of people who die of each cause every year). Later, we will be using a longer dataset with different subgroups and we will show you how to summarise comprehensive datasets yourself.

```
library(tidyverse)
gbd_short = read_csv("data/global_burden_disease_cause-year.csv")
```

⁴Global Burden of Disease Collaborative Network. Global Burden of Disease Study 2017 (GBD 2017) Results. Seattle, United States: Institute for Health Metrics and Evaluation (IHME), 2018. Available from <http://ghdx.healthdata.org/gbd-results-tool>.

TABLE 2.2: Deaths per year from three broad disease categories (short version of the Global Burden of Disease example dataset).

year	cause	deaths_millions
1990	Communicable diseases	15.36
1990	Injuries	4.25
1990	Non-communicable diseases	26.71
1995	Communicable diseases	15.11
1995	Injuries	4.53
1995	Non-communicable diseases	29.27
2000	Communicable diseases	14.81
2000	Injuries	4.56
2000	Non-communicable diseases	31.01
2005	Communicable diseases	13.89
2005	Injuries	4.49
2005	Non-communicable diseases	32.87
2010	Communicable diseases	12.51
2010	Injuries	4.69
2010	Non-communicable diseases	35.43
2015	Communicable diseases	10.88
2015	Injuries	4.46
2015	Non-communicable diseases	39.28
2017	Communicable diseases	10.38
2017	Injuries	4.47
2017	Non-communicable diseases	40.89

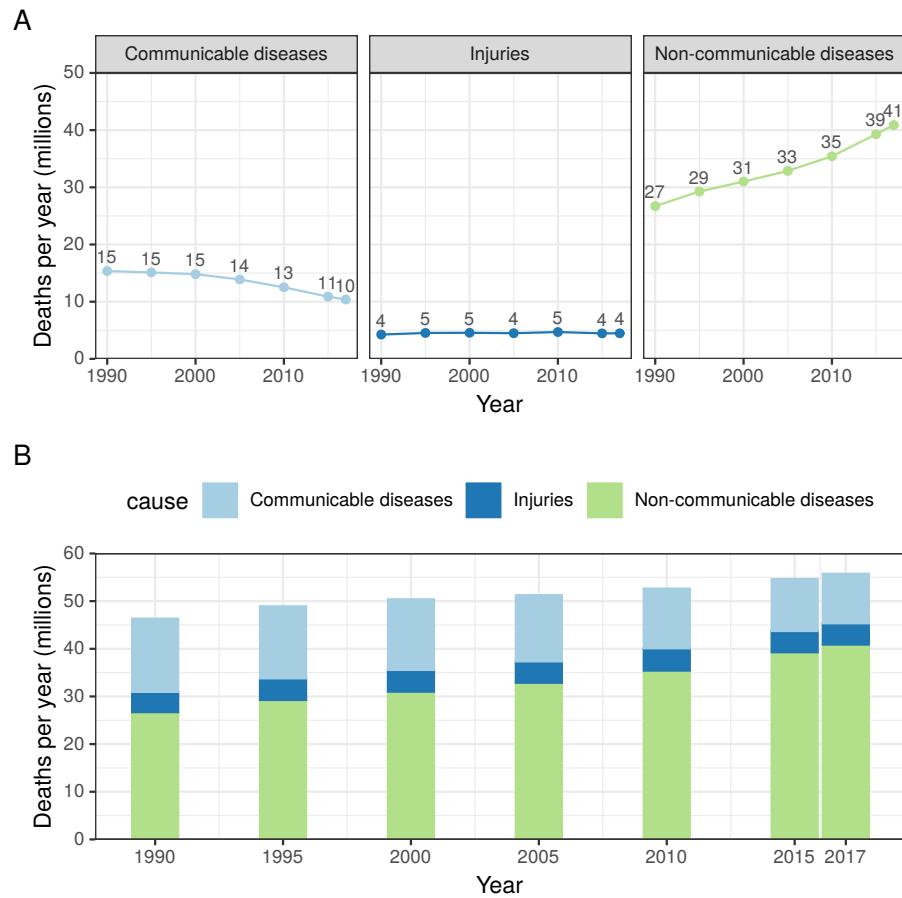


FIGURE 2.5: Causes of death from the Global Burden of Disease dataset (Table 2.2). Data on (B) is the same as (A) but stacked to show the total (sum) of all causes.

2.6 Operators for filtering data

Operators are symbols that tell R how to handle different pieces of data or objects. We have already introduced three: `$` (selects a column), `=` (assigns values or results to a variable), and the pipe - `%>%` (sends data into a function).

Other common operators are the ones we use for filtering data - these are called comparison and logical operators. This may be for creating subgroups, or for excluding outliers or incomplete cases.

The comparison operators that work with numeric data are relatively straightforward: `>`, `<`, `>=`, `<=`. The first two check whether your values are greater or less than another value, the last two check for "greater than or equal to" and "less than or equal to". These operators are most commonly spotted inside the `filter()` function:

```
gbd_short %>%
  filter(year < 1995)

## # A tibble: 3 x 3
##   year cause           deaths_millions
##   <dbl> <chr>                  <dbl>
## 1 1990 Communicable diseases     15.4
## 2 1990 Injuries                 4.25
## 3 1990 Non-communicable diseases 26.7
```

Here we send the data (`gbd_short`) to the `filter()` and ask it to retain all years that are less than 1995. The resulting tibble only includes the year 1990. Now, if we use the `<=` (less than or equal to) operator, both 1990 and 1995 pass the filter:

```
gbd_short %>%
  filter(year <= 1995)

## # A tibble: 6 x 3
##   year cause           deaths_millions
##   <dbl> <chr>                  <dbl>
## 1 1990 Communicable diseases     15.4
```

```
## 2 1990 Injuries          4.25
## 3 1990 Non-communicable diseases 26.7
## 4 1995 Communicable diseases 15.1
## 5 1995 Injuries          4.53
## 6 1995 Non-communicable diseases 29.3
```

Furthermore, the values either side of the operator could both be variables, e.g., `mydata %>% filter(var2 > var1)`.

To filter for values that are equal to something, we use the `==` operator.

```
gbd_short %>%
  filter(year == 1995)
```

```
## # A tibble: 3 x 3
##   year cause           deaths_millions
##   <dbl> <chr>            <dbl>
## 1 1995 Communicable diseases     15.1
## 2 1995 Injuries          4.53
## 3 1995 Non-communicable diseases 29.3
```

This reads, take the GBD dataset, send it to the filter and keep rows where year is equal to 1995.

Accidentally using the single equals `=` when double equals is necessary `==` is a very common mistake and still happens to the best of us. It happens so often that the error the `filter()` function gives when using the wrong one also reminds us what the correct one was:

```
gbd_short %>%
  filter(year = 1995)
```

```
## `year` (`year = 1995`) must not be named, do you need `==`?
```

The answer to 'do you need `==`?' is almost always "Yes R, I do, thank you".

But that's just because `filter()` is a clever cookie and is used to

this very common mistake. There are other useful functions we use these operators in, but they don't always know to tell us that we've just confused = for ==. So if you get an error when checking for an equality between variables, always check your == operators first.

R also has two operators for combining multiple comparisons: & and |, which stand for AND and OR, respectively. For example, we can filter to only keep the earliest and latest years in the dataset:

```
gbd_short %>%
  filter(year == 1995 | year == 2017)

## # A tibble: 6 x 3
##   year cause           deaths_millions
##   <dbl> <chr>                  <dbl>
## 1 1995 Communicable diseases      15.1
## 2 1995 Injuries                  4.53
## 3 1995 Non-communicable diseases 29.3
## 4 2017 Communicable diseases     10.4
## 5 2017 Injuries                  4.47
## 6 2017 Non-communicable diseases 40.9
```

This reads: take the GBD dataset, send it to the filter and keep rows where year is equal to 1995 OR year is equal to 2017.

Using specific values like we've done here (1995/2017) is called "hard-coding", which is fine if we know for sure that we will not want to use the same script on an updated dataset. But a cleverer way of achieving the same thing is to use the `min()` and `max()` functions:

```
gbd_short %>%
  filter(year == max(year) | year == min(year))

## # A tibble: 6 x 3
##   year cause           deaths_millions
##   <dbl> <chr>                  <dbl>
## 1 1990 Communicable diseases      15.4
## 2 1990 Injuries                  4.25
## 3 1990 Non-communicable diseases 26.7
## 4 2017 Communicable diseases     10.4
## 5 2017 Injuries                  4.47
## 6 2017 Non-communicable diseases 40.9
```

TABLE 2.3: Filtering operators.

Operators	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to
<code>&</code>	AND
<code> </code>	OR

2.6.1 Worked examples

Filter the dataset to only include the year 2000. Save this in a new variable using the assignment operator.

```
mydata_year2000 = gbd_short %>%
  filter(year == 2000)
```

Let's practice combining multiple selections together.

Reminder: ‘|’ means OR and ‘&’ means AND.

From `gbd_short`, select the lines where year is either 1990 or 2017 and cause is “Communicable diseases”:

```
new_data_selection = gbd_short %>%
  filter((year == 1990 | year == 2013) & cause == "Communicable diseases")

# Or we can get rid of the extra brackets around the years
# by moving cause into a new filter on a new line:

new_data_selection = gbd_short %>%
  filter(year == 1990 | year == 2013) %>%
  filter(cause == "Communicable diseases")
```

The hash symbol (#) is used to add free text comments to R code. R will not try to run these lines, they will be ignored. Comments are an essential part of any programming code and these are “Dear Diary” notes to your future self.

2.7 The combine function: `c()`

The combine function as it's name implies is used to combine several values. It is especially useful when used with the `%in%` operator to filter for multiple values. Remember how the `gbd_short` cause column had three different causes in it:

```
gbd_short$cause %>% unique()

## [1] "Communicable diseases"      "Injuries"
## [3] "Non-communicable diseases"
```

Say we wanted to filter for communicable and non-communicable diseases.⁵ We could use the OR operator `|` like this:

```
gbd_short %>%
  # also filtering for a single year to keep the result concise
  filter(year == 1990) %>%
  filter(cause == "Communicable diseases" | cause == "Non-communicable diseases")

## # A tibble: 2 x 3
##   year cause           deaths_millions
##   <dbl> <chr>              <dbl>
## 1 1990 Communicable diseases        15.4
## 2 1990 Non-communicable diseases    26.7
```

But that means we have to type in `cause` twice (and more if we had other values we wanted to include). This is where the `%in%` operator together with the `c()` function come in handy:

```
gbd_short %>%
  filter(year == 1990) %>%
  filter(cause %in% c("Communicable diseases", "Non-communicable diseases"))

## # A tibble: 2 x 3
##   year cause           deaths_millions
##   <dbl> <chr>              <dbl>
```

⁵In this example, it would just be easier to used the “not equal” operator, `filter(cause != “Injuries”)`, but imagine your column had more than just three different values in it.

```
## 1 1990 Communicable diseases           15.4
## 2 1990 Non-communicable diseases       26.7
```

2.8 Missing values (NAs) and filters

Filtering for missing values (NAs) needs special attention and care. Remember the small example tibble from Table 2.1 - it has some NAs in columns `var2` and `var3`:

```
mydata
```

```
## # A tibble: 4 × 5
##       id sex     var1  var2  var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     1 Male      4    NA      2
## 2     2 Female    1     4      1
## 3     3 Female    2     5     NA
## 4     4 Male      3    NA     NA
```

If we now want to filter for rows where `var2` is missing, `filter(var2 == NA)` is not the way to do it, it will not work.

Since R is a programming language, it can be a bit stubborn with things like these. When you ask R to do a comparison using `==` (or `<`, `>`, etc.) it expects a value on each side, but `NA` is not a value, it is the lack thereof. The way to filter for missing values is using the `is.na()` function:

```
mydata %>%
  filter(is.na(var2))
```

```
## # A tibble: 2 × 5
##       id sex     var1  var2  var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     1 Male      4    NA      2
## 2     4 Male      3    NA     NA
```

We send `mydata` to the filter and keep rows where `var2` is `NA`. Note the double brackets at the end: that's because the inner one belongs to

`is.na()`, and the outer one to `filter()`. Missing out a closing bracket is also a common source of errors, and still happens to the best of us.

If filtering for rows where `var2` is not missing, we do this⁶

```
mydata %>%
  filter(!is.na(var2))
```

```
## # A tibble: 2 x 5
##       id sex     var1 var2 var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     2 Female     1     4     1
## 2     3 Female     2     5    NA
```

In R, the exclamation mark (!) means “not”.

Sometimes you want to drop a specific value (e.g. an outlier) from the dataset like this. The small example tibble `mydata` has 4 rows, with the values for `var2` as follows: NA, 4, 5, NA. We can exclude the row where `var2` is equal to 5 by using the “not equals” (`!=`)⁷:

```
mydata %>%
  filter(var2 != 5)
```

```
## # A tibble: 1 x 5
##       id sex     var1 var2 var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     2 Female     1     4     1
```

However, you’ll see that by doing this, R drops the rows where `var2` is NA as well, as it can’t be sure these missing values were not equal to 5.

If you want to keep the missing values, you need to make use of the OR (`|`) operator and the `is.na()` function:

⁶In this simple example, `mydata %>% filter(! is.na(var2))` could be replaced by a shorthand: `mydata %>% drop_na(var2)`, but it is important to understand how the `!` and `is.na()` work as there will be more complex situations where using these is necessary.

⁷`filter(var2 != 5)` is equivalent to `filter(var2 == 5)`

```
mydata %>%
  filter(var2 != 5 | is.na(var2))

## # A tibble: 3 x 5
##   id sex    var1  var2  var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     1 Male      4    NA      2
## 2     2 Female    1      4      1
## 3     4 Male      3    NA     NA
```

Being caught out by missing values, either in filters or other functions is very common (remember mydata\$var2 %>% mean() returns NA unless you add na.rm = TRUE). This is also why we insist that you always plot your data first - outliers will reveal themselves and NA values usually become obvious too.

Another thing we do to stay safe around filters and missing values is saving the results and making sure the number of rows still add up:

```
subset1 = mydata %>%
  filter(var2 == 5)

subset2 = mydata %>%
  filter(! var2 == 5)

subset1

## # A tibble: 1 x 5
##   id sex    var1  var2  var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     3 Female    2      5     NA

subset2

## # A tibble: 1 x 5
##   id sex    var1  var2  var3
##   <int> <chr> <dbl> <dbl> <dbl>
## 1     2 Female    1      4      1
```

If the numbers are small, you can now quickly look at RStudio's **Environment** tab and figure out whether the number of observations (rows) in `subset1` and `subset2` add up to the whole dataset

(`mydata`). Or use the `nrow()` function to ask R to tell you what the number of rows is in each dataset:

Rows in `mydata`:

```
nrow(mydata)
```

```
## [1] 4
```

Rows in `subset1`:

```
nrow(subset1)
```

```
## [1] 1
```

Rows in `subset2`:

```
nrow(subset2)
```

```
## [1] 1
```

Asking R whether adding these two up equals the original size:

```
nrow(subset1) + nrow(subset2) == nrow(mydata)
```

```
## [1] FALSE
```

As expected, this returns FALSE - because we didn't add special handling for missing values. Let's create a third subset only including rows where `var3` is NA:

Rows in `subset2`:

```
subset3 = mydata %>%
  filter(is.na(var2))
```

```
nrow(subset1) + nrow(subset2) + nrow(subset3) == nrow(mydata)
```

```
## [1] TRUE
```

2.9 Variable types and why we care

There are three broad types of data:

- continuous (numbers), in R: numeric, double, or integer;
- categorical, in R: character, factor, or logical (TRUE/FALSE);
- date/time, in R: POSIXct date-time⁸.

Values within a column all have to be the same type, but a tibble can of course hold columns of different types. Generally, R is very good at figuring out what type of data you have (in programming, this ‘figuring out’ is called ‘parsing’).

For example, when reading in data, it will tell you what it assumed for the columns:

```
library(tidyverse)
typesdata = read_csv("data/typesdata.csv")

## Parsed with column specification:
## cols(
##   id = col_character(),
##   group = col_character(),
##   measurement = col_double(),
##   date = col_datetime(format = "")
## )

typesdata

## # A tibble: 3 × 4
##   id     group    measurement date
##   <chr> <chr>        <dbl> <dttm>
## 1 ID1   Control      1.8  2017-01-02 12:00:00
## 2 ID2   Treatment    4.5  2018-02-03 13:00:00
## 3 ID3   Treatment    3.7  2019-03-04 14:00:00
```

This means that a lot of the time you do not have to worry about

⁸Portable Operating System Interface (POSIX) is a set of computing standards. There’s nothing more to understand about this other than when R starts shouting “POSIXct this or POSIXlt that” at you, check your date and time variables

those little `<chr>` vs `<dbl>` vs `<s3: POSIXct>` labels, R knows what its doing. But in cases of irregular or faulty input data, or when doing a lot of calculations and modifications your data, we need to be aware of these different types to be able to find and fix mistakes.

For example, consider a very similar file as above but with a couple of data entry issues:

```
typesdata_faulty = read_csv("data/typesdata_faulty.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_character(),
##   group = col_character(),
##   measurement = col_character(),
##   date = col_character()
## )
```

```
typesdata_faulty
```

```
## # A tibble: 3 x 4
##   id     group    measurement date
##   <chr>  <chr>      <chr>       <chr>
## 1 ID1    Control    1.8        02-Jan-17 12:00
## 2 ID2    Treatment  4.5        03-Feb-18 13:00
## 3 ID3    Treatment  3.7 or 3.8  04-Mar-19 14:00
```

Notice R parsed both measurement and date as characters. The first one is a data entry issue: the person taking the measurement couldn't decide which value to note down (maybe the scale was shifting between the two values) so they included both values and text “or” in the cell.

A numeric variable will also get parsed as a categorical variable if it contains certain typos, e.g., if entered as “3..7” instead of “3.7”.

The reason R didn't automatically make sense of the date column is that it can't tell which is the date and which is the year: **02-Jan-17** could stand for *02-Jan-2017* as well as *2002-Jan-17*.

Therefore, while a lot of the time you do not have to worry about variable types and can just get on with your analysis, it is impor-

tant to understand what the different types are to be ready to deal with them when issues arise.

Since health datasets are generally full of categorical data, it is crucial to understand the difference between characters and factors (both are types of categorical variables in R with pros and cons).

So here we go.

2.10 Numeric variables (continuous)

Number are straightforward to handle and don't usually cause trouble. R usually refers to numbers as `numeric` (or `num`), but sometimes it really gets its nerd on and also calls numbers `integer` or `double`.⁹ It doesn't usually matter whether R is classifying your continuous data `numeric/num/double/int`, but it is good to be aware of these different terms as you will see them in R messages.

FRIENDLY WARNING: What's about to follow is a bit dry. Furthermore, it is not essential for complete beginners - you might want to continue reading from **Characters**. Before you leave, take a mental note that sometimes numbers in R have more decimal places than it seems, and that can cause funny behaviour when using the double equals operator (`==`).

Something to note about numbers is that R doesn't usually print more than 6 decimal places, but that doesn't mean they don't exist.

⁹Integers are numbers without decimal places (e.g., 1, 2, 3), whereas `double` stands for “Double-precision floating-point” format (e.g., 1.234, 5.67890).

For example, from the `typedata` tibble, we're taking the `measurement` column and sending it to the `mean()` function. R then calculates the mean and tells us what it is with 6 decimal places:

```
typedata$measurement %>% mean()
```

```
## [1] 3.333333
```

Let's save that in a new object:

```
measurement_mean = typedata$measurement %>% mean()
```

But when using the double equals operator to check if this is equivalent to a fixed value (you might do this when comparing to a threshold, or even another mean value), R returns `FALSE`:

```
measurement_mean == 3.333333
```

```
## [1] FALSE
```

Now this doesn't seem right, does it - R clearly told us just above that the mean of this variable is 3.333333 (reminder: the actual values in the measurement column are 1.8, 4.5, 3.7). The reason the above statement is `FALSE` is because `measurement_mean` is quietly holding more than 6 decimal places.

One way to go about this is to round the mean to a reasonable number of decimal places:

```
round(measurement_mean, 3)
```

```
## [1] 3.333
```

The second argument of `round()` specifies the number of decimal places you want your number(s) rounded to. So when using `round()` in the equality statement like this, we get the expected `TRUE`:

```
round(measurement_mean, 3) == 3.333
```

```
## [1] TRUE
```

Which is usually fine, especially if you've finished applying calculations on that number. But when you indent to use it if further calculations, then rounding should be left to the very end - to minimise rounding errors. This is where the `near()` function comes in handy:

```
library(tidyverse)
near(measurement_mean, 3.333, 0.001)
```

```
## [1] TRUE
```

The first two arguments for `near()` are the numbers you are comparing, the third argument is the precision you are interested in. So if the numbers are equal within that precision, it returns `TRUE`. This means you get the expected result without having to round the numbers off.

2.11 Character variables (categorical, IDs, free text)

Characters (sometimes referred to as *strings* or *character strings*) in R are letters, words, or even whole sentences (an example of this may be free text comments). Characters are displayed in-between `""` (or `''`).

A very useful function for quickly investigating categorical variables is the `count()` function:

```
library(tidyverse)
typesdata %>%
  count(group)
```

```
## # A tibble: 2 x 2
##   group     n
##   <chr>    <int>
## 1 Control      1
## 2 Treatment    2
```

`count()` can accept multiple variables and will count up the number of observations in each subgroup, e.g., `mydata %>% count(var1, var2)`.

Another helpful option to `count` is `sort = TRUE`, which will order the result putting the highest count (`n`) to the top.

```
typesdata %>%
  count(group, sort = TRUE)

## # A tibble: 2 × 2
##   group     n
##   <chr>   <int>
## 1 Treatment    2
## 2 Control      1
```

`count()` and its `sort = TRUE` option are also useful for identifying duplicate IDs or misspellings in your data. With this example `tibble` (`typesdata`) that only has three rows, it is easy to see that the `id` column is a unique identifier whereas the `group` column is a categorical variable. You can check everything by just eyeballing the `tibble` using the built in Viewer tab (click on the dataset in the Environment tab).

But for larger datasets, you need to know how to check and then clean data programmatically - can't go through 1000s of values checking if they're all the way you expect without unexpected duplicates or typos. For most variables (categorical or numeric) we recommend always plotting your data before starting analysis. But to check for duplicates in a unique identifier, use `count()` with `sort = TRUE`:

```
# all ids are unique:
typesdata %>%
  count(id, sort = TRUE)

## # A tibble: 3 × 2
##   id     n
##   <chr> <int>
## 1 ID1      1
## 2 ID2      1
## 3 ID3      1
```

```
# we add in a duplicate row where id = ID3,
# then count again:
typesdata %>%
  add_row(id = "ID3") %>%
  count(id, sort = TRUE)

## # A tibble: 3 × 2
##   id      n
##   <chr> <int>
## 1 ID3      2
## 2 ID1      1
## 3 ID2      1
```

2.12 Factor variables (categorical)

Factors are fussy characters. Factors are fussy because they have something called **levels**. Levels are all the unique values a factor variable could take - e.g. like when we looked at `typesdata$group %>% unique()`. Using factors rather than just characters can be useful because:

- The values factor levels can take is fixed. For example, once you tell R that `typesdata$group` is a factor with two levels: Control and Treatment, combining it with other datasets with different spellings are abbreviations for the same variable would get you a warning. This can be very helpful and useful, but it can also be a nuisance when you really do want to add in another option for a `factor` variable.
- Levels have an order. When running statistical tests on grouped data (e.g., Control vs Treatment, Adult vs Child) and the variable is just a character, not a factor, R will use the alphabetically first as the reference level. Converting a character column into a factor column enables us to define and change the order of its levels. Level order also affect plots: by default, categorical variables (i.e., think of a barplot) get ordered alphabetically, but if this is not the order we want them in, we have to make

it into a factor before we plot it. The plot will then know how the order it better.

So overall, since health data is often categorical and has a reference (comparison) level, then factors are an essential way to work with these data in R. Nevertheless, the fussiness of factors can sometimes be unhelpful or even frustrating. It takes experience as an R user to know when is it easiest to keep your variables as characters and when to convert them to factors. A lot more about factor handling will be covered later in the book.

2.13 Date/time variables

R is very good for working with dates. For example, it can calculate the number of days/weeks/months between two dates, or it can be used to find a future date is (i.e., “what’s the date exactly 60 days from now?”). It also knows about time zones and is happy to parse dates in pretty much any format - as long as you tell R how your date is formatted (e.g., day before month, month name abbreviated, year in 2 or 4 digits, etc.). Since R displays dates and times between quotes (""), they look similar to characters. However, it is important to know whether R has understood which of your columns contain date/time information, as which are just normal characters.

```
library(lubridate) # lubridate makes working with dates easier
current_datetime = Sys.time()
current_datetime

## [1] "2019-08-29 18:01:38 BST"

my_datetime = "2020-12-01 12:00"
my_datetime

## [1] "2020-12-01 12:00"
```

When printed, the two objects - `current_datetime` and `my_datetime` seem to have the a very similar format. But if we try to calculate the difference between these two dates, we get an error:

```
my_datetime - current_datetime
```

```
## Error in `-.POSIXt`(my_datetime, current_datetime): can only subtract from "POSIXt" objects
```

That's because when we assigned a value to `my_datetime`, R assumed the simpler type for it - so a character. We can check what the type of an object or variable is using the `class()` function:

```
current_datetime %>% class()
```

```
## [1] "POSIXct" "POSIXt"
```

```
my_datetime %>% class()
```

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

So we need to tell R that `my_datetime` does indeed include date/time information so we can then use it in calculations:

```
my_datetime_converted = ymd_hm(my_datetime)  
my_datetime_converted
```

```
## [1] "2020-12-01 12:00:00 UTC"
```

Calculating the difference will now work:

```
my_datetime_converted - current_datetime
```

```
## Time difference of 459.7905 days
```

Since R knows this is a difference between two date/time objects, it prints the in a nicely readable way. Furthermore, the result has its own type, it is a “difftime”.

```
my_datesdiff = my_datetime_converted - current_datetime
my_datesdiff %>% class()
```

```
## [1] "difftime"
```

This is useful if we want to apply this time difference on another date, e.g.:

```
ymd_hm("2021-01-02 12:00") + my_datesdiff
```

```
## [1] "2022-04-07 06:58:21 UTC"
```

But if we want to use the number of days in a normal calculation, e.g., what if a measurement increased by 560 arbitrary units during this time period. We might want to calculate the increase per day like this:

```
560/my_datesdiff
```

```
## Error in `/.difftime`(560, my_datesdiff): second argument of / cannot be a "difftime" object
```

Doesn't work, does it. We need to convert `my_datesdiff` (which is a difftime value) into a numeric value by using the `as.numeric()` function:

```
560/as.numeric(my_datesdiff)
```

```
## [1] 1.217946
```

The lubridate package comes with several convenient functions for parsing dates, e.g., `ymd()`, `mdy()`, `ymd_hm()`, etc. - for a full list see lubridate.tidyverse.org.

However, if your date/time variable comes in an extra special format, then use the `parse_date_time()` function where the second argument specifies the format using these helpers:

Notation	Meaning	Example
%d	day as number	01-31

Notation	Meaning	Example
%m	month as number	01-12
%B	month name	January-December
%b	abbreviated month	Jan-Dec
%Y	4-digit year	2019
%y	2-digit year	19
%H	hours	12
%M	minutes	01
%A	weekday	Monday-Sunday
%a	abbreviated weekday	Mon-Sun

For example:

```
parse_date_time("12:34 07/Jan'20", "%H:%M %d/%b'%y")
```

```
## [1] "2020-01-07 12:34:00 UTC"
```

Furthermore, the same date/time helpers can be used to rearrange your date and time for printing:

```
Sys.time()
```

```
## [1] "2019-08-29 18:01:38 BST"
```

```
Sys.time() %>% format("%H:%M on %B-%d (%Y)")
```

```
## [1] "18:01 on August-29 (2019)"
```

You can even add plain text into the `format()` function, R will know to put the right date/time values where the % are:

```
Sys.time() %>% format("Happy days, the current time is %H:%M %B-%d (%Y)!")
```

```
## [1] "Happy days, the current time is 18:01 August-29 (2019)!"
```

2.14 Creating new columns - `mutate()`

The function for adding new columns (or making changes to existing ones) to a tibble is called `mutate()`. As a reminder, this is what `typesdata` looked like:

```
typesdata
```

```
## # A tibble: 3 x 4
##   id    group      measurement date
##   <chr> <chr>        <dbl> <dttm>
## 1 ID1   Control      1.8  2017-01-02 12:00:00
## 2 ID2   Treatment    4.5  2018-02-03 13:00:00
## 3 ID3   Treatment    3.7  2019-03-04 14:00:00
```

Let's say we decide to divide the column `measurement` by 2. A very quick way to see these values would be to pull them out using the `$` operator and then divide by 2:

```
typesdata$measurement
```

```
## [1] 1.8 4.5 3.7
```

```
typesdata$measurement/2
```

```
## [1] 0.90 2.25 1.85
```

But this becomes very cumbersome once we want to combine multiple variables from the same tibble in a calculation. So the `mutate()` is the way to go here:

```
typesdata %>%
  mutate(measurement/2)
```

```
## # A tibble: 3 x 5
##   id    group      measurement date      `measurement/2`
##   <chr> <chr>        <dbl> <dttm>        <dbl>
## 1 ID1   Control      1.8  2017-01-02 12:00:00      0.9
## 2 ID2   Treatment    4.5  2018-02-03 13:00:00      2.25
## 3 ID3   Treatment    3.7  2019-03-04 14:00:00      1.85
```

Notice how the `mutate()` above returns the whole tibble with a new column called `measurement/2`. This is quite nice of `mutate()` already, but it would be best to give columns names that don't include characters other than underscores (`_`) or dots (`.`). So let's assign a more standard name for this new column:

```
typesdata %>%
  mutate(measurement_half = measurement/2)

## # A tibble: 3 x 5
##   id    group      measurement date           measurement_half
##   <chr> <chr>      <dbl> <dttm>                <dbl>
## 1 ID1   Control     1.8  2017-01-02 12:00:00        0.9
## 2 ID2   Treatment   4.5  2018-02-03 13:00:00       2.25
## 3 ID3   Treatment   3.7  2019-03-04 14:00:00       1.85
```

Better. You can see that R likes the name we gave it a bit better as it's now removed the back-ticks from around it. Overall, back-ticks can be used to call out non-standard column names, so if you are forced to read in data with, e.g., spaces in column names, then the back-ticks enable calling column names that would otherwise error¹⁰:

```
mydata$`Nasty column name`

# or

mydata %>%
  select(`Nasty column name`)
```

But as usual, if it gets printed, it doesn't get saved. We have two options - we can either overwrite the `typesdata` tibble (by changing the first line to `typesdata = typesdata %>%`), or we can create a new one (that appears in your Environment):

```
typesdata_modified = typesdata %>%
  mutate(measurement_half = measurement/2)

typesdata_modified
```

¹⁰If this happens to you a lot, then check out `library(janitor)` and its function `clean_names()` for automatically tidying non-standard column names.

```
## # A tibble: 3 x 5
##   id     group    measurement date           measurement_half
##   <chr>  <chr>      <dbl> <dttm>                <dbl>
## 1 ID1    Control     1.8  2017-01-02 12:00:00       0.9
## 2 ID2    Treatment    4.5  2018-02-03 13:00:00      2.25
## 3 ID3    Treatment    3.7  2019-03-04 14:00:00      1.85
```

The `mutate()` function can also be used to create a new column with a single constant value, which in return can be used to calculate a difference for each of the existing dates:

```
library(lubridate)
typesdata %>%
  mutate(reference_date = ymd_hm("2020-01-01 12:00"),
         dates_difference = reference_date - date) %>%
  select(date, reference_date, dates_difference)
```

```
## # A tibble: 3 x 3
##   date           reference_date      dates_difference
##   <dttm>        <dttm>            <drtn>
## 1 2017-01-02 12:00:00 2020-01-01 12:00:00 1094.0000 days
## 2 2018-02-03 13:00:00 2020-01-01 12:00:00 696.9583 days
## 3 2019-03-04 14:00:00 2020-01-01 12:00:00 302.9167 days
```

(We are then using the `select()` function to only choose the three relevant columns.)

Finally, the `mutate` function can be used to create a new column with a summarised value in it, e.g. the mean of another column:

```
typesdata %>%
  mutate(mean_measurement = mean(measurement))
```

```
## # A tibble: 3 x 5
##   id     group    measurement date           mean_measurement
##   <chr>  <chr>      <dbl> <dttm>                <dbl>
## 1 ID1    Control     1.8  2017-01-02 12:00:00       3.33
## 2 ID2    Treatment    4.5  2018-02-03 13:00:00       3.33
## 3 ID3    Treatment    3.7  2019-03-04 14:00:00       3.33
```

Which in return can be useful for calculating a standardized measurement (i.e. relative to the mean):

```
typesdata %>%
  mutate(mean_measurement = mean(measurement)) %>%
```

```
mutate(measurement_relative = measurement / mean_measurement) %>%
  select(matches("measurement"))

## # A tibble: 3 x 3
##   measurement mean_measurement measurement_relative
##       <dbl>           <dbl>                 <dbl>
## 1        1.8          3.33                0.54
## 2        4.5          3.33                1.35
## 3        3.7          3.33                1.11
```

2.14.1 Worked example/exercise

Round the difference to 0 decimal places using the `round()` function inside a `mutate()`. Then add a clever `matches("date")` inside the `select()` function to choose all matching columns.

Solution:

```
typesdata %>%
  mutate(reference_date = ymd_hm("2020-01-01 12:00"),
         dates_difference = reference_date - date) %>%
  mutate(dates_difference = round(dates_difference)) %>%
  select(matches("date"))
```

```
## # A tibble: 3 x 3
##   date             reference_date     dates_difference
##   <dttm>           <dttm>           <drtn>
## 1 2017-01-02 12:00:00 2020-01-01 12:00:00 1094 days
## 2 2018-02-03 13:00:00 2020-01-01 12:00:00  697 days
## 3 2019-03-04 14:00:00 2020-01-01 12:00:00  303 days
```

You can shorten this by adding the `round()` function directly around the subtraction, so the third line becomes `dates_difference = round(reference_date - date)) %>%`. But sometimes writing calculations out longer than the absolute minimum can make them easier to understand when you return to an old script months later. Furthermore, we didn't have to save the `reference_date` as a new column, the calculation could have used the value directly: `mutate(dates_difference = ymd_hm("2020-01-01 12:00") - date) %>%`. But again, defining it makes it clearer for future self what was done. And it makes `reference_date` available for reuse in more complicated calculations within the tibble.

2.15 Conditional calculations - `if_else()`

And finally, we combine the filtering operators (`==`, `>`, `<`, etc) with the `if_else()` function to create new columns based on a condition.

```
typesdata %>%
  mutate(above_threshold = if_else(measurement > 3,
                                    "Above three",
                                    "Below three"))

## # A tibble: 3 x 5
##   id    group      measurement date      above_threshold
##   <chr> <chr>       <dbl> <dttm>        <chr>
## 1 ID1   Control     1.8  2017-01-02 12:00:00 Below three
## 2 ID2   Treatment   4.5  2018-02-03 13:00:00 Above three
## 3 ID3   Treatment   3.7  2019-03-04 14:00:00 Above three
```

We are sending `typesdata` into a `mutate()` function, we are creating a new column called `above_threshold` based on whether `measurement` is greater or less than 3. The first argument to `if_else()` is a condition (in this case that `measurement` is greater than 3), the second argument is the value if the condition is TRUE, and the third argument is the value if the condition is FALSE. Look at each line in the tibble above and convince yourself that the `threshold` variable worked as expected. Then look at the two closing brackets - `)` - at the end of the code and convince yourself they both need to be there.

`if_else()` and missing values tip: for rows with missing values (NAs), the condition returns neither TRUE or FALSE, it returns NA. And that might be fine, but if you want to assign a specific group/label for missing values in the new variable, you can add a fourth argument to `if_else()`, e.g., `if_else(measurement > 3, "Above three", "Below three", "Value missing")`.

2.16 Create labels - `paste()`

The `paste()` function is used to add characters together. It also works with numbers and dates which will automatically be converted to characters before being pasted together into a single label. See this example where we use all variables from `typesdata` to create a new column called `plot_label` (we `select()` for printing space):

```
typesdata %>%
  mutate(plot_label = paste(id,
                            "was last measured at", date,
                            ", and the value was", measurement)) %>%
  select(plot_label)

## # A tibble: 3 × 1
##   plot_label
##   <chr>
## 1 ID1 was last measured at 2017-01-02 12:00:00 , and the value was 1.8
## 2 ID2 was last measured at 2018-02-03 13:00:00 , and the value was 4.5
## 3 ID3 was last measured at 2019-03-04 14:00:00 , and the value was 3.7
```

The `paste` is also useful when pieces of information are stored in different columns. For example, consider this made-up tibble:

```
pastedata = tibble(year = c(2007, 2008, 2009),
                    month = c("Jan", "Feb", "March"),
                    day = c(1, 2, 3))

pastedata

## # A tibble: 3 × 3
##   year month   day
##   <dbl> <chr> <dbl>
## 1 2007 Jan     1
## 2 2008 Feb     2
## 3 2009 March   3
```

We can use `paste()` to combine these into a single column:

```
pastedata %>%
  mutate(date = paste(day, month, year, sep = "-"))
```

```
## # A tibble: 3 x 4
##   year month   day date
##   <dbl> <chr> <dbl> <chr>
## 1 2007 Jan     1 1-Jan-2007
## 2 2008 Feb     2 2-Feb-2008
## 3 2009 March   3 3-March-2009
```

By default, `paste()` adds a space between the each value, but we can use the `sep =` argument to specify a different separator. Sometimes it is useful to use `paste0()` which does not add anything between the values (no space, no dash, etc.).

We can now tell R that the date column should be parsed as such:

```
library(lubridate)

pastedata %>%
  mutate(date = paste(day, month, year, sep = "-")) %>%
  mutate(date = dmy(date))
```

```
## # A tibble: 3 x 4
##   year month   day date
##   <dbl> <chr> <dbl> <date>
## 1 2007 Jan     1 2007-01-01
## 2 2008 Feb     2 2008-02-02
## 3 2009 March   3 2009-03-03
```

2.17 Joining multiple datasets

It is common that different pieces of information might be kept in different files or tables and that you want to combine them together. For example, consider you have some demographic information (`id`, `sex`, `age`) in one file:

```
library(tidyverse)
patientdata = read_csv("data/patient_data.csv")
patientdata
```

```
## # A tibble: 6 x 3
##       id sex      age
##   <dbl> <chr>    <dbl>
```

```
## 1     1 Female    24
## 2     2 Male     59
## 3     3 Female   32
## 4     4 Female   84
## 5     5 Male    48
## 6     6 Female   65
```

And another one with some lab results (`id`, `measurement`):

```
labsdata = read_csv("data/labs_data.csv")
labsdata

## # A tibble: 4 × 2
##       id measurement
##   <dbl>      <dbl>
## 1     5      3.47
## 2     6      7.31
## 3     8      9.91
## 4     7      6.11
```

Notice how these datasets are not only different size (6 rows in `patientdata`, 4 rows in `labsdata`), but include information on different patients: `patientdata` has ids 1, 2, 3, 4, 5, 6, `labsdata` has ids 5, 6, 8, 7.

A comprehensive way to join these is to use `full_join()` retaining all information from both tibbles (and matching up rows by shared columns, in this case `id`):

```
full_join(patientdata, labsdata)

## Joining, by = "id"

## # A tibble: 8 × 4
##       id sex     age measurement
##   <dbl> <chr>  <dbl>      <dbl>
## 1     1 Female    24        NA
## 2     2 Male     59        NA
## 3     3 Female   32        NA
## 4     4 Female   84        NA
## 5     5 Male    48      3.47
## 6     6 Female   65      7.31
## 7     8 <NA>      NA      9.91
## 8     7 <NA>      NA      6.11
```

However, if we are only interested in matching information, we use the inner join:

```
inner_join(patientdata, labsdata)
```

```
## Joining, by = "id"

## # A tibble: 2 × 4
##       id sex      age measurement
##   <dbl> <chr>    <dbl>      <dbl>
## 1     5 Male     48        3.47
## 2     6 Female   65        7.31
```

And finally, if we want to retain all information from one tibble, we use either the `left_join()` or the `right_join()`:

```
left_join(patientdata, labsdata)
```

```
## Joining, by = "id"

## # A tibble: 6 × 4
##       id sex      age measurement
##   <dbl> <chr>    <dbl>      <dbl>
## 1     1 Female   24        NA
## 2     2 Male     59        NA
## 3     3 Female   32        NA
## 4     4 Female   84        NA
## 5     5 Male     48        3.47
## 6     6 Female   65        7.31
```

```
right_join(patientdata, labsdata)
```

```
## Joining, by = "id"

## # A tibble: 4 × 4
##       id sex      age measurement
##   <dbl> <chr>    <dbl>      <dbl>
## 1     5 Male     48        3.47
## 2     6 Female   65        7.31
## 3     8 <NA>      NA        9.91
## 4     7 <NA>      NA        6.11
```

2.17.1 Further notes about the joins

- The joins functions (`full_join()`, `inner_join()`, `left_join()`, `right_join()`) will automatically look for matching column names. You can use the `by` = argument to specify by hand. This

is especially useful if the columns are named differently in the datasets, e.g. `left_join(data1, data2, by = c("id" = "patient_id"))`.

- The rows do not have to be ordered, the joins match on values within the rows, not the order of the rows within the tibble.
- Joins are used to combine different variables (columns) into a single tibble. **If you are getting more data of the same variables, use `bind_rows()` instead:**

```
patientdata_new = read_csv("data/patient_data_updated.csv")
patientdata_new
```

```
## # A tibble: 2 × 3
##       id sex      age
##   <dbl> <chr>    <dbl>
## 1     7 Female    38
## 2     8 Male      29
```

```
bind_rows(patientdata, patientdata_new)
```

```
## # A tibble: 8 × 3
##       id sex      age
##   <dbl> <chr>    <dbl>
## 1     1 Female    24
## 2     2 Male      59
## 3     3 Female    32
## 4     4 Female    84
## 5     5 Male      48
## 6     6 Female    65
## 7     7 Female    38
## 8     8 Male      29
```

Finally, it is important to understand how joins behave if there are multiple matches within the tibbles. For example, if patient id 4 had a second measurement as well:

```
labsdata_updated = labsdata %>%
  add_row(id = 5, measurement = 2.49)
labsdata_updated
```

```
## # A tibble: 5 × 2
##       id measurement
##   <dbl>        <dbl>
## 1     5         3.47
```

```
## 2      6      7.31
## 3      8      9.91
## 4      7      6.11
## 5      5      2.49
```

When we now do a `left_join()` with our main tibble - `patientdata`:

```
left_join(patientdata, labsdata_updated)
```

```
## Joining, by = "id"
## # A tibble: 7 x 4
##       id   sex     age measurement
##   <dbl> <chr>   <dbl>      <dbl>
## 1     1 Female    24        NA
## 2     2 Male      59        NA
## 3     3 Female    32        NA
## 4     4 Female    84        NA
## 5     5 Male      48      3.47
## 6     5 Male      48      2.49
## 7     6 Female    65      7.31
```

We get 7 rows, instead of 6 - as patient id 5 now appears twice with the two different measurements. So it is important to keep either know your datasets very well or keep an eye on the number of rows to make sure any increases/decreases in the tibble sizes are as you expect them to be.

3

Summarising data

“The Answer to the Great Question... Of Life, the Universe and Everything... Is... Forty-two,” said Deep Thought, with infinite majesty and calm. Douglas Adams, The Hitchhiker’s Guide to the Galaxy

In this chapter you will find out how to:

- summarise data using: `group_by()`, `summarise()`, and `mutate()`
- reshape data between the wide and long formats: `spread()` and `gather()`
- `select()` columns and `arrange()` (sort) rows

The exercises at the end of this chapter combine all of the above to give context and show you more worked examples.

3.0.1 Dataset: Global Burden of Disease (year, cause, sex, income, deaths)

The Global Burden of Disease dataset used in this chapter is more detailed than the one we used previously. For each year, the total number of deaths from the three broad disease categories are also separated into sex and World Bank income categories. This means that we have 24 rows for each year, and that the total number of deaths per year is the sum of these 24 rows:

```
library(tidyverse)
gbd_full = read_csv("data/global_burden_disease_cause-year-sex-income.csv")

# Creating a single-year tibble for printing and simple examples:
gbd2017 = gbd_full %>%
  filter(year == 2017)
```

TABLE 3.1: Deaths per year (2017) from three broad disease categories, sex, and World Bank country-level income groups.

cause	year	sex	income	deaths millions
Communicable diseases	2017	Female	High	0.26
Communicable diseases	2017	Female	Upper-Middle	0.55
Communicable diseases	2017	Female	Lower-Middle	2.92
Communicable diseases	2017	Female	Low	1.18
Communicable diseases	2017	Male	High	0.29
Communicable diseases	2017	Male	Upper-Middle	0.73
Communicable diseases	2017	Male	Lower-Middle	3.10
Communicable diseases	2017	Male	Low	1.35
Injuries	2017	Female	High	0.21
Injuries	2017	Female	Upper-Middle	0.43
Injuries	2017	Female	Lower-Middle	0.66
Injuries	2017	Female	Low	0.12
Injuries	2017	Male	High	0.40
Injuries	2017	Male	Upper-Middle	1.16
Injuries	2017	Male	Lower-Middle	1.23
Injuries	2017	Male	Low	0.26
Non-communicable diseases	2017	Female	High	4.68
Non-communicable diseases	2017	Female	Upper-Middle	7.28
Non-communicable diseases	2017	Female	Lower-Middle	6.27
Non-communicable diseases	2017	Female	Low	0.92
Non-communicable diseases	2017	Male	High	4.65
Non-communicable diseases	2017	Male	Upper-Middle	8.79
Non-communicable diseases	2017	Male	Lower-Middle	7.30
Non-communicable diseases	2017	Male	Low	1.00

The best way to investigate a dataset is of course to plot it. We have added a couple of notes as comments (the lines starting with a #) for those who can't wait to get to the next chapter where the code for plotting will be introduced and explained in detail. Overall, you shouldn't waste time trying to understand this code here but to look at the different groups within this new dataset.

```
gbd2017 %>%
  # without the mutate(... = fct_relevel())
  # the panels get ordered alphabetically
  mutate(income = fct_relevel(income,
    "Low",
    "Lower-Middle",
    "Upper-Middle",
    "High")) %>%
  # defining the variables using ggplot(aes(...)):
  ggplot(aes(x = sex, y = deaths_millions, fill = cause)) +
  # type of geom to be used: column (that's a type of barplot):
  geom_col(position = "dodge") +
  # facets for the income groups:
  facet_wrap(~income, ncol = 4) +
  # move the legend to the top of the plot (default is "right"):
  theme(legend.position = "top")
```

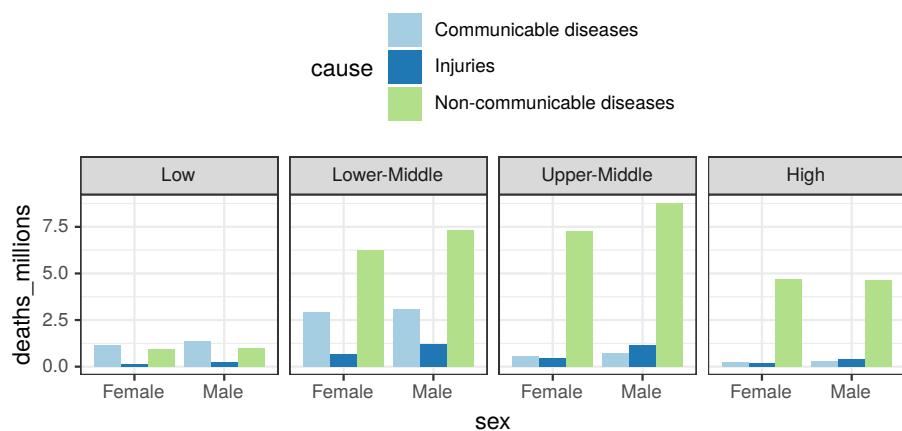


FIGURE 3.1: Global Burden of Disease data with subgroups: cause, sex, World Bank income group.

3.1 Aggregating: `group_by()`, `summarise()`

To quickly calculate the total number of deaths in 2017, we can select the column and send it into the `sum()` function:

```
gbd2017$deaths_millions %>% sum()
## [1] 55.74
```

But a much cleverer way of summarising data is using the `summarise()` function:

```
gbd2017 %>%
  summarise(sum(deaths_millions))

## # A tibble: 1 × 1
##   `sum(deaths_millions)`
##       <dbl>
## 1      55.74
```

And this is indeed equal to the number of deaths per year we were looking at in the shorter version of this data (Deaths from the three causes were 10.38, 4.47, 40.89 which adds to 55.74).

`sum()` is a function that adds numbers together, whereas `summarise()` is a clever and efficient way of creating summarised tibbles. The main strength of `summarise()` is how it works together with the `group_by()` function. We use `group_by()` to tell `summarise()` which sub-groups to apply the calculations on. In the above example, without `group_by()`, `summarise` just works on the whole dataset, yielding the same result as just sending a single column into the `sum()` function. But if we add `group_by()` like this, e.g., for the `cause` variable:

```
gbd2017 %>%
  group_by(cause) %>%
  summarise(sum(deaths_millions))

## # A tibble: 3 × 2
##   cause           `sum(deaths_millions)`
##   <fct>             <dbl>
## 1 All causes       55.74
## 2 Noncommunicable, injur... 40.89
## 3 Injury, violence, and self-harm 10.38
```

```
##   <chr>                      <dbl>
## 1 Communicable diseases      10.38
## 2 Injuries                   4.47
## 3 Non-communicable diseases  40.89
```

Furthermore, `group_by()` is happy accept multiple grouping variables. So by just copying and editing the above code, we can quickly get summarised totals across multiple grouping variables (by just adding `sex` inside the `group_by()` after `cause`):

```
gbd2017 %>%
  group_by(cause, sex) %>%
  summarise(sum(deaths_millions))

## # A tibble: 6 x 3
## # Groups:   cause [3]
##   cause              sex `sum(deaths_millions)`
##   <chr>            <chr>                  <dbl>
## 1 Communicable diseases Female        4.91
## 2 Communicable diseases Male         5.47
## 3 Injuries           Female        1.42
## 4 Injuries           Male         3.05
## 5 Non-communicable diseases Female    19.15
## 6 Non-communicable diseases Male     21.74
```

3.2 Add new columns: `mutate()`

Let's give the summarised column a better name, e.g. `deaths_per_group`. And, if we use `ungroup()` we can use `mutate()` to add a total deaths column, and we can then use it to calculate a percentage:

```
gbd2017 %>%
  group_by(cause, sex) %>%
  summarise(deaths_pergroups = sum(deaths_millions)) %>%
  ungroup() %>%
  mutate(deaths_total = sum(deaths_pergroups))

## # A tibble: 6 x 4
##   cause              sex   deaths_pergroups  deaths_total
##   <chr>            <dbl>             <dbl>
## 1 Communicable diseases 4.91            20.32
## 2 Injuries           5.47            20.32
## 3 Non-communicable diseases 19.15          20.32
```

```

## 1 Communicable diseases   Female      4.91      55.74
## 2 Communicable diseases   Male       5.47      55.74
## 3 Injuries                Female      1.42      55.74
## 4 Injuries                Male       3.05      55.74
## 5 Non-communicable diseases Female    19.15      55.74
## 6 Non-communicable diseases Male     21.74      55.74

```

3.2.1 percentages formatting: `percent()`

So `summarise()` condenses a tibble, whereas `mutate()` retains its current size and adds columns. We can also further lines to `mutate()` to calculate the percentage of each group:

```

# percent() function for formatting percentages come from library(scales)
library(scales)
gbd2017 %>%
  group_by(cause, sex) %>%
  summarise(deaths_pergrroups = sum(deaths_millions)) %>%
  ungroup() %>%
  mutate(deaths_total     = sum(deaths_pergrroups),
        deaths_relative = percent(deaths_pergrroups/deaths_total))

## # A tibble: 6 x 5
##   cause           sex   deaths_pergrroups deaths_total deaths_relative
##   <chr>          <chr>            <dbl>        <dbl> <chr>
## 1 Communicable diseases Female      4.91      55.74  8.8%
## 2 Communicable diseases Male       5.47      55.74  9.8%
## 3 Injuries          Female      1.42      55.74  2.5%
## 4 Injuries          Male       3.05      55.74  5.5%
## 5 Non-communicable diseases Female    19.15      55.74 34.4%
## 6 Non-communicable diseases Male     21.74      55.74 39.0%

```

The `percent()` function that comes from `library(scales)` is a very handy way of formatting percentages, but you have to keep in mind that it changes the column from a number (denoted `<dbl>`) to a character (`<chr>`). The `percent()` function is basically equivalent to:

```

# using values from the first row as an example:
round(100*4.91/55.74, 1) %>% paste0("%")

```

```
## [1] "8.8%"
```

This is very convenient for final presentation of number, but if you

intend to do further calculations/plot/sort the percentages just calculate them as fractions with:

```
mydata %>%
  mutate(deaths_relative = deaths_pergrroups/deaths_total)
```

and convert to nicely formatted percentages later:

```
mydata %>%
  mutate(deaths_percentage = percent(deaths_relative))
```

3.3 *summarise()* VS *mutate()*

So far we've shown you examples of using `summarise()` on grouped data (so following `group_by()`) and `mutate()` on the whole dataset (either without using `group_by()` at all, or resetting the grouping information with `ungroup()`).

But here's the thing: `mutate()` is also happy to work on grouped data!

Let's save the aggregated (one of the `summarise()` examples from above) in a new tibble, and let's `arrange()` the rows based on `sex`, just for easier viewing (it was previously sorted/arranged by `cause`).

The `arrange()` function sorts the rows within a tibble:

```
gbd_summarised = gbd2017 %>%
  group_by(cause, sex) %>%
  summarise(deaths_pergrroups = sum(deaths_millions)) %>%
  arrange(sex)

gbd_summarised
```

```
## # A tibble: 6 x 3
## # Groups:   cause [3]
##   cause                 sex   deaths_pergrroups
##   <chr>                <chr>          <dbl>
## 1 AIDS, HIV and other  female      1.00
## 2 Malaria              female      0.98
## 3 Tuberculosis          female      0.95
## 4 Lower respiratory tract infection  female      0.90
## 5 Diarrhoea, cholera and other enteric infections  female      0.85
## 6 Malaria              male        0.80
```

```
## 1 Communicable diseases   Female      4.91
## 2 Injuries                Female      1.42
## 3 Non-communicable diseases Female    19.15
## 4 Communicable diseases   Male       5.47
## 5 Injuries                Male       3.05
## 6 Non-communicable diseases Male     21.74
```

You should also notice that `summarise()` drops all variables that are not listed in `group_by()` or created inside it. So `year`, `income`, and `deaths_millions` exist in `gbd2017`, but they do not exist in `gbd_summarised`.

We now want to calculate the percentage of deaths from each cause for each gender. We could use `summarise()` to calculate the totals:

```
gbd_summarised_sex =
  gbd_summarised %>%
  group_by(sex) %>%
  summarise(deaths_persex = sum(deaths_pergroups))

gbd_summarised_sex

## # A tibble: 2 x 2
##   sex   deaths_persex
##   <chr>      <dbl>
## 1 Female      25.48
## 2 Male        30.26
```

But that drops the `cause` and `deaths_pergroups` columns. One way would be to now use a join on `gbd_summarised` and `gbd_summarised_sex`:

```
full_join(gbd_summarised, gbd_summarised_sex)

## Joining, by = "sex"

## # A tibble: 6 x 4
## # Groups:   cause [3]
##   cause           sex   deaths_pergroups deaths_persex
##   <chr>          <chr>      <dbl>        <dbl>
## 1 Communicable diseases Female      4.91        25.48
## 2 Injuries         Female      1.42        25.48
## 3 Non-communicable diseases Female    19.15      25.48
## 4 Communicable diseases   Male       5.47        30.26
## 5 Injuries          Male       3.05        30.26
## 6 Non-communicable diseases Male     21.74      30.26
```

And joining different summaries together is a good idea, especially

if the individual pipelines are quite long (e.g., over 5 lines of `%>%`) - as it is reasonable to save the interim results in separate tibbles (like we've saved `gbd_summarised` and `gbd_summarised_sex`), check that they have worked as expected, and then join together.

But in simpler examples, we can use `mutate()` with `group_by()` to achieve the same result as the `full_join()` above:

```
gbd_summarised %>%
  group_by(sex) %>%
  mutate(deaths_persex = sum(deaths_pergroups))
```

```
## # A tibble: 6 x 4
## # Groups:   sex [2]
##   cause           sex   deaths_pergroups deaths_persex
##   <chr>          <chr>        <dbl>       <dbl>
## 1 Communicable diseases Female      4.91      25.48
## 2 Injuries        Female      1.42      25.48
## 3 Non-communicable diseases Female    19.15     25.48
## 4 Communicable diseases Male       5.47      30.26
## 5 Injuries        Male       3.05      30.26
## 6 Non-communicable diseases Male     21.74     30.26
```

So `mutate()` calculates the sums within each grouping variable (in this example just `group_by(sex)`) and puts the results in a new column without condensing the tibble down or removing any of the existing columns.

Let's combine all of this together into a single pipeline and calculate the percentages per cause for each gender:

```
gbd2017 %>%
  group_by(cause, sex) %>%
  summarise(deaths_pergroups = sum(deaths_millions)) %>%
  group_by(sex) %>%
  mutate(deaths_persex = sum(deaths_pergroups),
         sex_cause_perc = percent(deaths_pergroups/deaths_persex)) %>%
  arrange(sex, deaths_pergroups)
```

```
## # A tibble: 6 x 5
## # Groups:   sex [2]
##   cause           sex   deaths_pergroups deaths_persex sex_cause_perc
##   <chr>          <chr>        <dbl>       <dbl> <chr>
## 1 Injuries        Fema~      1.42      25.48  5.6%
## 2 Communicable diseases Fema~     4.91      25.48 19.3%
## 3 Non-communicable dis~ Fema~    19.15     25.48 75.2%
```

```
## 4 Injuries           Male      3.05      30.26 10.1%
## 5 Communicable diseases Male     5.47      30.26 18.1%
## 6 Non-communicable dis~ Male    21.74     30.26 71.8%
```

3.4 Common arithmetic functions - `sum()`, `mean()`, `median()`, etc.

Statistics is what R does, so if there is a statistical function you can think of, it will exist in R.

The most common ones are:

- `sum()`
- `mean()`
- `median()`
- `min()`, `max()`
- `sd()` - standard deviation
- `IQR()` - inter-quartile range

The import thing to remember about all of these is that if any of the values is NA (not applicable/not available), these functions will return an NA. Either deal with your missing values beforehand (recommended) or add the `na.rm = TRUE` argument into any of the above functions to ask R to ignore missing values. More discussion and examples around missing data can be found in Chapters 2 and ??.

```
mynumbers = c(1, 2, NA)
sum(mynumbers)
```

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

```
sum(mynumbers, na.rm = TRUE)
```

```
## [1] 3
```

Overall, R's unwillingness to implicitly average over observations

with missing values should be considered helpful, not an unnecessary pain. Thing is, if you don't know exactly where your missing values are/how many, you might end up comparing the averages of very different groups (if the values are not missing and random or the sample size is small). So the `na.rm = TRUE` is fine to use if quickly exploring and cleaning data, or you've already investigated missing values and are convinced the existing ones are representative. But it is rightfully not a default so get used to typing `na.rm = TRUE` when using these functions.

3.5 Reshaping data - long vs wide format

So far, all of the example we've shown you have been using 'tidy' data. Data is 'tidy' where each variable is in its own column, and each observation is in its own row. This long looking format is efficient to use in data analysis and visualisation, it can also be referred to as "computer readable". But sometimes when presenting data in tables for humans to have a look, or when collecting data directly into a Spreadsheet, it can be convenient to have data in a wide format.

```
gbd_wide = read_csv("data/global_burden_disease_wide-format.csv")
gbd_long = read_csv("data/global_burden_disease_cause-year-sex.csv")
```

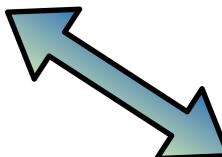
TABLE 3.2: Global Burden of Disease data in human-readable wide format. This is not tidy data.

cause	Female-1990	Female-2017	Male-1990	Male-2017
Communicable diseases	7.30	4.91	8.06	5.47
Injuries	1.41	1.42	2.84	3.05
Non-communicable diseases	12.80	19.15	13.91	21.74

Tables 3.3 and 3.2 contain the exact same information, but in long (tidy) and wide formats, respectively.

TABLE 3.3: Global Burden of Disease data in analysis-friendly long format. This is tidy data.

cause	year	sex	deaths_millions
Communicable diseases	1990	Female	7.30
Communicable diseases	2017	Female	4.91
Communicable diseases	1990	Male	8.06
Communicable diseases	2017	Male	5.47
Injuries	1990	Female	1.41
Injuries	2017	Female	1.42
Injuries	1990	Male	2.84
Injuries	2017	Male	3.05
Non-communicable diseases	1990	Female	12.80
Non-communicable diseases	2017	Female	19.15
Non-communicable diseases	1990	Male	13.91
Non-communicable diseases	2017	Male	21.74



Developed countries:		1990	1995	2000	2005	2010	2013
Communicable diseases		0.6	0.7	0.7	0.7	0.7	0.7
Injuries		0.9	1	0.9	0.9	0.8	0.8
Non-communicable diseases		9.7	10.8	10.7	11.1	11.3	11.6

Developing countries:		1990	1995	2000	2005	2010	2013
Communicable diseases		15.5	14.8	14.1	13.2	11.8	11.1
Injuries		3.5	3.6	3.8	3.9	4.1	4.3
Non-communicable diseases		17.3	19.1	20.8	22.5	24.2	25.9

location	cause	year	deaths_millions
Developed	Communicable diseases	1990	0.6
Developed	Communicable diseases	1995	0.7
Developed	Communicable diseases	2000	0.7
Developed	Communicable diseases	2005	0.7
Developed	Communicable diseases	2010	0.7
Developed	Communicable diseases	2013	0.7
Developed	Injuries	1990	0.9
Developed	Injuries	1995	1
Developed	Injuries	2000	0.9
Developed	Injuries	2005	0.9
Developed	Injuries	2010	0.8
Developed	Injuries	2013	0.8
Developed	Non-communicable diseases	1990	9.7
Developed	Non-communicable diseases	1995	10.8
Developed	Non-communicable diseases	2000	10.7
Developed	Non-communicable diseases	2005	11.1
Developed	Non-communicable diseases	2010	11.3
Developed	Non-communicable diseases	2013	11.6
Developing	Communicable diseases	1990	15.5
Developing	Communicable diseases	1995	14.8

FIGURE 3.2: Same data in the long ('tidy', necessary for efficient analysis) and wide (easier for human-readability/presentation/manual data entry) formats. TODO: replace with updated data.

3.5.1 `spread()` values from rows into columns

If we want to take the data from 3.3 and put some of the numbers next to each other for easier comparison, then `spread()` is the function to do it. It means we want to spread a variable into columns, and it needs just two arguments: the column we want to spread, and the column where the values are.

```
gbd_long %>%
  spread(year, deaths_millions)

## # A tibble: 6 x 4
##   cause           sex   `1990` `2017`
##   <chr>          <chr>   <dbl>   <dbl>
## 1 Communicable diseases Female    7.3    4.91
## 2 Communicable diseases Male     8.06   5.47
## 3 Injuries        Female   1.41   1.42
## 4 Injuries        Male    2.84   3.05
## 5 Non-communicable diseases Female 12.8   19.15
## 6 Non-communicable diseases Male   13.91  21.74
```

In this example, we are sending `gbd_long` into the `spread(year, deaths_millions)` to put the year variable into different columns, the values to fill the new columns with are `deaths_millions`. This means we can quickly eyeball how the number of deaths have changed from 1990 to 2017 for each cause category and sex. Whereas if we wanted to quickly look at the difference in the number of deaths for Females and Males, we can just the `sex` variable instead, so `spread(sex, deaths_millions)`. Furthermore, we can now add a `mutate()` to show this difference in a new column:

```
gbd_long %>%
  spread(sex, deaths_millions) %>%
  mutate(Male - Female)

## # A tibble: 6 x 5
##   cause           year Female  Male `Male - Female`
##   <chr>          <dbl>   <dbl>   <dbl>            <dbl>
## 1 Communicable diseases 1990    7.3    8.06      0.76
## 2 Communicable diseases 2017    4.91   5.47      0.5600
## 3 Injuries          1990    1.41   2.84      1.430
## 4 Injuries          2017    1.42   3.05      1.63
## 5 Non-communicable diseases 1990   12.8   13.91     1.110
## 6 Non-communicable diseases 2017   19.15  21.74     2.59
```

All of these differences are positive which means every year, more men die than women. Which make sense, as more boys are also born than girls.

And what if we want to spread both `year` and `sex` at the same time, so to create table 3.2 from Table 3.3? Since `spread()` can only accept two columns - first that is to be spread into columns, second where the values come from, we need to combine something beforehand. This is what the `unite()` function is for:

```
gbd_long %>%
  unite(sex_year, c(sex, year)) %>%
  slice(1:2)

## # A tibble: 2 x 3
##   cause           sex_year   deaths_millions
##   <chr>          <chr>                <dbl>
## 1 Communicable diseases Female_1990        7.3
## 2 Communicable diseases Female_2017        4.91
```

We are using the `slice(1:2)` to select the first two rows - just for efficient printing (`:` in R is a shorthand for creating sequential numbers, e.g. `1:4` is 1, 2, 3, 4).

We can then `spread()` the united column:

```
gbd_long %>%
  unite(sex_year, c(sex, year)) %>%
  spread(sex_year, deaths_millions)

## # A tibble: 3 x 5
##   cause           Female_1990 Female_2017 Male_1990 Male_2017
##   <chr>            <dbl>      <dbl>     <dbl>      <dbl>
## 1 Communicable diseases    7.3       4.91      8.06      5.47
## 2 Injuries             1.41       1.42      2.84      3.05
## 3 Non-communicable diseases 12.8      19.15     13.91     21.74
```

Both `spread()` and `unite()` have a few optional arguments that you may be useful for you. For example, `spread(..., fill = 0)` is used to fill empty cells (default is `fill = NA`). Or `unite(..., sep = " ")` can be used to change the separator that gets put between the values (e.g. you may want “Female-1990” or “Female: 1990” instead of the default “`_`”). Remember that pressing F1 when your cursor is on

a function opens it up in the Help tab where these extra options are listed.

The `unite()` is a very convenient function for pasting values from multiple columns together, but if you want to do something more special (i.e. also round numbers or add different separators between multiple different columns), then the `paste()` function inside `mutate()` will give you that extra flexibility and control.

So for example, this:

```
gbd_long %>%
  unite(sex_year, c(sex, year)) %>%
  slice(1)

## # A tibble: 1 × 3
##   cause           sex_year   deaths_millions
##   <chr>          <chr>            <dbl>
## 1 Communicable diseases Female_1990      7.3
```

is similar to:

```
gbd_long %>%
  mutate(sex_year = paste(sex, year, sep = "_")) %>%
  slice(1)

## # A tibble: 1 × 5
##   cause           year sex   deaths_millions sex_year
##   <chr>          <dbl> <chr>            <dbl> <chr>
## 1 Communicable diseases 1990 Female      7.3 Female_1990
```

They're similar but not exactly the same as `unite()` drops the original columns (and only keeps the new united one), whereas `mutate()` creates a new column and keeps all existing ones as well. To make them equivalent, you could either add `drop = FALSE` inside `unite()` (keeping all columns) or `%>% select(-sex, -year)` after the `mutate()` (to drop/deselect these).

3.5.2 `gather()` values from columns to rows

The opposite of `spread()` is `gather()`. If you're lucky enough, your data comes from a proper database (e.g., REDCap) and is already

in the long and tidy format. But if you do get landed with something that looks like the Table 3.2, you'll need to know how to gather and separate the variables currently spread across different columns into the tidy format (each column is a variable, each row is an observation).

We could try and run `gather()` without any extra arguments (again, using `slice(1:6)` just for shorter printing, the first 6 lines this time):

```
gbd_wide %>%
  gather() %>%
  slice(1:6)

## # A tibble: 6 x 2
##   key      value
##   <chr>    <chr>
## 1 cause    Communicable diseases
## 2 cause    Injuries
## 3 cause    Non-communicable diseases
## 4 Female-1990 7.3
## 5 Female-1990 1.41
## 6 Female-1990 12.8
```

So it gathers all column names into a new variable called `key`, and puts everything in the rows into a column called `value`. However, the `cause` variable already was how we wanted it - in a column of its own, so we don't want this gathered together the `deaths_millions` values. So we can tell `gather()` to leave it where it is:

```
gbd_wide %>%
  gather(sex_year, deaths_millions, -cause) %>%
  slice(1:6)

## # A tibble: 6 x 3
##   cause           sex_year  deaths_millions
##   <chr>          <chr>            <dbl>
## 1 Communicable diseases Female-1990     7.3
## 2 Injuries        Female-1990     1.41
## 3 Non-communicable diseases Female-1990   12.8
## 4 Communicable diseases Female-2017     4.91
## 5 Injuries        Female-2017     1.42
## 6 Non-communicable diseases Female-2017  19.15
```

Now there, because selection (or deselection) of columns needs to be the fourth argument to `gather()` (first on is the data that gets

piped - `%>%` in, second and third the names of the new columns), we also need to include the names of the new columns before we can specify that we want `-cause` to stay where it is.

And finally, we need to use `separate()` to put `sex` and `year` into their own columns:

```
gbd_wide %>%
  gather(sex_year, deaths_millions, -cause) %>%
  separate(sex_year, into = c("sex", "year"), convert = TRUE) %>%
  slice(1:6)

## # A tibble: 6 x 4
##   cause           sex     year deaths_millions
##   <chr>          <chr>   <int>        <dbl>
## 1 Communicable diseases Female  1990        7.3
## 2 Injuries         Female  1990       1.41
## 3 Non-communicable diseases Female  1990      12.8
## 4 Communicable diseases Female  2017       4.91
## 5 Injuries         Female  2017       1.42
## 6 Non-communicable diseases Female  2017      19.15
```

It is important to notice the quotes around the new column names: `into = c("sex", "year")`. Most tidyverse functions don't want to use quotes around column names, so this can be confusing. But these columns don't exist yet, so in this case, `sex` and `year` are not variables, they are new names. We've also added `convert = TRUE` to `separate()` so `year` would get converted into a numeric variable. The combination of, e.g., “Female-1990” is a character variable, so after separating them both `sex` and `year` would still be classified as characters. But the `convert = TRUE` recognises that `year` is a number and will appropriately convert it into an integer.

When working with large datasets with a lot of columns that need gathering, then the `select()` helpers are extremely useful.

3.6 select() columns

The `select()` function can be used to choose, rename or reorder columns of a tibble.

For the following `select()` examples, let's create a new tibble called `gbd_2rows` by taking the first 2 rows of `gbd_full` (just for shorter printing):

```
gbd_2rows = gbd_full %>%
  slice(1:2)

gbd_2rows

## # A tibble: 2 × 5
##   cause           year sex   income   deaths_millions
##   <chr>          <dbl> <chr> <chr>        <dbl>
## 1 Communicable diseases 1990 Female High      0.21
## 2 Communicable diseases 1990 Female Upper-Middle 1.150
```

Let's `select()` two of these columns:

```
gbd_2rows %>%
  select(cause, deaths_millions)

## # A tibble: 2 × 2
##   cause           deaths_millions
##   <chr>            <dbl>
## 1 Communicable diseases 0.21
## 2 Communicable diseases 1.150
```

We can also use `select()` to rename the columns we are choosing:

```
gbd_2rows %>%
  select(cause, deaths = deaths_millions)

## # A tibble: 2 × 2
##   cause           deaths
##   <chr>            <dbl>
## 1 Communicable diseases 0.21
## 2 Communicable diseases 1.150
```

The `function rename()` is similar to `select()`, but it keeps all variables whereas `select()` only kept the ones we mentioned:

```
gbd_2rows %>%
  rename(deaths = deaths_millions)

## # A tibble: 2 x 5
##   cause           year sex   income   deaths
##   <chr>          <dbl> <chr> <chr>     <dbl>
## 1 Communicable diseases 1990 Female High      0.21
## 2 Communicable diseases 1990 Female Upper-Middle 1.150
```

`select()` can also be used to reorder the columns in your tibble. Moving columns around is not relevant in data analysis (as any of the functions we showed you above, as well as plotting, only look at the column names, and not their positions in the tibble), but it is useful for organising your tibble for easier viewing.

So if we use `select` like this:

```
gbd_2rows %>%
  select(year, sex, income, cause, deaths_millions)

## # A tibble: 2 x 5
##   year sex   income   cause       deaths_millions
##   <dbl> <chr> <chr> <chr>     <dbl>
## 1 1990 Female High   Communicable diseases      0.21
## 2 1990 Female Upper-Middle Communicable diseases 1.150
```

The columns are reordered.

If you want to move specific column(s) to the front to the tibble, do:

```
gbd_2rows %>%
  select(year, sex, everything())

## # A tibble: 2 x 5
##   year sex   cause       income   deaths_millions
##   <dbl> <chr> <chr> <chr>     <dbl>
## 1 1990 Female Communicable diseases High      0.21
## 2 1990 Female Communicable diseases Upper-Middle 1.150
```

And this is where the true power of `select()` starts to come

out. In addition to listing the columns explicitly (e.g., `mydata %>% select(year, cause...)`) there are several special functions that can be used inside `select()`. These special functions are called select helpers, and the first select helper we used is `everything()`.

The most common select helpers are `starts_with()`/`ends_with()` and `matches()` (but there are several others that may be useful to you, so press F1 on `select()` for a full list, or search the web for more examples).

Let's say you can remember, whether the deaths column was called `deaths_millions` or just `deaths` OR `deaths_mil`, or maybe there are other columns that include the word “deaths” that you want to `select()`:

```
gbd_2rows %>%
  select(starts_with("deaths"))
```

```
## # A tibble: 2 × 1
##   deaths_millions
##       <dbl>
## 1          0.21
## 2         1.150
```

Note how “deaths” needs to be quoted inside `starts_with()` - as it’s a word to look for, not the real name of a column/variable.

3.6.1 using select helpers in other functions

The select helpers may seem a bit extra in the simple example above, but they become invaluable when working with, for example, biobank data - as you’ll be landed with hundreds of columns, often named something like `hosp_ep_01`, `hosp_ep_02`, `hosp_ep_03`, `hosp_ep_04`, `hosp_ep_05`, `hosp_ep_06...`, so `starts_with("hosp_ep")` can really become your new best friend.

Furthermore, the select helpers can be used in some of the other tidyverse functions as well, namely `mutate_at()`, `summarise_at()`, and especially `gather()`:

```
# made-up biobank-like tibble:  
gather_helperdata = tibble(id = "ID-1",  
                           already = "Yes",  
                           tidy = TRUE,  
                           hosp_ep_01 = "Y92.241",  
                           hosp_ep_02 = "W61.43",  
                           hosp_ep_03 = "V91.07",  
                           hosp_ep_04 = "V95.43")  
  
gather_helperdata  
  
## # A tibble: 1 × 7  
##   id    already  tidy  hosp_ep_01 hosp_ep_02 hosp_ep_03 hosp_ep_04  
##   <chr> <chr>   <lgl> <chr>      <chr>      <chr>      <chr>  
## 1 ID-1  Yes     TRUE  Y92.241    W61.43    V91.07    V95.43  
  
gather_helperdata %>%  
  gather(episode, diagnosis, starts_with("hosp_ep"))  
  
## # A tibble: 4 × 5  
##   id    already  tidy  episode  diagnosis  
##   <chr> <chr>   <lgl> <chr>    <chr>  
## 1 ID-1  Yes     TRUE  hosp_ep_01 Y92.241  
## 2 ID-1  Yes     TRUE  hosp_ep_02 W61.43  
## 3 ID-1  Yes     TRUE  hosp_ep_03 V91.07  
## 4 ID-1  Yes     TRUE  hosp_ep_04 V95.43
```

3.7 `arrange()` rows

The `arrange()` functions sorts rows based on the column(s) you want. By default, it arranges the tibble ascendingly:

```
gbd_long %>%  
  arrange(deaths_millions) %>%  
  # first 3 rows just for printing:  
  slice(1:3)  
  
## # A tibble: 3 × 4  
##   cause      year sex  deaths_millions  
##   <chr>     <dbl> <chr>          <dbl>  
## 1 Injuries  1990 Female        1.41
```

```
## 2 Injuries 2017 Female      1.42
## 3 Injuries 1990 Male       2.84
```

For numeric variables, we can just use a - to sort descendingly:

```
gbd_long %>%
  arrange(-deaths_millions) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 4
##   cause           year sex   deaths_millions
##   <chr>          <dbl> <chr>            <dbl>
## 1 Non-communicable diseases 2017 Male        21.74
## 2 Non-communicable diseases 2017 Female     19.15
## 3 Non-communicable diseases 1990 Male        13.91
```

Whereas the - doesn't work for categorical variables, they need to be put in `desc()` for arranging descendingly:

```
gbd_long %>%
  arrange(desc(sex)) %>%
  # printing rows 1, 2, 11, and 12
  slice(1, 2, 11, 12)
```

```
## # A tibble: 4 x 4
##   cause           year sex   deaths_millions
##   <chr>          <dbl> <chr>            <dbl>
## 1 Communicable diseases 1990 Male        8.06
## 2 Communicable diseases 2017 Male        5.47
## 3 Non-communicable diseases 1990 Female    12.8
## 4 Non-communicable diseases 2017 Female    19.15
```

3.7.1 factor levels

`arrange()` sorts characters alphabetically, whereas factors will be sorted by the order of their levels. Let's make the cause column into a factor:

```
gbd_factored = gbd_long %>%
  mutate(cause = factor(cause))
```

When we first create a factor, its levels will be ordered alphabetically:

```
gbd_factored$cause %>% levels()  
  
## [1] "Communicable diseases"      "Injuries"  
## [3] "Non-communicable diseases"
```

But we can now use `fct_relevel()` inside `mutate()` to change the order of these levels:

```
gbd_factored = gbd_factored %>%  
  mutate(cause = cause %>%  
        fct_relevel("Injuries"))  
  
gbd_factored$cause %>% levels()  
  
## [1] "Injuries"                  "Communicable diseases"  
## [3] "Non-communicable diseases"
```

`fct_relevel()` brings the level(s) listed in it to the front.

So if we use `arrange()` on `gbd_factored`, the `cause` column will be sorted based on the order of its levels, not alphabetically. This is especially useful in two places:

- plotting - categorical variables that are characters will be ordered alphabetically (e.g., think barplots), regardless of whether the rows are arranged or not.
- statistical tests - the reference level of categorical variables that are characters is the alphabetically first (e.g., what the odds ratio is relative to).

However, making a character column into a factor gives us power to give its levels a non-alphabetical order, giving us control over plotting order or defining our reference levels for use in statistical tests.

3.8 Exercise - `spread()`

Using the GBD dataset with variables `cause`, `year` (1990 and 2017 only), `sex` (as shown in Table 3.3):

```
gbd_long = read_csv("data/global_burden_disease_cause-year-sex.csv")
```

Spread the `cause` variable into columns using the `deaths_millions` as values:

TABLE 3.4: Exercise: putting the cause variable into the wide format using `spread`.

year	sex	Communicable diseases	Injuries	Non-communicable diseases
1990	Female	7.30	1.41	12.80
1990	Male	8.06	2.84	13.91
2017	Female	4.91	1.42	19.15
2017	Male	5.47	3.05	21.74

Solution

```
gbd_long = read_csv("data/global_burden_disease_cause-year-sex.csv")
gbd_long %>%
  spread(cause, deaths_millions)
```

3.9 Exercise - `group_by()`, `summarise()`

Read in the full GBD dataset with variables `cause`, `year`, `sex`, `income`, `deaths_millions`.

```
gbd_full = read_csv("data/global_burden_disease_cause-year-sex-income.csv")
glimpse(gbd_full)
```

```
## Observations: 168
## Variables: 5
## $ cause      <chr> "Communicable diseases", "Communicable disease...
## $ year       <dbl> 1990, 1990, 1990, 1990, 1990, 1990, 1990...
## $ sex        <chr> "Female", "Female", "Female", "Female", "Male"...
## $ income     <chr> "High", "Upper-Middle", "Lower-Middle", "Low",...
## $ deaths_millions <dbl> 0.21, 1.15, 4.43, 1.51, 0.26, 1.35, 4.73, 1.72...
```

Year 2017 of this dataset was shown in Table 3.1, the full dataset has seven times as many observations as Table 3.1 since it includes information about multiple years: 1990, 1995, 2000, 2005, 2010, 2015, 2017.

Investigate these code examples:

```
summary_data1 =
  gbd_full %>%
  group_by(year) %>%
  summarise(total_per_year = sum(deaths_millions))
```

```
summary_data1

## # A tibble: 7 x 2
##   year total_per_year
##   <dbl>        <dbl>
## 1 1990        46.32
## 2 1995        48.91
## 3 2000        50.38
## 4 2005        51.25
## 5 2010        52.63
## 6 2015        54.62
## 7 2017        55.74
```

```
summary_data2 =
  gbd_full %>%
  group_by(year, cause) %>%
  summarise(total_per_cause = sum(deaths_millions))
```

```
summary_data2

## # A tibble: 21 x 3
## # Groups:   year [7]
##   year cause          total_per_cause
##   <dbl> <chr>            <dbl>
## 1 1990 Communicable diseases    15.36
## 2 1990 Injuries             4.25
## 3 1990 Non-communicable diseases 26.71
## 4 1995 Communicable diseases    15.11
```

```

## 5 1995 Injuries          4.53
## 6 1995 Non-communicable diseases 29.27
## 7 2000 Communicable diseases    14.81
## 8 2000 Injuries            4.56
## 9 2000 Non-communicable diseases 31.01
## 10 2005 Communicable diseases   13.89
## # ... with 11 more rows

```

You should recognise that:

- `summary_data1` includes the total number of deaths per year.
- `summary_data2` includes the number of deaths per cause per year.
- `summary_data1 =` means we are creating a new tibble called `summary_data1` and saving (=) results into it. If `summary_data1` was a tibble that already existed, it would get overwritten.
- `gbd_full` is the data being sent to the `group_by()` and then `summarise()` functions.
- `group_by()` tells `summarise()` that we want aggregated results for each year.
- `summarise()` then creates a new variable called `total_per_year` that sums the deaths from each different observation (subcategory) together.
- Calling `summary_data1` on a separate line gets it printed.
- We then do something very similar in `summary_data2`.

Compare the number of rows (observations) and number of columns (variables) of `gbd_full`, `summary_data1`, and `summary_data2`.

You should notice that: * `summary_data2` has exactly 3 times as many rows (observations) as `summary_data1`. Why? * `gbd_full` has 5 variables, whereas the summarised tibbles have 2 and 3. Which variables got dropped? How?

Answers

- `gbd_full` has 168 observations (rows),
- `summary_data1` has 7,
- `summary_data2` has 21.

`summary_data1` was grouped by year, therefore it includes a (summarised) value for each year in the original dataset. `summary_data2`

was grouped by year and cause (Communicable diseases, Injuries, Non-communicable diseases), so it has 3 values for each year.

The columns a `summarise()` function returns are: variables listed in `group_by()` + variables created inside `summarise()` (e.g., in this case `deaths_peryear`). All others get aggregated.

3.10 Exercise - `full_join()`, `percent()`

For each cause, calculate its percentage to total deaths in each year.

Hint: Use `full_join()` on `summary_data1` and `summary_data2`, and then use `mutate()` to add a new column called `percentage`.

Example result for a single year:

```
## Joining, by = "year"  
## # A tibble: 3 x 5  
##   year total_per_year cause      total_per_cause percentage  
##   <dbl>        <dbl> <chr>          <dbl> <chr>  
## 1 1990        46.32 Communicable diseases    15.36 33.2%  
## 2 1990        46.32 Injuries            4.25  9.2%  
## 3 1990        46.32 Non-communicable diseases 26.71 57.7%
```

Solution

```
## Joining, by = "year"  
## # A tibble: 21 x 5  
##   year total_per_year cause      total_per_cause percentage  
##   <dbl>        <dbl> <chr>          <dbl> <chr>  
## 1 1990        46.32 Communicable diseases    15.36 33.2%  
## 2 1990        46.32 Injuries            4.25  9.2%  
## 3 1990        46.32 Non-communicable diseases 26.71 57.7%  
## 4 1995        48.91 Communicable diseases    15.11 30.9%  
## 5 1995        48.91 Injuries            4.53  9.3%  
## 6 1995        48.91 Non-communicable diseases 29.27 59.8%  
## 7 2000        50.38 Communicable diseases    14.81 29.4%  
## 8 2000        50.38 Injuries            4.56  9.1%  
## 9 2000        50.38 Non-communicable diseases 31.01 61.6%  
## 10 2005       51.25 Communicable diseases    13.89 27.1%  
## # ... with 11 more rows
```

3.11 Exercise - `mutate()`, `summarise()`

Instead of creating the two summarised tibbles and using a `full_join()`, achieve the same result as in the previous Exercise by with a single pipeline using `summarise()` and then `mutate()`.

Hint: you have to do it the either way round, so `group_by(year, cause) %>% summarise(...)` first, then `group_by(year) %>% mutate()`.

Bonus: `select()` columns `year`, `cause`, `percentage`, then `spread()` the `cause` variable using `percentage` as values.

Solution

```
gbd_full %>%
  # aggregate to deaths per cause per year using summarise()
  group_by(year, cause) %>%
  summarise(total_per_cause = sum(deaths_millions)) %>%
  # then add a column of yearly totals using mutate()
  group_by(year) %>%
  mutate(total_per_year = sum(total_per_cause)) %>%
  # add the percentage column
  mutate(percentage = percent(total_per_cause/total_per_year)) %>%
  # select the final variables and spread for better viewing
  select(year, cause, percentage) %>%
  spread(cause, percentage)
```

```
## # A tibble: 7 x 4
## # Groups:   year [7]
##   year `Communicable diseases` Injuries `Non-communicable diseases`
##   <dbl> <chr>          <chr>      <chr>
## 1 1990 33.2%           9.2%       57.7%
## 2 1995 30.9%           9.3%       59.8%
## 3 2000 29.4%           9.1%       61.6%
## 4 2005 27.1%           8.8%       64.1%
## 5 2010 23.8%           8.9%       67.3%
## 6 2015 19.9%           8.2%       71.9%
## 7 2017 18.6%           8.0%       73.4%
```

Note that your pipelines shouldn't be much longer than this, and we often save interim results into separate tibbles for checking (like we did with `summary_data1` and `summary_data2`, making sure the number of rows are what we expect and spot checking that the calculation worked as expected).

R doesn't do what you want it to do, it does what you ask it to do. Testing and spot checking is essential as you will make mistakes. We sure do.

Do not feel like you should be able to just bash out these clever pipelines without a lot of trial and error first.

3.12 Exercise - `filter()`, `summarise()`, `spread()`

Still working with `gbd_full`:

- Filter for 1990.
- Calculate the total number of deaths in the different income groups (High, Upper-Middle, Lower-Middle, Low). Hint: use `group_by(income)` and `summarise(new_column_name = sum(variable))`.
- Calculate the total number of deaths within each income group for males and females. Hint: this is as easy as adding `, sex` to `group_by(income)`.
- `spread()` the `income` column.

Solution

```
gbd_full %>%
  filter(year == 1990) %>%
  group_by(income, sex) %>%
  summarise(total_deaths = sum(deaths_millions)) %>%
  spread(income, total_deaths)
```

```
## # A tibble: 2 x 5
##   sex     High   Low `Lower-Middle` `Upper-Middle`
##   <chr>  <dbl> <dbl>        <dbl>        <dbl>
## 1 Female  4.140  2.22        8.47       6.68
## 2 Male    4.46   2.57        9.83      7.950
```



4

Different types of plots

What I can not create, I do not understand.
Richard Feynman

There are a few different plotting packages in R, but the most elegant and versatile one is `ggplot2`¹. **gg** stands for grammar of graphics which means that we can make a plot by describing it one component at a time. In other words, we build a plot by adding layers to it.

This does not have to be many layers, the simplest `ggplot()` consist of just two components:

- the variables to be plotted,
- a geometrical object (e.g., point, line, bar, box, etc.).

`ggplot()` calls geometrical objects **geoms**.

Figure 4.1 shows some example steps for building a scatter plot, including changing its appearance ('theme') and facetting - an efficient way of creating separate plots for subgroups.

¹The name of the package is `ggplot2`, but the function is called `ggplot()`. For everything you've ever wanted to know about the grammar of graphics in R, see [Wickham \(2016\)](#) .

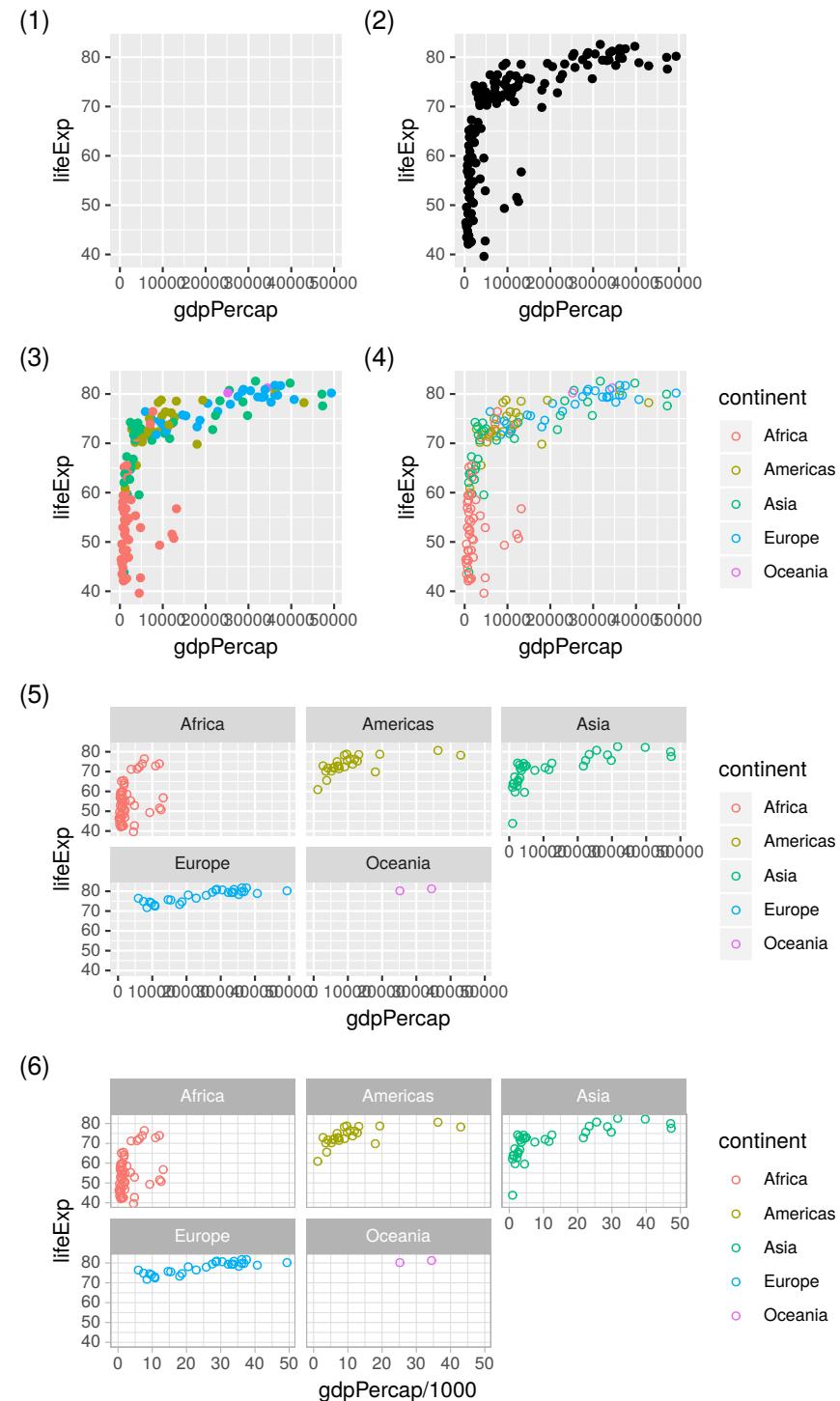


FIGURE 4.1: Example steps for building and modifying a ggplot. (1) initialising the canvas, defining variables, (2) adding points, (3) colouring points by continent, (4) changing point type, (5) facetting, (6) changing the plot theme and the scale of the x variable.

4.1 Data - gapminder

We are using the gapminder dataset (<https://www.gapminder.org/data>) that has been put into an R package by Bryan (2017) so we can load it with `library(gapminder)`.

```
library(tidyverse)
library(gapminder)

glimpse(gapminder)

## Observations: 1,704
## Variables: 6
## $ country <fct> Afghanistan, Afghanistan, Afghanistan, Afghanistan, ...
## $ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia...
## $ year     <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992...
## $ lifeExp  <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.8...
## $ pop      <int> 8425333, 9240934, 10267083, 11537966, 13079460, 1488...
## $ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 78...
```

The dataset includes 1704 observations (rows) of 6 variables (columns: country, continent, year, lifeExp, pop, gdpPercap). `country`, `continent`, and `year` could be thought of as grouping variables, whereas `lifeExp` (life expectancy), `pop` (population), and `gdpPercap` (Gross Domestic Product per capita) are values.

The years in this dataset span 1952 to 2007 with 5-year intervals (so a total of 12 different years). It includes 142 countries from 5 continents (Asia, Europe, Africa, Americas, Oceania).

The way for you to double check that all of the numbers we've quoted above are correct, run these lines:

```
library(tidyverse)
library(gapminder)
gapminder$year %>% unique()
gapminder$country %>% n_distinct()
gapminder$continent %>% unique()
```

Let's create a new shorter tibble called `gapminder2007` that only includes data for the year 2007.

```
gapminder2007 = gapminder %>%
  filter(year == 2007)

gapminder2007

## # A tibble: 142 x 6
##   country   continent year lifeExp      pop gdpPercap
##   <fct>     <fct>    <int>   <dbl>     <int>     <dbl>
## 1 Afghanistan Asia     2007    43.8  31889923     975.
## 2 Albania     Europe   2007    76.4  3600523      5937.
## 3 Algeria     Africa   2007    72.3  33333216     6223.
## 4 Angola      Africa   2007    42.7  12420476     4797.
## 5 Argentina   Americas 2007    75.3  40301927    12779.
## 6 Australia   Oceania  2007    81.2  20434176    34435.
## 7 Austria     Europe   2007    79.8  8199783     36126.
## 8 Bahrain     Asia     2007    75.6  708573      29796.
## 9 Bangladesh  Asia     2007    64.1  150448339    1391.
## 10 Belgium    Europe   2007    79.4  10392226    33693.
## # ... with 132 more rows
```

The new tibble - `gapminder2007` - now shows up in your Environment tab, whereas `gapminder` does not. Running `library(gapminder)` makes it available to use (so the funny line below is not necessary for any of the code in this chapter to work), but to have it appear in your normal Environment tab you'll need to run this funny looking line:

```
# loads the gapminder dataset from the package environment
# into your Global Environment
gapminder = gapminder
```

Both `gapminder` and `gapminder2007` now show up in the Environment tab and can be clicked on/quickly viewed as usual.

4.2 Anatomy of ggplot explained

We will now explain the six steps shown in Figure 4.1. Note that you only need the first two to make a plot, the rest are just to show you further functionality and optional customisations.

- (1) Start by defining the variables, e.g., `ggplot(aes(x = var1, y = var2))`:

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap, y = lifeExp))
```

This creates the first plot in Figure 4.1.

Although the above code is equivalent to:

```
ggplot(gapminder2007, aes(x = gdpPercap, y = lifeExp))
```

we tend to put the data first and then use the pipe (`%>%`) to send it to the `ggplot()` function. This becomes useful when we add further data wrangling functions between the data and the `ggplot()`. For example, our plotting pipelines often look like this:

```
data %>%
  filter(...) %>%
  mutate(...) %>%
  ggplot(aes(...)) +
  ...
```

The lines that come before the `ggplot()` function are piped, whereas from `ggplot()` onwards you have to use `+`. This is because we are now adding different layers and customisations to the same plot.

`aes()` stands for **aesthetics** - things we can see. Variables are always inside the `aes()` function, which in turn is inside a `ggplot()`. Take a moment to appreciate the double closing brackets `))` - the first one belongs to `aes()`, the second one to `ggplot()`.

- (2) Choose and add a geometrical object

Let's ask `ggplot()` to draw a point for each observation by adding `geom_point()`:

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

We have now created the second plot in Figure 4.1, a scatter plot.

If we copy the above code and change just one thing - the `x` variable from `gdpPercap` to `continent` (which is a categorical variable) - we get what's called a strip plot. This means we are now plotting a continuous variable (`lifeExp`) against a categorical one (`continent`). But the thing to note is that the rest of the code stays exactly the same, all we did was change the `x =`.

```
gapminder2007 %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  geom_point()
```

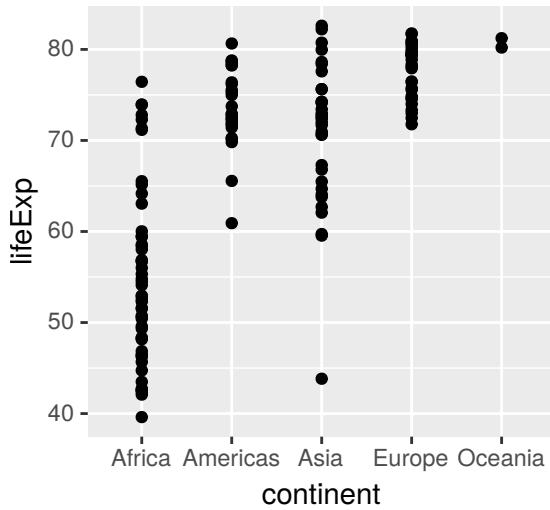


FIGURE 4.2: A strip plot using `geom_point()`.

(3) specifying further variables inside `aes()`

Going back to the scatter plot (`lifeExp` vs `gdpPercap`), let's use `continent` to give the points some colour. We can do this by adding `colour = continent` inside the `aes()`:

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
  geom_point()
```

This creates the third plot in Figure 4.1. It uses the default colour scheme and will automatically include a legend. Still with just two lines of code (`ggplot(...)` + `geom_point()`).

(4) specifying aesthetics outside `aes()`

It is very important to understand the difference between including `ggplot` arguments inside or outside of the `aes()` function.

The main aesthetics (things we can see) are: **x**, **y**, **colour**, **fill**, **shape**, **size**, and any of these could appear inside or outside the `aes()` function. Press F1 on, e.g., `geom_point()`, to see the full list of aesthetics that can be used with this geom (this opens the Help tab).

Variables (so columns of your dataset) have to be defined inside `aes()`. Whereas to apply a modification on everything, we can set an aesthetic to a constant value outside of `aes()`.

For example, Figure 4.3 shows a selection of the point shapes built into R. The default shape used by `geom_point()` is number 16.

0 □ 1 ○ 2 △ 4 × 8 ★ 15 ■ 16 ● 17 ▲ 21 ○ 22 □ 23 ◆

FIGURE 4.3: A selection of shapes for plotting. Shapes 0, 1, and 2 are hollow, whereas for shapes 21, 22, and 23 we can define both a colour and a fill (for these shapes, colour is the border around the fill).

To make all of the points in our figure hollow, let's set their shape to 1. We do this by adding `shape = 1` inside the `geom_point()`:

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
  geom_point(shape = 1)
```

This creates the fourth plot in Figure 4.1.

(5) From one plot to multiple with a single extra line

Faceting is a way to efficiently create the same plot for subgroups

within the dataset. For example, we can separate each continent into its own facet by adding `facet_wrap(~continent)` to our plot:

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
  geom_point(shape = 1) +
  facet_wrap(~continent)
```

This creates the fifth plot in Figure 4.1. Note that we have to use the tilde (~) in `facet_wrap()`. There is a similar function called `facet_grid()` that will create a grid of plots based on two grouping variables, e.g., `facet_grid(var1~var2)`. Furthermore, facets are happy to quickly separate data based on a condition (so something you would usually use in a filter).

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
  geom_point(shape = 1) +
  facet_wrap(~pop > 50000000)
```

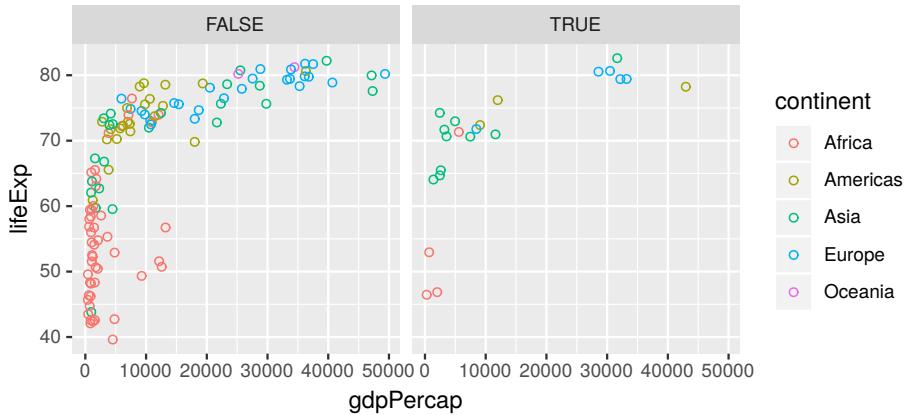


FIGURE 4.4: Using a filtering condition (e.g., population > 50 million) directly inside a `facet_wrap()`.

On this plot, the facet `FALSE` includes countries with a population less than 50 million people, and the facet `TRUE` includes countries with a population greater than 50 million people.

The tilde (~) in R denotes dependency. It is mostly used by sta-

tistical models to define dependent and explanatory variables, you will see it a lot in the second part of this book.

(6) Grey to white background - changing the theme

Overall, we can customise every single thing on a ggplot. Font type, colour, size or thickness or any lines or numbers, background, you name it. But a very quick way to change the appearance of a ggplot is to apply a different theme. The signature ggplot theme has a light grey background and white grid lines (Figure 4.5).

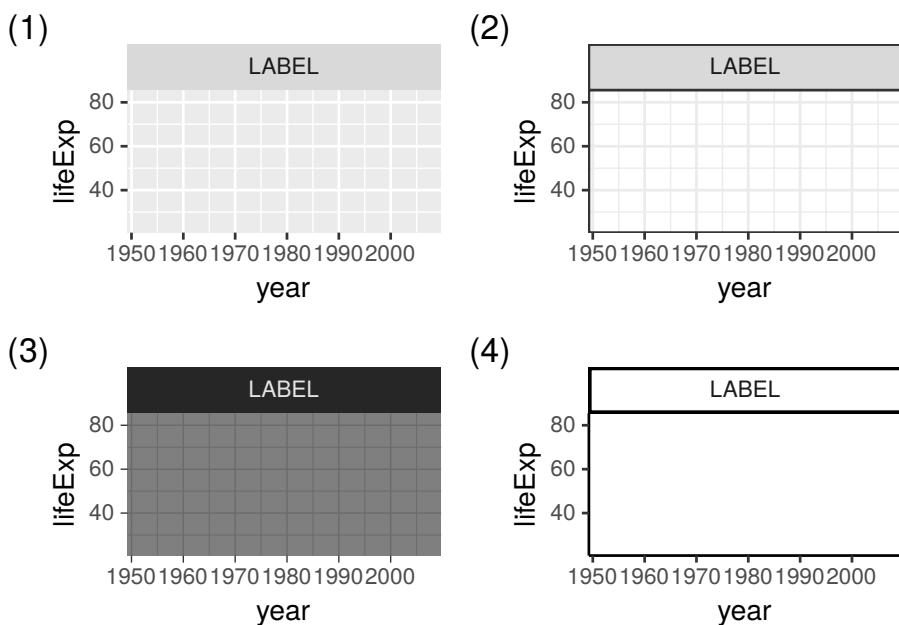


FIGURE 4.5: Some of the built-in ggplot themes (1) default (2) `theme_bw()`, (3) `theme_dark()`, (4) `theme_classic()`.

As a final step, we are adding `theme_bw()` (“background white”) to give the plot a different look. We have also divided the `gdpPercap` by 1000 (making the units “thousands of dollars per capita”). Note that you can apply calculations directly on ggplot variables (so how we’ve done `x = gdpPercap/1000` here).

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap/1000, y = lifeExp, colour = continent)) +
```

```
geom_point(shape = 1) +
facet_wrap(~continent) +
theme_bw()
```

This creates the last plot in Figure 4.1.

This is how `ggplot()` works - you can build a plot by adding or modifying things one by one.

4.3 Set your theme - grey vs white

If you find yourself always adding the same theme to your plot (i.e. we really like the `+ theme_bw()`), you can use `theme_set()` so your chosen theme is applied to every plot you draw:

```
theme_set(theme_bw())
```

In fact, we usually have these two lines at the top of every script:

```
library(tidyverse)
theme_set(theme_bw())
```

Furthermore, we can customise anything that appears in a `ggplot()` from axis fonts to the exact grid lines, and much more. That's what Chapter 5: Fine tuning plots is all about, but in here we are focussing on the basic functionality and how different geoms work. But from now on, `+ theme_bw()` is automatically applied on everything we make.

4.4 Scatter plots/bubble plots

The ggplot anatomy (Section 4.2) covered both scatter and strip plots (both created with `geom_point()`). Another cool thing about this geom is that adding a size aesthetic makes it into a bubble plot. For example, let's size the points by population.

As you would expect from a “grammar of graphics plot”, this is as simple as adding `size = pop` as an aesthetic:

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap/1000, y = lifeExp, size = pop)) +
  geom_point()
```

With increased bubble sizes, there is some overplotting, so let's make the points hollow (`shape = 1`) and slightly transparent (`alpha = 0.5`):

```
gapminder2007 %>%
  ggplot(aes(x = gdpPercap/1000, y = lifeExp, size = pop)) +
  geom_point(shape = 1, alpha = 0.5)
```

The resulting bubble plots are shown in Figure 4.6.

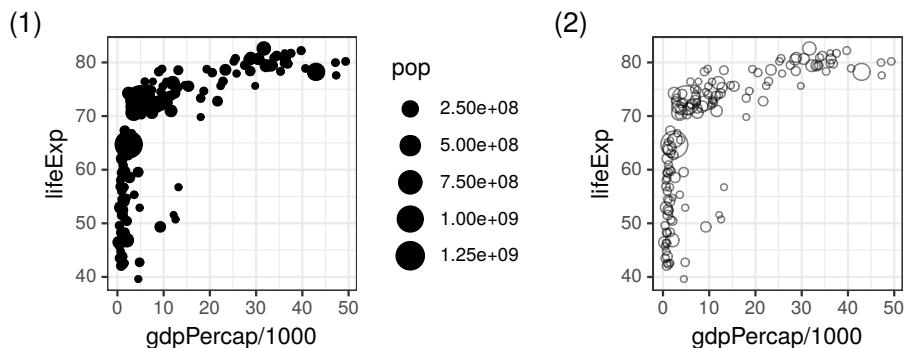


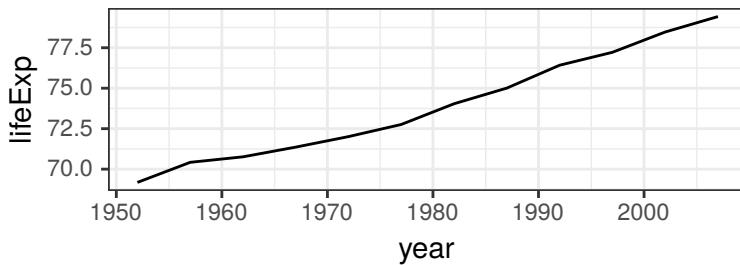
FIGURE 4.6: Turn the scatter plot from Figure 4.1:(2) to a bubble plot by (1) adding `size = pop` inside the `aes()`, (2) make the points hollow and transparent.

Alpha is an aesthetic to make geoms transparent, its values can range from 0 (invisible) to 1 (solid).

4.5 Line plots/time series plots

Let's plot the life expectancy in the United Kingdom over time:

```
gapminder %>%
  filter(country == "United Kingdom") %>%
  ggplot(aes(x = year, y = lifeExp)) +
  geom_line()
```



As a recap, the steps in the code above are:

- Using the `gapminder` data send into a `filter()`
- inside the `filter()`, our condition is `country == "United Kingdom"`
- We initialise `ggplot()` and define our main variables: `aes(x = year, y = lifeExp)`
- we are using a new geom - `geom_point()`.

This is identical to how we used `geom_point()`. In fact, by just changing `line` to `point` in the code above works - and instead of a continuous line you'll get a point at every 5 years as in the dataset.

But what if we want to draw multiple lines, e.g., for each country in the dataset. Let's send the whole dataset to `ggplot()` and `geom_line()`:

```
gapminder %>%
  ggplot(aes(x = year, y = lifeExp)) +
  geom_line()
```

The reason you see this weird zig-zag in Figure 4.7:1 is that, using the above code, `ggplot()` does not know which points to connect with which. Yes, you know you want a line for each country, but you haven't told it that. So for drawing multiple lines, we need to add a `group` aesthetic, in this case `group = country`:

```
gapminder %>%
  ggplot(aes(x = year, y = lifeExp, group = country)) +
  geom_line()
```

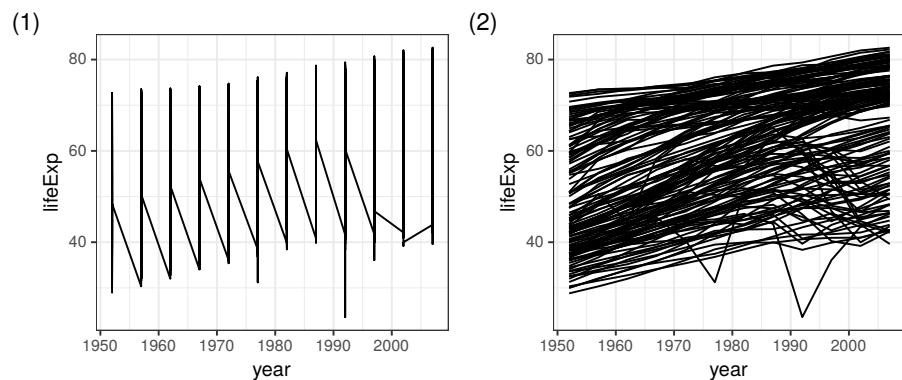
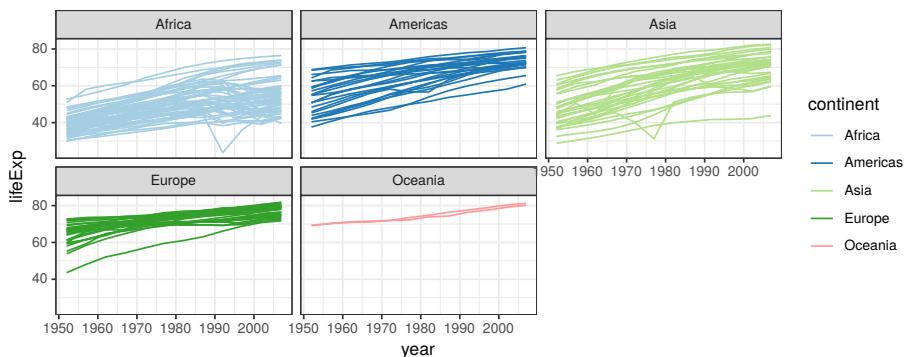


FIGURE 4.7: The ‘zig-zag plot’ is a common mistake: using `geom_line()` (1) without a `group =` aesthetic, (2) after adding `group = country`.

This code works as expected (Figure 4.7:2) - yes there is a lot of overplotting but that's just because we've included 142 lines on a single plot.

4.5.1 Exercise

Follow the step-by-step instructions to transform (Figure 4.7:2) into this:



- Colour lines by continents: add `colour = continent` inside `aes()`
- Continents on separate facets: + `facet_wrap(~continent)`
- Use a nicer colour scheme: + `scale_colour_brewer(palette = "Paired")`

4.6 Bar plots

There are two geoms for making bar plots - `geom_col()` and `geom_bar()`. In short: `geom_col()` plots values from your data directly (you define the `x` and `y` values), whereas `geom_bar()` will only take `x` values, the height of the bar (`y`) is the subgroups within `x` counted up. `geom_bar()` is basically a histogram for a categorical variable.

`geom_col()`:

- requires two variables `aes(x = , y =)`
- `x` is categorical, `y` is continuous (numeric)

Let's plot the life expectancies in 2007 in these three countries:

```
gapminder2007 %>%
  filter(country %in% c("United Kingdom", "France", "Germany")) %>%
  ggplot(aes(x = country, y = lifeExp)) +
  geom_col()
```

This gives us Figure 4.8:1. We have also created another cheeky one using the same code but changing the scale of the y axis to be more dramatic (Figure 4.8:2).

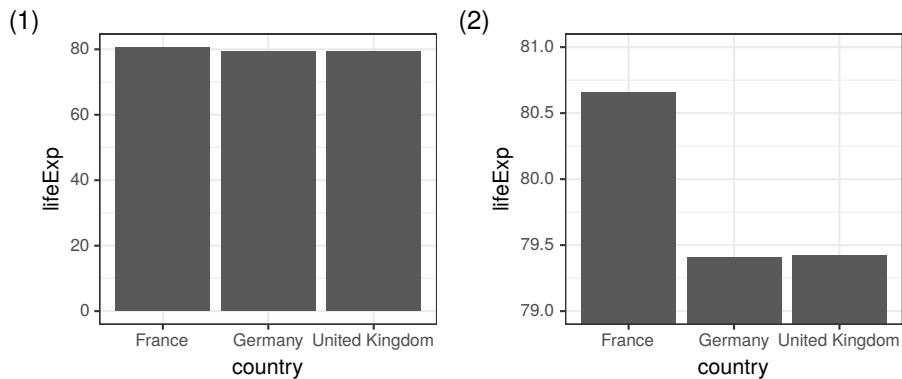


FIGURE 4.8: Bar plots using `geom_bar()`: (1) using the code example, (2) same plot but with `+ coord_cartesian(ylim=c(79, 81))` to manipulate the scale into something a lot more dramatic.

`geom_bar()`:

- requires a single variable `aes(x =)`
- this `x` should be a categorical variable
- `geom_bar()` then counts up the number of observations (rows) for this variable and plots them as bars.

Our `gapminder2007` tibble has a row for each country (see end of Section 4.1 to remind yourself). Therefore, if we use the `count()` function on the `continent` variable, we are counting up the number of countries on each continent (in this dataset²):

```
gapminder2007 %>%
  count(continent)
```

```
## # A tibble: 5 x 2
##   continent     n
##   <fct>     <int>
## 1 Africa      52
## 2 Americas    25
## 3 Asia        33
## 4 Europe      30
## 5 Oceania     2
```

So `geom_bar()` basically runs the `count()` function and plots it (see

²The number of countries in this dataset is 142, whereas the United Nations have 193 member states

how the bars on Figure 4.9 are the same height as the values from `count(continent)`.

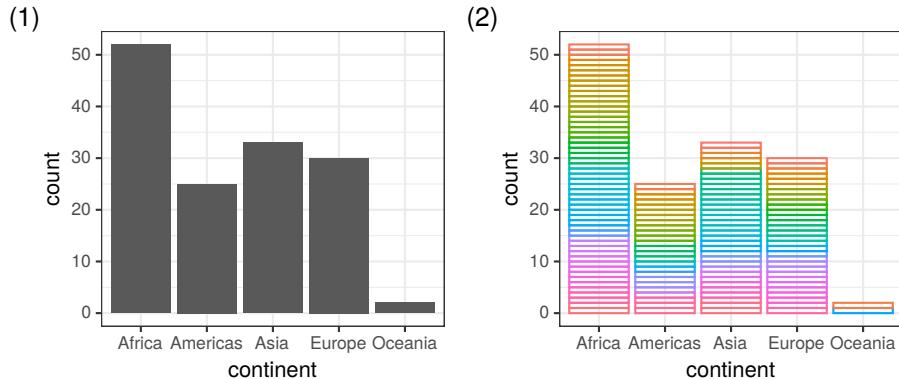


FIGURE 4.9: `geom_bar()` counts up the number of observations for each group. (1) `gapminder2007 %>% ggplot(aes(x = continent)) + geom_bar()`, (2) same + a little bit of magic to reveal the underlying data.

The first barplot in Figure 4.9 is produced with just this:

```
gapminder2007 %>%
  ggplot(aes(x = continent)) +
  geom_bar()
```

Whereas on the second one, we've asked `geom_bar()` to reveal the components (countries) in a colourful way:

```
gapminder2007 %>%
  ggplot(aes(x = continent, colour = country)) +
  geom_bar(fill = NA) +
  theme(legend.position = "none")
```

We have added `theme(legend.position = "none")` to remove the legend - it includes all 142 countries and is not very informative in this case. We're only including the colours for a bit of fun.

We're also removing the fill by setting it to NA (`fill = NA`). Note how we defined `colour = country` inside the `aes()` (as it's a variable), but we put the fill inside `geom_bar()` as a constant. This was explained

in more detail in steps (3) and (4) in the ggplot anatomy Section (4.2).

4.6.1 colour vs fill

Figure 4.9 also reveals the difference between a colour and a fill. Colour is the border around a geom, whereas fill is inside it. Both can either be set based on a variable in your dataset (this means `colour =` or `fill =` needs to be inside the `aes()` function), or they could be set to a fixed colour.

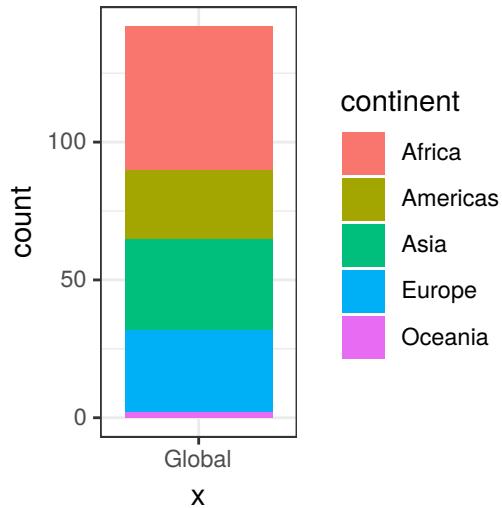
R has an amazing knowledge about different colours. In addition to knowing what is “white”, “yellow”, “red”, “green” etc. (meaning we can simply do `geom_bar(fill = "green")`) it also knows what “aquamarine”, “blanchedalmond”, “coral”, “deeppink”, “lavender”, “deepskyblue” look like (amongst many many others, search the internet for “R colours” for a full list).

We can also use HEX colour codes, for example, `geom_bar(fill = "#FF0099")` is a very pretty pink.

4.6.2 Proportions

Whether using `geom_bar()` or `geom_col()`, we can use fill to display proportions within bars. Furthermore, sometimes it’s useful to set the `x` value to a constant - to get everything plotted together rather than separated by a variable. So we are using `aes(x = "Global", fill = continent)`, note that “Global” could be any word - since it’s quoted `ggplot()` won’t go looking for it in the dataset:

```
gapminder2007 %>%
  ggplot(aes(x = "Global", fill = continent)) +
  geom_bar()
```



There are more examples of bar plots in Chapter 8.

4.6.3 Exercise

Create Figure 4.10 of life expectancies in European countries (year 2007).

Hints:

- If `geom_bar()` doesn't work try `geom_col()` or vice versa.
- `coord_flip()` to make the bars horizontal (it flips the `x` and `y` axes).
- `x = country` gets the country bars plotted in the alphabetical order, use `x = fct_reorder(country, lifeExp)` still inside the `aes()` to order the bars by their `lifeExp` values. Or try one of the other variables. (`pop`, `gdpPercap`) as the second argument to `fct_reorder()`.
- when using `fill = NA` also need to include a colour, we're using `colour = "deepskyblue"` inside the `geom_col()`.

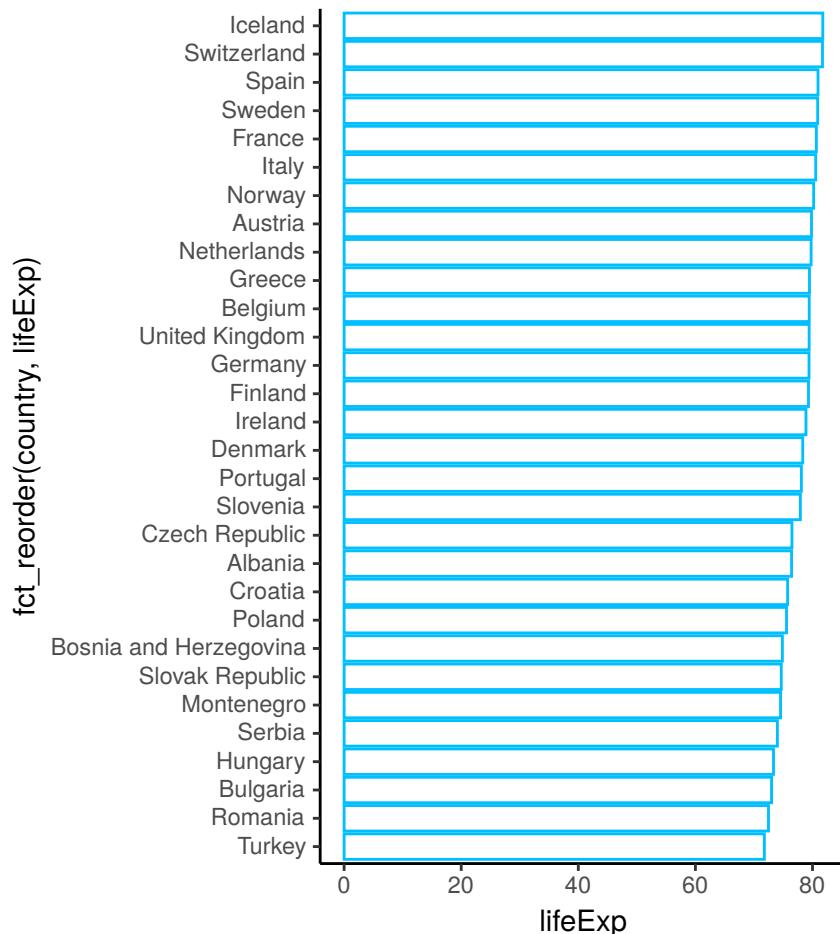
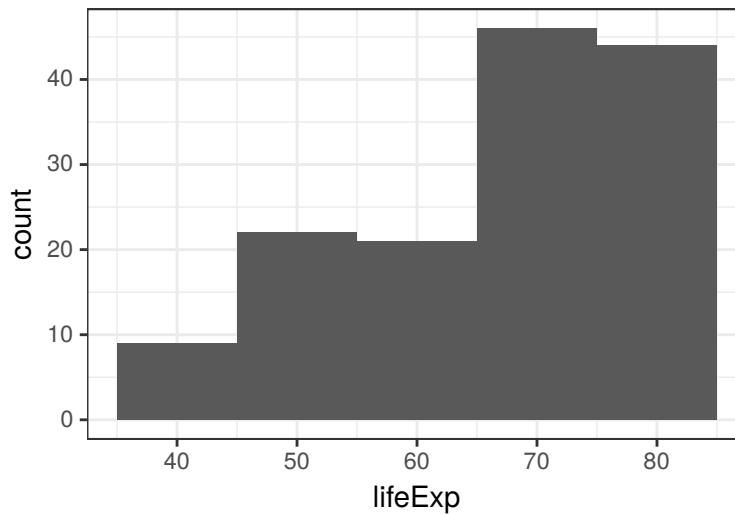


FIGURE 4.10: Barplot Exercise. Life expectancies in European countries in year 2007 from the Gapminder dataset.

4.7 Histograms

A histogram displays the distribution of values within a continuous variable. In the example below, we are taking the life expectancy (`aes(x = lifeExp)`) and telling the histogram to count the observations up in “bins” of 10 years (`geom_histogram(binwidth = 10)`):

```
gapminder2007 %>%
  ggplot(aes(x = lifeExp)) +
  geom_histogram(binwidth = 10)
```



We can see that most countries in the world have a life expectancy of \sim 70-80 years (in 2007), and that the distribution of life expectancies globally is not normally distributed. Setting the binwidth is optional, using just `geom_histogram()` works well too -by default, it will divide your data into 30 bins.

There are more examples of histograms in Chapter 6. There are two other geoms that are useful for plotting distributions: `geom_density()` and `geom_freqpoly()`.

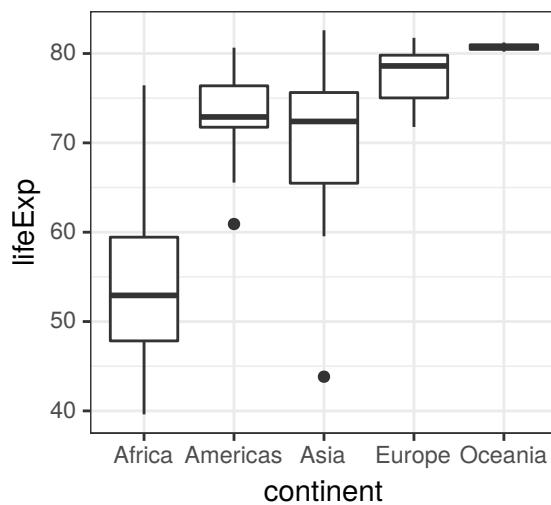
4.8 Box plots

Box plots are a way to quickly visualise summary statistics of a continuous outcome variable (such as life expectancy in the gapminder dataset).

Box plots include:

- the median (middle line in the box)
- inter-quartile range (IQR, top and bottom parts of the boxes - this is where 50% of your data is)
- whiskers (the black lines extending to the lowest and highest values that are still within 1.5*IQR)
- outliers (any observations outwith the whiskers)

```
gapminder2007 %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  geom_boxplot()
```



4.9 Multiple geoms, multiple `aes()`?

One of the coolest things about `ggplot()` is that we can plot multiple geoms on top of each other!

Let's add individual data points on top of the box plots:

```
gapminder2007 %>%
  ggplot(aes(x = continent, y = lifeExp)) +
```

```
geom_boxplot() +
geom_point()
```

This makes Figure 4.11:1.

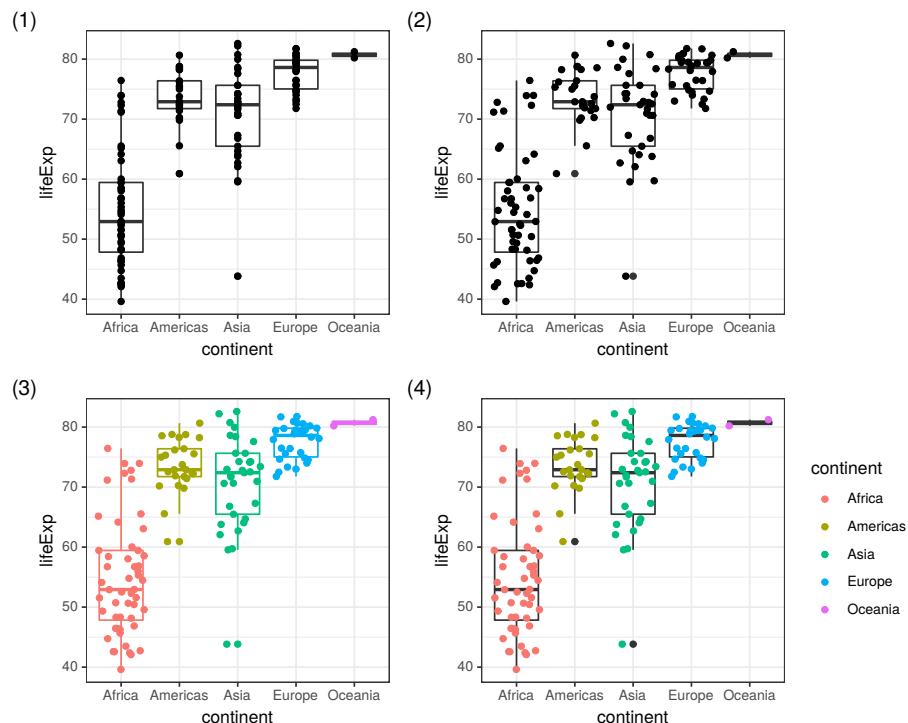


FIGURE 4.11: Multiple geoms together. (1) `geom_boxplot() + geom_point()`, (2) `geom_boxplot() + geom_jitter()`, (3) colour aesthetic inside `ggplot(aes())`, (4) colour aesthetic inside `geom_jitter(aes())`.

The only thing we've changed in (2) is replacing `geom_point()` with `geom_jitter()` - this spreads the points out to reduce overplotting.

But what's really exciting is the difference between (3) and (4) in Figure 4.11. Spot it!

```
# (3)
gapminder2007 %>%
  ggplot(aes(x = continent, y = lifeExp, colour = continent)) +
  geom_boxplot() +
```

```
geom_jitter()

# (4)
gapminder2007 %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  geom_jitter(aes(colour = continent))
```

This is new: `aes()` inside a geom, not just at the top! In the code for (4) you can see `aes()` in two places - at the top and inside the `geom_jitter()`. And `colour = continent` was only included in the second `aes()`. This means that the jittered points get a colour, but the box plots will be drawn without (so just black). An this is exactly what we see on 4.11³.

4.9.1 Worked example - three geoms together

Let's combine three geoms by including text labels on top of the box plot + points from above.

We are creating a new tibble called `label_data` filtering for the maximum life expectancy countries at each continent (`group_by(continent)`):

```
label_data = gapminder2007 %>%
  group_by(continent) %>%
  filter(lifeExp == max(lifeExp)) %>%
  select(country, continent, lifeExp)

# since we filtered for lifeExp == max(lifeExp)
# these are the maximum life expectancy countries at each continent:
label_data

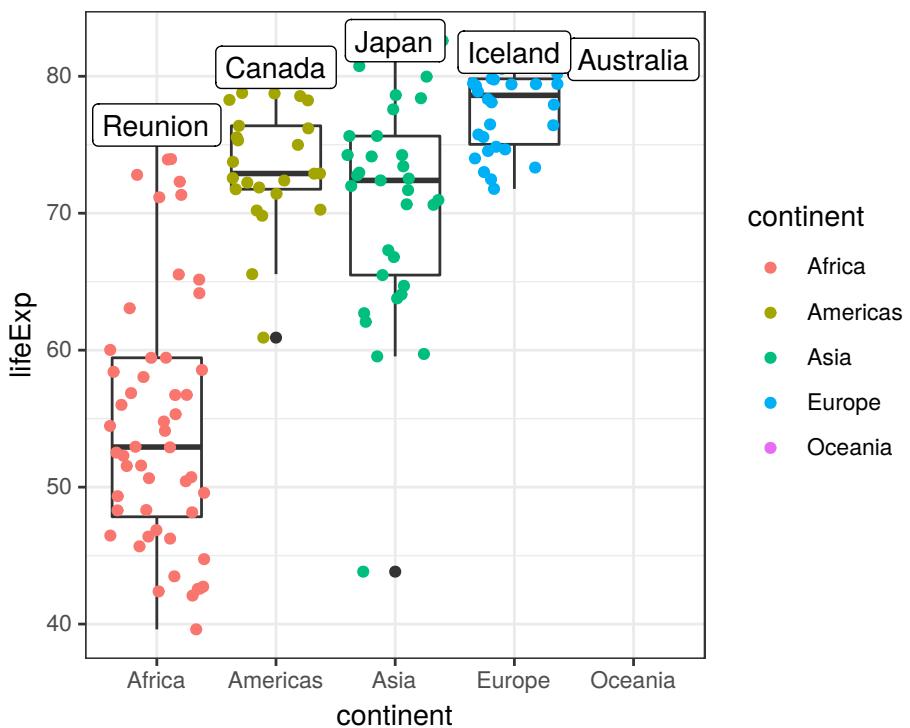
## # A tibble: 5 x 3
## # Groups:   continent [5]
##   country   continent lifeExp
##   <fct>     <fct>      <dbl>
## 1 Australia Oceania     81.2
```

³Nerd alert: the variation added by `geom_jitter()` is random, which means that when you recreate the same plots the points will appear in slightly different locations to ours. To make identical ones, add `position = position_jitter(seed = 1)` inside `geom_jitter()`.

```
## 2 Canada      Americas      80.7
## 3 Iceland     Europe       81.8
## 4 Japan       Asia        82.6
## 5 Reunion     Africa      76.4
```

The first two geoms are from the previous example (`geom_boxplot()` and `geom_jitter()`). Note that `ggplot()` plots them in the order they are in the code - so box plots at the bottom, jittered points on the top. We are then adding `geom_label()` with its own data option (`data = label_data`) as well as a new aesthetic (`aes(label = country)`):

```
gapminder2007 %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  # First geom - boxplot
  geom_boxplot() +
  # Second geom - jitter with its own aes(colour = )
  geom_jitter(aes(colour = continent)) +
  # Third geom - label, with its own dataset (label_data) and aes(label = )
  geom_label(data = label_data, aes(label = country))
```



A few suggested experiments to try with the 3-geom plot code above:

- remove `data = label_data`, from `geom_label()` and you'll get all 142 labels (so it will plot a label for the whole `gapminder2007` dataset).
- change from `geom_label()` to `geom_text()` - it works similarly but doesn't have the border and background behind the country name.
- change `label = country` to `label = lifeExp`, this plots the maximum value, rather than the country name.

4.10 All other types of plots

In this chapter we have introduced some of the most common geoms, as well as explained how `ggplot()` works. In fact, `ggplot` has 52 different geoms for you to use, see its documentation for a full list: <https://ggplot2.tidyverse.org>

And with the ability of combining multiple geoms together on the same plot, the possibilities really are endless. Furthermore, the `plotly` Graphic Library (<https://plot.ly/ggplot2/>) can make some of your ggplots interactive, meaning you can use your mouse to hover over the point or zoom and subset interactively.

The two most important things to understand about `ggplot()` are:

- Variables (columns in your dataset) need to be inside `aes()`
- `aes()` can be both at the top - `data %>% ggplot(aes())` - as well as inside a geom (e.g., `geom_point(aes())`). This distinction is useful when combining multiple geoms. All your geoms will “know about” the top-level `aes()` variables, but including `aes()` variables inside a specific geom means it only applies to that one.

4.11 Solutions

4.4.1

```
library(tidyverse)
library(gapminder)

gapminder %>%
  ggplot(aes(x      = year,
             y      = lifeExp,
             group = country,
             colour = continent)) +
  geom_line() +
  facet_wrap(~continent) +
  theme_bw() +
  scale_colour_brewer(palette = "Paired")
```

4.5.3

```
library(tidyverse)
library(gapminder)

gapminder %>%
  filter(year == 2007) %>%
  filter(continent == "Europe") %>%
  ggplot(aes(x = fct_reorder(country, lifeExp), y = lifeExp)) +
  geom_col(colour = "deepskyblue", fill = NA) +
  coord_flip() +
  theme_classic()
```

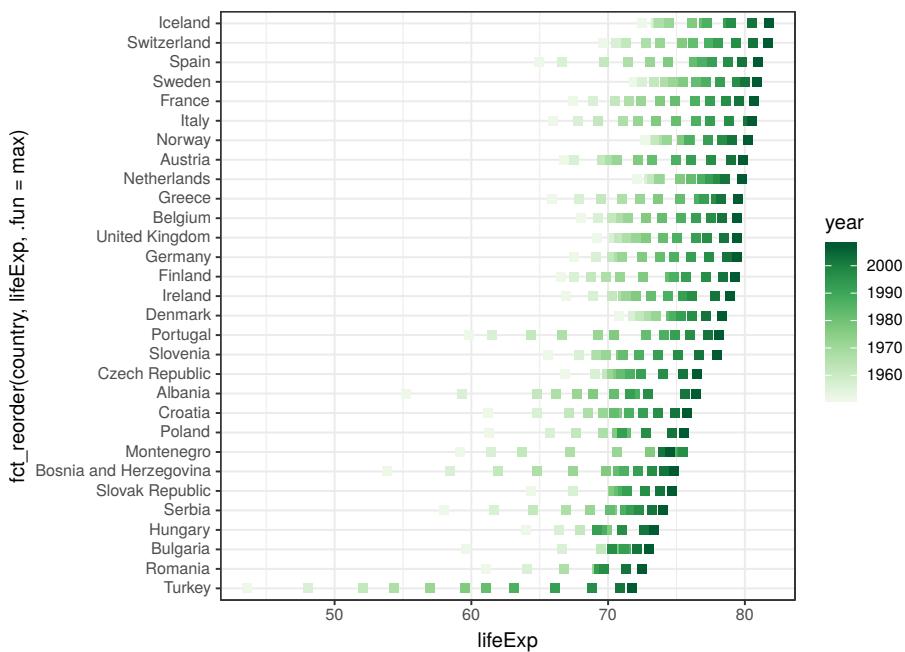
4.12 Extra: Advanced examples

There are two examples of how just a few lines of `ggplot()` code and the basic geoms introduced in this chapter can be used to make very different things. Let your imagination fly free when using `ggplot()`!

The first one shows how the life expectancies in European countries

have increased by plotting a square (`geom_point(shape = 15)`) for each observation (year) in the dataset.

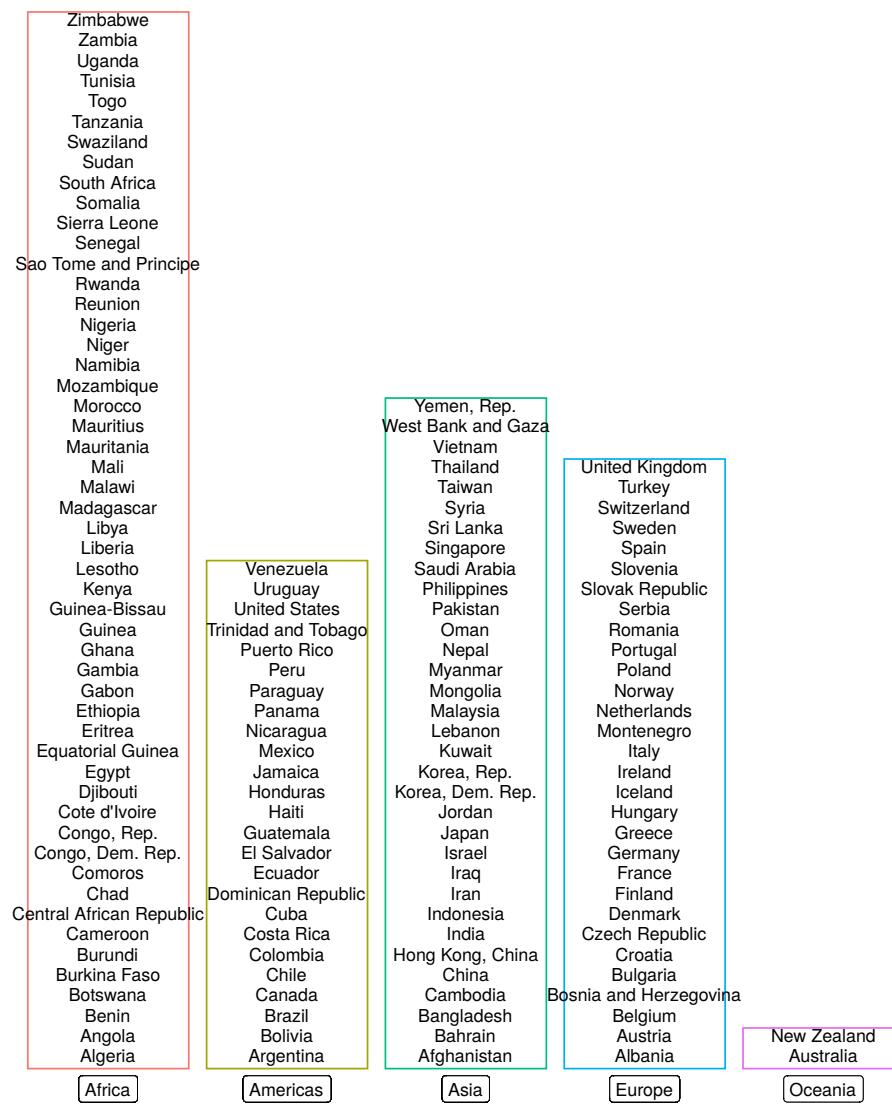
```
gapminder %>%
  filter(continent == "Europe") %>%
  ggplot(aes(y = fct_reorder(country, lifeExp, .fun=max),
             x = lifeExp,
             colour = year)) +
  geom_point(shape = 15, size = 2) +
  scale_colour_distiller(palette = "Greens", direction = 1) +
  theme_bw()
```



In the second example, we're using `group_by(continent)` followed by `mutate(country_number = seq_along(country))` to create a new column with numbers 1, 2, 3, etc for countries within continents. We are then using these as y coordinates for the text labels (`geom_text(aes(y = country_number...))`).

```
gapminder2007 %>%
  group_by(continent) %>%
  mutate(country_number = seq_along(country)) %>%
  ggplot(aes(x = continent)) +
```

```
geom_bar(aes(colour = continent), fill = NA, show.legend = FALSE) +
  geom_text(aes(y = country_number, label = country), vjust = 1) +
  geom_label(aes(label = continent), y = -1) +
  theme_void()
```



5

Fine tuning plots

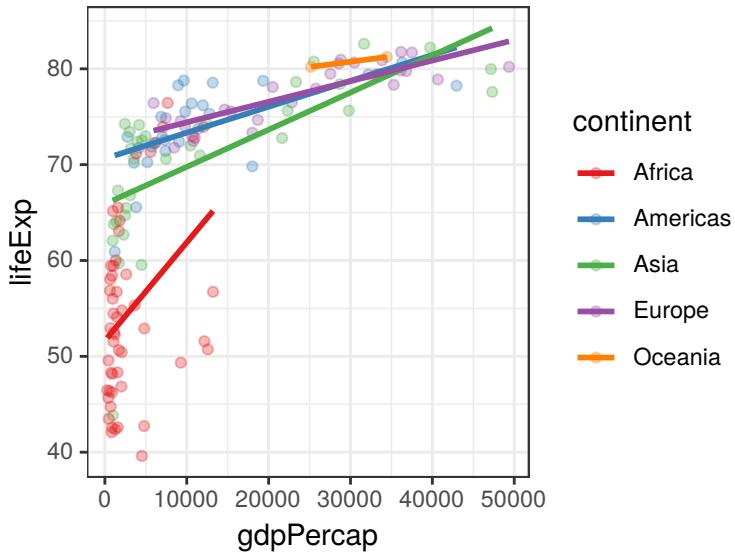
5.1 Data and initial plot

We can save a `ggplot()` object into a variable (usually called `p` but can be any name). This then appears in the Environment tab. To plot it it needs to be recalled on a separate line. Saving a plot into a variable allows us to modify it later (e.g., `p + theme_bw()`).

```
library(gapminder)
library(tidyverse)

p0 = gapminder %>%
  filter(year == 2007) %>%
  group_by(continent, year) %>%
  ggplot(aes(y = lifeExp, x = gdpPercap, colour = continent)) +
  geom_point(alpha = 0.3) +
  theme_bw() +
  geom_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette = "Set1")

p0
```



5.2 Scales

5.2.1 Logarithmic

```
p1 = p0 + scale_x_log10()
```

5.2.2 Expand limits

Specify the value you want to be included:

```
p2 = p0 + expand_limits(y = 0)
```

Or two:

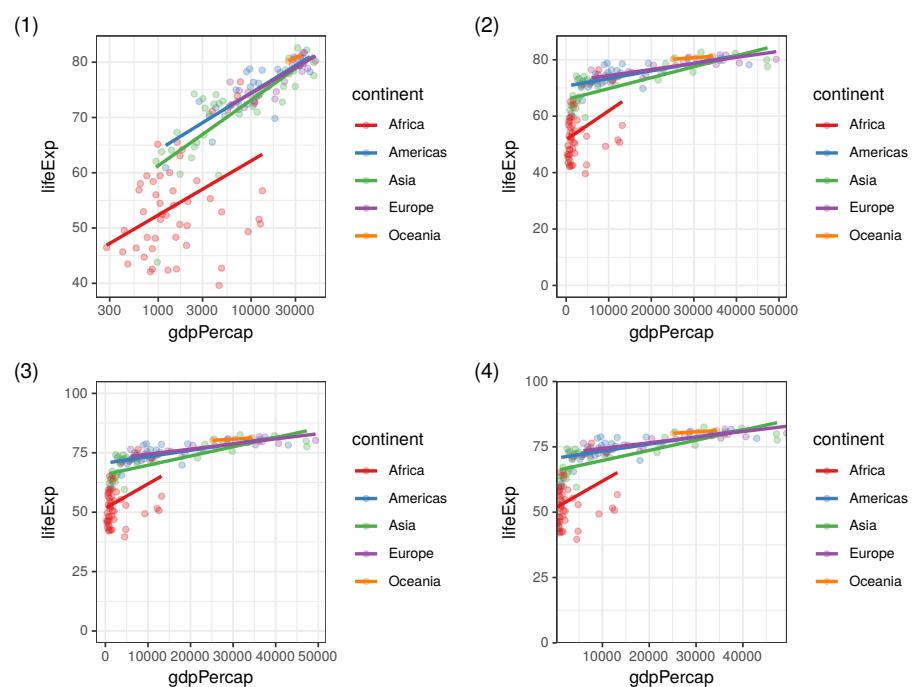
```
p3 = p0 + expand_limits(y = c(0, 100))
```

By default, `ggplot()` adds some padding around the included area

(see how the scale doesn't start from 0, but slightly before). You can remove this padding with the expand option:

```
p4 = p0 +
  expand_limits(y = c(0, 100)) +
  coord_cartesian(expand = FALSE)
```

```
library(patchwork)
p1 + p2 + p3 + p4 + plot_annotation(tag_levels = "1", tag_prefix = "(", tag_suffix = ")")
```



5.2.3 Zoom in

```
p1 = p0 +
  coord_cartesian(ylim = c(70, 85), xlim = c(20000, 40000))
```

5.2.4 Exercise

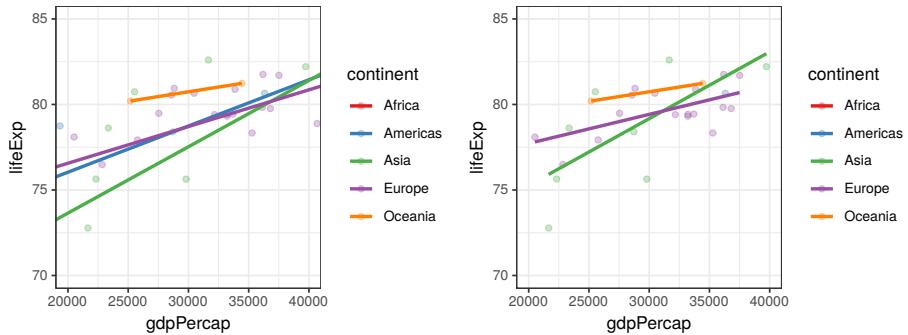
How is this one different to the previous?

```
p2 = p0 +
  scale_y_continuous(limits = c(70, 85)) +
  scale_x_continuous(limits = c(20000, 40000))
```

Answer: the first one zooms in, still retaining information about the excluded points when calculating the linear regression lines. The second one removes the data (as the warnings say), calculating the linear regression lines only for the visible points.

```
p1 + p2
```

```
## Warning: Removed 114 rows containing non-finite values (stat_smooth).
## Warning: Removed 114 rows containing missing values (geom_point).
```



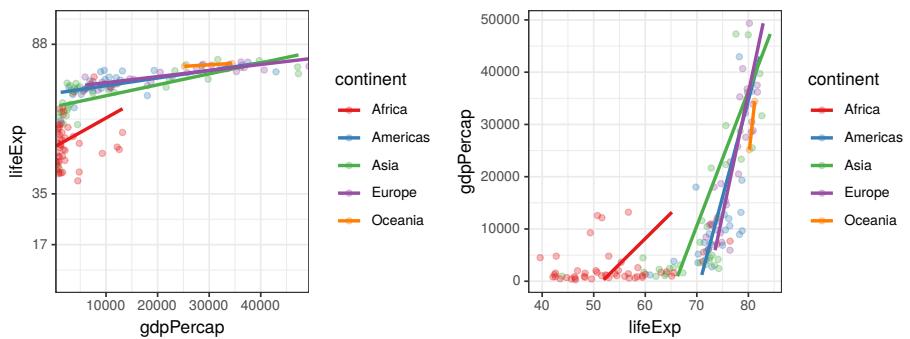
5.2.5 Axis ticks

```
p1 = p0 +
  coord_cartesian(ylim = c(0, 100), expand = 0) +
  scale_y_continuous(breaks = c(17, 35, 88))
```

5.2.6 Swap the axes

```
p2 = p0 +
  coord_flip()
```

p1 + p2



5.3 Colours

5.3.1 Using the Brewer palettes:

```
p1 = p0 +
  scale_color_brewer(palette = "Paired")
```

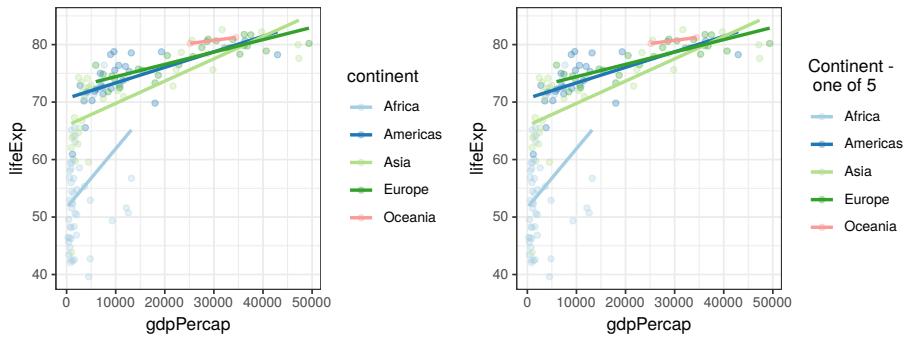
5.3.2 Legend title

`scale_colour_brewer()` is also a convenient place to change the legend title:

```
p2 = p0 +
  scale_color_brewer("Continent - \n one of 5", palette = "Paired")
```

Note the `\n` inside the new legend title - new line.

```
p1 + p2
```



5.3.3 Choosing colours manually

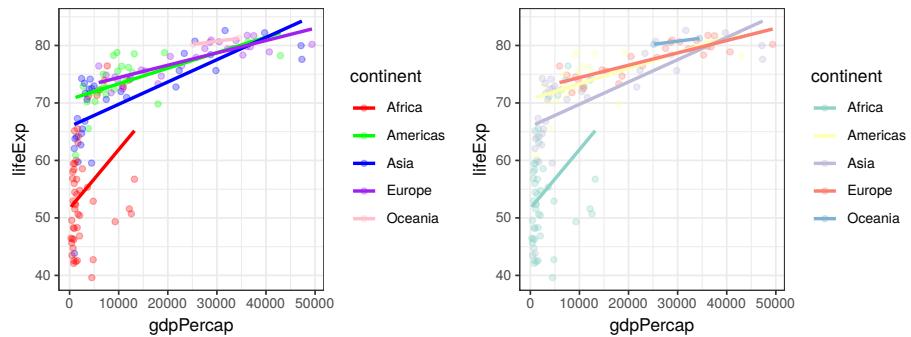
Use words:

```
p1 = p0 +
  scale_color_manual(values = c("red", "green", "blue", "purple", "pink"))
```

Or HEX codes (either from <http://colorbrewer2.org/> or any other resource):

```
p2 = p0 +
  scale_color_manual(values = c("#8dd3c7", "#ffffb3", "#bebada",
                                "#fb8072", "#80b1d3"))
```

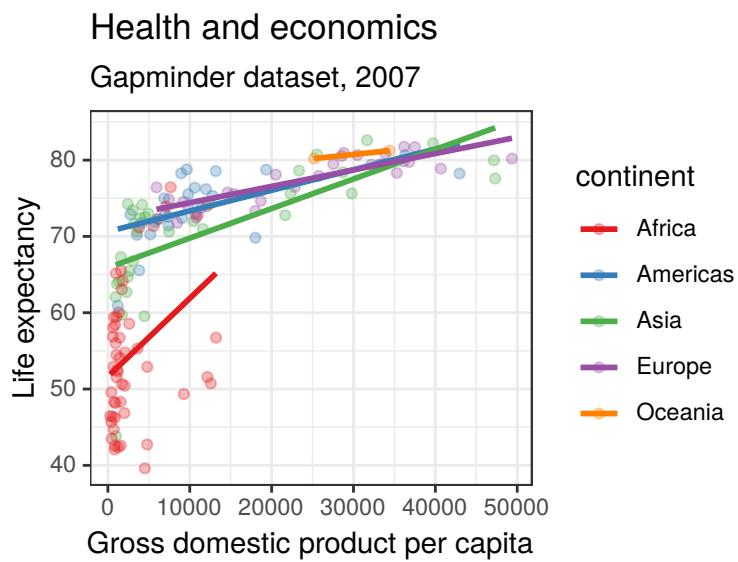
```
p1 + p2
```



Note that <http://colorbrewer2.org/> also has options for *Colourblind safe* and *Print friendly*.

5.4 Titles and labels

```
p0 +  
  labs(x = "Gross domestic product per capita",  
       y = "Life expectancy",  
       title = "Health and economics",  
       subtitle = "Gapminder dataset, 2007")
```



5.4.1 Annotation

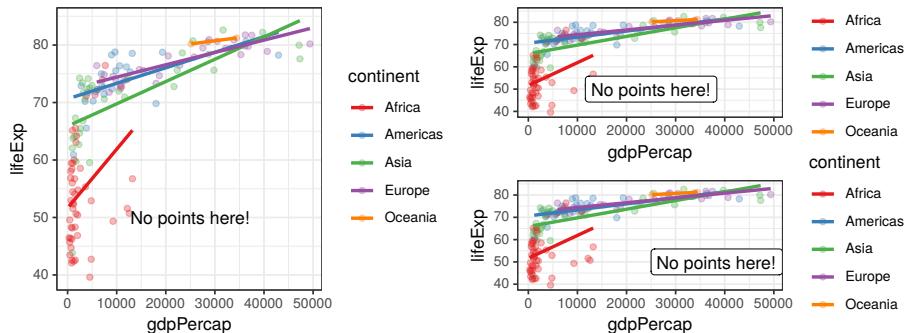
```
p1 = p0 +  
  annotate("text",  
          x = 25000,  
          y = 50,  
          label = "No points here!")
```

```
p2 = p0 +  
  annotate("label",  
          x = 25000,
```

```
y = 50,
label = "No points here!")
```

```
p3 = p0 +
  annotate("label",
    x = 25000,
    y = 50,
    label = "No points here!",
    hjust = 0)
```

`p1 + p2 / p3`



`hjust` stands for horizontal justification. It's default value is 0.5 (see how the label was centered at 25,000 - our chosen x location), 0 means the label goes to the right from 25,000, 1 would make it end at 25,000.

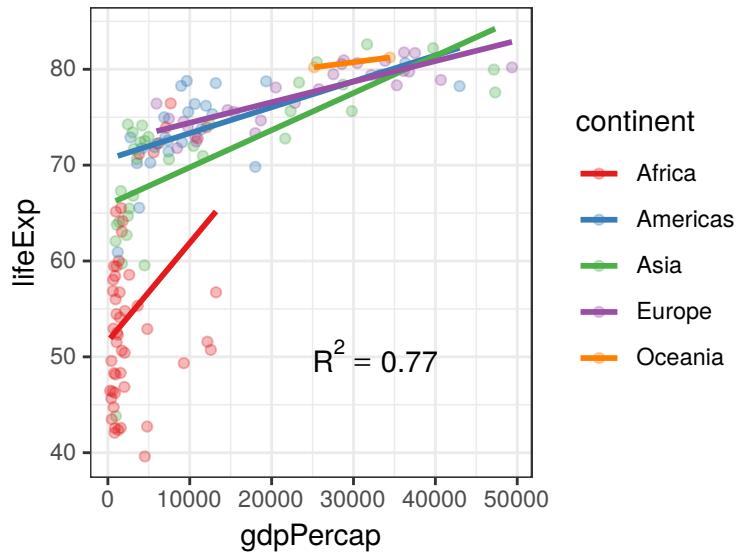
5.4.2 Annotation with a superscript and a variable

```
fit_glance = tibble(r.squared = 0.7693465)

plot_rsquared = paste0(
  "R^2 == ",
  fit_glance$r.squared %>% round(2))

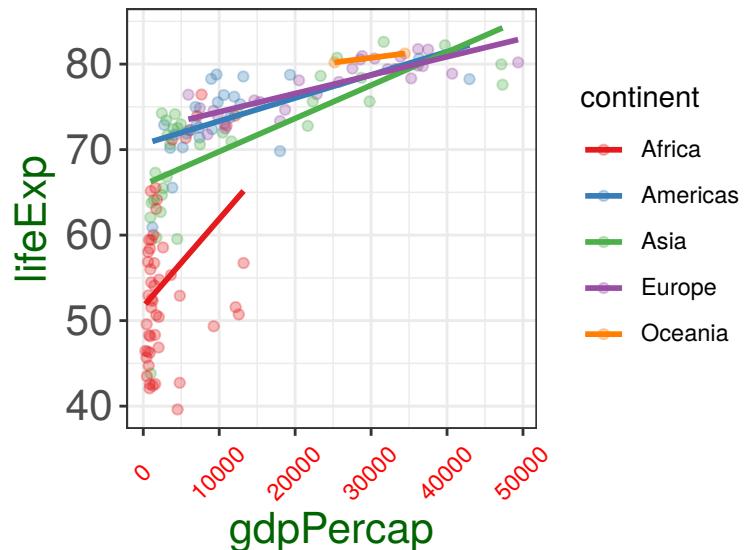
p0 +
```

```
annotate("text",
  x = 25000,
  y = 50,
  label = plot_rsquared, parse = TRUE,
  hjust = 0)
```



5.5 Text size

```
p0 +
  theme(axis.text.y = element_text(size = 16),
        axis.text.x = element_text(colour = "red", angle = 45, vjust = 0.5),
        axis.title = element_text(size = 16, colour = "darkgreen")
      )
```



5.5.1 Legend position

Use the following words: "right", "left", "top", "bottom", OR "none" to remove the legend.

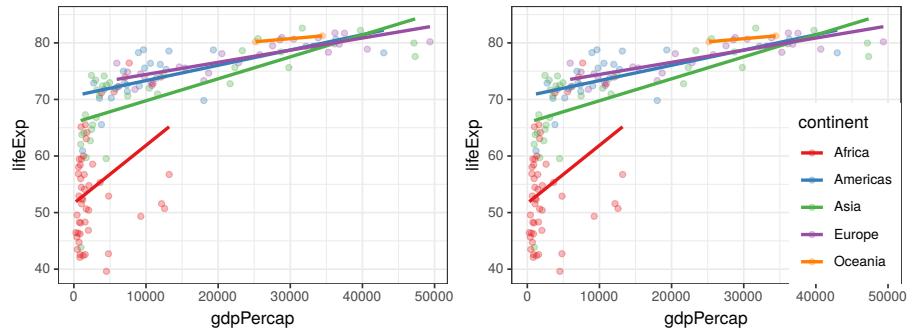
```
p1 = p0 +
  theme(legend.position = "none")
```

Or use relative coordinates (0–1) to give it an -y location:

```
p2 = p0 +
  theme(legend.position      = c(1,0),
        legend.justification = c(1,0)) #bottom-right corner
```

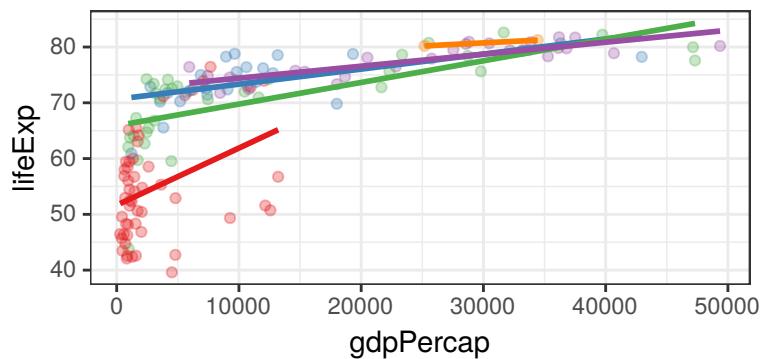
```
p3 = p0 +
  theme(legend.position = "top") +
  guides(colour = guide_legend(ncol = 2))
```

```
p1 + p2
```



p3

continent Africa Americas Asia	Europe Oceania
---	-------------------

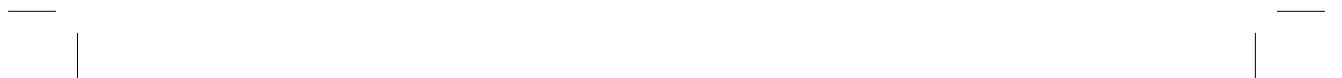


5.6 Saving your plot

```
ggsave(p0, file = "my_saved_plot.png", width = 5, height = 4)
```

Part II

Data analysis



In the second part of this book, we focus specifically on the business of data analysis. That is, formulating clear questions and seeking to answer them using available datasets.

Again, we emphasise the importance of understanding the underlying data through visualisation, rather than relying on statistical tests or, heaven forbid, the p-value alone.

There are five chapters. Testing for continuous outcome variables (6) leads naturally into Linear regression (7). We would expect the majority of actual analysis done by readers to be using the methods in chapter 7 rather than 6. Similarly, Testing for categorical outcome variables (8) leads naturally to Logistic regression (9), where we would expect the majority of work to focus. Chapters 6 and 8 however do provide helpful reminders of how to prepare data for these analyses and shouldn't be skipped. Time-to-event data introduces survival analysis and includes sections on the manipulation of dates.



6

Working with continuous outcome variables

Continuous data can be measured.
Categorical data can be counted.

6.1 Continuous data

Continuous data is everywhere in healthcare. From physiological measures in patients such as systolic blood pressure or pulmonary function tests, through to populations measures like life expectancy or disease incidence, the analysis of continuous outcome measures is common and important.

Our goal in most health data questions, is to draw a conclusion on a comparison between groups. For instance, understanding differences in life expectancy between the year 2002 and 2007 is more useful than simply describing the average life expectancy across all of time.

The basis for comparisons between continuous measures is the *distribution* of the data. That word, as many which have a statistical flavour, brings on the sweats in a lot of people. It needn't. By distribution, we are simply referring to the shape of the data.

6.2 The Question

The examples in this chapter all use the data introduced previously from the amazing Gapminder project¹. We will start by looking at the life expectancy of populations over time and in different geographical regions.

6.3 Get the data

```
# Load packages
library(tidyverse)
library(finalfit)
library(gapminder)

# Create object mydata from object gapminder
mydata = gapminder
```

6.4 Check the data

It is vital that data is carefully inspected when first read (for help reading data into R see 2.5). The three functions below provide a clear summary allowing errors or miscoding to be quickly identified. It is particularly important to ensure that any missing data is identified (see Chapter 13. If you don't do this you will regret it! There are many times when an analysis has got to a relatively advanced stage before research realised the dataset was incomplete.

¹<https://www.gapminder.org/>

```
glimpse(mydata) # each variable as line, variable type, first values
```

```
## Observations: 1,704
## Variables: 6
## $ country    <fct> Afghanistan, Afghanistan, Afghanistan, Afghanistan, ...
## $ continent   <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia...
## $ year        <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992...
## $ lifeExp     <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.8...
## $ pop         <int> 8425333, 9240934, 10267083, 11537966, 13079460, 1488...
## $ gdpPercap   <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 78...
```

```
missing_glimpse(mydata) # missing data for each variable
```

```
##           label var_type   n missing_n missing_percent
## country      country    <fct> 1704      0          0.0
## continent   continent   <fct> 1704      0          0.0
## year        year       <int> 1704      0          0.0
## lifeExp     lifeExp    <dbl> 1704      0          0.0
## pop         pop       <int> 1704      0          0.0
## gdpPercap   gdpPercap <dbl> 1704      0          0.0
```

```
ff_glimpse(mydata) # summary statistics for each variable
```

```
## Continuous
##           label var_type   n missing_n missing_percent      mean
## year        year       <int> 1704      0          0.0      1979.5
## lifeExp    lifeExp    <dbl> 1704      0          0.0       59.5
## pop         pop       <int> 1704      0          0.0  29601212.3
## gdpPercap   gdpPercap <dbl> 1704      0          0.0      7215.3
##                   sd      min quartile_25      median quartile_75
## year          17.3  1952.0    1965.8      1979.5      1993.2
## lifeExp       12.9   23.6     48.2       60.7       70.8
## pop          106157896.7 60011.0  2793664.0  7023595.5  19585221.8
## gdpPercap    9857.5   241.2    1202.1     3531.8      9325.5
##                   max
## year          2007.0
## lifeExp        82.6
## pop          1318683096.0
## gdpPercap     113523.1
##
## Categorical
##           label var_type   n missing_n missing_percent levels_n
## country      country    <fct> 1704      0          0.0      142
## continent   continent   <fct> 1704      0          0.0          5
##                   levels
## country
## continent "Africa", "Americas", "Asia", "Europe", "Oceania"
##                   levels_count      levels_percent
```

```
## country - -
## continent 624, 300, 396, 360, 24 36.6, 17.6, 23.2, 21.1, 1.4
```

TABLE 6.1: Gapminder dataset, ff_glimpse: continuous

label	var_type	n	missing_n	mean	sd	median
year	<int>	1704	0	1979.5	17.3	1979.5
lifeExp	<dbl>	1704	0	59.5	12.9	60.7
pop	<int>	1704	0	29601212.3	106157896.7	7023595.5
gdpPercap	<dbl>	1704	0	7215.3	9857.5	3531.8

TABLE 6.2: Gapminder dataset, ff_glimpse: categorical

label	var_type	n	missing_n	levels_n	levels	levels_count
country	<fct>	1704	0	142	-	-
continent	<fct>	1704	0	5	"Africa", "Americas", "Asia", "Europe", "Oceania"	624, 300, 396, 360, 24

As can be seen, there are 6 variables, 4 are continuous and 2 are categorical. The categorical variables are already identified as `factors`. There are no missing data.

6.5 Plot the data

We will start by comparing life expectancy between the 5 continents of the world in two different years. Always plot your data first. Never skip this step! We are particularly interested in the distribution. There's that word again. The shape of the data. Is it normal? Is it skewed? Does it differ between regions and years?

There are three useful plots which can help here:

- Histograms: examine shape of data and compare groups;
- Q-Q plots: are data normally distributed?
- Box-plots: identify outliers, compare shape and groups.

6.5.1 Histogram

```
mydata %>%
  filter(year %in% c(2002, 2007)) %>%
  ggplot(aes(x = lifeExp)) +
    # remember aes()
    geom_histogram(bins = 20) +
    # histogram with 20 bars
    facet_grid(year ~ continent)
    # add scale="free" for axes to vary
```

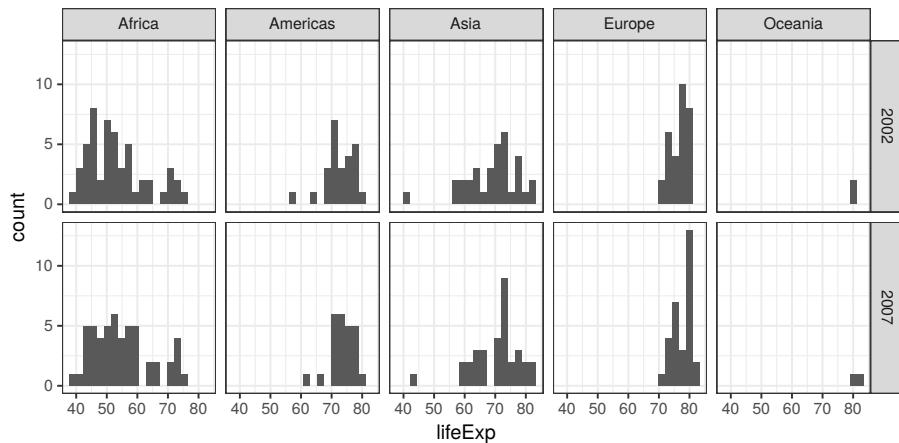


FIGURE 6.1: Histogram: country life expectancy by continent and year

What can we see? That life expectancy in Africa is lower than in other regions. That we have little data for Oceania given there are only two countries included, Australia and New Zealand. That Africa and Asia have great variability in life expectancy by country than in the Americas or Europe. That the data follow a reasonably normal shape, with Africa 2002 a little right skewed.

6.5.2 Q-Q plot

A quantile-quantile sounds complicated but is not. It is simply a graphical method for comparing the distribution (think shape) of our own data to a theoretical distribution, such as the normal distribution. In this context, quantiles are just cut points which divide our data into bins each containing the same number of ob-

servations. For example, if we have the life expectancy for 100 countries, then quartiles (note the quar-) for life expectancy are the three ages which split the observations into 4 groups each containing 25 countries. A Q-Q plot simply plots the quantiles for our data against the theoretical quantiles for a particular distributions (the default shown below is the normal distribution). If our data follow that distribution (e.g. normal), then we get a 45 degree line on the plot.

```
mydata %>%
  filter(year %in% c(2002, 2007)) %>%
  ggplot(aes(sample = lifeExp)) +
    geom_qq() +                         # Q-Q plot requires `sample`  

    geom_qq_line() +                      # defaults to normal distribution  

    geom_qq_line() +                      # add 45 degree line  

    facet_grid(year ~ continent)
```

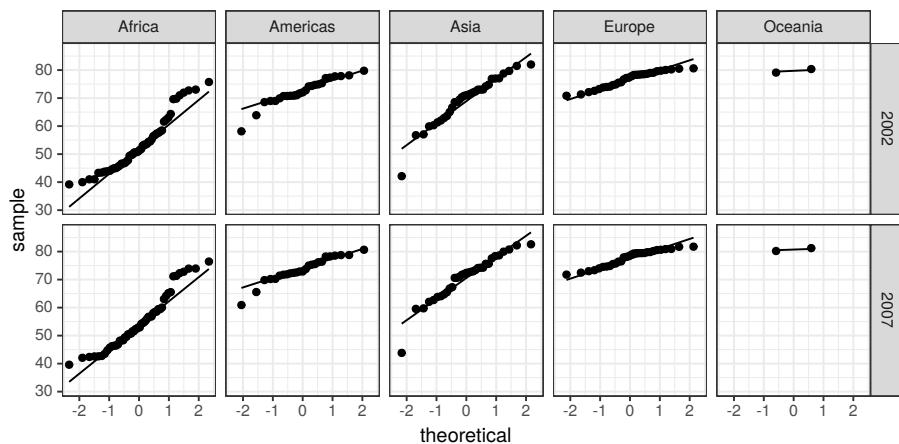


FIGURE 6.2: Q-Q plot: country life expectancy by continent and year

What can we see? We are looking to see if the data follow the 45 degree line which is included in the plot. These do reasonably, except for Africa which is curved upwards at each end, suggesting a skew.

We are frequently asked about performing a hypothesis test to check the assumption of normality, such as the Shapiro-Wilk normality test. We do not recommend this, simply because it is often

non-significant when the number of observations is small but the data look skewed, and often significant when the number of observations is high but the data look reasonably normal on inspection of plots. It is therefore not useful in practice - common sense should prevail.

6.5.3 Boxplot

Boxplots are our preferred method for comparing a continuous variable such as life expectancy with a categorical explanatory variable. It is much better than a bar plot, or a bar plot with error bars, sometimes called a dynamite plot.

The box represents the median and interquartile range (where 50% of the data sits). The lines (whiskers) by default are 1.5 times the interquartile range. Outliers are represented as points.

Thus it contains information, not only on central tenancy (median), but on the variation in the data and the distribution of the data, for instance a skew should be obvious.

```
mydata %>%
  filter(year %in% c(2002, 2007)) %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  facet_grid(. ~ year) # spread by year, note `.`
```

What can we see? The median life expectancy is lower in Africa than in any other continent. The variation in life expectancy is greatest in Africa and smallest in Oceania. The data in Africa looks skewed, particularly in 2002 - the lines/whiskers are unequal lengths.

We can add further arguments

```
mydata %>%
  filter(year %in% c(2002, 2007)) %>%
  ggplot(aes(x = factor(year), y = lifeExp)) +
  geom_boxplot(aes(fill = continent)) + # add colour to boxplots
  geom_jitter(alpha = 0.4) + # alpha = transparency
```

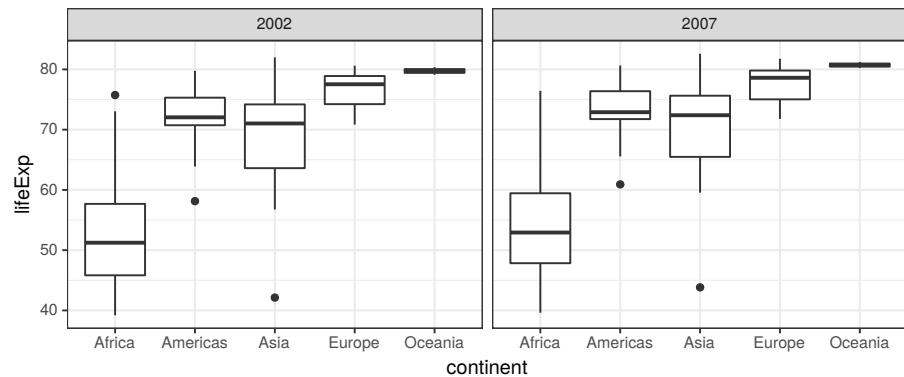


FIGURE 6.3: Boxplot: country life expectancy by continent and year

```

facet_grid(. ~ continent) +
  theme(legend.position = "none") +
  xlab("Year") +
  ylab("Life expectancy (years)") +
  ggtitle(
    "Life expectancy by continent in 2002 v 2007") # add title
  
```

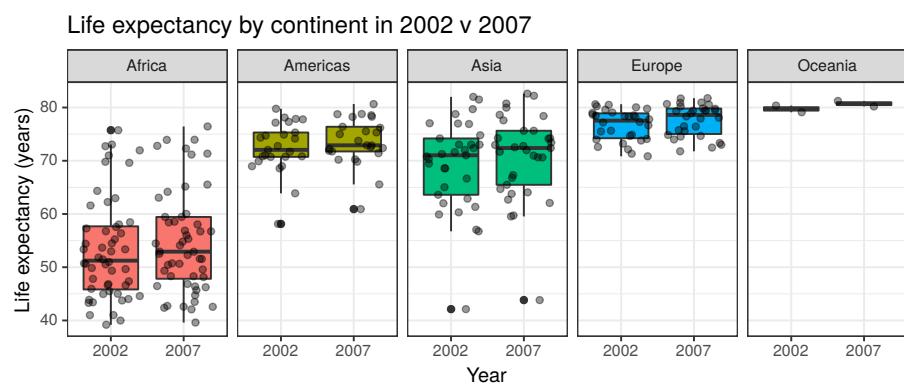


FIGURE 6.4: Boxplot with jitter points: country life expectancy by continent and year

6.6 Compare the means of two groups

6.6.1 T-test

A *t*-test is used to compare the means of two groups of continuous variables. Volumes have been written about this elsewhere, and we won't rehearse it here.

There are various variations on the *t*-test. We will use two here. The most useful in our context is a two-sample test of independent groups. Repeated-measures data, such as comparing the same countries between years, can be analysed using a paired *t*-test.

6.6.2 Two-sample *t*-tests

Referring to the first figure, let's compare life expectancy between Asia and Europe for 2007. What is imperative, is that you decide what sort of difference exists by looking at the boxplot, rather than relying on the *t*-test output. The median for Europe is clearly higher than in Asia. The distributions overlap, but it looks likely that Europe has a higher life expectancy than Asia.

```
ttest_data = mydata %>%                                # save as object testdata
  filter(year == 2007) %>%
  filter(continent %in% c("Asia", "Europe")) # Asia/Europe only

ttest_result =
  t.test(lifeExp ~ continent, data = ttest_data) # Base R t.test
ttest_result

## 
## Welch Two Sample t-test
##
## data: lifeExp by continent
## t = -4.6468, df = 41.529, p-value = 3.389e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -9.926525 -3.913705
## sample estimates:
##   mean in group Asia mean in group Europe
##             70.72848           77.64860
```

The Welch two-sample t-test is the most flexible and copes with differences in variance (variability) between groups, as in this example. The difference in means is provided at the bottom of the output. The *t*-value, degrees of freedom (df) and p-value are all provided. The p-value is 0.00003.

The base R output is not that easy to utilise. For reference, the results can be explored and exported. However, more straightforward methods are provided below.

```
names(ttest_result) # Names of elements of result object

## [1] "statistic"    "parameter"    "p.value"      "conf.int"     "estimate"
## [6] "null.value"   "stderr"        "alternative"  "method"       "data.name"

str(ttest_result) # Details of result object

## List of 10
## $ statistic : Named num -4.65
##   ..- attr(*, "names")= chr "t"
## $ parameter : Named num 41.5
##   ..- attr(*, "names")= chr "df"
## $ p.value   : num 3.39e-05
## $ conf.int  : num [1:2] -9.93 -3.91
##   ..- attr(*, "conf.level")= num 0.95
## $ estimate   : Named num [1:2] 70.7 77.6
##   ..- attr(*, "names")= chr [1:2] "mean in group Asia" "mean in group Europe"
## $ null.value : Named num 0
##   ..- attr(*, "names")= chr "difference in means"
## $ stderr     : num 1.49
## $ alternative: chr "two.sided"
## $ method     : chr "Welch Two Sample t-test"
## $ data.name  : chr "lifeExp by continent"
## - attr(*, "class")= chr "htest"

ttest_result$p.value # Extracted element of result object

## [1] 3.38922e-05
```

The `broom` package provides useful methods for ‘tidying’ common model outputs into a `tibble`.

The whole analysis can be constructed as a single piped function.

```

library(broom)
mydata %>%
  filter(year == 2007) %>%
  filter(continent %in% c("Asia", "Europe")) %>% # Asia/Europe only
  t.test(lifeExp ~ continent, data = .) %>%
  tidy()

## # A tibble: 1 x 10
##   estimate estimate1 estimate2 statistic p.value parameter conf.low
##   <dbl>     <dbl>     <dbl>     <dbl>    <dbl>     <dbl>
## 1 -6.92      70.7     77.6    -4.65 3.39e-5     41.5    -9.93
## # ... with 3 more variables: conf.high <dbl>, method <chr>,
## #   alternative <chr>

```

6.6.3 When pipe sends data to the wrong place: use , `data = .` to direct it

In the code above, the `, data = .` bit is necessary because the pipe usually sends data to the beginning of function brackets. So `mydata %>% t.test(lifeExp ~ continent)` would be equivalent to `t.test(mydata, lifeExp ~ continent)`. However, this is not an order that `t.test()` will accept. `t.test()` wants us to specify the formula first, and then wants the data these variables are present in. So we have to use the `.` to tell the pipe to send the data to the second argument of `t.test()`, not the first.

6.6.4 Paired *t*-tests

Consider that we want to compare the difference in life expectancy in Asian countries between 2002 and 2007. The overall difference is not impressive in the boxplot.

We can plot differences at the country level directly.

```

paired_data = mydata %>%
  filter(year %in% c(2002, 2007)) %>% # 2002 and 2007 only
  filter(continent == "Asia")           # Asia only

paired_data %>%
  ggplot(aes(x = year, y = lifeExp,

```

```
group = country)) +      # for individual country lines
geom_line()
```

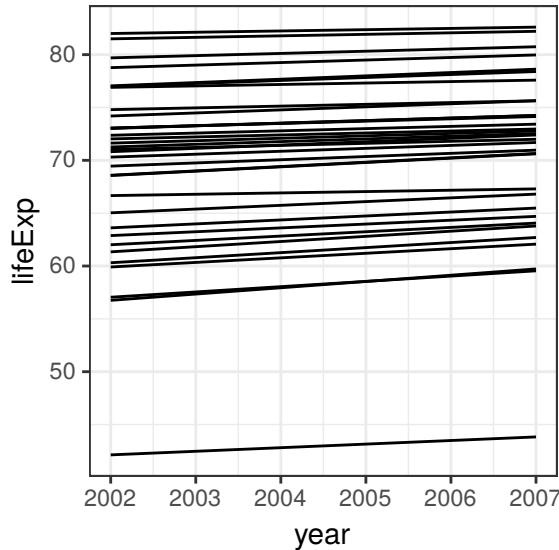


FIGURE 6.5: Line plot: Change in life expectancy in Asian countries from 2002 to 2007

What is the difference in life expectancy for each individual country? We don't usually have to produce this directly, but here is one method.

```
paired_table = paired_data %>%      # save object paired_data
  select(country, year, lifeExp) %>% # select vars interest
  spread (year, lifeExp) %>%        # make wide table
  mutate(
    dlifeExp = `2007` - `2002`        # difference in means
  )
```

```
paired_table
```

```
## # A tibble: 33 x 4
##   country     `2002` `2007` dlifeExp
##   <fct>       <dbl>   <dbl>   <dbl>
## 1 Afghanistan 42.1    43.8    1.70
## 2 Bahrain      74.8    75.6    0.84
## 3 Bangladesh   62.0    64.1    2.05
## 4 Cambodia     56.8    59.7    2.97
```

```
## 5 China          72.0  73.0  0.933
## 6 Hong Kong, China 81.5  82.2  0.713
## 7 India          62.9  64.7  1.82
## 8 Indonesia      68.6  70.6  2.06
## 9 Iran            69.5  71.0  1.51
## 10 Iraq           57.0  59.5  2.50
## # ... with 23 more rows
```

```
# Mean of difference in years
paired_table %>% summarise( mean(dlifeExp) )
```

```
## # A tibble: 1 × 1
##   `mean(dlifeExp)`
##   <dbl>
## 1 1.49
```

On average, therefore, there is an increase in life expectancy of 1.5 years in Asian countries between 2002 and 2007. Let's test whether this number differs from zero with a paired *t*-test.

```
paired_data %>%
  t.test(lifeExp ~ year, data = .) # Include paired = TRUE

##
## Welch Two Sample t-test
##
## data: lifeExp by year
## t = -0.74294, df = 63.839, p-value = 0.4602
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -5.513722 2.524510
## sample estimates:
## mean in group 2002 mean in group 2007
##             69.23388             70.72848
```

The results show a highly significant difference. As an exercise you can repeat this analysis simply comparing the means in an unpaired manner. The resulting p-value is `R paired_data %>% t.test(lifeExp ~ year, data = .)$p.value`.

Why is there such a difference between the two approaches? This emphasises just how important it is to plot the data first. The average difference of 1.5 years is highly consistent between countries, as shown on the line plot, and this differs from zero. It is up to

you the investigator to interpret the relevance of the effect size of 1.5 y in reporting the finding.

6.7 Compare the mean of one group

6.7.1 One sample *t*-tests

We can use a *t*-test to determine whether the mean of a distribution is different to a specific value.

The paired *t*-test above is equivalent to a one-sample *t*-test on the calculated difference in life expectancy being different to zero.

```
t.test(paired_table$dlifeExp)

##
##  One Sample t-test
##
## data:  paired_table$dlifeExp
## t = 14.338, df = 32, p-value = 1.758e-15
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  1.282271 1.706941
## sample estimates:
## mean of x
##  1.494606
```

We can compare to values other than zero. For instance, we can test whether the mean life expectancy in each continent was significantly different to 77 years in 2007. We have included some extra code here to demonstrate how to run multiple base R tests in one pipe function.

```
mydata %>%
  filter(year == 2007) %>%          # 2007 only
  group_by(continent) %>%           # split by continent
  do(
    t.test(. $lifeExp, mu = 77) %>% # dplyr function
    tidy()                          # compare mean to 77 years
  )                                 # tidy into tibble
```

```
## # A tibble: 5 x 9
## # Groups: continent [5]
##   continent estimate statistic p.value parameter conf.low conf.high method
##   <fct>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl> <chr>
## 1 Africa      54.8    -16.6   3.15e-22      51     52.1    57.5 One S~
## 2 Americas     73.6    -3.82  8.32e- 4      24     71.8    75.4 One S~
## 3 Asia         70.7    -4.52  7.88e- 5      32     67.9    73.6 One S~
## 4 Europe       77.6     1.19  2.43e- 1      29     76.5    78.8 One S~
## 5 Oceania      80.7     7.22  8.77e- 2      1     74.2    87.3 One S~
## # ... with 1 more variable: alternative <chr>
```

group_modify option - decision on whether to switch to this needed

```
mydata %>%
  filter(year == 2007) %>%          # 2007 only
  group_by(continent) %>%           # split by continent
  group_modify(            # dplyr function
    ~ t.test(.\$lifeExp, mu = 77) %>% # compare mean to 77 years
    tidy()                         # tidy into tibble
  )
```

```
## # A tibble: 5 x 9
## # Groups: continent [5]
##   continent estimate statistic p.value parameter conf.low conf.high method
##   <fct>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl> <chr>
## 1 Africa      54.8    -16.6   3.15e-22      51     52.1    57.5 One S~
## 2 Americas     73.6    -3.82  8.32e- 4      24     71.8    75.4 One S~
## 3 Asia         70.7    -4.52  7.88e- 5      32     67.9    73.6 One S~
## 4 Europe       77.6     1.19  2.43e- 1      29     76.5    78.8 One S~
## 5 Oceania      80.7     7.22  8.77e- 2      1     74.2    87.3 One S~
## # ... with 1 more variable: alternative <chr>
```

The mean life expectancy for Europe and Oceania do not differ from 77, while the others to to varying degrees. In particular, look at the confidence intervals of the tables and whether they include or exclude 77.

6.8 Compare the means of more than two groups

It may be that our question is set around a hypothesis involving more than two groups. For example, we may be interested in com-

paring life expectancy across 3 continents such as the Americas, Europe and Asia.

6.8.1 Plot the data

```
mydata %>%
  filter(year == 2007) %>%
  filter(continent %in%
         c("Americas", "Europe", "Asia")) %>%
  ggplot(aes(x = continent, y=lifeExp)) +
  geom_boxplot()
```

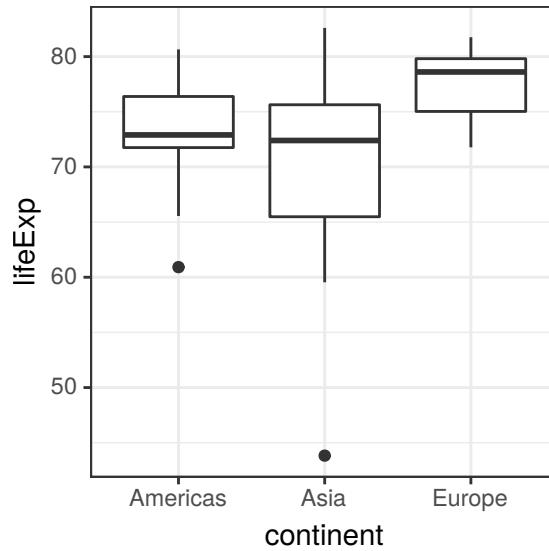


FIGURE 6.6: Boxplot: Life expectancy in selected continents for 2007

6.8.2 ANOVA

Analysis of variance is a collection of statistical tests which can be used to test the difference in means between two or more groups.

In base R form, it produces an ANOVA table which includes an F-test. This so-called omnibus test tells you whether there are

any differences in the comparison of means of the included groups. Again, it is important to plot carefully and be clear what question you are asking.

```
aov_data = mydata %>%
  filter(year == 2007) %>%
  filter(continent %in% c("Americas", "Europe", "Asia"))

fit = aov(lifeExp ~ continent, data = aov_data)
summary(fit)

##          Df Sum Sq Mean Sq F value    Pr(>F)
## continent     2 755.6   377.8   11.63 3.42e-05 ***
## Residuals    85 2760.3    32.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can conclude from this, that there is a difference in the means between at least two pairs of the included continents. As above, the output can be neatened up using the `tidy` function.

```
library(broom)
mydata %>%
  filter(year == 2007) %>%
  filter(continent %in% c("Americas", "Europe", "Asia")) %>%
  aov(lifeExp~continent, data = .) %>%
  tidy()

## # A tibble: 2 x 6
##   term      df sumsq meansq statistic   p.value
##   <chr>     <dbl> <dbl>   <dbl>     <dbl>     <dbl>
## 1 continent     2   756.   378.     11.6  0.0000342
## 2 Residuals    85  2760.   32.5     NA      NA
```

6.8.3 Assumptions

As with the normality assumption of the *t*-test, there are assumptions of the ANOVA model). These are covered in detail in the linear regression chapter and will not be repeated here. Suffice to say that diagnostic plots can be produced to check that the assumptions are fulfilled.

```
par(mfrow=c(2,2))
plot(fit)
```

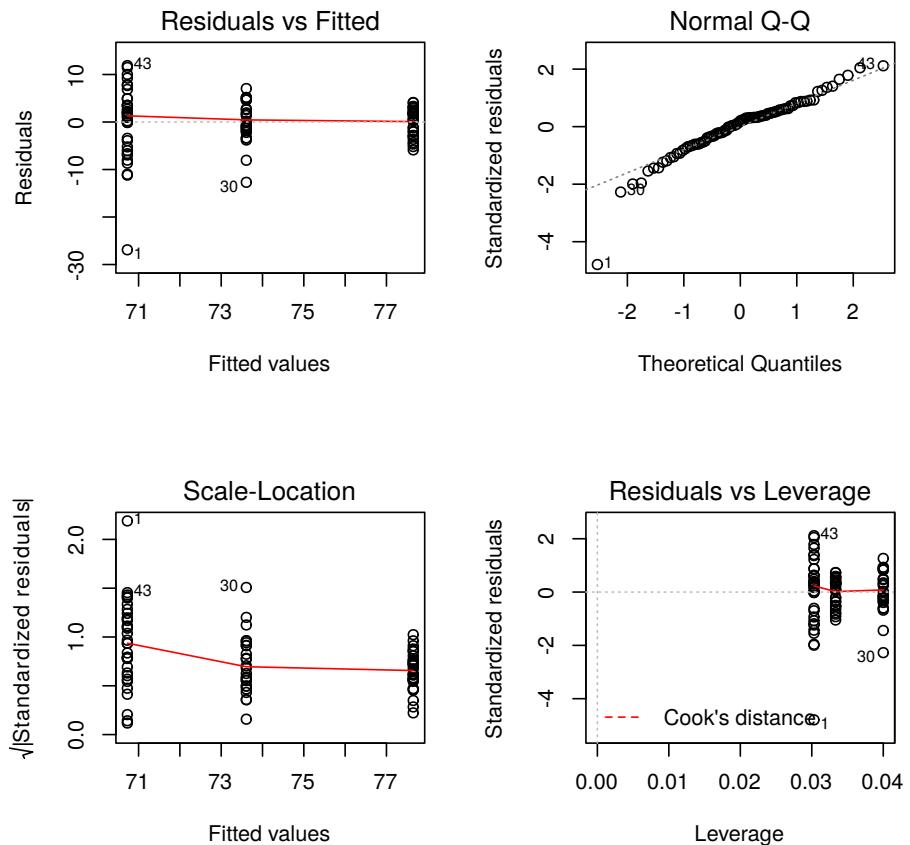


FIGURE 6.7: Diagnostic plots: ANOVA model of life expectancy by continent for 2007

```
par(mfrow=c(1,1))
```

6.8.4 Pairwise testing and multiple comparisons

When the F-test is significant, we will often want to proceed to try and determine where the differences lie. This should of course

be obvious from the boxplot you have made. However, some are fixated on the p-value!

```
pairwise.t.test(aov_data$lifeExp, aov_data$continent,
                 p.adjust.method = "bonferroni")
```

```
##  
##  Pairwise comparisons using t tests with pooled SD  
##  
##  data:  aov_data$lifeExp and aov_data$continent  
##  
##          Americas Asia  
## Asia     0.180   -  
## Europe  0.031  1.9e-05  
##  
## P value adjustment method: bonferroni
```

A matrix of pairwise p-values is produced. Here we can see that there is good evidence of a difference in means between Europe and Asia.

The p-values are corrected for multiple comparisons. When performing a hypothesis test at the 5% level ($\alpha = 0.05$), there is a 5% chance of a type 1 error. That is, a 1 in 20 chance of concluding a difference exists when it in fact does not (formally, this is a 1 in 20 chance of rejecting a true null hypothesis). As more simultaneous statistical tests are performed, the chance of a type 1 error increases.

There are three approaches to this. The first, is not to perform any correction at all. Some advocate that the best approach is simply to present the results of all the tests that were performed, and let the sceptical reader make adjustments for themselves. This is attractive, but presupposes a sophisticated readership who will take the time to consider the results in their entirety.

The second and classical approach, is to control for the so-called family-wise error rate. The “Bonferroni” correction is probably the most famous and most conservative, where the threshold for significance is lowered in proportion to the number of comparisons made. For example, if three comparisons are made, the threshold for significance is lowered to 0.017. Equivalently, any particular p-value can be multiplied by 3 and the value compared to a threshold

of 0.05, as is done above. The Bonferroni method is particular conservative, meaning that type 2 errors may occur (failure to identify true differences, or false negatives) in favour of minimising type 1 errors (false positives).

The third newer approach controls false-discovery rate. The development of these methods has been driven in part by the needs of areas of science where many different statistical tests are performed at the same time, for instance, examining the influence of 1000 genes simultaneously. In these hypothesis-generating settings, a higher tolerance to type 1 errors may be preferable to missing potential findings through type 2 errors. You can see in our example, that the p-values are lower with the `fdr` correction when compared to the `Bonferroni` correction.

```
pairwise.t.test(aov_data$lifeExp, aov_data$continent,
                .adjust.method = "fdr")
```

```
## 
##  Pairwise comparisons using t tests with pooled SD
##
##  data:  aov_data$lifeExp and aov_data$continent
##
##          Americas Asia
## Asia     0.060   -
## Europe  0.016  1.9e-05
##
## P value adjustment method: fdr
```

Try not to get too hung up on this. Be sensible. Plot the data and look for differences. Focus on effect size. For instance, what is the actual difference in life expectancy in years, rather than the p-value of a comparison test. Choose a method which fits with your overall aims. If you are generating hypotheses which you will proceed to test with other methods, the `fdr` approach may be preferable. If you are trying to capture robust effect and want to minimise type 2 errors, use a family-wise approach.

6.9 Non-parametric data

What if your data is different shape to normal or the ANOVA assumptions are not fulfilled (see linear regression chapter). As always, be sensible! Would your data be expected to be normally distributed given the data-generating process? For instance, if you examining length of hospital stay it is likely that your data are highly right skewed - most patients are discharged from hospital in a few days while a smaller number stay for a long time. Is a comparison of means ever going to be the correct approach here? Perhaps you should consider a time-to-event analysis for instance (see chapter ??).

If a comparison of means approach is reasonable, but the normality assumption is not fulfilled there are two approaches,

1. Transform the data;
2. Perform non-parametric tests.

6.9.1 Transforming data

Remember, the Welch *t*-test is reasonably robust to divergence from the normality assumption, so small deviations can be safely ignored.

Otherwise, the data can be transformed to another scale to deal with a skew. A natural *log* scale is common.

TABLE 6.3: Transformations that can be applied to skewed data

Distribution	Transformation	Function
Moderate right skew (+)	Square-root	<code>sqr()</code>
Substantial right skew (++)	Natural log*	<code>log()</code>
Substantial right skew (+++)	Base-10 log*	<code>log10()</code>

Note:

For left skewed data, subtract all values from a constant greater than the maximum value.

* If data contain zero values, add a small constant to all values.

```
africa_data = mydata %>%
  filter(year == 2002) %>%
  filter(continent == "Africa") %>%
  select(country, lifeExp) %>%
  mutate(
    lifeExp_log = log(lifeExp)
  )
head(africa_data) # inspect
```

```
## # A tibble: 6 x 3
##   country     lifeExp lifeExp_log
##   <fct>       <dbl>      <dbl>
## 1 Algeria     71.0       4.26
## 2 Angola      41.0       3.71
## 3 Benin        54.4       4.00
## 4 Botswana    46.6       3.84
## 5 Burkina Faso 50.6       3.92
## 6 Burundi     47.4       3.86
```

```
africa_data %>%
  gather(key, lifeExp, -country) %>% # gather vals to same column
  ggplot(aes(x = lifeExp)) +
  geom_histogram(bins = 15) + # make histogram
  facet_grid(. ~ key, scales = "free") # facet & axes free to vary
```

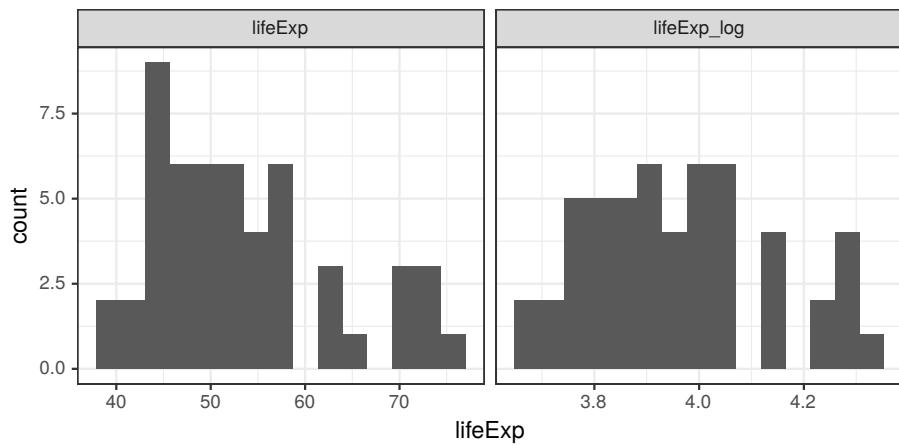


FIGURE 6.8: Histogram: Log transformation of life expectancy for countries in Africa 2002

This has worked well here. The right skew on the Africa data has been dealt with by the transformation. A parametric test such as a *t*-test can now be performed.

6.9.2 Non-parametric test for comparing two groups

The Mann-Whitney U test is also called the Wilcoxon rank-sum test and uses a rank-based method to compare two groups (note the Wilcoxon signed-rank test is for paired data). We can use it to test for a difference in life expectancies for African countries between 1982 and 2007. Let's do a histogram, Q-Q plot and boxplot first.

```
africa_plot = mydata %>%
  filter(year %in% c(1982, 2007)) %>%
  filter(continent %in% c("Africa"))      # only 1982 and 2007
                                            # only Africa

p1 = africa_plot %>%                      # save plot as p1
  ggplot(aes(x = lifeExp)) +
  geom_histogram(bins = 15) +
  facet_grid(. ~ year)

p2 = africa_plot %>%                      # save plot as p2
  ggplot(aes(sample = lifeExp)) +
  geom_qq() +
  geom_qq_line() +
  facet_grid(. ~ year)

p3 = africa_plot %>%                      # save plot as p3
  ggplot(aes(x = factor(year),
             y = lifeExp)) +
  geom_boxplot(aes(fill = factor(year))) + # colour boxplot
  geom_jitter(alpha = 0.4) +               # add data points
  theme(legend.position = "none")          # remove legend

library(patchwork)                          # great for combining plots
p1 / p2 | p3
```

The data is a little skewed based on the histograms and Q-Q plots. The difference between 1982 and 2007 is not particularly striking on the boxplot.

```
africa_plot %>%
  wilcox.test(lifeExp ~ year, data = .)

## Wilcoxon rank sum test with continuity correction
## data: lifeExp by year
```

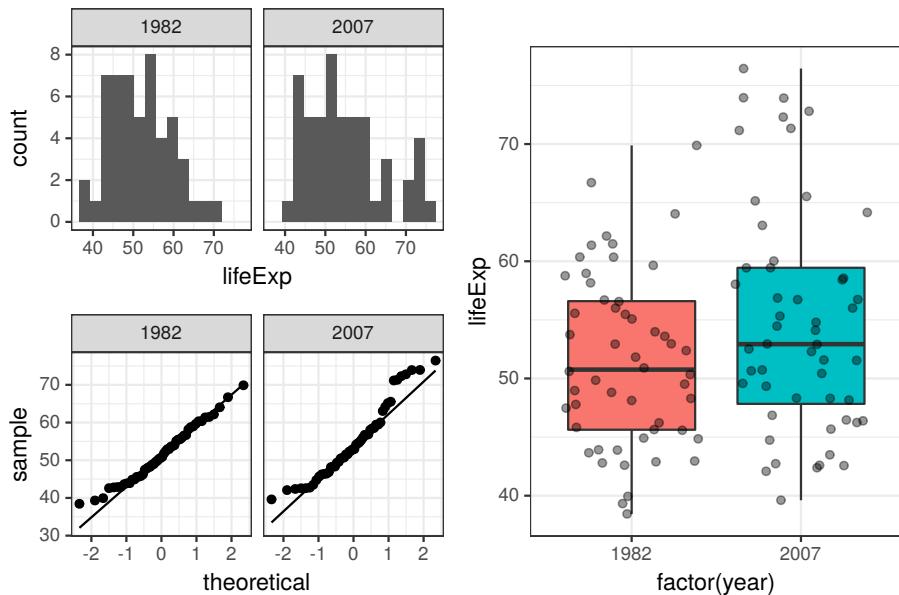


FIGURE 6.9: Panels plots: histogram, Q-Q, boxplot for life expectancy in Africa 1992 v 2007

```
## W = 1130, p-value = 0.1499
## alternative hypothesis: true location shift is not equal to 0
```

6.9.3 Non-parametric test for comparing more than two groups

The non-parametric equivalent to ANOVA, is the Kruskal-Wallis test. It can be used in base R, or via the finalfit package below.

```
library(broom)
mydata %>%
  filter(year == 2007) %>%
  filter(continent %in% c("Americas", "Europe", "Asia")) %>%
  kruskal.test(lifeExp ~ continent, data = .) %>%
  tidy()

## # A tibble: 1 x 4
##   statistic  p.value parameter method
##     <dbl>     <dbl>     <int> <chr>
## 1      21.6  0.0000202       2 Kruskal-Wallis rank sum test
```

6.10 Finalfit approach

The finalfit package provides an easy to use interface for performing non-parametric hypothesis tests. Any number of explanatory variables can be tested against a so-called dependent variable. In this case, this is equivalent to a typical Table 1 in healthcare study.

```
dependent = "year"
explanatory = c("lifeExp", "pop", "gdpPercap")
mydata %>%
  filter(year %in% c(1982, 2007)) %>%          # only 1982 and 2007
  filter(continent == "Africa") %>%              # only Africa
  mutate(
    year = factor(year)                          # change year to factor
  ) %>%
  summary_factorlist(dependent, explanatory,
                      cont = "median", p = TRUE)
```

TABLE 6.4: Life expectancy, population and GDPperCap in Africa 1982 v 2007

label	levels	1982	2007	p
lifeExp	Median (IQR)	50.8 (11.0)	52.9 (11.6)	0.150
pop	Median (IQR)	5668228.5 (8218654.0)	10093310.5 (16454428.0)	0.032
gdpPercap	Median (IQR)	1323.7 (1958.9)	1452.3 (3130.6)	0.506

6.11 Conclusions

Continuous data is frequently encountered in a healthcare setting. Liberal use of plotting is required to really understand the underlying data. Comparisons can easily made between two or more groups of data, but always remember what you are actually trying to analyse and don't become fixated on the p-value. In the next chapter, we will explore the comparison of two continuous variables together with multivariable models of datasets.

6.12 Exercises

6.12.1 Exercise 1

Make a histogram, Q-Q plot, and a box-plot for the life expectancy for a continent of your choice, but for all years. Do the data appear normally distributed?

6.12.2 Exercise 2

1. Select any 2 years in any continent and perform a *t*-test to determine whether mean life expectancy is significantly different. Remember to plot your data first.
2. Extract only the p-value from your `t.test()` output.

6.12.3 Exercise 3

In 2007, in which continents did mean life expectancy differ from 70.

6.12.4 Exercise 4

1. Use ANOVA to determine if the population changed significantly through the 1990s/2000s in individual continents.
-

6.13 Exercise solutions

```
# Exercise 1
## Make a histogram, Q-Q plot, and a box-plot for the life expectancy
```

```
## for a continent of your choice, but for all years.
## Do the data appear normally distributed?

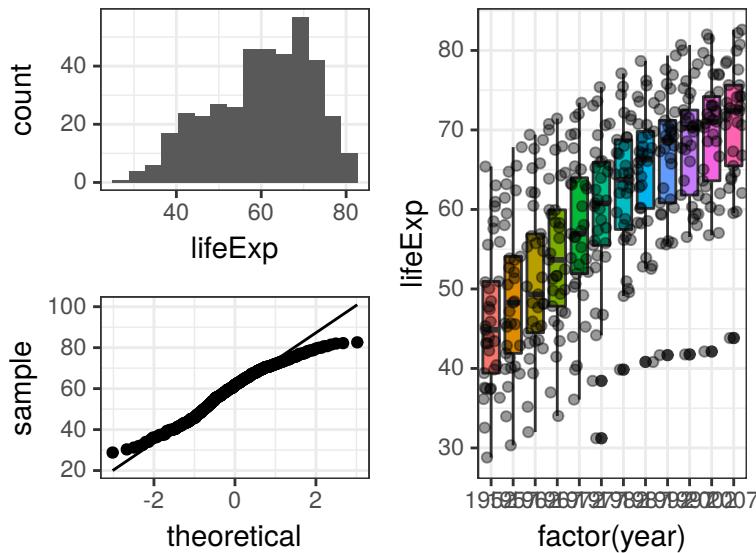
asia_plot = mydata %>%
  filter(continent %in% c("Asia"))

p1 = asia_plot %>%
  ggplot(aes(x = lifeExp)) +
  geom_histogram(bins = 15) ++
  #facet_grid(. ~ year) # no facet

p2 = asia_plot %>%
  ggplot(aes(sample = lifeExp)) +           # `sample` for Q-Q plot
  geom_qq() +
  geom_qq_line() ++
  #facet_grid(. ~ year) # no facet

p3 = asia_plot %>%
  ggplot(aes(x = factor(year), y = lifeExp)) + # year as factor
  geom_boxplot(aes(fill = factor(year))) +
  geom_jitter(alpha = 0.4) +
  theme(legend.position = "none")

library(patchwork)
p1 / p2 | p3
```



```
# Exercise 2
## Select any 2 years in any continent and perform a *t*-test to
```

```

## determine whether mean life expectancy is significantly different.
## Remember to plot your data first.

asia_years = mydata %>%
  filter(continent %in% c("Asia")) %>%
  filter(year %in% c(1952, 1972))

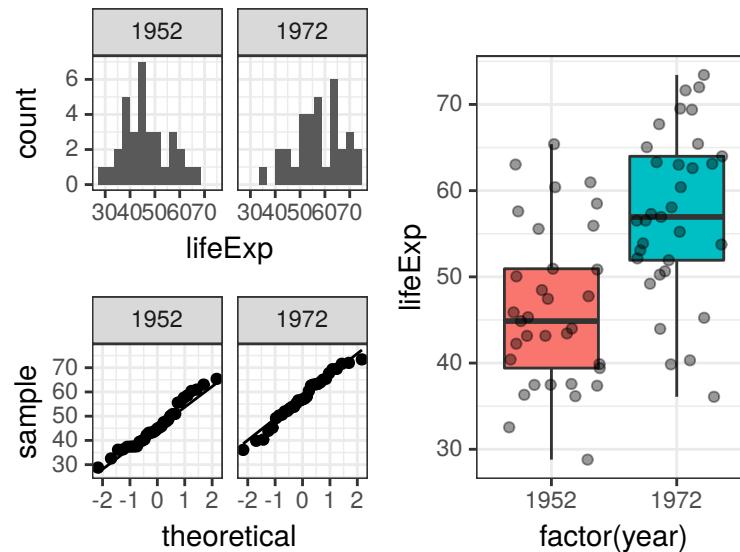
p1 = asia_years %>%
  ggplot(aes(x = lifeExp)) +
  geom_histogram(bins = 15) +
  facet_grid(. ~ year)

p2 = asia_years %>%
  ggplot(aes(sample = lifeExp)) +
  geom_qq() +
  geom_qq_line() +
  facet_grid(. ~ year)

p3 = asia_years %>%
  ggplot(aes(x = factor(year), y = lifeExp)) +
  geom_boxplot(aes(fill = factor(year))) +
  geom_jitter(alpha = 0.4) +
  theme(legend.position = "none")

library(patchwork)
p1 / p2 | p3

```



```
asia_years %>%
  t.test(lifeExp ~ year, data = .)

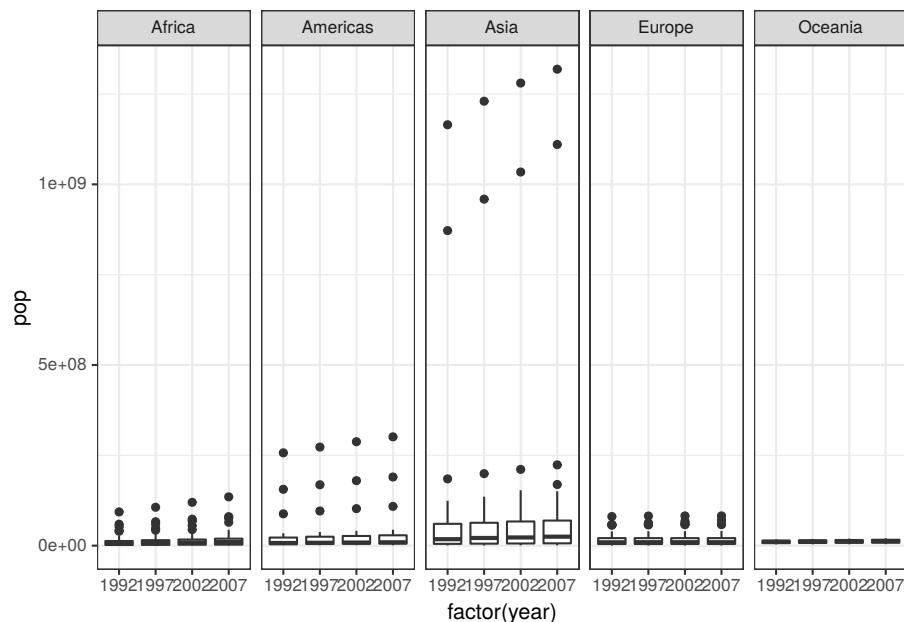
##
## Welch Two Sample t-test
##
## data: lifeExp by year
## t = -4.7007, df = 63.869, p-value = 1.428e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -15.681981 -6.327769
## sample estimates:
## mean in group 1952 mean in group 1972
##           46.31439          57.31927

# Exercise 3
## In 2007, in which continents did mean life expectancy differ from 70
mydata %>%
  filter(year == 2007) %>%
  group_by(continent) %>%
  group_modify(
    ~ t.test(.lifeExp, mu = 70) %>% tidy() # Sometimes awkward in the tidyverse
  )

## # A tibble: 5 x 9
## Groups:   continent [5]
##   continent estimate statistic p.value parameter conf.low conf.high method
##   <fct>      <dbl>     <dbl>    <dbl>     <dbl>    <dbl>    <chr>
## 1 Africa      54.8    -11.4  1.33e-15      51     52.1    57.5 One S~
## 2 Americas    73.6     4.06  4.50e- 4      24     71.8    75.4 One S~
## 3 Asia        70.7     0.525 6.03e- 1      32     67.9    73.6 One S~
## 4 Europe      77.6     14.1  1.76e-14      29     76.5    78.8 One S~
## 5 Oceania     80.7     20.8  3.06e- 2      1     74.2    87.3 One S~
## # ... with 1 more variable: alternative <chr>

# Exercise 4
## Use Kruskal-Wallis to determine if the mean population changed
## significantly through the 1990s/2000s in individual continents.

mydata %>%
  filter(year >= 1990) %>%
  ggplot(aes(x = factor(year), y = pop)) +
  geom_boxplot() +
  facet_grid(. ~ continent)
```



```
mydata %>%
  filter(year >= 1990) %>%
  group_by(continent) %>%
  group_modify(
    ~ kruskal.test(pop ~ factor(year), data = .) %>% tidy()
  )
```

```
## # A tibble: 5 x 5
##   continent statistic p.value parameter method
##   <fct>      <dbl>    <dbl>     <int> <chr>
## 1 Africa       2.10    0.553      3 Kruskal-Wallis rank sum test
## 2 Americas     0.847   0.838      3 Kruskal-Wallis rank sum test
## 3 Asia          1.57    0.665      3 Kruskal-Wallis rank sum test
## 4 Europe        0.207   0.977      3 Kruskal-Wallis rank sum test
## 5 Oceania       1.67    0.644      3 Kruskal-Wallis rank sum test
```

7

Linear regression

Smoking is one of the leading causes of statistics.
Fletcher Knebel

7.1 Regression

Regression is a method by which we can determine the existence and strength of the relationship between two or more variables. This can be thought of as drawing lines, ideally straight lines, through data points.

Linear regression is our method of choice for examining continuous outcome variables. Broadly, there are often two separate goals in regression:

- Prediction: fitting a predictive model to an observed dataset. Using that model to make predictions about an outcome from a new set of explanatory variables;
- Explanation: fit a model to explain the inter-relationships between a set of variables.

Figure 7.1 unifies the terms we will use throughout. A clear scientific question should define our `explanatory variable of interest (x)`, which sometimes gets called an exposure, predictor, or independent variable. Our outcome of interest will be referred to as the

dependent variable or outcome (y); it is sometimes referred to as the response. In simple linear regression, there is a single explanatory variable and a dependent variable, and we will sometimes refer to this as *univariable linear regression*. When there is more than one explanatory variable, we will call this *multivariable regression*. Avoid the term *multivariate regression*, which suggests more than one dependent variable. We don't use this method and we suggest you don't either!

Note that the dependent variable is always continuous, it cannot be a categorical variable. The explanatory variables can be either continuous or categorical.

7.1.1 The Question (1)

We will illustrate our examples of linear regression using a classical question which is important to many of us! This is the relationship between coffee consumption and blood pressure (and therefore cardiovascular events, such as myocardial infarction and stroke). There has been a lot of backwards and forwards over decades about whether coffee is harmful, has no effect, or is in fact beneficial. Figure 7.1 shows a linear regression example. Each point is a person. The explanatory variable is average number of cups of coffee per day (x) and systolic blood pressure as the dependent variable (y). This next bit is important! These data are made up, fake, randomly generated, fabricated, not real. So please do not alter your coffee habit on the basis of these plots!

7.1.2 Fitting a regression line

Simple linear regression uses the *ordinary least squares* method for fitting. The details are beyond the scope here, but if you want to get out the linear algebra/matrix maths you did in high school, an enjoyable afternoon can be spent proving to yourself how it actually works.

Figure 7.2 aims to make this easy to understand. The maths defines

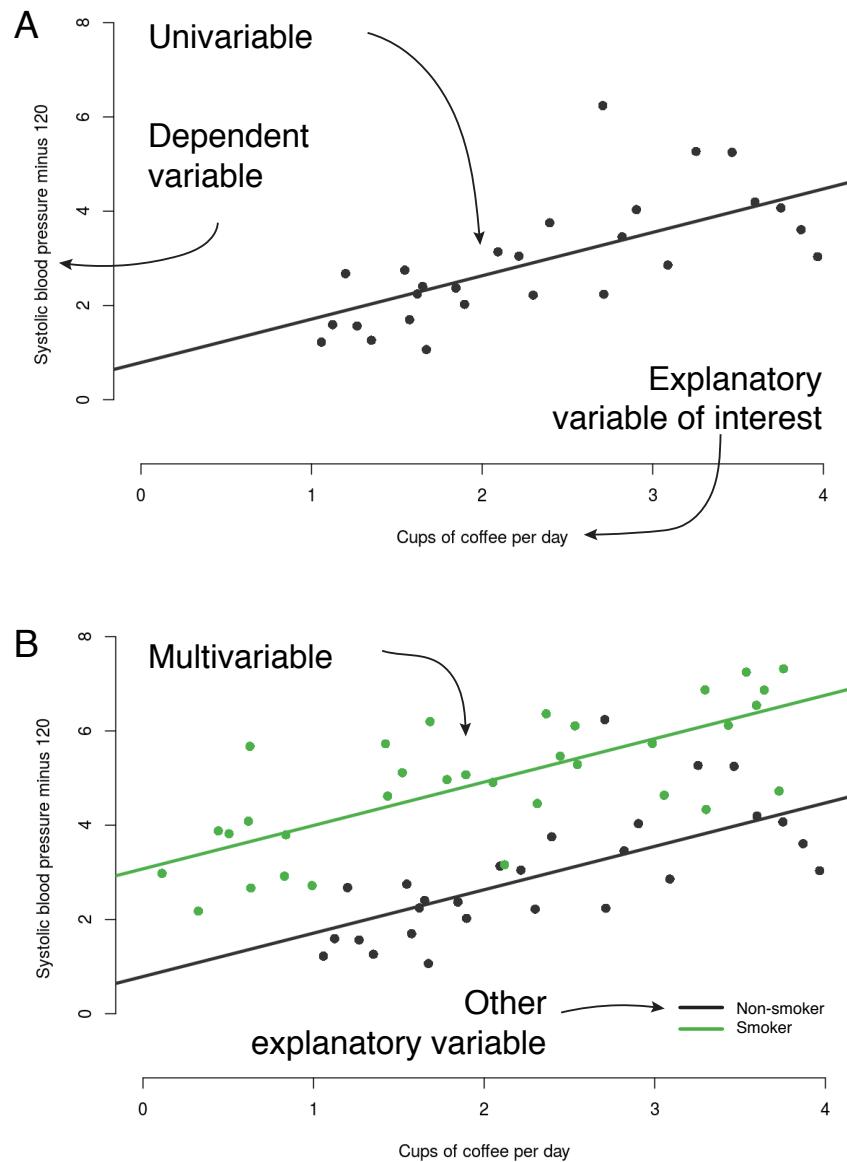


FIGURE 7.1: The anatomy of a regression plot.

a line which best fits the data provided. For the line to fit best, the distances between it and the observed data should be as small as possible. The distance from each observed point to the line is called a *residual* - one of those statistical terms that bring on the sweats. It just refers to the “residual error” left over after the line is fitted.

You can use the simple regression shiny app¹ to explore the concept. We want the residuals to be as small as possible. We can square each residual (to get rid of minuses and make the algebra more convenient) and add them up. If this number is as small as possible, the line is fitting as best it can. Or in more formal language, we want to minimise the sum of squared residuals.

7.1.3 When the line fits well

Linear regression modelling has four main assumptions:

1. Linear relationship between predictors and outcome;
2. Independence of residuals;
3. Normal distribution of residuals;
4. Equal variance of residuals.

You can use the simple regression diagnostics shiny app² to get a handle on these.

Figure 7.3 shows diagnostic plots from the app, which we will run ourselves below.

Linear relationship

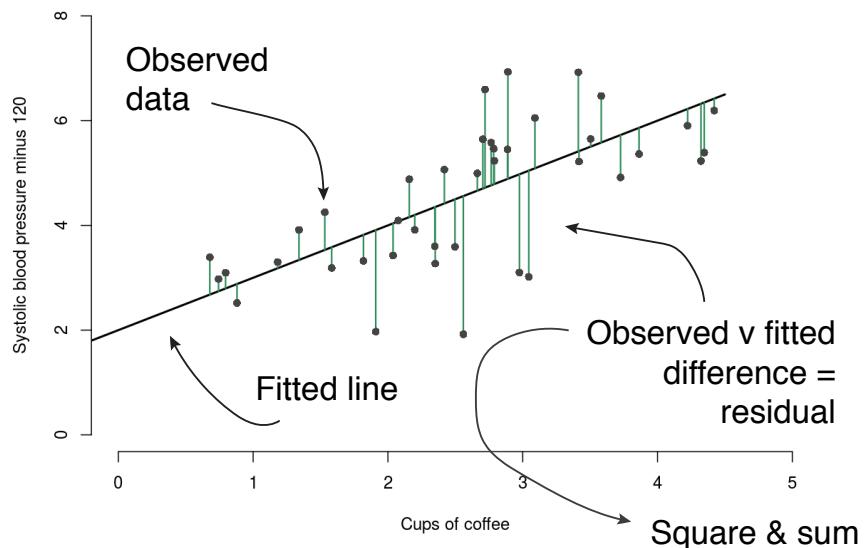
A simple scatter plot should show a linear relationship between the explanatory and the dependent variable, as in figure 7.3A. If the data describe a non-linear pattern (figure 7.3B), then a straight line is not going to fit it well. In this situation, an alternative model should be considered, such as including a quadratic (x^2) term.

¹https://argosshare.is.ed.ac.uk/simple_regression

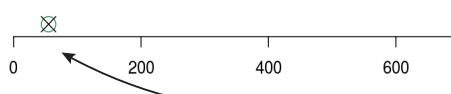
²https://argosshare.is.ed.ac.uk/simple_regression_diagnostics

A

Linear model of systolic blood pressure by coffee consumption

**B**

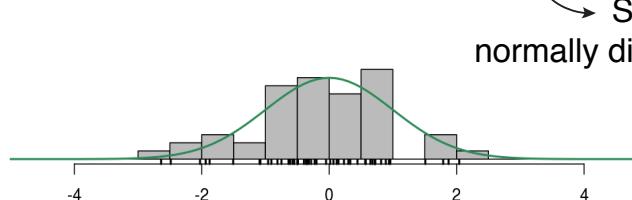
Sum of Squares of Residuals



This is minimised
for best fitting line

C

Distribution of Residuals



Should be
normally distributed

FIGURE 7.2: How a regression line is fitted.

Independence of residuals

The observations and therefore the residuals should be independent. This is more commonly a problem in time series data, where observations may be correlated across time with each other (auto-correlation).

Normal distribution of residuals

The observations should be normally distributed around the fitted line. This means that the residuals should show a normal distribution with a mean of zero (figure 7.3A). If the observations are not equally distributed around the line, the histogram of residuals will be skewed and a normal Q-Q plot will show residuals diverging from the 45 degree line (figure 7.3B) (see section 6.5.2).

Equal variance of residuals

The distribution (spread) of the observations around the fitted line should be the same on the left side as on the right side. Look at the fan-shaped data on the simple regression diagnostics shiny app³. This should be obvious on the residuals vs. fitted values plot, as well as the histogram and normal Q-Q plot.

This is really all about making sure that the line you draw through your data points is valid. It is about ensuring that the regression line is appropriate across the range of the explanatory variable and dependent variable. It is about understanding the underlying data, rather than relying on a fancy statistical test that gives you a *p*-value.

7.1.4 The fitted line and the linear equation

We promised to keep the equations to a minimum, but this one is so important it needs to be included. But it is easy to understand, so fear not.

Figure 7.4 links the fitted line, the linear equation, and the output from R. Some of this will likely be already familiar to you.

³https://argosshare.is.ed.ac.uk/simple_regression_diagnostics

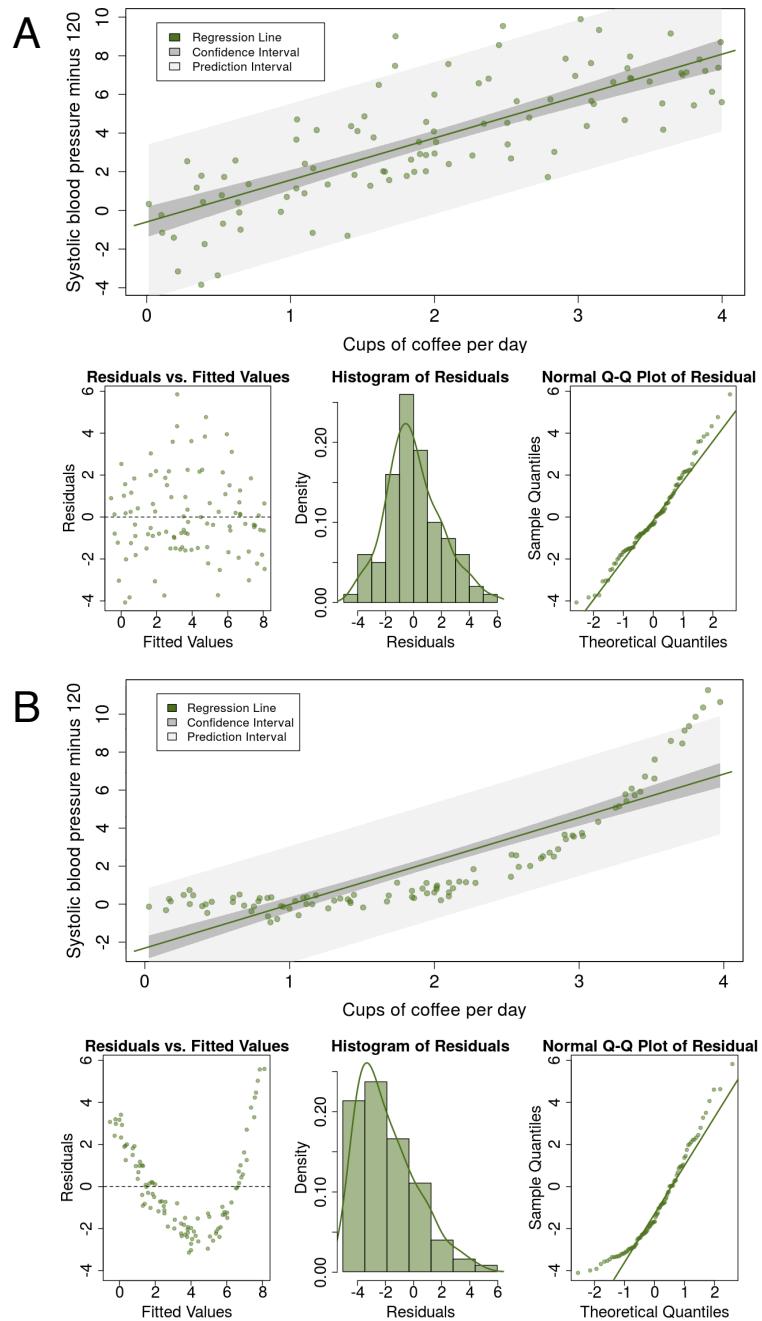


FIGURE 7.3: Regression diagnostics. Does this also appear in the contents. What about this?

Figure 7.4A shows a scatter plot with fitted lines from a multivariable linear regression model. The plot is taken from the multivariable regression shiny app⁴. Remember, these data are simulated and are not real. This app will really help you understand different regression models, more on this below. The app allows us to specify “the truth” with the sliders on the left hand side. For instance, we can set the *intercept* = 1, meaning that when all $x = 0$, the value of the dependent variable, $y = 1$.

Our model has a continuous explanatory variable of interest (average coffee consumption) and a further categorical variable (smoking). In the example the truth is set as *intercept* = 1, $\beta_1 = 1$ (true effect of coffee on blood pressure, gradient/slope of line), and $\beta_2 = 2$ (true effect of smoking on blood pressure). The points on the plot are simulated following the addition of random noise.

Figure 7.4B shows the default output in R for this linear regression model. Look carefully and make sure you are clear how the fitted lines, the linear equation, and the R output fit together. In this example, the random sample from our true population specified above shows *intercept* = 0.67, $\beta_1 = 1.00$ (coffee), and $\beta_2 = 2.48$ (smoking). A *p*-value is provided ($Pr(> |t|)$), which is the result of a null hypothesis significance test for the gradient of the line being equal to zero.

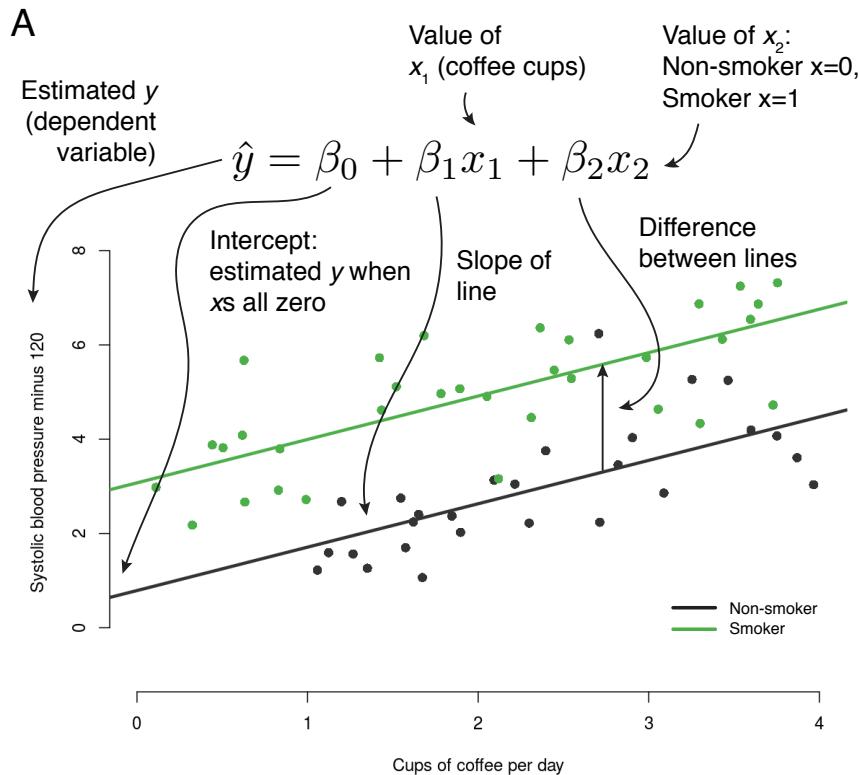
7.1.5 Effect modification

Effect modification occurs when the size of the effect of the explanatory variable of interest (exposure) on the outcome (dependent variable) differs depending on the level of a third variable. Said another way, this is a situation in which an explanatory variable differentially (positively or negatively) modifies the observed effect of another explanatory variable on the outcome.

Again, this is best thought about using the concrete example provided in the multivariable regression shiny app⁵.

⁴https://argosshare.is.ed.ac.uk/multi_regression/

⁵https://argosshare.is.ed.ac.uk/multi_regression/



B Linear regression (`lm`) output

```

Call:
lm(formula = y ~ coffee + smoking, data = df) ← Function call

Residuals:
    Min      1Q  Median      3Q     Max 
-1.4589 -0.6176  0.0043  0.6715  1.8748 

Coefficients:
            Estimate Std. Error t value Pr(>|t|) 
(Intercept) 0.68661   0.24292   2.827  0.00648 ** 
coffee       1.00496   0.09781  10.275 1.38e-14 *** 
smoking      2.47581   0.22694  10.910 1.40e-15 *** 
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8724 on 57 degrees of freedom
Multiple R-squared:  0.8144, Adjusted R-squared:  0.8079 
F-statistic: 125.1 on 2 and 57 DF,  p-value: < 2.2e-16

```

$s\hat{B}P = \beta_0 + \beta_{coffee}x_{coffee} + \beta_{smoking}x_{smoking}$

Function call

Distribution of residuals

Results

Adjusted R²

FIGURE 7.4: Linking the fitted line, regression equation and R output.

Figure 7.5 shows three potential causal pathways.

In the first, smoking is not associated with the outcome (blood pressure) or our explanatory variable of interest (coffee consumption).

In the second, smoking is associated with elevated blood pressure, but not with coffee consumption. This is an example of effect modification.

In the third, smoking is associated with elevated blood pressure and with coffee consumption. This is an example of confounding.

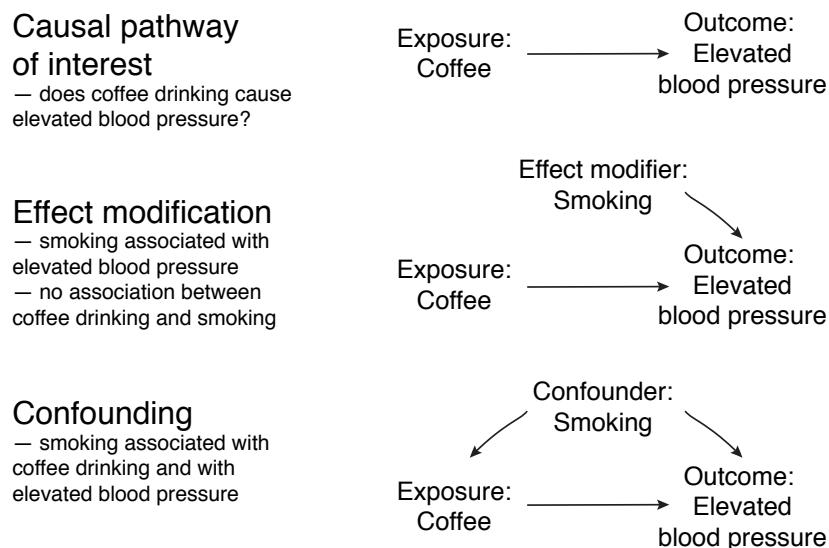


FIGURE 7.5: Causal pathways, effect modification and confounding.

Additive vs. multiplicative effect modification (interaction)

Depending on the field you work, will depend on which set of terms you use. Effect modification can be additive or multiplicative. We refer to multiplicative effect modification as simply including a statistical interaction.

Figure 7.6 should make it clear exactly how these work. The data

have been set-up to include an interaction term. What does this mean?

- $\text{intercept} = 1$: the blood pressure (\hat{y}) for non-smokers who drink no coffee (all $x = 0$);
- $\beta_1 = 1$ (`coffee`): the additional blood pressure for each cup of coffee drunk by non-smokers (slope of the line when $x_2 = 0$);
- $\beta_2 = 1$ (`smoking`): the difference in blood pressure between non-smokers and smokers who drink no coffee ($x_1 = 0$);
- $\beta_3 = 1$ (`coffee:smoking` interaction): the blood pressure (\hat{y}) in addition to β_1 and β_2 , for each cup of coffee drunk by smokers ($x_2 = 1$).

You may have to read that a couple of times in combination with looking at Figure 7.6.

With the additive model, the fitted lines for non-smoking vs smoking are constrained to be parallel. Look at the equation in Figure 7.6B and convince yourself that the lines can never be anything other than parallel.

A statistical interaction (or multiplicative effect modification) is a situation where the effect of an explanatory variable on the outcome is modified in non-additive manner. In other words using our example, the fitted lines are no longer constrained to be parallel.

If we had not checked for an interaction effect, we would have inadequately described the true relationship between these three variables.

What does this mean back in reality? Well it may be biologically plausible for the effect of smoking on blood pressure to increase multiplicatively due to a chemical interaction between cigarette smoke and caffeine, for example.

Note, we are just trying to find a model which best describes the underlying data. All models are approximations of reality.

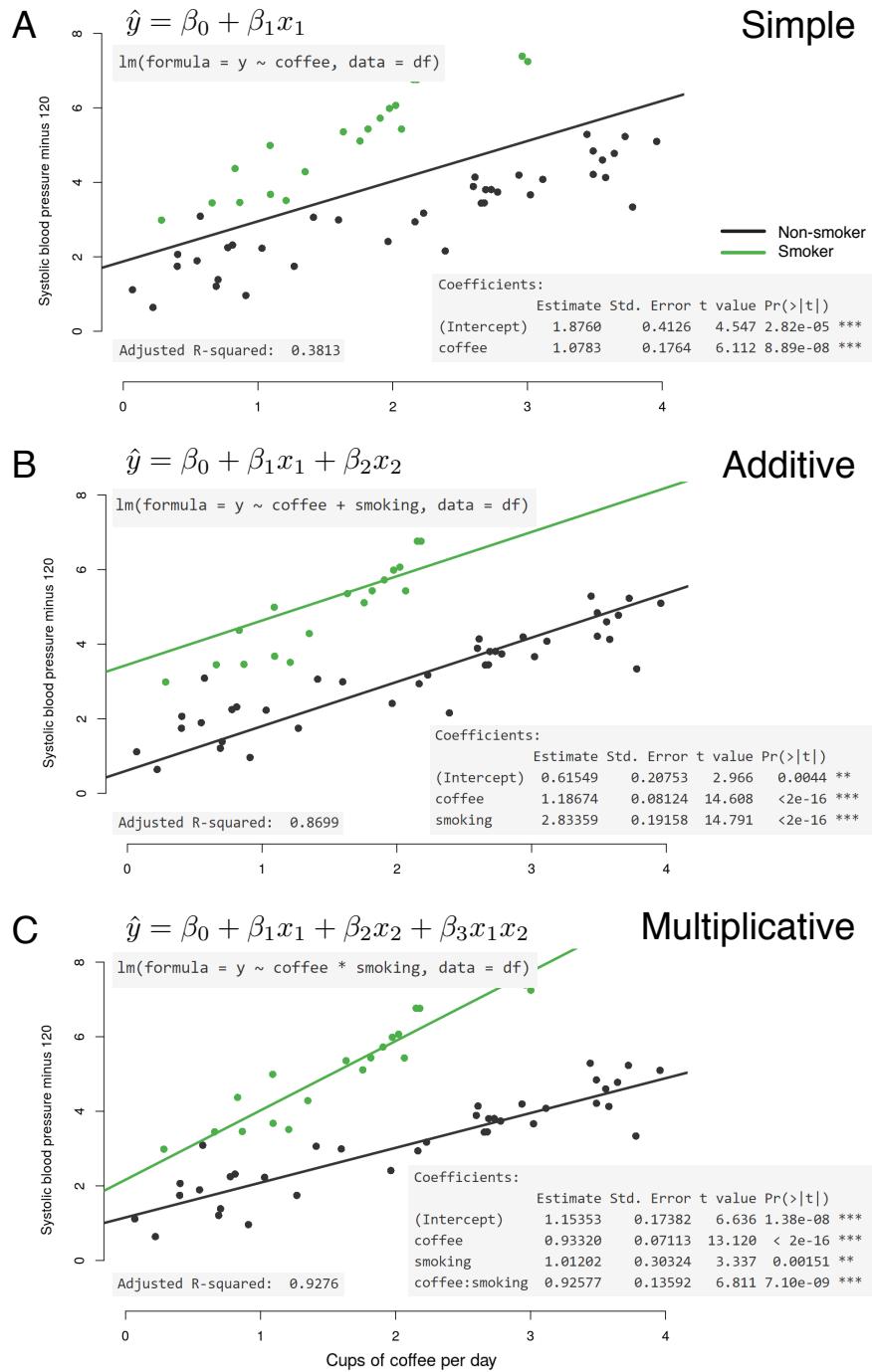


FIGURE 7.6: Multivariable linear regression with additive and multiplicative effect modification.

7.1.6 R-squared and model fit

\index{r-squared}

Figure 7.6 includes a further metric from the R output: `Adjusted R-squared`.

R-squared is another measure of how close the data are to the fitted line. It is also known as the coefficient of determination and represents proportion of the dependent variable which is explained by the explanatory variable(s). So 0.0 indicates that none of the variability in the dependent is explained by the explanatory (no relationship between data points and fitted line) and 1.0 indicates that the model explains all of the variability in the dependent (fitted line follows data points exactly).

R provides the `r-squared` and the `Adjusted R-squared`. The adjusted R-squared includes a penalty the more explanatory variables are included in the model. So if the model includes variables which do not contribute to the description of the dependent variable, the adjusted R-squared will be lower.

Looking again at Figure 7.6, in A, a simple model of coffee alone does not describe the data well (adjusted R-squared 0.38). Add smoking to the model improves the fit as can be seen by the fitted lines (0.87). But a true interaction exists in the actual data. By including this interaction in the model, the fit is very good indeed (0.93).

7.1.7 Confounding

The last important concept to mention here is confounding. Confounding is a situation in which the association between an explanatory variable (exposure) and outcome (dependent variable) is distorted by the presence of another explanatory variable.

In our example, confounding exists if there is an association between smoking and blood pressure AND smoking and coffee con-

sumption (Figure 7.5C). This exists simply if smokers drink more coffee than non-smokers.

Figure 7.7 shows this really clearly. The underlying data have been altered so that those who drink more than two cups of coffee per day also smoke and those who drink fewer than two cups per day do not smoke. A true effect of smoking on blood pressure is entered, but NO effect of coffee on blood pressure.

If we simply fit blood pressure by coffee consumption (Figure 7.7A), then we may mistakenly conclude a relationship between coffee consumption and blood pressure. But this does not exist, because the ground truth we have set is that no relationship exists between coffee and blood pressure. We are simply seeing the effect of smoking on blood pressure, which is confounding the effect of coffee on blood pressure.

If we include the confounder in the model by adding smoking, the true relationship becomes apparent. Two parallel flat lines indicating no effect of coffee on blood pressure, but a relationship between smoking and blood pressure. This procedure is often referred to as controlling for or adjusting for confounders.

7.1.8 Summary

We have intentionally spent some time going through the principles and applications of linear regression because it is so important. A firm grasp of these concepts lead to an easy understanding of other regression procedures, such as logistic regression and Cox Proportional Hazards regression.

We will now perform all this ourselves in R using a gapminder dataset which you are familiar with from preceding chapters.

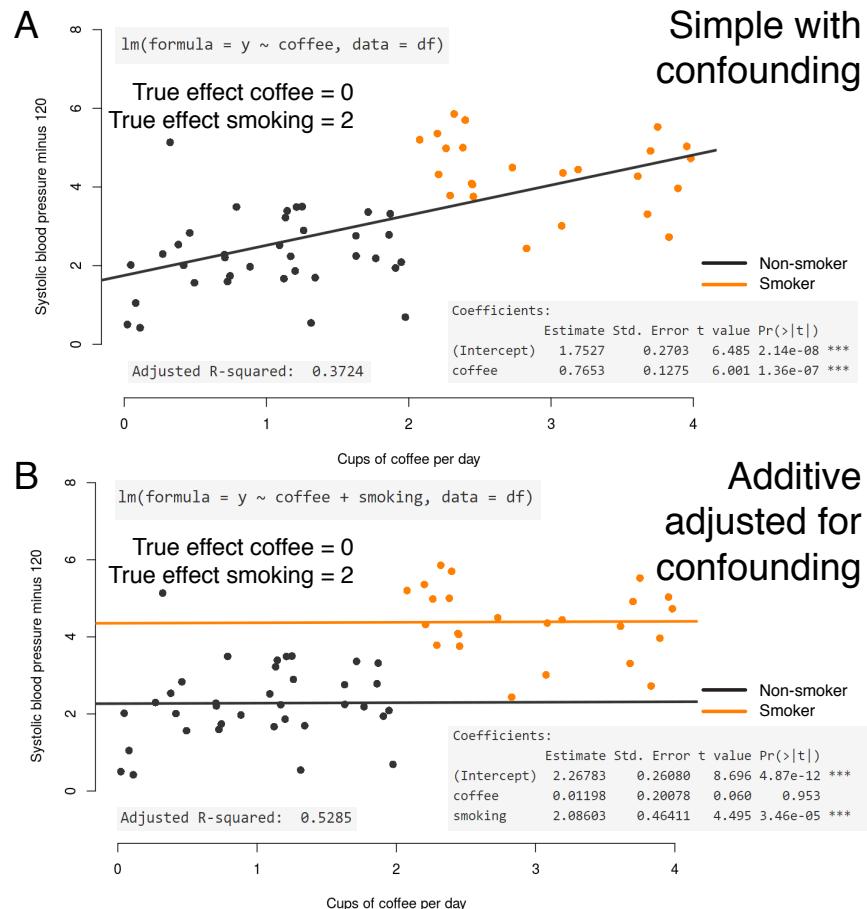


FIGURE 7.7: Multivariable linear regression with confounding of coffee drinking by smoking.

7.2 Fitting simple models

7.2.1 The Question (2)

We are interested in modelling the change in life expectancy for different countries over the past 60 years.

7.2.2 Get the data

```
library(tidyverse)
library(gapminder) # dataset
library(lubridate) # handles dates
library(finalfit)
library(broom)

theme_set(theme_bw())
mydata = gapminder
```

7.2.3 Check the data

Always check a new dataset, as described in section @ref{chap06-h2-check}.

```
glimpse(mydata) # each variable as line, variable type, first values
missing_glimpse(mydata) # missing data for each variable
ff_glimpse(mydata) # summary statistics for each variable
```

7.2.4 Plot the data

Let's plot the life expectancies in European countries over the past 60 years, focussing on the UK and Turkey. We can add in simple best fit lines using `ggplot` directly.

```
p1 = mydata %>%
  filter(continent == "Europe") %>% # Europe only
  ggplot(aes(x = year, y = lifeExp)) + # lifeExp~year
  geom_point() + # plot points
  facet_wrap(~ country) + # facet by country
  scale_x_continuous(
    breaks = c(1960, 2000)) # adjust x-axis

p2 = p1 + geom_smooth(method = "lm") # add regression line

library(patchwork)
p1 + p2
```

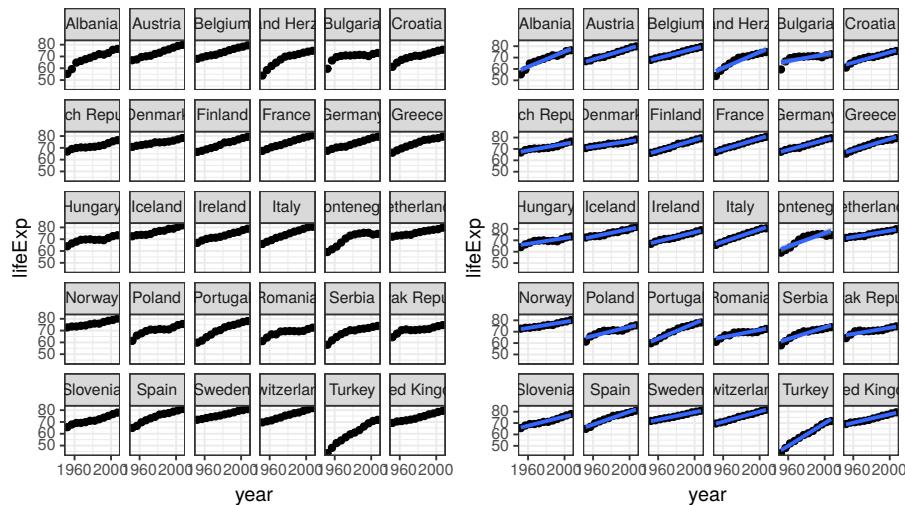


FIGURE 7.8: Scatterplot with fitted line plot: Life expectancy by year in European countries

7.2.5 Simple linear regression

As you can see, `ggplot()` is very happy to run and plot linear regression models for us. While this is sometimes convenient, we usually want to build, run, and explore these models ourselves. We can then investigate the intercepts and the slope coefficients (linear increase per year):

First let's plot two countries to compare, Turkey and United Kingdom

```
mydata %>%
  filter(country %in% c("Turkey", "United Kingdom")) %>%
  ggplot(aes(x = year, y = lifeExp, colour = country)) +
  geom_point()
```

The two non-parallel lines may make you think of what has been discussed above.

First, let's model the two countries separately.

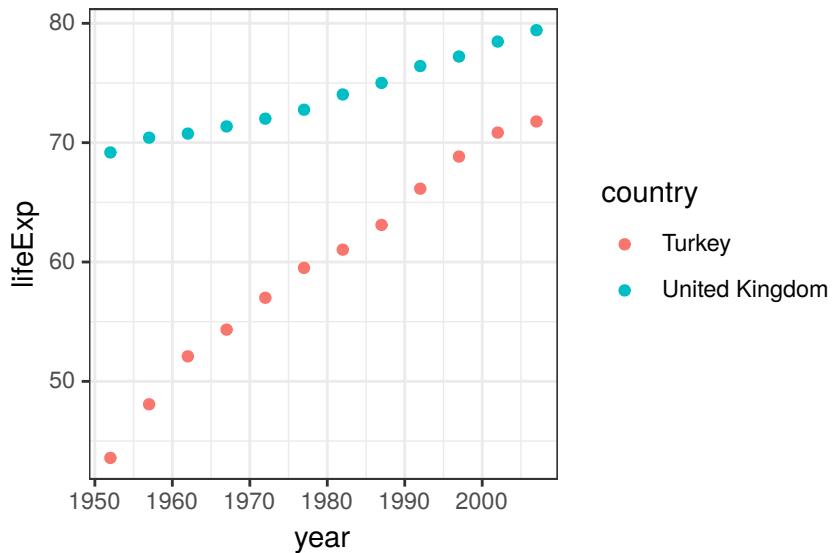


FIGURE 7.9: Scatterplot: Life expectancy by year Turkey and Europe.

```
# United Kingdom
fit_uk = mydata %>%
  filter(country == "United Kingdom") %>%
  lm(lifeExp~year, data = .)

fit_uk %>%
  summary()

##
## Call:
## lm(formula = lifeExp ~ year, data = .)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -0.69767 -0.31962  0.06642  0.36601  0.68165 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -2.942e+02  1.464e+01 -20.10 2.05e-09 ***
## year        1.860e-01  7.394e-03   25.15 2.26e-10 ***
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4421 on 10 degrees of freedom
## Multiple R-squared:  0.9844, Adjusted R-squared:  0.9829 
## F-statistic: 632.5 on 1 and 10 DF,  p-value: 2.262e-10
```

```
# Turkey
fit_turkey = mydata %>%
  filter(country == "Turkey") %>%
  lm(lifeExp~year, data = .)

fit_turkey %>%
  summary()

##
## Call:
## lm(formula = lifeExp ~ year, data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -2.4373 -0.3457  0.1653  0.9008  1.1033 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -924.58989   37.97715 -24.35 3.12e-10 ***
## year         0.49724    0.01918   25.92 1.68e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.147 on 10 degrees of freedom
## Multiple R-squared:  0.9853, Adjusted R-squared:  0.9839 
## F-statistic: 671.8 on 1 and 10 DF,  p-value: 1.681e-10
```

Accessing the coefficients of linear regression

A simple linear regression model will return two coefficients - the intercept and the slope (the second returned value). Compare this to the `summary()` output above.

```
fit_uk$coefficients

## (Intercept)      year
## -294.1965876   0.1859657
```

```
fit_turkey$coefficients

## (Intercept)      year
## -924.5898865   0.4972399
```

In this example, the intercept is telling us that life expectancy at year 0 in the United Kingdom (some 2000 years ago) was -294 years. While this is mathematically correct (based on the data we

have), it obviously makes no sense in practice. It is important at all stages of data analysis, to keep “sense checking” your results.

To make the intercepts meaningful, we will add in a new column called `year_from1952` and re-run `fit_uk` and `fit_turkey` using `year_from1952` instead of `year`.

```
mydata = mydata %>%
  mutate(year_from1952 = year - 1952)

fit_uk = mydata %>%
  filter(country == "United Kingdom") %>%
  lm(lifeExp ~ year_from1952, data = .)

fit_turkey = mydata %>%
  filter(country == "Turkey") %>%
  lm(lifeExp ~ year_from1952, data = .)
```

```
fit_uk$coefficients
```

```
##   (Intercept) year_from1952
##     68.8085256    0.1859657
```

```
fit_turkey$coefficients
```

```
##   (Intercept) year_from1952
##     46.0223205    0.4972399
```

Now, the updated results tell us that in year 1952, the life expectancy in the United Kingdom was 68 years. Note that the slope (0.18) does not change. There was nothing wrong with the original model and the results were correct, the intercept was just not very useful.

Accessing all model information `tidy()` and `glance()`

In the `fit_uk` and `fit_turkey` examples above, we were using `fit_uk %>% summary()` to get R to print out a summary of the model. This summary is not, however, in a rectangular shape so we can't easily access the values or put them in a table/use as information on plot labels.

We use the `tidy()` function from `library(broom)` to get the explanatory variable specific values in a nice tibble:

```
fit_uk %>% tidy()
```

```
## # A tibble: 2 x 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>    <dbl>     <dbl>    <dbl>
## 1 (Intercept) 68.8     0.240     287.  6.58e-21
## 2 year_from1952 0.186    0.00739    25.1  2.26e-10
```

In the `tidy()` output, the column `estimate` includes both the intercepts and slopes.

And we use the `glance()` function to get overall model statistics (mostly the r.squared).

```
fit_uk %>% glance()
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value df logLik AIC BIC
##       <dbl>        <dbl> <dbl>     <dbl>    <int> <dbl> <dbl> <dbl>
## 1     0.984        0.983 0.442     633. 2.26e-10     2 -6.14 18.3 19.7
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

7.2.6 Multivariable linear regression

Multivariable linear regression includes more than one explanatory variable. There are a few ways to include more variables, depending on whether they should share the intercept and how they interact:

Simple linear regression (exactly one predictor variable):

```
myfit = lm(lifeExp ~ year, data = mydata)
```

Multivariable linear regression (additive):

```
myfit = lm(lifeExp ~ year + country, data = mydata)
```

Multivariable linear regression (interaction):

```
myfit = lm(lifeExp ~ year * country, data = mydata)
```

This equivalent to: `myfit = lm(lifeExp ~ year + country + year:country, data = mydata)`

These examples of multivariable regression include two variables: `year` and `country`, but we could include more by adding them with `+`.

In this particular setting, it will become obvious which model is appropriate. So we have complete control over the model being fitted, we will use the `predict()` function directly to obtain our fitted line, rather than leaving it up to `ggplot`.

Model 1: year only

```
mydata_UK_T = mydata %>%
  filter(country %in% c("Turkey", "United Kingdom"))

fit_both1 = mydata_UK_T %>%
  lm(lifeExp ~ year_from1952, data = .)
fit_both1

## 
## Call:
## lm(formula = lifeExp ~ year_from1952, data = .)
## 
## Coefficients:
##   (Intercept)  year_from1952
##             57.4154          0.3416

pred_both1 = predict(fit_both1)

mydata_UK_T %>%
  bind_cols(pred_both = pred_both1) %>%
  ggplot() +
  geom_point(aes(x = year, y = lifeExp, colour = country)) +
  geom_line(aes(x = year, y = pred_both))
```

By fitting year only, the model ignores country. This gives us a fitted line which is the average of life expectancy in the UK and Turkey. This may be desirable, depending on the question. But here we want to best describe the data.

Model 2: year + country

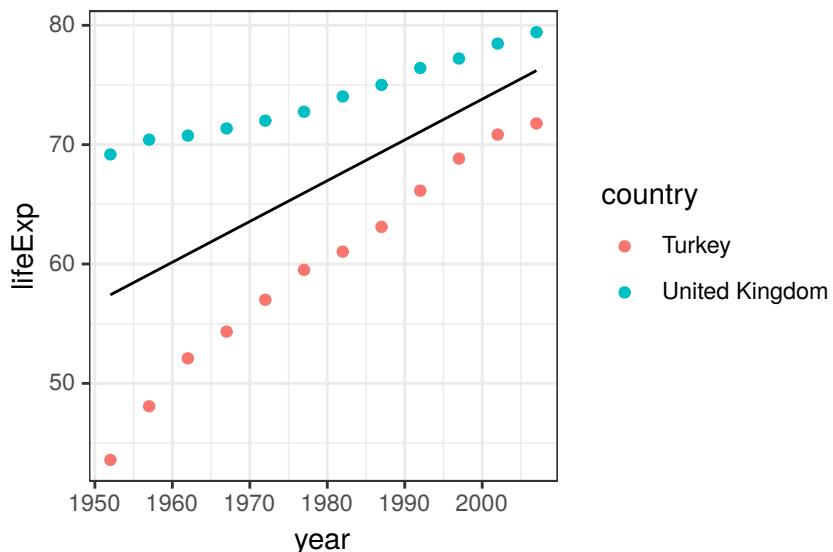


FIGURE 7.10: Scatter and line plot. Life expectancy in Turkey and the UK - univariable fit.

```
fit_both2 = mydata_UK_T %>%
  lm(lifeExp ~ year_from1952 + country, data = .)
fit_both2
```

```
## 
## Call:
## lm(formula = lifeExp ~ year_from1952 + country, data = .)
## 
## Coefficients:
##             (Intercept)      year_from1952  countryUnited Kingdom
##                   50.3023                  0.3416                  14.2262
```

```
pred_both2 = predict(fit_both2)

mydata_UK_T %>%
  bind_cols(pred_both = pred_both2) %>%
  ggplot() +
  geom_point(aes(x = year, y = lifeExp, colour = country)) +
  geom_line(aes(x = year, y = pred_both, colour = country))
```

This is better, by including country in the model, we now have fitted lines better represent the data. However, the lines are constrained to be parallel. This is discussed in detail above. We need

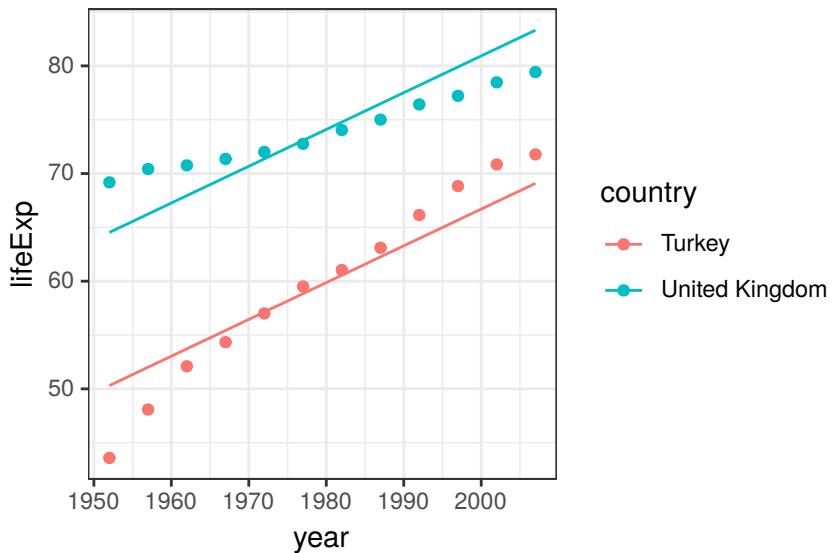


FIGURE 7.11: Scatter and line plot. Life expectancy in Turkey and the UK - multivariable additive fit.

to include an interaction term to allow the effect of year on life expectancy to vary by country in a non-additive manner.

*Model 3: year * country*

```
fit_both3 = mydata_UK_T %>%
  lm(lifeExp ~ year_from1952 * country, data = .)
fit_both3

## 
## Call:
## lm(formula = lifeExp ~ year_from1952 * country, data = .)
## 
## Coefficients:
##                               (Intercept)                  year_from1952
##                                         46.0223                      0.4972
##                               countryUnited Kingdom  year_from1952:countryUnited Kingdom
##                                         22.7862                      -0.3113

pred_both3 = predict(fit_both3)

mydata_UK_T %>%
  bind_cols(pred_both = pred_both3) %>%
```

```
ggplot() +
  geom_point(aes(x = year, y = lifeExp, colour = country)) +
  geom_line(aes(x = year, y = pred_both, colour = country))
```

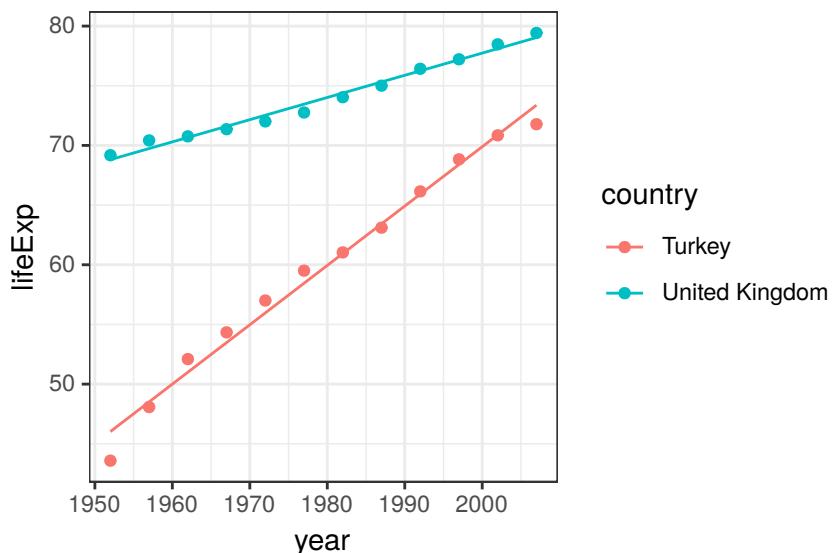


FIGURE 7.12: Scatter and line plot. Life expectancy in Turkey and the UK - multivariable multiplicative fit.

This fits the data much better than the previous two models. You can check the R-squared using `summary(fit_both1)`.

Pro tip

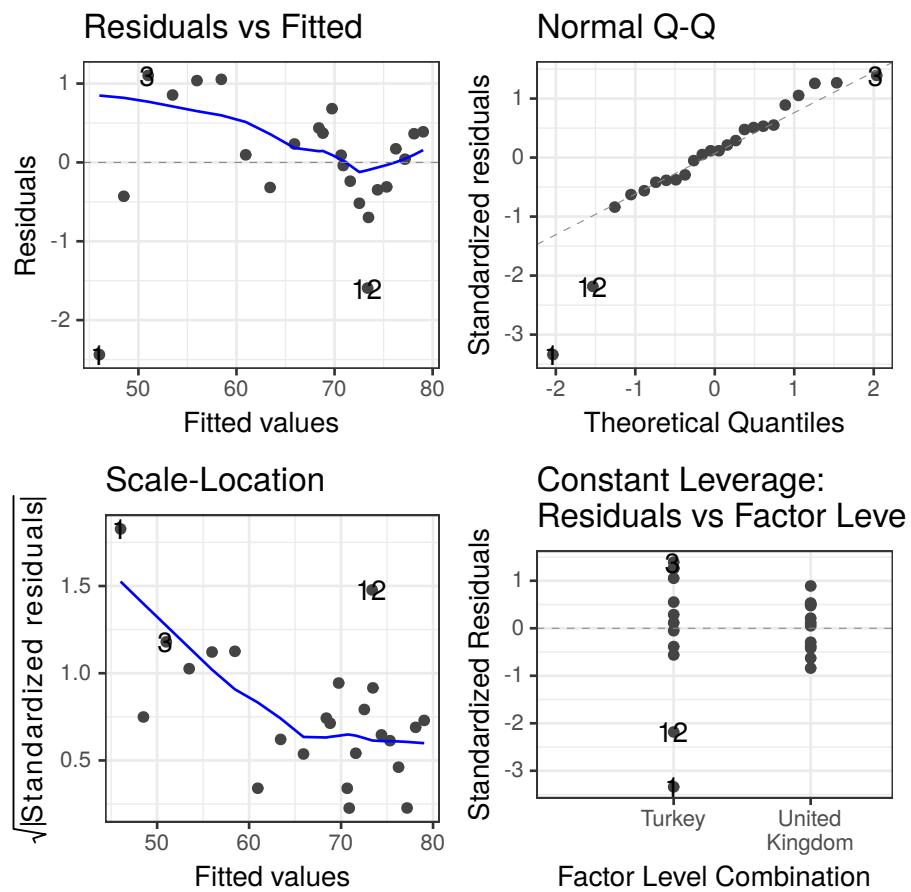
```
library(purrr)
list(fit_both1, fit_both2, fit_both3) %>%
  map_df(glance, .id = "fit")
```

```
## # A tibble: 3 x 12
##   fit r.squared adj.r.squared sigma statistic p.value    df logLik  AIC
##   <chr>     <dbl>        <dbl>    <dbl>    <dbl> <int> <dbl> <dbl>
## 1 1      0.373       0.344  7.98     13.1 1.53e-3     2  -82.9 172.
## 2 2      0.916       0.908  2.99     114. 5.18e-12     3  -58.8 126.
## 3 3      0.993       0.992  0.869    980. 7.30e-22     4  -28.5  67.0
## # ... with 3 more variables: BIC <dbl>, deviance <dbl>, df.residual <int>
```

7.2.7 Check assumptions

The assumptions of linear regression can be checked with diagnostic plots, either by passing the fitted object (`lm()` output) to base R `plot()`, or by using the more convenient function below.

```
library(ggfortify)
autoplot(fit_both3)
```



There is no clear problem with the residuals, as we would expect from the scatterplot with fitted lines.

7.3 Fitting more complex models

7.3.1 The Question (3)

Finally in this chapter, we are going to fit a more complex linear regression model. Here, we will discuss variable selection and introduce the Akaike Information Criterion (AIC).

We will introduce a new dataset: The Western Collaborative Group Study. This classic includes data from 3154 healthy young men aged 39-59 from the San Francisco area who were assessed for their personality type. It aimed to capture the occurrence of coronary heart disease over the following 8.5 years.

We will use it however to explore the relationship between systolic blood pressure (`sbp`) and personality type (`personality_2L`), accounting for potential confounders such as weight (`weight`). Now this is just for fun - don't write in! The study was designed to look at cardiovascular events as the outcome, not blood pressure. But it is a convenient to use blood pressure as a continuous outcome from this dataset, even if that was not the intention of the study.

Personality type is A: aggressive and B: passive.

7.3.2 Model fitting principles

We suggest building statistical models on the basis of the following six pragmatic principles:

1. As few explanatory variables should be used as possible (parsimony);
2. Explanatory variables associated with the outcome variable in previous studies should be accounted for;
3. Demographic variables should be included in model exploration;
4. Population stratification should be incorporated if available;

5. Interactions should be checked and included if influential;
6. Final model selection should be performed using a “criterion-based approach”
 - minimise the Akaike information criterion (AIC)
 - maximise the adjusted R-squared value.

This is not the law, but it probably should be. These principles are sensible as we will discuss through the rest of this book. We strongly suggest you do not use automated methods of variable selection. These are often “forward selection” or “backward elimination” methods for including or excluding particular variables on the basis of a statistical property.

In certain settings, these approaches may be found to work. However, they create an artificial distance between you and the problem you are working on. They give you a false sense of certainty that the model you have created is in some sense valid. And quite frequently, they will get it wrong.

Alternatively, you can follow the six principles above.

A variable may have previously been shown to strongly predict an outcome (think smoking and risk of cancer). This should give you good reason to consider it in your model. But perhaps you think that previous studies were incorrect, or that the variable is confounded by another. All this is fair, but it will be expected that this new knowledge is clearly demonstrated by you, so do not omit these variables before you start.

There are particular variables that are so commonly associated with particular outcomes in healthcare that they should almost always be included at the start. Age, sex, social class, and comorbidity for instance are commonly associated with survival, before you start looking at your explanatory variable of interest or checking that your randomised controlled trial is indeed balanced.

Patients are often clustered by a particular grouping variable, such as treating hospital. There will be commonalities between these patients that are likely not fully explained by your observed variables.

To estimate the coefficients of your variables of interest most accurately, clustering should be accounted for in the analysis.

As demonstrated above, the purpose of the model is to provide a best fit approximation of the underlying data. Effect modification and interactions commonly exist in health datasets, and should be incorporated if present.

Finally, we want to assess how well our models fit the data with ‘model checking’. The effect of adding or removing one variable to the coefficients of the other variables in the model is very important, and will be discussed later. Measures of goodness-of-fit such as the AIC, can also be of great use when deciding which model choice is most valid.

7.3.3 AIC

The Akaike Information Criterion (AIC) is an alternative goodness-of-fit measure. In that sense, it is similar to the R-squared, but it has a different statistical basis. It is useful because it can be used to help guide the best fit in generalised linear models such as logistic regression (see chapter 9). It is based on the likelihood but is also penalised for the number of variables present in the model. We aim to have as small an AIC as possible. The value of the number itself has no inherent meaning, but it is used to compare different models.

7.3.4 Get the data

```
mydata = finalfit::wcgs #F1 here for details
```

7.3.5 Check the data

As always, when you receive a new dataset, carefully check that it does not contain errors.

TABLE 7.1: WCGS data, ff_glimpse: continuous

label	var_type	n	missing_n	mean	sd	median
Subject ID	<int>	3154	0	10477.9	5877.4	11405.5
Age (years)	<int>	3154	0	46.3	5.5	45.0
Height (inches)	<int>	3154	0	69.8	2.5	70.0
Weight (pounds)	<int>	3154	0	170.0	21.1	170.0
Systolic BP (mmHg)	<int>	3154	0	128.6	15.1	126.0
Diastolic BP (mmHg)	<int>	3154	0	82.0	9.7	80.0
Cholesterol (mg/100 ml)	<int>	3142	12	226.4	43.4	223.0
Cigarettes/day	<int>	3154	0	11.6	14.5	0.0
Time to CHD event	<int>	3154	0	2683.9	666.5	2942.0

TABLE 7.2: WCGS data, ff_glimpse: categorical

label	var_type	n	missing_n	levels_n	levels	levels_count
Personality type	<fct>	3154	0	4	"A1", "A2", "B3", "B4"	264, 1325, 1216, 349
Personality type	<fct>	3154	0	2	"B", "A"	1565, 1589
Smoking	<fct>	3154	0	2	"Non-smoker", "Smoker"	1652, 1502
Corneal arcus	<fct>	3152	2	2	"No", "Yes", "(Missing)"	2211, 941, 2
CHD event	<fct>	3154	0	2	"No", "Yes"	2897, 257
Type CHD	<fct>	3154	0	4	"No", "MI_SD", "Silent_MI", "Angina"	2897, 135, 71, 51

7.3.6 Plot the data

```
mydata %>%
  ggplot(aes(y = sbp, x = weight,
             colour = personality_2L)) +
    # Personality type
    geom_point(alpha = 0.2) +
    # Add transparency
    geom_smooth(method = "lm", se = FALSE)
```

From this plot we can see that there is a weak relationship between weight and blood pressure. In addition, there is really no meaningful effect of personality type on blood pressure. This is really important because, as you will see below, we are about to find some highly statistically significant effects in a model.

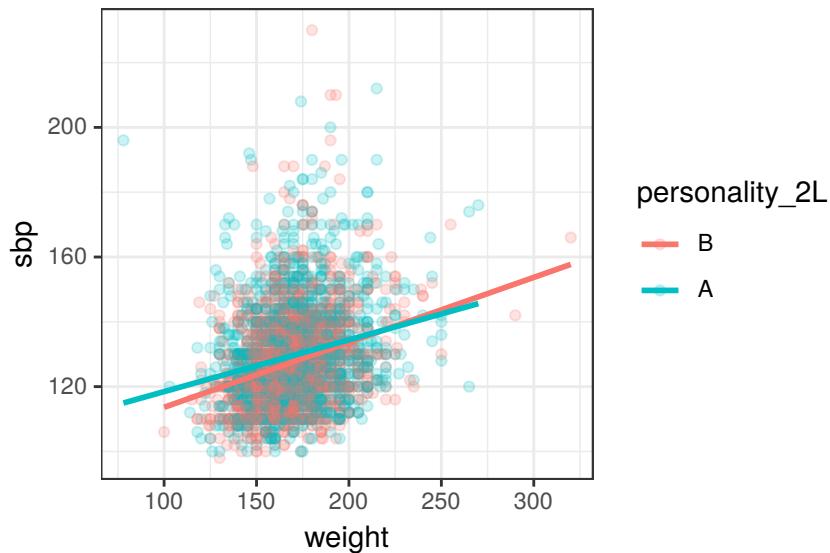


FIGURE 7.13: Scatter and line plot. Systolic blood pressure by weight and personality type.

7.3.7 Linear regression with finalfit

Finalfit is our own package and provides a convenient set of functions for fitting regression models with results presented in final tables.

There are a host of features with example code at the `finalfit` website⁶.

Here we will use the all-in-one `finalfit()` function, which takes a dependent variable and one or more explanatory variables. The appropriate regression for the dependent variable is performed, from a choice of linear, logistic, and Cox Proportional Hazards regression. Summary statistics, together with a univariable and a multivariable regression analysis are produced in a final results table.

⁶<https://finalfit.org>

```
dependent = "sbp"
explanatory = c("personality_2L")
fit_sbpl = mydata %>%
  finalfit(dependent, explanatory, metrics = TRUE)
```

TABLE 7.3: Linear regression: Systolic blood pressure by personality type.

Dependent: Systolic BP (mmHg)	Mean (sd)	Coefficient (univariable)	Coefficient (multivariable)
Personality type	B 127.5 (14.4)	-	-
	A 129.8 (15.7)	2.32 (1.26 to 3.37, p<0.001)	2.32 (1.26 to 3.37, p<0.001)

TABLE 7.4: Model metrics: Systolic blood pressure by personality type.

Number in dataframe = 3154, Number in model = 3154, Missing = 0, Log-likelihood = -13031.39, AIC = 26068.8, R-squared = 0.0059, Adjusted R-squared = 0.0056

Let's look first at our explanatory variable of interest, personality type. When a factor is entered into a regression model, the default is to compare each level of the factor with a “reference level”. This reference level can be easily changed (see section ??). Alternatives methods are available (sometimes called *contrasts*), but the default method is likely to be what you want almost all the time. Note this is sometimes referred to as creating a “dummy variable”.

It can be seen that the mean blood pressure for type A is higher than for type B. As there is only one variable, the univariable and multivariable analyses are the same (the multivariable column can be removed if desired by including `select(-5) #5th column` in the piped function).

Although the difference is numerically quite small (2.3 mmHg), it is statistically significant partly because of the large number of patients in the study. The optional `metrics = TRUE` output gives us the number of rows (in this case subjects) included in the model. This is important as frequently people forget that in standard regression models, missing data from any variable results in the entire row being excluded from the analysis (see chapter 13).

Note the AIC and Adjusted R-squared results. The adjusted R-squared is very low - the model only explains only 0.6% of the variation in systolic blood pressure. This is to be expected, given our scatterplot above.

Let's now include subject weight, which we have hypothesised may influence blood pressure.

```
dependent = "sbp"
explanatory = c("weight", "personality_2L")
fit_sbp2 = mydata %>%
  finalfit(dependent, explanatory, metrics = TRUE)
```

TABLE 7.5: Multivariable linear regression: Systolic blood pressure by personality type and weight.

Dependent: Systolic BP (mmHg)		Mean (sd)	Coefficient (univariable)	Coefficient (multivariable)
Weight (pounds)	[78,320]	128.6 (15.1)	0.18 (0.16 to 0.21, p<0.001)	0.18 (0.16 to 0.20, p<0.001)
Personality type	B	127.5 (14.4)	-	-
	A	129.8 (15.7)	2.32 (1.26 to 3.37, p<0.001)	1.99 (0.97 to 3.01, p<0.001)

TABLE 7.6: Multivariable linear regression metrics: Systolic blood pressure by personality type and weight.

Number in dataframe = 3154, Number in model = 3154, Missing = 0, Log-likelihood = -12928.82, AIC = 25865.6, R-squared = 0.068, Adjusted R-squared = 0.068

The output shows us the range for weight (78 to 320 pounds) and the mean (standard deviation) systolic blood pressure for the whole cohort.

The coefficient and 95% confidence interval are provided by default. This is interpreted as: for each pound increase in weight, there is on average a corresponding increase of 0.18 mmHg in systolic blood pressure.

Note the difference in the interpretation of continuous and categorical variables in the regression model output (Figure 7.6).

The adjusted R-squared is now higher - the personality and weight together explain 6.8% of the variation in blood pressure.

The AIC is also slightly lower meaning this new model better fits the data.

There is little change in the size of the coefficients for each variable in the multivariable analysis, meaning that they are reasonably independent. As an exercise, check the the distribution of weight by personality type using a boxplot.

Let's now add in other variables that may influence systolic blood pressure.

```
dependent = "sbp"
explanatory = c("personality_2L", "weight", "age",
               "height", "chol", "smoking")
fit_sbp3 = mydata %>%
  finalfit(dependent, explanatory, metrics = TRUE)
```

TABLE 7.7: Multivariable linear regression: Systolic blood pressure by available explanatory variables.

Dependent: Systolic BP (mmHg)		Mean (sd)	Coefficient (univariable)	Coefficient (multivariable)
Personality type	B	127.5 (14.4)	-	-
	A	129.8 (15.7)	2.32 (1.26 to 3.37, p<0.001)	1.44 (0.44 to 2.43, p=0.005)
Weight (pounds)	[78,320]	128.6 (15.1)	0.18 (0.16 to 0.21, p<0.001)	0.24 (0.21 to 0.27, p<0.001)
Age (years)	[39,59]	128.6 (15.1)	0.45 (0.36 to 0.55, p<0.001)	0.43 (0.33 to 0.52, p<0.001)
Height (inches)	[60,78]	128.6 (15.1)	0.11 (-0.10 to 0.32, p=0.302)	-0.84 (-1.08 to -0.61, p<0.001)
Cholesterol (mg/100 ml)	[103,645]	128.6 (15.1)	0.04 (0.03 to 0.05, p<0.001)	0.03 (0.02 to 0.04, p<0.001)
Smoking	Non-smoker	128.6 (15.6)	-	-
	Smoker	128.7 (14.6)	0.08 (-0.98 to 1.14, p=0.883)	0.95 (-0.05 to 1.96, p=0.063)

TABLE 7.8: Model metrics: Systolic blood pressure by available explanatory variables.

Number in dataframe = 3154, Number in model = 3142, Missing = 12, Log-likelihood = -12772.04, AIC = 25560.1, R-squared = 0.12, Adjusted R-squared = 0.12

Age, height, serum cholesterol, and smoking status have been added. Some of the variation explained by personality type has been taken up by these new variables - personality is now associated with an average change of blood pressure of 1.4 mmHg.

The adjusted R-squared now tells us that 12% of the variation in blood pressure is explained by the model, which is an improvement.

Look out for variables that show large changes in effect size or a change in the direction of effect when going from a univariable to multivariable model. This means that the other variables in the model are having a large effect on this variable and the cause of this should be explored. For instance, in this example the effect of height changes size and direction. This is because of the close association between weight and height. For instance, it may be more sensible to work with body mass index ($weight/height^2$) rather than the two separate variables. This can be created easily.

In general, variables that are highly correlated with each other should be treated carefully in regression analysis. This is called collinearity and can lead to unstable estimates of coefficients. For more on this, see section 9.4.2.

```
mydata = mydata %>%
  mutate(
    bmi = ((weight*0.4536) / (height*0.0254)^2) %>%
      ff_label("BMI")
  )
```

Weight and height can be replaced in the model with BMI.

```
explanatory = c("personality_2L", "bmi", "age",
               "chol", "smoking")

fit_sbp4 = mydata %>%
  finalfit(dependent, explanatory, metrics = TRUE)
```

TABLE 7.9: Multivariable linear regression: Systolic blood pressure using BMI.

Dependent: Systolic BP (mmHg)		Mean (sd)	Coefficient (univariable)	Coefficient (multivariable)
Personality type	B	127.5 (14.4)	-	-
	A	129.8 (15.7)	2.32 (1.26 to 3.37, p<0.001)	1.51 (0.51 to 2.50, p=0.003)
BMI	[11.2,39]	128.6 (15.1)	1.69 (1.50 to 1.89, p<0.001)	1.65 (1.46 to 1.85, p<0.001)
Age (years)	[39,59]	128.6 (15.1)	0.45 (0.36 to 0.55, p<0.001)	0.41 (0.32 to 0.50, p<0.001)
Cholesterol (mg/100 ml)	[103,645]	128.6 (15.1)	0.04 (0.03 to 0.05, p<0.001)	0.03 (0.02 to 0.04, p<0.001)
Smoking	Non-smoker	128.6 (15.6)	-	-
	Smoker	128.7 (14.6)	0.08 (-0.98 to 1.14, p=0.883)	0.98 (-0.03 to 1.98, p=0.057)

On the principle of parsimony, we may want to remove variables

TABLE 7.10: Model metrics: Systolic blood pressure using BMI.

Number in dataframe = 3154, Number in model = 3142, Missing = 12, Log-likelihood = -12775.03, AIC = 25564.1, R-squared = 0.12, Adjusted R-squared = 0.12

which are not contributing much to the model. For instance, let's compare models with and without the inclusion of smoking. This can be easily done using the `finalfit explanatory_multi` option.

```
dependent = "sbp"
explanatory = c("personality_2L", "bmi", "age",
               "chol", "smoking")
explanatory_multi = c("bmi", "personality_2L", "age",
                      "chol")
fit_sbp5 = mydata %>%
  finalfit(dependent, explanatory,
           explanatory_multi,
           keep_models = TRUE, metrics = TRUE)
```

TABLE 7.11: Multivariable linear regression: Systolic blood pressure by available explanatory variables and reduced model.

Dependent: Systolic BP (mmHg)		Mean (sd)	Coefficient (univariable)	Coefficient (multivariable)	Coefficient (multivariable reduced)
Personality type	B	127.5 (14.4)	-	-	-
	A	129.8 (15.7)	2.32 (1.26 to 3.37, p<0.001)	1.51 (0.51 to 2.50, p=0.003)	1.56 (0.57 to 2.56, p=0.002)
BMI	[11.2,39]	128.6 (15.1)	1.69 (1.50 to 1.89, p<0.001)	1.65 (1.46 to 1.85, p<0.001)	1.62 (1.43 to 1.82, p<0.001)
Age (years)	[39,59]	128.6 (15.1)	0.45 (0.36 to 0.55, p<0.001)	0.41 (0.32 to 0.50, p<0.001)	0.41 (0.32 to 0.50, p<0.001)
Cholesterol (mg/100 ml)	[103,645]	128.6 (15.1)	0.04 (0.03 to 0.05, p<0.001)	0.03 (0.02 to 0.04, p<0.001)	0.03 (0.02 to 0.04, p<0.001)
Smoking	Non-smoker	128.6 (15.6)	-	-	-
	Smoker	128.7 (14.6)	0.08 (-0.98 to 1.14, p=0.883)	0.98 (-0.03 to 1.98, p=0.057)	-

TABLE 7.12: Model metrics: Systolic blood pressure by available explanatory variables (top) with reduced model (bottom).

Number in dataframe = 3154, Number in model = 3142, Missing = 12, Log-likelihood = -12775.03, AIC = 25564.1, R-squared = 0.12, Adjusted R-squared = 0.12
 Number in dataframe = 3154, Number in model = 3142, Missing = 12, Log-likelihood = -12776.83, AIC = 25565.7, R-squared = 0.12, Adjusted R-squared = 0.12

This results in little change in the other coefficients and very little change in the AIC. We will consider the reduced model the final model.

We can check the assumptions as above.

```
dependent = "sbp"
explanatory_multi = c("bmi", "personality_2L", "age",
                      "chol")
mydata %>%
  lmmulti(dependent, explanatory_multi) %>%
  autoplot()
```

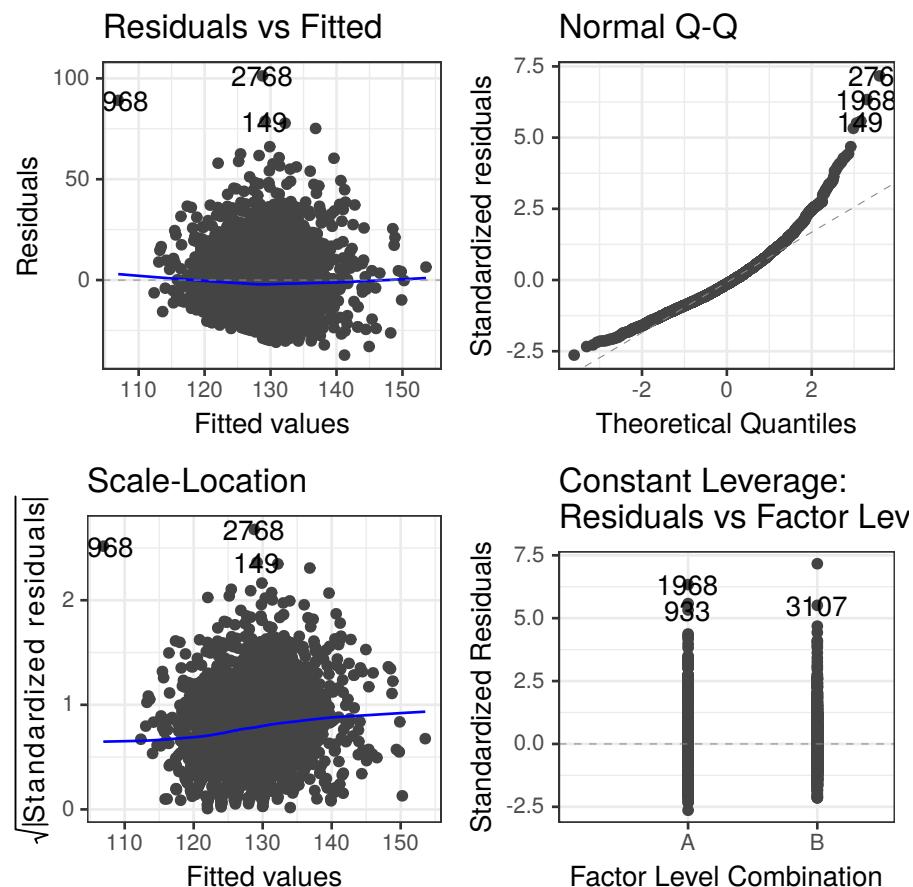


FIGURE 7.14: Diagnostic plots: linear regression model of systolic blood pressure.

An important message in the results relates to the highly significant p -values in the table above. Should we conclude that in a "multivariable regression model controlling for BMI, age, and serum

cholesterol, blood pressure was significantly elevated in those with a Type A personality (1.56 (0.57 to 2.56, $p=0.002$) compared with Type B? The p -value looks impressive, but the actual difference in blood pressure is only 1.6 mmHg. Even at a population level, that seems unlikely to be clinically significant. Which fits with our first thoughts when we saw the scatter plot.

This serves to emphasise our most important point. Our focus should be on understanding the underlying data itself, rather than relying on complex multidimensional modelling procedures. By making liberal use of upfront plotting, together with further visualisation as you understand the data, you will likely be able to draw most of the important conclusions that the data has to offer. Use modelling to quantify and confirm this, rather than the primary method of data exploration.

7.3.8 Summary

Time spent truly understanding linear regression is well spent. Not because you will spend a lot of time making linear regression models in health data science (we rarely do), but because it the essential foundation for understanding more advanced statistical models.

It can even be argued that all common statistical tests are linear models⁷. This great post demonstrates beautifully how the statistical tests we are most familiar with (such as t-test, Mann-Whitney U test, ANOVA, chi-squared test) can simply be considered as special cases of linear models, or a close approximations.

Regression is fitting lines, preferably straight, through data points. Make $\hat{y} = \beta_0 + \beta_1 x_1$ a close friend.

⁷<https://lindeloev.github.io/tests-as-linear>

8

Working with categorical outcome variables

Suddenly Christopher Robin began to tell Pooh about some of the things: People called Kings and Queens and something called Factors ... and Pooh said "Oh!" and thought how wonderful it would be to have a Real Brain which could tell you things.

A.A. Milne, *The House at Pooh Corner* (1928)

8.1 Factors

We said earlier that continuous data can be measured and categorical data can be counted, which is useful to remember. Categorical data can be a:

- Factor
 - a fixed set of names/strings or numbers
 - these may have an inherent order (1st, 2nd 3rd) - ordinal factor
 - or may not (female, male)
- Character
 - sequences of letters, numbers, and symbols
- Logical
 - containing only TRUE or FALSE

Health data is awash with factors. Whether it is outcomes like death, recurrence, or readmission. Or predictors like cancer stage, deprivation quintile, smoker yes/no. It is essential therefore to

be comfortable manipulating factors and dealing with outcomes which are categorical.

8.2 The Question

We will use the classic “Survival from Malignant Melanoma” dataset which is included in the `boot` package. The data consist of measurements made on patients with malignant melanoma, a type of skin cancer. Each patient had their tumour removed by surgery at the Department of Plastic Surgery, University Hospital of Odense, Denmark between 1962 and 1977.

We are interested in the association between tumour ulceration and death from melanoma.

8.3 Get the data

The help page (`?boot::melanoma`) gives us the data dictionary. This includes the definition of each variable and the coding used.

```
mydata = boot::melanoma
```

8.4 Check the data

As always, check any new dataset carefully before you start analysis.

```

library(tidyverse)
library(finalfit)
mydata %>% glimpse()

## Observations: 205
## Variables: 7
## $ time      <dbl> 10, 30, 35, 99, 185, 204, 210, 232, 232, 279, 295, 3...
## $ status     <dbl> 3, 3, 2, 3, 1, 1, 1, 3, 1, 1, 1, 1, 3, 1, 1, 1, 1...
## $ sex        <dbl> 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1...
## $ age         <dbl> 76, 56, 41, 71, 52, 28, 77, 60, 49, 68, 53, 64, 68, ...
## $ year        <dbl> 1972, 1968, 1977, 1968, 1965, 1971, 1972, 1974, 1968...
## $ thickness   <dbl> 6.76, 0.65, 1.34, 2.90, 12.08, 4.84, 5.16, 3.22, 12....
## $ ulcer       <dbl> 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...

mydata %>% ff_glimpse()

## Continuous
##           label var_type  n missing_n missing_percent    mean      sd
## time          time     <dbl> 205      0             0.0 2152.8 1122.1
## status        status    <dbl> 205      0             0.0   1.8   0.6
## sex           sex      <dbl> 205      0             0.0   0.4   0.5
## age            age     <dbl> 205      0             0.0   52.5  16.7
## year          year     <dbl> 205      0             0.0 1969.9   2.6
## thickness    thickness <dbl> 205      0             0.0   2.9   3.0
## ulcer         ulcer    <dbl> 205      0             0.0   0.4   0.5
##               min quartile_25 median quartile_75      max
## time          10.0    1525.0  2005.0      3042.0 5565.0
## status        1.0     1.0     2.0       2.0     3.0
## sex           0.0     0.0     0.0       1.0     1.0
## age           4.0     42.0    54.0      65.0    95.0
## year          1962.0  1968.0  1970.0      1972.0 1977.0
## thickness    0.1     1.0     1.9       3.6    17.4
## ulcer         0.0     0.0     0.0       1.0     1.0
##
## Categorical
## data frame with 0 columns and 205 rows

```

As can be seen, all of the variables are currently coded as continuous/numeric. The `<dbl>` stands for ‘double’, meaning numeric which comes from ‘double-precision floating point’, an awkward computer science term.

8.5 Recode the data

It is really important that variables are correctly coded for all plotting and analysis functions. Using the data dictionary, we will convert the categorical variables to factors.

In the section below, we convert the continuous variables to `factors`, then use the `forcats` package to recode the factor levels.

```
library(forcats)
mydata = mydata %>%
  mutate(sex.factor =
    sex %>%
    factor() %>%
    fct_recode(
      "Female" = "0",
      "Male" = "1") %>%
    ff_label("Sex"),           # Label for finalfit tables

    ulcer.factor =
    ulcer %>%
    factor() %>%
    fct_recode(
      "Present" = "1",
      "Absent" = "0") %>%
    ff_label("Ulcerated tumour"),

    status.factor =
    status %>%
    factor() %>%
    fct_recode("Died melanoma" = "1",
              "Alive" = "2",
              "Died - other causes" = "3") %>%
    ff_label("Status"))
```

8.6 Should I convert a continuous variable to a categorical variable?

This is a common question and something which is frequently done. Take for instance the variable years. Is it better to leave it as a continuous variable, or to chop it into categories, e.g. 30 to 39 etc.?

The clear disadvantage in doing this is that information is being thrown away. Which feels like a bad thing to be doing. This is particularly important if categories being created are large. For instance, if age was dichotomised to “young” and “old” at say 42 years (the current median age in Europe), then it is likely that information which is likely to be relevant to a number of analyses has been discarded. Secondly, it is unforgivable practice to repeatedly cut a continuous variable in different ways in order to obtain a statistically significant result. This is most commonly done in tests of diagnostic accuracy, where a threshold for considering a continuous test result positive is chosen *post hoc* to maximise sensitivity/specificity, but not then validated in an independent cohort.

But there are also advantages. Say the relationship between age and an outcome is not linear, but rather u-shaped, then fitting a regression line is more difficult. While if the age has been cut into 10 year bands and entered into a regression as a factor, then the non-linearity is already accounted for. Secondly, sometimes when communicating the results of an analysis to a lay audience, then using a rhetorical representation can make this easier. For instance, an odds of death 1.8 times greater in 70 year olds compared with 40 year olds may be easier to grasp than 1.02 times per year.

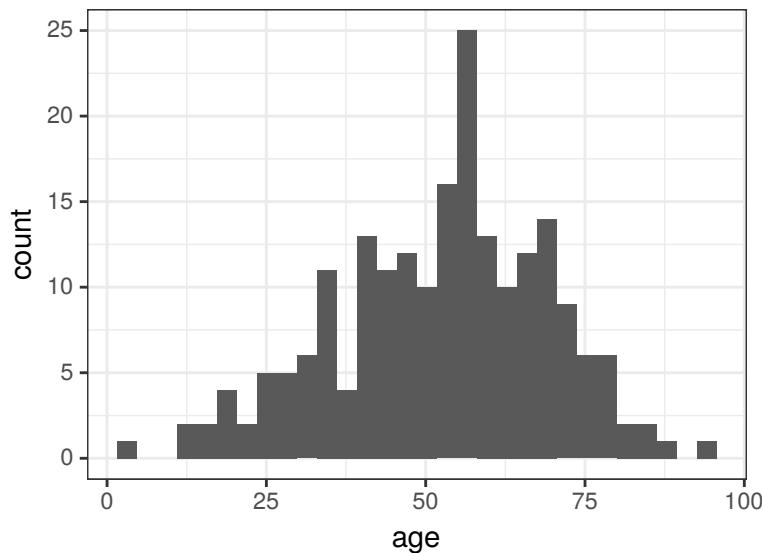
So the answer. Do not do it unless you have to. Plot and understand the continuous variable first. If you do it, try not to throw away too much information.

```
# Summary of age
mydata$age %>%
  summary()
```

```
##      Min. 1st Qu. Median     Mean 3rd Qu.     Max.
##      4.00   42.00  54.00  52.46  65.00  95.00
```

```
mydata %>%
  ggplot(aes(x = age)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



There are different ways in which a continuous variable can be converted to a factor. You may wish to create a number of intervals of equal length. The `cut()` function can be used for this.

Figure 8.1 illustrates different options for this. We suggest not using the `label` option in the function, to avoid errors should the underlying data change or when the code is copied and reused. A better practice is to recode the levels using `fct_recode` as above.

The intervals in the output are standard mathematical notation. A square bracket indicates the value is included in the interval and a round bracket that the value is excluded.

Note the requirement for `include.lowest = TRUE` when you specify breaks yourself and the the lowest cut-point is also the lowest data value. This should be clear in Figure 8.1.

```

x = 1:9
cut(x, breaks = 3)
Levels: (0.991,4] (4,7] (7,10]
      1   2   3   4   5   6   7   8   9   10
      (0.991,4] (0.991,4] (0.991,4] (0.991,4] (4,7] (4,7] (4,7] (7,10] (7,10] (7,10]

cut(x, breaks = 3, right = FALSE)
Levels: [0.991,4) [4,7) [7,10)
      1   2   3   4   5   6   7   8   9   10
      [0.991,4) [0.991,4) [0.991,4) [4,7) [4,7) [4,7) [7,10) [7,10) [7,10) [7,10)

cut(x, breaks = c(1, 4, 7, 10))
Levels: (1,4] (4,7] (7,10]
      1   2   3   4   5   6   7   8   9   10
      <NA> (1,4] (1,4] (1,4] (4,7] (4,7] (4,7] (7,10] (7,10] (7,10] (7,10]

cut(x, breaks = c(1, 4, 7, 10), include.lowest = TRUE)
Levels: [1,4] (4,7] (7,10]
      1   2   3   4   5   6   7   8   9   10
      [1,4] [1,4] [1,4] [1,4] (4,7] (4,7] (4,7] (7,10] (7,10] (7,10]

```

FIGURE 8.1: `cut` a continuous variable into a categorical variable.

8.6.1 Equal intervals vs quantiles

Be clear in your head whether you wish to cut the data so the intervals are of equal length. Or whether you wish to cut the data so there are equal proportions of cases (patients) in each level.

Equal intervals:

```

mydata = mydata %>%
  mutate(
    age.factor =
      age %>%
      cut(4)
  )
mydata$age.factor %>%
  summary()

## # (3.91,26.8] (26.8,49.5] (49.5,72.2] (72.2,95.1]
## #           16            68           102            19

```

Quantiles:

```
mydata = mydata %>%
  mutate(
    age.factor =
      age %>%
      Hmisc:::cut2(g=4)
  )
mydata$age.factor %>%
  summary()

## [ 4,43) [43,55) [55,66) [66,95]
##      55         49         53         48

# Or?
# mydata$age.factor %>%
#   fct_count()
```

Using the cut function, a continuous variable can be converted

```
mydata = mydata %>%
  mutate(
    age.factor =
      age %>%
      cut(breaks = c(4,20,40,60,95), include.lowest = TRUE) %>%
      fct_recode(
        "<=20" = "[4,20]",
        "21 to 40" = "(20,40]",
        "41 to 60" = "(40,60]",
        ">60" = "(60,95]"
      ) %>%
      ff_label("Age (years)")
  )
head(mydata$age.factor)

## [1] >60      41 to 60 41 to 60 >60      41 to 60 21 to 40
## Levels: <=20 21 to 40 41 to 60 >60
```

8.7 Plot the data

We are interested in the association between tumour ulceration and death from melanoma. To start then, we simply count the number of patients with ulcerated tumours who died. It is useful to plot this as counts but also as proportions. It is proportions you are comparing, but you really want to know the absolute numbers as well.

```
p1 = mydata %>%
  ggplot(aes(x = ulcer.factor, fill=status.factor)) +
  geom_bar() +
  theme(legend.position = "none")

p2 = mydata %>%
  ggplot(aes(x = ulcer.factor, fill=status.factor)) +
  geom_bar(position = "fill") +
  ylab("proportion")

library(patchwork)
p1 + p2
```

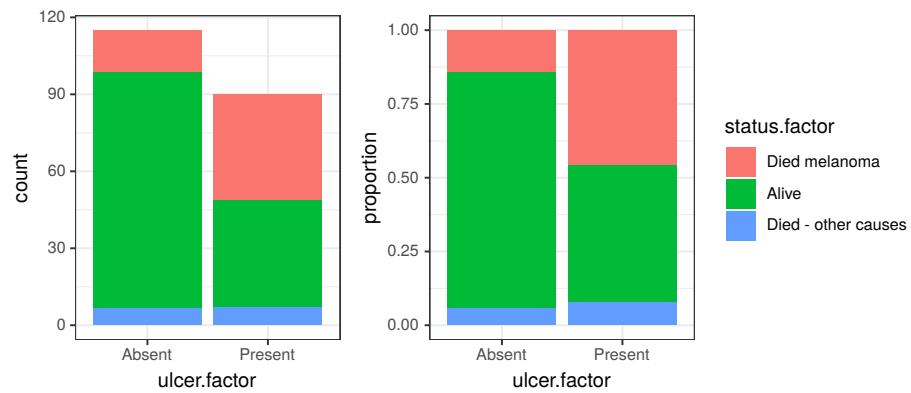


FIGURE 8.2: Bar chart: outcome after surgery for patients with ulcerated melanoma.

It should be obvious that more died from melanoma in the ulcerated tumour group compared with the non-ulcerated tumour group. The stacking is orders from top to bottom by default. This

can be easily adjusted by changing the order of the levels within the factor (see re-levelling below). This default order works well for binary variables - the “yes” or “1” is lowest and can be easily compared. This ordering of this particular variable is unusual - it would be more common to have for instance `alive = 0, died = 1`. One quick option is to just reverse the order of the levels in the plot.

```
p1 = mydata %>%
  ggplot(aes(x = ulcer.factor, fill=status.factor)) +
  geom_bar(position = position_stack(reverse = TRUE)) +
  theme(legend.position = "none")

p2 = mydata %>%
  ggplot(aes(x = ulcer.factor, fill=status.factor)) +
  geom_bar(position = position_fill(reverse = TRUE)) +
  ylab("proportion")

library(patchwork)
p1 + p2
```

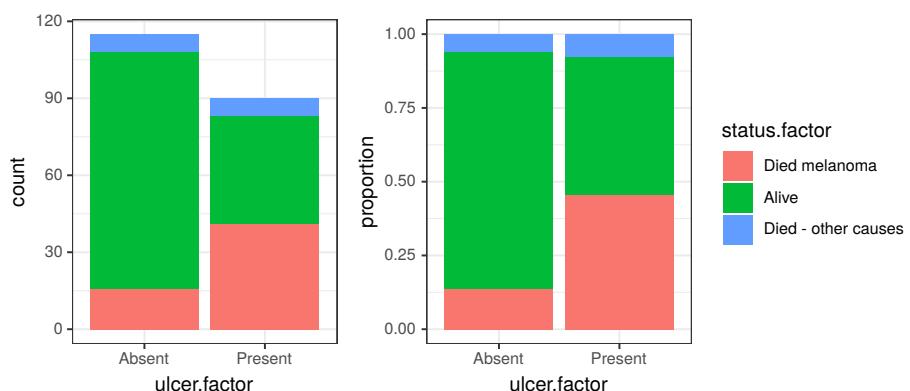


FIGURE 8.3: Bar chart: outcome after surgery for patients with ulcerated melanoma, reversed levels.

Just from the plot then, death from melanoma in the ulcerated tumour group is around 40% and in the non-ulcerated group around 13%. The number of patients included in the study is not huge, however, this still looks like a real difference given its size.

We also may be interested in exploring potential effect modification, interactions and confounders. Again, we urge you to first visualise these, rather than going straight to a model.

```

p1 = mydata %>%
  ggplot(aes(x = ulcer.factor, fill=status.factor)) +
  geom_bar(position = position_stack(reverse = TRUE)) +
  facet_grid(sex.factor ~ age.factor) +
  theme(legend.position = "none")

p2 = mydata %>%
  ggplot(aes(x = ulcer.factor, fill=status.factor)) +
  geom_bar(position = position_fill(reverse = TRUE)) +
  facet_grid(sex.factor ~ age.factor) +
  theme(legend.position = "bottom")

p1 / p2

```

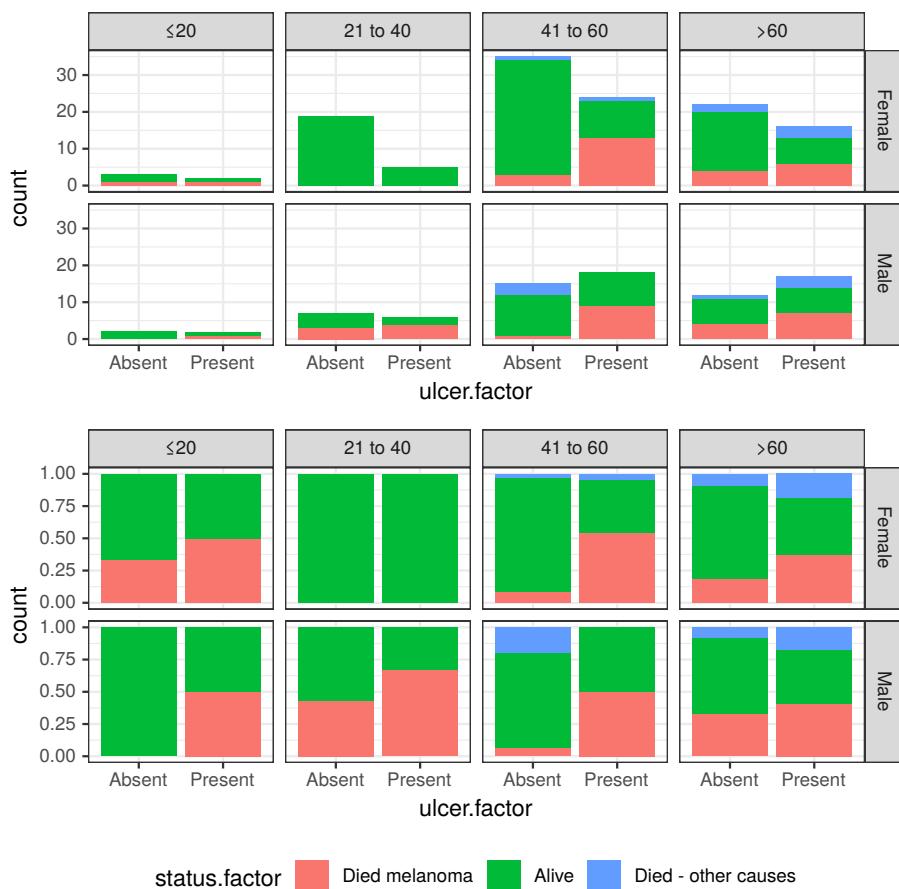


FIGURE 8.4: Facetted bar plot: outcome after surgery for patients with ulcerated melanoma aggregated by sex and age.

8.8 Group factor levels together - `fct_collapse()`

Our question relates to the association between tumour ulceration and death from melanoma. The outcome measure has three levels as can be seen. There are a number of ways of approaching this, which are discussed in detail in chapter 10 (REF). For our purposes here, we will generate a disease-specific mortality variable, by combining “Died - other causes” and “Alive”.

```
mydata = mydata %>%
  mutate(
    status_dss = fct_collapse(
      status.factor,
      "Alive" = c("Alive", "Died - other causes"))
  )
```

8.9 Change the order of values within a factor - `fct_relevel()`

The default order for levels with `factor()` is alphabetical. We often want to reorder the levels in a factor when plotting, or when performing an regression analysis and we want to specify the reference level.

The order can be checked using `levels()`.

```
mydata$status_dss %>% levels()
```

```
## [1] "Died melanoma" "Alive"
```

The reason “Alive” is second, rather than alphabetical, is it was recoded from “2” and that order was retained. If, however, we want to make comparisons relative to “Alive”, we need to move it to the front by using `fct_relevel()`.

```
mydata = mydata %>%
  mutate(status_dss = status_dss %>%
    fct_relevel("Alive"))
)
```

Any number of factor levels can be specified in `fct_relevel()`.

8.10 Summarising factors with Finalfit

Our own `Finalfit` package provides convenient functions to summarise and compare factors, producing final tables for publication.

```
library(finalfit)
mydata %>%
  summary_factorlist(dependent = "status_dss",
                     explanatory = "ulcer.factor")
```

TABLE 8.1: Two-by-two table with Finalfit: Died with melanoma by tumour ulceration status.

label	levels	Alive	Died melanoma
Ulcerated tumour	Absent	99 (86.1)	16 (13.9)
	Present	49 (54.4)	41 (45.6)

`Finalfit` is useful for summarising multiple variables. We often want to summarise more than one factor or continuous variable against our `dependent` variable of interest. Think of Table 1 in a journal article.

Any number of continuous or categorical explanatory variables can be added.

```
library(finalfit)
mydata %>%
  summary_factorlist(dependent = "status_dss",
```

```

explanatory =
  c("ulcer.factor", "age.factor",
    "sex.factor", "thickness")
)
## Warning in chisq.test(tab, correct = FALSE): Chi-squared approximation may
## be incorrect

```

TABLE 8.2: Multiple variables by outcome: Outcome after surgery for melanoma by patient and disease factors.

label	levels	Alive	Died melanoma
Ulcerated tumour	Absent	99 (86.1)	16 (13.9)
	Present	49 (54.4)	41 (45.6)
Age (years)	20	6 (66.7)	3 (33.3)
	21 to 40	30 (81.1)	7 (18.9)
	41 to 60	66 (71.7)	26 (28.3)
	>60	46 (68.7)	21 (31.3)
Sex	Female	98 (77.8)	28 (22.2)
	Male	50 (63.3)	29 (36.7)
thickness	Mean (SD)	2.4 (2.5)	4.3 (3.6)

8.11 Pearson's chi-squared and Fisher's exact tests

Pearson's chi-squared (χ^2) test of independence is used to determine whether two categorical variables are independent in a given population. Independence here means that the relative frequencies of one variable are the same over all levels of another variable.

A common setting for this is the classic 2x2 table. This refers to two categorical variables each with two levels, such as is shown in Table REF above. The null hypothesis of independence for this particular question is no difference in the proportion of patients with ulcerated tumours who die (45.6%) compared with non-ulcerated

tumours (13.9%). From the raw frequencies, there seems to be a large difference, as we noted in the plot we made above.

8.11.1 Base R

Base R has reliable functions for all common statistical tests, but they are sometimes a little inconvenient to perform or extract results from.

A table of counts can be constructed, either using the `$` to identify columns, or using the `with()` function.

```
table(mydata$ulcer.factor, mydata$status_dss) # both give same result
with(mydata, table(ulcer.factor, status_dss))

##
##          Alive Died melanoma
##  Absent      99      16
##  Present     49      41
```

When working with older R functions, a useful shortcut is the exposition pipe-operator (`%$%`) from the `magrittr` package, home of the standard forward pipe-operator (`%>%`). The exposition pipe-operator exposes data frame/tibble columns on the left to the function which follows on the right. It's easier to see in action by making a table of counts.

```
library(magrittr)
mydata %$%           # note $ sign here
  table(ulcer.factor, status_dss) # No dollar, no data = .

##
##          status_dss
## ulcer.factor Alive Died melanoma
##    Absent      99      16
##    Present     49      41
```

The counts table can be passed to `prop.table()` for a proportions.

```
mydata %$%
  table(ulcer.factor, status_dss) %>%
  prop.table(margin = 1)      # 1: row, 2: column etc.
```

```
##           status_dss
## ulcer.factor   Alive Died melanoma
##     Absent  0.8608696  0.1391304
##     Present 0.5444444  0.4555556
```

Similarly, the counts table can be passed to `chisq.test()` to perform the chi-squared test.

```
mydata %$%      # note $ sign here
  table(ulcer.factor, status_dss) %>%
    chisq.test()
```

```
##
## Pearson's Chi-squared test with Yates' continuity correction
##
## data: .
## X-squared = 23.631, df = 1, p-value = 1.167e-06
```

The result can be extracted into a tibble using the `tidy()` function from the `broom` package.

```
library(broom)
mydata %$%      # note $ sign here
  table(ulcer.factor, status_dss) %>%
    chisq.test() %>%
    tidy()
```

```
## # A tibble: 1 x 4
##   statistic   p.value parameter method
##       <dbl>     <dbl>     <int> <chr>
## 1     23.6 0.00000117     1 Pearson's Chi-squared test with Yates' co~
```

The base R default statistic uses the Yates' continuity correction. The standard interpretation assumes that the discrete probability of observed counts in the table can be approximated by the continuous chi-squared distribution. This introduces some error. A suggested correction involves subtracting 0.5 from the absolute difference between each observed and expected value. This is particularly helpful when counts are low. This can be removed if desired, `chisq.test(..., correct = FALSE)`.

8.12 Fisher's exact test

A commonly stated assumption of the chi-squared test is the requirement to have an expected count in each table cell of at least 5 in a 2x2 table. For larger tables, all expected counts should be > 1 and no more than 20% of all cells should have expected counts < 5 . If this assumption is not fulfilled, an alternative test is Fisher's exact test. For instance, if we are testing across a 2x4 table created from our `age.factor` variable and `status_dss`, then we receive a warning.

```
mydata %$%      # note $ sign here
  table(age.factor, status_dss) %>%
    chisq.test()
```

```
## Warning in chisq.test(.): Chi-squared approximation may be incorrect
##
## Pearson's Chi-squared test
##
## data: .
## X-squared = 2.0198, df = 3, p-value = 0.5683
```

Switch to Fisher's exact test

```
mydata %$%      # note $ sign here
  table(age.factor, status_dss) %>%
    fisher.test()
```

```
##
## Fisher's Exact Test for Count Data
##
## data: .
## p-value = 0.5437
## alternative hypothesis: two.sided
```

8.13 Chi-squared / Fisher's exact test using Finalfit

It is easier using Finalfit. Including `p = TRUE` in `summary_factorlist()` adds a hypothesis test to each included comparison. This defaults to chi-squared tests with no continuity correct for categorical variables and a Kruskal-Wallis non-parametric test to continuous variables.

```
library(finalfit)
mydata %>%
  summary_factorlist(dependent = "status_dss",
                     explanatory = "ulcer.factor",
                     p = TRUE)
```

TABLE 8.3: Two-by-two table with chi-squared test using final fit: Outcome after surgery for melanoma by tumour ulceration status.

label	levels	Alive	Died melanoma	p
Ulcerated tumour	Absent	99 (86.1)	16 (13.9)	<0.001
	Present	49 (54.4)	41 (45.6)	

Add further variables.

```
mydata %>%
  summary_factorlist(dependent = "status_dss",
                     explanatory =
                       c("ulcer.factor", "age.factor",
                         "sex.factor", "thickness"),
                     p = TRUE)

## Warning in chisq.test(tab, correct = FALSE): Chi-squared approximation may
## be incorrect
```

Switch to Fisher's exact test.

```
mydata %>%
  summary_factorlist(dependent = "status_dss",
                     explanatory =
```

TABLE 8.4: Multiple variables by outcome with hypothesis tests: Outcome after surgery for melanoma by patient and disease factors (chi-squared test).

label	levels	Alive	Died	melanoma	p
Ulcerated tumour	Absent	99 (86.1)	16 (13.9)	<0.001	
	Present	49 (54.4)	41 (45.6)		
Age (years)	20	6 (66.7)	3 (33.3)	0.568	
	21 to 40	30 (81.1)	7 (18.9)		
	41 to 60	66 (71.7)	26 (28.3)		
Sex	>60	46 (68.7)	21 (31.3)		0.024
	Female	98 (77.8)	28 (22.2)		
	Male	50 (63.3)	29 (36.7)		
thickness	Mean (SD)	2.4 (2.5)	4.3 (3.6)	<0.001	

```
c("ulcer.factor", "age.factor",
  "sex.factor", "thickness"),
p = TRUE,
catTest = catTestfisher)
```

```
mydata %>%
  summary_factorlist(dependent = "status_dss",
                     explanatory =
                       c("ulcer.factor", "age.factor",
                         "sex.factor", "thickness"),
                     p = TRUE,
                     catTest = catTestfisher) %>%
mykable(caption = "Multiple variables by outcome with hypothesis tests: Outcome after surgery f")
```

Other options can be included.

```
mydata %>%
  summary_factorlist(dependent = "status_dss",
                     explanatory =
                       c("ulcer.factor", "age.factor",
                         "sex.factor", "thickness"),
                     p = TRUE,
                     catTest = catTestfisher,
                     digits =
                       c(1, 1, 4, 2), #1: mean/median, 2: SD/IQR
                         # 3: p-value, 4: count percentage
                     na_include = TRUE, # include missing in table and tests
```

TABLE 8.5: Multiple variables by outcome with hypothesis tests: Outcome after surgery for melanoma by patient and disease factors (Fisher's exact test).

label	levels	Alive	Died	melanoma	p
Ulcerated tumour	Absent	99 (86.1)	16 (13.9)	<0.001	
	Present	49 (54.4)	41 (45.6)		
Age (years)	20	6 (66.7)	3 (33.3)	0.544	
	21 to 40	30 (81.1)	7 (18.9)		
	41 to 60	66 (71.7)	26 (28.3)		
	>60	46 (68.7)	21 (31.3)		
Sex	Female	98 (77.8)	28 (22.2)	0.026	
	Male	50 (63.3)	29 (36.7)		
thickness	Mean (SD)	2.4 (2.5)	4.3 (3.6)	<0.001	

```
add_dependent_label = TRUE
)
```

TABLE 8.6: Multiple variables by outcome with hypothesis tests: options including missing data, rounding, and labels.

Dependent: Status		Alive	Died	melanoma	p
Ulcerated tumour	Absent	99 (86.09)	16 (13.91)	<0.0001	
	Present	49 (54.44)	41 (45.56)		
Age (years)	20	6 (66.67)	3 (33.33)	0.5437	
	21 to 40	30 (81.08)	7 (18.92)		
	41 to 60	66 (71.74)	26 (28.26)		
	>60	46 (68.66)	21 (31.34)		
Sex	Female	98 (77.78)	28 (22.22)	0.0263	
	Male	50 (63.29)	29 (36.71)		
thickness	Mean (SD)	2.4 (2.5)	4.3 (3.6)	<0.0001	

8.14 Exercise

Using `finalfit` create a summary table with “status.factor” as the dependent variable and the following as explanatory variables:

- `sex.factor`
- `ulcer.factor`
- `age.factor`
- `thickness`

Try changing the table to show `median` and `interquartile range` instead of mean and sd.

8.15 Exercise

Better for this to be in chapter 3 By changing one and only one line in the following block create firstly a new table showing the breakdown of `status.factor` by age and secondly the breakdown of `status.factor` by sex:

```
mydata %>%
  count(ulcer.factor, status.factor) %>%
  group_by(status.factor) %>%
  mutate(total = sum(n)) %>%
  mutate(percentage = round(100*n/total, 1)) %>%
  mutate(count_perc = paste0(n, " (", percentage, "%)")) %>%
  select(-total, -n, -percentage) %>%
  spread(status.factor, count_perc)
```

8.16 Exercise

Now produce these tables using the `summary_factorlist` function from the `library(finalfit)` package.

8.17 Exercise

Reproduce the plot from 6.1 but this time with row-wise percentages instead of col(column)-wise percentages.

9

Logistic regression

All generalizations are false, including this one.
Mark Twain

9.1 Generalised linear modelling

Do not start here! The material covered in this chapter is best understood after having read linear regression (REF Chapter 7) and working with categorical outcome variables (chapter 8).

Generalised linear modelling is an extension to the linear modelling we are now familiar with. It allows the principles of linear regression to be applied when outcomes are not continuous numeric variables.

9.2 Binary logistic regression

A regression analysis is a statistical approach to estimating the relationships between variables, often by drawing straight lines through data points. For instance, we may try to predict blood pressure in a group of patients based on their coffee consumption

(REF Chap 7 PLOT). As blood pressure and coffee consumption can be considered on a continuous scale, this is an example of simple linear regression.

Logistic regression is an extension of this, where the variable being predicted is *categorical*. We will focus on binary logistic regression, where the variable being predicted has two levels, e.g. yes or no, 0 or 1, dead or alive. Other types of logistic regression include ‘ordinal’, when the outcome variable has >2 ordered levels, and ‘multinomial’, where the outcome variable has >2 levels with no inherent order. We will only deal with binary logistic regression, and when we use the ‘logistic regression’ that is what we are referring to.

We have good reason. In healthcare we are often interested in an event (like death) occurring or not occurring. Binary logistic regression can tell us the probability of this outcome occurring in a patient with a particularly set of characteristics.

Although in binary logistic regression the outcome must have two levels, remember that the predictors (explanatory variables) can be either continuous or categorical.

9.2.1 The Question (1)

As in previous chapters, we will use concrete examples when discussing the principles of the approach. We return to our example of coffee drinking (REF The Question (1) chapter 7). Yes, we are a little obsessed with coffee.

Our outcome variable was previously blood pressure. We will now consider our outcome as the occurrence of a cardiovascular (CV) event over a 10-year period. A cardiovascular event includes the diagnosis of ischemic heart disease, a heart attack (myocardial infarction), or a stroke (cerebrovascular accident). The occurrence of a cardiovascular event is clearly a binary condition, it either happens or it does not. This is ideal therefore for modelling using binary logistic regression. But remember, the data are completely

simulated and not based on anything in the real world. It's just for fun!

9.2.2 Odds and probabilities

To understand logistic regression we need to remind ourselves about odds and probability. Odds and probabilities can get confusing so get them straight with Figure 9.1.

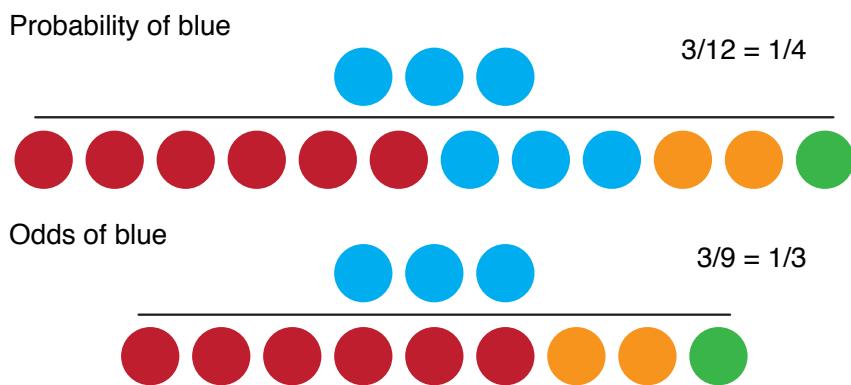


FIGURE 9.1: Probability vs odds.

In many situations, there is no particular reason to prefer one to the other. However, humans seem to have a preference for expressing chance in terms of probabilities, while odds have particular mathematical properties which makes them useful in regression.

When a probability is 0, the odds are 0. When a probability is between 0 and 0.5, the odds are less than 1.0. As a probability increases from 0.5 to 1.0, the odds increase from 1.0 to approach infinity.

Thus the range of probability is 0 to 1 and the range of odds is 0 to $+\infty$.

Odds and probabilities can easily be interconverted. For example, if the odds of a patient dying from a disease are 1/3 (in horse racing this is stated as '3 to 1 against'), then the probability of

death (also known as risk) is 0.25 (or 25%). Odds of 1 to 1 equal 50%.

$Odds = \frac{p}{1-p}$, where p is the probability of the outcome occurring.

$$Probability = \frac{odds}{odds+1}.$$

9.2.3 Odds ratios

Another important term to remind ourselves of is the ‘odds ratio’. Why? Because in a logistic regression the slopes of fitted lines (coefficients) can be interpreted as odds ratios. This is very useful when interpreting the association of a particular predictor with an outcome.

For a given categorical predictor such as smoking, the difference in chance of the outcome occurring for smokers vs. non-smokers can be expressed as a ratio of odds or odds ratio Figure 9.2. For example, if the odds of a smoker have a CV event are 1.5 and the odds of a non-smoker are 1.0, then the odds of a smoker having an event are 1.5-times greater than a non-smoker, odds ratio = 1.5.

				Odds CV event Smoker
		a	b	$= b/d$
CV event Yes	No	a	b	
	Yes	c	d	
		Smoking No	Smoking Yes	Odds CV event smoker vs non-smoker
				$= \frac{a/c}{b/d} = \frac{ad}{bc}$ Odds ratio

FIGURE 9.2: Odds ratios.

An alternative is a ratio of probabilities which is called a risk ratio or relative risk. We will continue to work with odds ratios given

they are an important expression of effect size in logistic regression analysis.

9.2.4 Fitting a regression line

Let's return to the task at hand. The difficulty in moving from a continuous to a binary outcome variable becomes obvious quickly. If our y -axis only has two values, say 0 and 1, then how can be fit line through our data points?

An assumption of linear regression is that dependent variable is continuous, unbounded and measured on an interval or ratio scale. Unfortunately, binary dependent variables fulfil none of these requirements.

The answer is what makes logistic regression so useful. Rather than estimating $y = 0$ or $y = 1$ from the x -axis, we estimate the *probability* of $y = 1$.

There is one more difficulty in this though. Probabilities can only exist between values of 0 and 1. The probability scale is therefore not linear - straight lines do not make sense on it.

As we saw above, the odds scale runs from 0 to $+\infty$. But here, probabilities from 0 to 0.5 are squashed into odds of 0 to 1, and probabilities from 0.5 to 1 have the expansive comfort of 1 to $+\infty$.

This is why we fit binary data on a *log-odds scale*.

A log-odds scale sounds incredibly off-putting to non-mathematicians, but it is the perfect solution!

- Log-odds run from $-\infty$ to $+\infty$;
- odds of 1 become log-odds of 0;
- a doubling and a halving of odds represent the same distance on the scale.

```
log(1)
```

```
## [1] 0
```

```
log(2)
```

```
## [1] 0.6931472
```

```
log(0.5)
```

```
## [1] -0.6931472
```

I'm sure some are shouting 'obviously' at the page. That is good!

This is wrapped up in a transformation (REF working with continuous data transform bit) using the so-called logit function. This can be skipped with no loss of understanding, but for those who just-gots-to-see, the logit function is,

$\log_e(\frac{p}{1-p})$, where p is the probability of the outcome occurring.

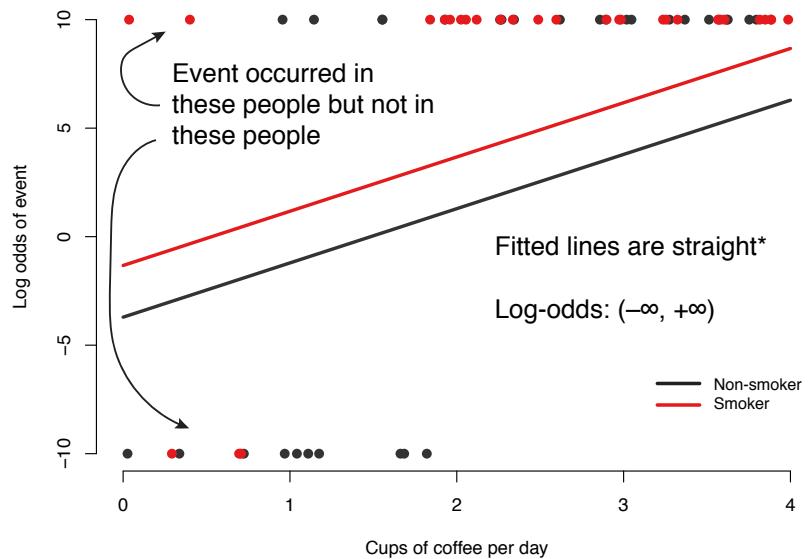
Figure 9.3 demonstrates the fitted lines from a logistic regression model of CV event by coffee consumption stratified by smoking on the log-odds scale (A) and the probability scale (B). We could conclude, for instance, that on average, non-smokers who drink 2 cups of coffee per day have a 50% chance of a cardiovascular event.

9.2.5 The fitted line and the logistic regression equation

Figure 9.4 links the logistic regression equation, the appearance of the fitted lines on the probability scale, and the output from a standard base R analysis. The dots at the top and bottom of the plot represent whether individual patients have had an event or not. The fitted line, therefore, represents the point-to-point probability of a patient with a particular set of characteristics having the event or not. Compare this to Figure 7.4 to be clear on the difference. The slope of the line is linear on the log-odds scale and these are presented in the output on the log-odds scale.

Thankfully, it is straightforward to convert these to odds ratios, a measure we can use to communicate effect size and direction effectively. Said in more technical language, the exponential of the

A Logistic regression: fitted lines log-odds scale



B Logistic regression: fitted lines probability scale

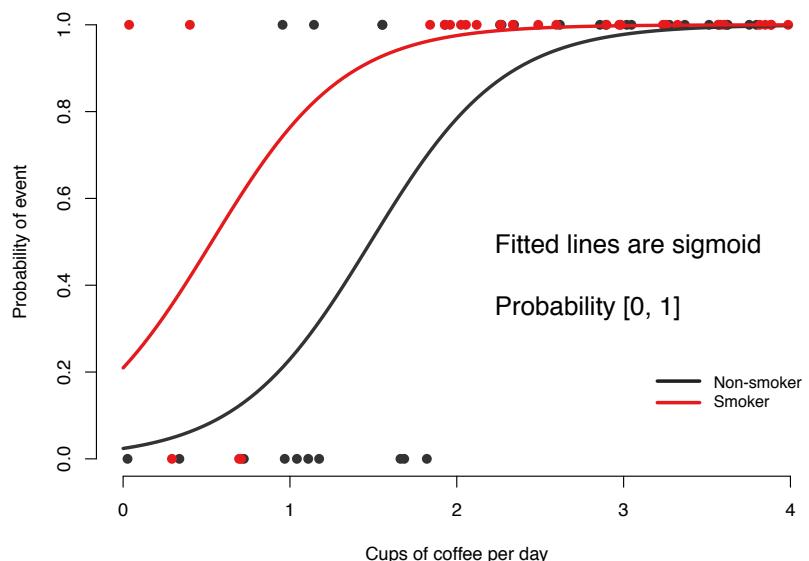


FIGURE 9.3: A logistic regression model of life-time cardiovascular event occurrence by coffee consumption stratified by smoking (simulated data). Fitted lines plotted on the log-odds scale (A) and probability scale (B). *lines are straight when no polynomials or splines are included in regression.

coefficient on the log-odds scale can be interpreted as an odds ratio.

For a continuous variable such as cups of coffee consumed, the odds ratio is the change in odds of a CV event associated with a 1 cup increase in coffee consumption. We are dealing with linear responses here, so the odds ratio is the same for an increase from 1 to 2 cups, or 3 to 4 cups etc. Remember that if the odds ratio for 1 unit of change is 1.5, then the odds ratio for 2 units of change is $\exp(\log(1.5) * 2) = 2.25$.

For a categorical variable such as smoking, the odds ratio is the change in odds of a CV event associated with smoking compared with not smoking (the reference level).

9.2.6 Effect modification and confounding

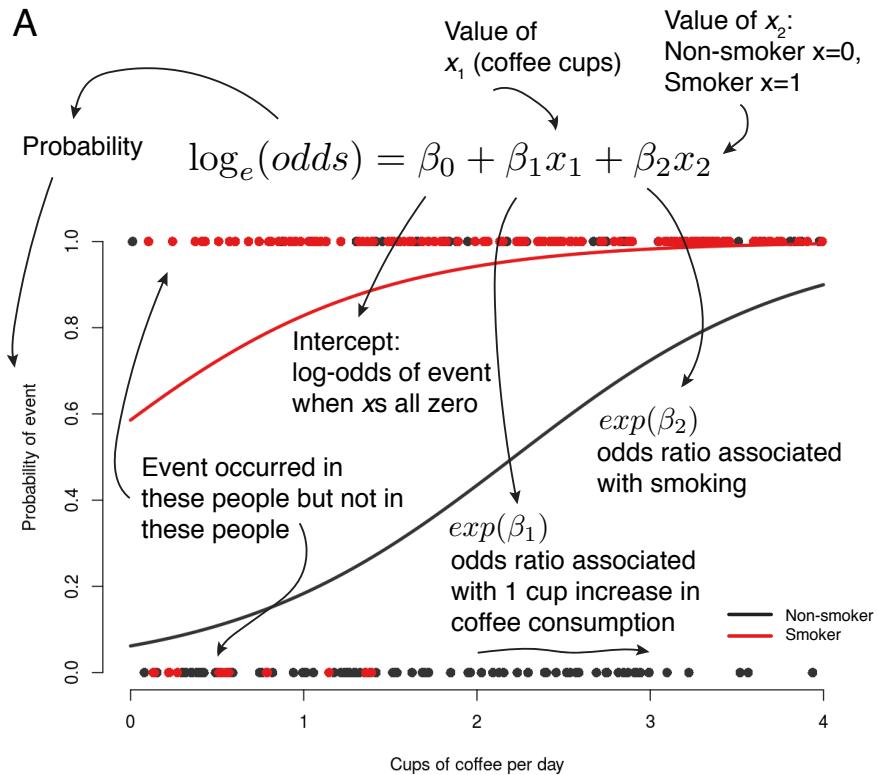
As with all multivariable regression models, logistic regression allows the incorporation of multiple variables which all may have direct effects on outcome or may confound the effect of another variable. This is explored in detail in (SEE CHAPTER 7 confounding).

Adjusting for effect modification and confounding allows us to isolate the direct effect of an explanatory variable of interest upon an outcome. In our example, we are interested in direct effect of coffee drinking on the occurrence of cardiovascular disease, independent of any association between coffee drinking and smoking.

Figure ?? demonstrates simple, additive and multiplicative models. Refer to Figure ?? and the discussion around it if this doesn't make sense.

Presented on the probability scale, the effect of the interaction is difficult to see. It is obvious on the log-odds scale that the fitted lines are no longer constrained to be parallel.

The interpretation of the interaction term is important. The exponential of the interaction coefficient term represents a ‘ratio-of-odds ratios’. This is easiest to see through a worked example.



B Logistic regression (glm) output

```

Call:
glm(formula = y ~ coffee + smoking, family = "binomial", data = df) ← Function call

Deviance Residuals:
    Min      1Q   Median      3Q     Max 
-2.1121 -0.5552  0.2567  0.6571  2.3538 ← Distribution of residuals

Coefficients:
            Estimate Std. Error z value Pr(>|z|) 
(Intercept) -2.7197    0.4398 -6.184 6.24e-10 ***
coffee       1.2284    0.1816  6.765 1.33e-11 *** 
smoking      3.0658    0.4066  7.540 4.71e-14 *** 
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 377.61 on 299 degrees of freedom
Residual deviance: 247.10 on 297 degrees of freedom
AIC: 253.1

```

$\log_e(\text{odds[event]}) = \beta_0 + \beta_{\text{coffee}}x_{\text{coffee}} + \beta_{\text{smoking}}x_{\text{smoking}}$

Results

Coffee:

$$\text{OR} = \exp(1.23) = 3.42$$

Smoking

$$\text{OR} = \exp(3.07) = 21.50$$

FIGURE 9.4: Linking the logistic regression fitted line, equation and R output.

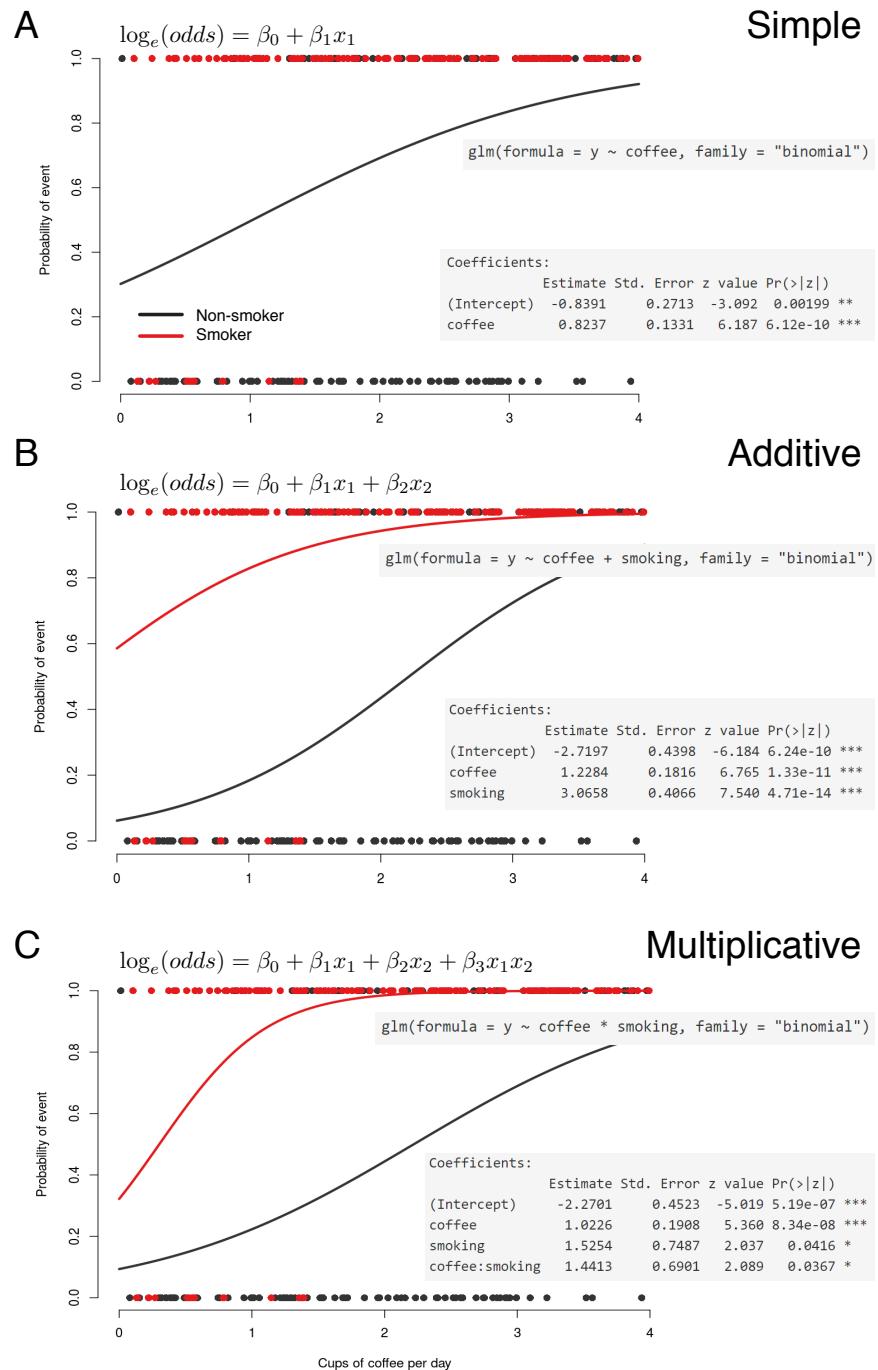


FIGURE 9.5: Multivariable logistic regression with additive and multiplicative effect modification.

In Figure 9.6 the effect of coffee on the odds of a cardiovascular event can be compared in smokers and non-smokers. The effect is now different given the inclusion of a significant interaction term. Please check back to the linear regression chapter if this is not making sense.

Multiplicative / interaction model

$\log_e(\text{odds}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$ <code>glm(formula = y ~ coffee * smoking, family = "binomial")</code>						
Coefficients:						
	Estimate	Std. Error	z value	Pr(> z)		
(Intercept)	-2.2701	0.4523	-5.019	5.19e-07 ***		
coffee	1.0226	0.1908	5.360	8.34e-08 ***		
smoking	1.5254	0.7487	2.037	0.0416 *		
coffee:smoking	1.4413	0.6901	2.089	0.0367 *		
					OR = $\exp(1.0226) = 2.78$	
					OR = $\exp(1.5254) = 4.60$	
					OR = $\exp(1.4413) = 4.23$	
Odds ratio (OR)						
Effect of each coffee in non-smokers					2.78	
Effect of each coffee in smokers					$(2.78 * 4.23) = 11.76$	
Effect of smoking in non-coffee drinkers					OR = 4.60	
Effect of smoking for each additional coffee					$(4.60 * 4.23) = 19.46$	

FIGURE 9.6: Multivariable logistic regression with interaction term. The exponential of the interaction term is a ratio-of-odds ratios (ROR).

9.3 Data preparation and exploratory analysis

9.3.1 The Question (2)

We will go on to explore the `boot::melanoma` dataset introduced in Chapter 8 (REF). The data consist of measurements made on patients after surgery to remove the melanoma in the University Hospital of Odense, Denmark between 1962 and 1977.

Malignant melanoma is aggressive and highly invasive, making it difficult to treat.

It's classically staged based on the depth of the tumour. The current TNM classification cut-offs are:

- T1: ≤ 1.0 mm depth
- T2: 1.1 to 2.0 mm depth
- T3: 2.1 to 4.0 mm depth
- T4: > 4.0 mm depth

This will be important in our analysis as we will create a new variable based upon this.

Using logistic regression, we will investigate factors associated with death from malignant melanoma with particular interest in tumour ulceration.

9.3.2 Get the data

The help page (`?boot::melanoma`) gives us the data dictionary. This includes the definition of each variable and the coding used.

```
melanoma = boot::melanoma
```

9.3.3 Check the data

As before, always carefully check and clean new dataset before you start the analysis.

```
library(tidyverse)
library(finalfit)
mydata %>% glimpse()
mydata %>% ff_glimpse()
```

9.3.4 Recode the data

We have done some of this already, but for this particular analysis we will recode some further variables.

```
library(tidyverse)
library(finalfit)
melanoma = melanoma %>%
```

```

  mutate(sex.factor =
    sex %>%
    factor() %>%
    fct_recode(
      "Female" = "0",
      "Male" = "1") %>%
    ff_label("Sex"),      # Label for finalfit tables

  ulcer.factor =
    ulcer %>%
    factor() %>%
    fct_recode(
      "Present" = "1",
      "Absent" = "0") %>%
    ff_label("Ulcerated tumour"),

  age = ff_label(age, "Age (years)"),
  year = ff_label(year, "Year"),

  status.factor =
    status %>%
    factor() %>%
    fct_recode("Died melanoma" = "1",
                 "Alive" = "2",
                 "Died - other" = "3") %>%
    fct_relevel("Alive") %>%
    ff_label("Status"),

  t_stage.factor =
    thickness %>%
    cut(breaks = c(0, 1.0, 2.0, 4.0,
                   max(thickness, na.rm=TRUE)),
          include.lowest = TRUE)
)

```

Check the `cut()` function has worked.

```

melanoma$t_stage.factor %>% levels()

## [1] "[0,1]"   "(1,2]"   "(2,4]"   "(4,17.4]"

```

Recode for ease.

```

melanoma = melanoma %>%
  mutate(
    t_stage.factor =
      fct_recode(t_stage.factor,

```

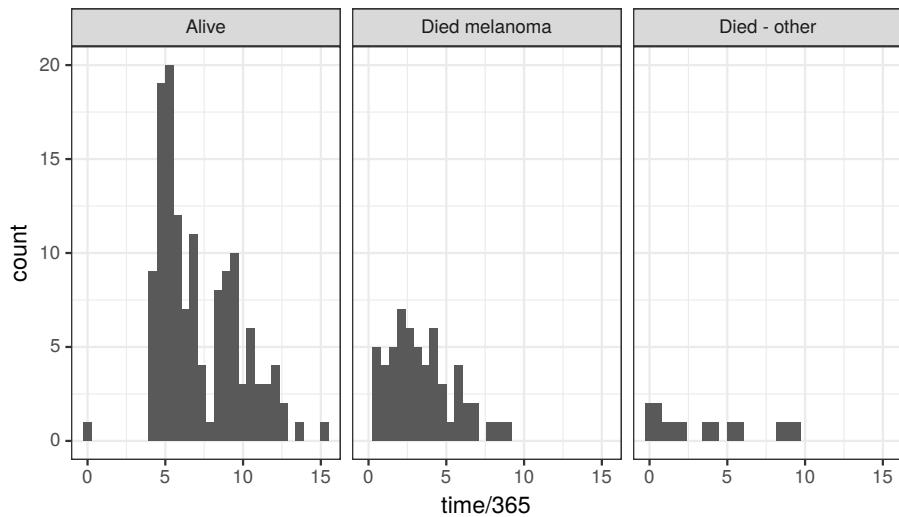
```
T1 = "[0,1]",  
T2 = "(1,2]",  
T3 = "(2,4]",  
T4 = "(4,17.4]" %>%  
  ff_label("T-stage")  
)
```

We will now consider our outcome variable. With a binary outcome and health data, we often have to make a decision as to *when* to determine if that variable has occurred or not. In the next chapter we will look at survival analysis where this requirement if not needed.

Our outcome of interest is death from melanoma, but we need to decide when to define this.

A quick histogram of `time` stratified by `status.factor` helps. We can see that most people who died from melanoma did so before 5 years. We can also see that the status of those who did not die is known beyond 5 years. Let's decide then to look at 5-year mortality from melanoma. The definition of this will be at 5 years after surgery, who had died from melanoma and who had not.

```
library(ggplot2)  
melanoma %>%  
  ggplot(aes(x = time/365)) +  
  geom_histogram() +  
  facet_grid(. ~ status.factor)  
  
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



```
# 5-year mortality
melanoma = melanoma %>%
  mutate(
    mort_5yr =
      if_else((time/365) < 5 &
              (status == 1),
              "Yes",           # then
              "No") %>%     # else
    fct_relevel("No") %>%
    ff_label("5-year survival")
  )
```

9.3.5 Plot the data

We are interested in the association between tumour ulceration and outcome.

```
p1 = melanoma %>%
  ggplot(aes(x = ulcer.factor, fill = mort_5yr)) +
  geom_bar() +
  theme(legend.position = "none")

p2 = melanoma %>%
  ggplot(aes(x = ulcer.factor, fill = mort_5yr)) +
  geom_bar(position = "fill") +
  ylab("proportion")
```

```
library(patchwork)
p1 / p2
```

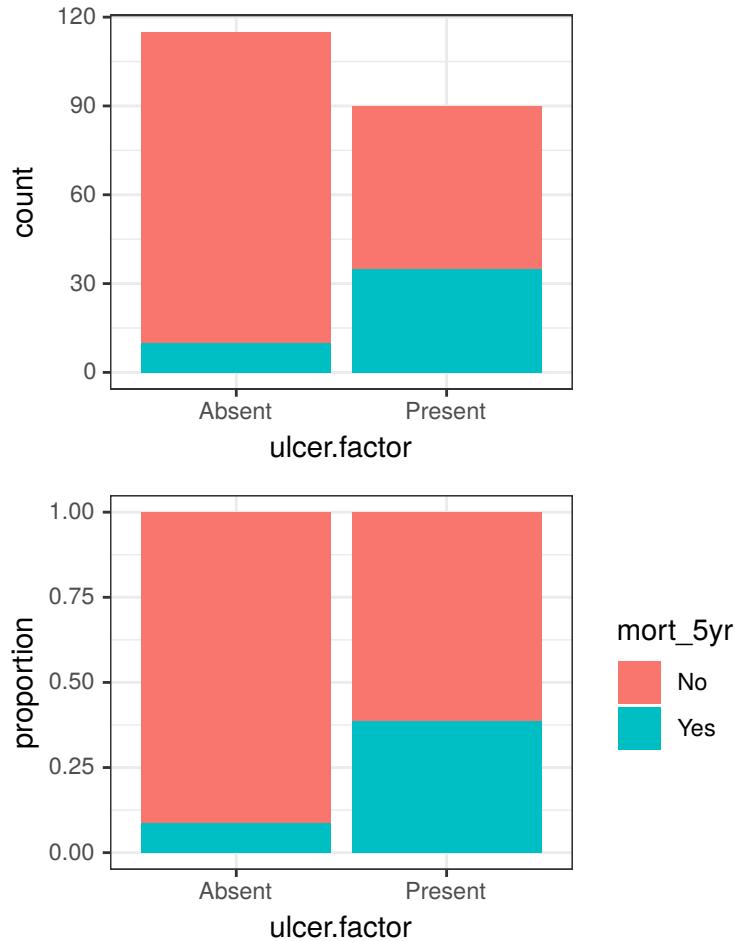


FIGURE 9.7: CAPTION

As we might have anticipated from our work in chapter 8, 5-year mortality is higher in patients with ulcerated tumours compared with those with non-ulcerated tumours.

We are also interested in other variables that may be associated with tumour ulceration. If they are also associated with our out-

come, then they will confound the estimate of the direct effect of tumour ulceration.

We can plot out these relationships, or tabulate them instead.

9.3.6 Tabulate data

We will use the convenient `Finalfit` function `summary_factorlist()` to look for difference across other variables by tumour ulceration.

```
library(finalfit)
dependent = "ulcer.factor"
explanatory = c("age", "sex.factor", "year", "t_stage.factor")
melanoma %>%
  summary_factorlist(dependent, explanatory, p = TRUE,
                     add_dependent_label = TRUE)
```

TABLE 9.1: Multiple variables by explanatory variable of interest: Malignant melanoma ulceration by patient and disease variables.

Dependent: Ulcerated tumour		Absent	Present	p
Age (years)	Mean (SD)	50.6 (15.9)	54.8 (17.4)	0.067
Sex	Female	79 (68.7)	47 (52.2)	0.016
	Male	36 (31.3)	43 (47.8)	
Year	Mean (SD)	1970.0 (2.7)	1969.8 (2.4)	0.382
T-stage	T1	51 (44.3)	5 (5.6)	<0.001
	T2	36 (31.3)	17 (18.9)	
	T3	21 (18.3)	30 (33.3)	
	T4	7 (6.1)	38 (42.2)	

It appears that patients with ulcerated tumours were older, more likely to be male, and had thicker/higher stage tumours. It is important therefore to consider inclusion of these variables in a regression model.

9.4 Model assumptions

Binary logistic regression is robust many assumptions which can cause problems in other statistical analyses. The main assumptions are:

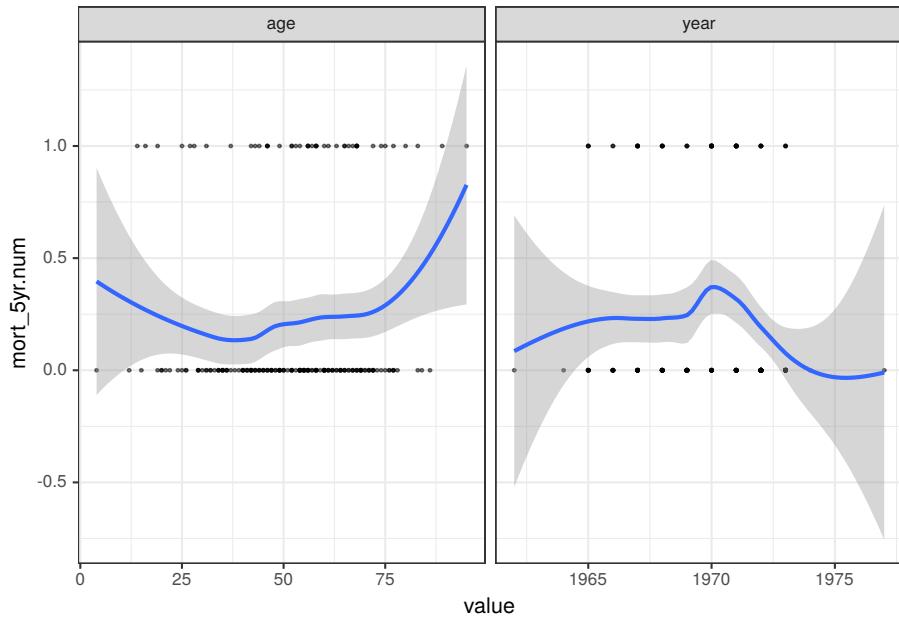
1. Binary dependent variable - this is obvious, but as above we need to check (alive, death from disease, death from other causes doesn't work);
2. Independence of observations - the observations should not be repeated measurements or matched data;
3. Linearity of continuous explanatory variables and the log-odds outcome - take age as an example. If the outcome, say death, gets more frequent or less frequent as age rises, the model will work well. However, say children and the elderly are at high risk of death, but those in middle years are not, then the relationship is not linear. Or more correctly, is not monotonic, meaning that the response does not only go in one direction.
4. No multicollinearity - explanatory variables should not be highly correlated with each other;

9.4.1 Linearity of continuous variables to the response

A graphical check of linearity can be performed using a best fit “loess” line. This is on the probability scale, so it is not going to be straight. But it should be monotonic - it should only ever go up or down.

```
library(tidyverse)
melanoma %>%
  mutate(
    mort_5yr.num = as.numeric(mort_5yr) - 1
  ) %>%
  select(mort_5yr.num, age, year) %>%
  gather(key = "predictors", value = "value", -mort_5yr.num) %>%
```

```
ggplot(aes(x = value, y = mort_5yr.num)) +
  geom_point(size = 0.5, alpha = 0.5) +
  geom_smooth(method = "loess") +
  facet_wrap(~predictors, scales = "free_x")
```



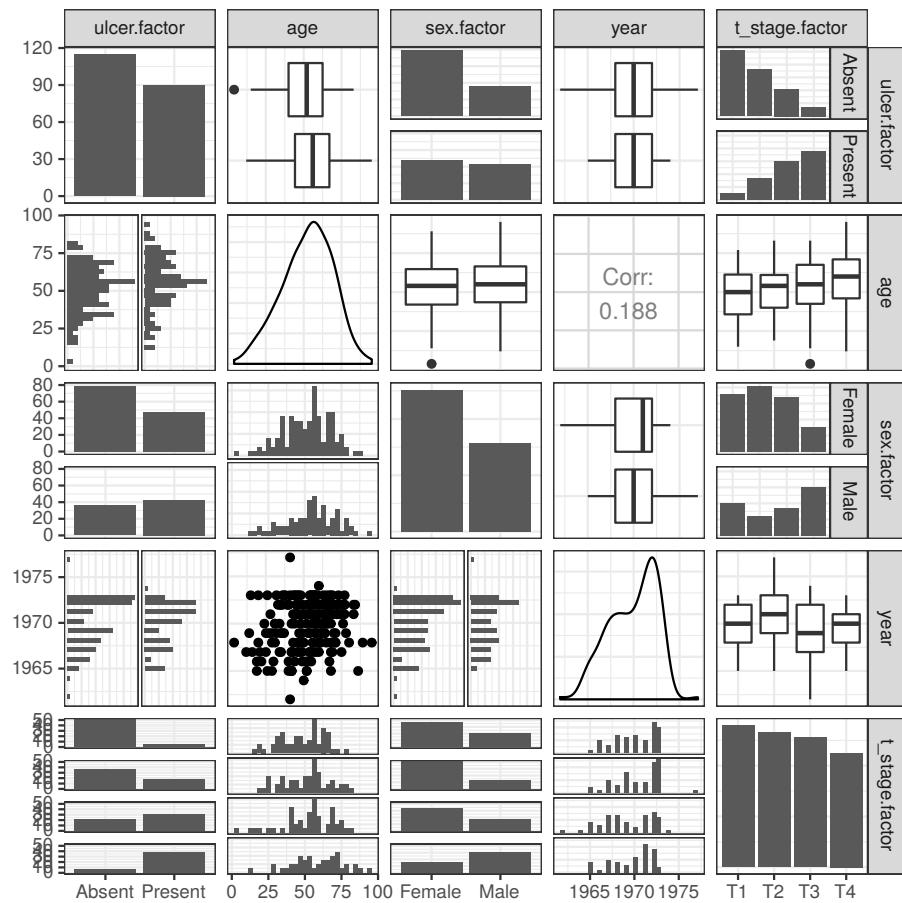
Age is interesting here, the relationship is u-shaped. The chance of death is higher in the young and the old compared with the middle-aged. This will need to be accounted for in any model including age as a predictor.

9.4.2 Multicollinearity

Multicollinearity occurs when two highly correlated explanatory variables are included in a model. If both variables describe the same thing, then their coefficients (ORs) can become unstable potentially leading to erroneous conclusions. Think about your variables before you start - would any be expected to be highly correlated?

The `ggpairs()` function from `library(GGally)` gives you all the plots you can dream of and more, but it is a lot:

```
library(GGally)
explanatory = c("ulcer.factor", "age", "sex.factor",
               "year", "t_stage.factor")
melanoma %>%
  remove_labels() %>% # ggpairs is older and doesn't like labels
  ggpairs(columns = explanatory)
```

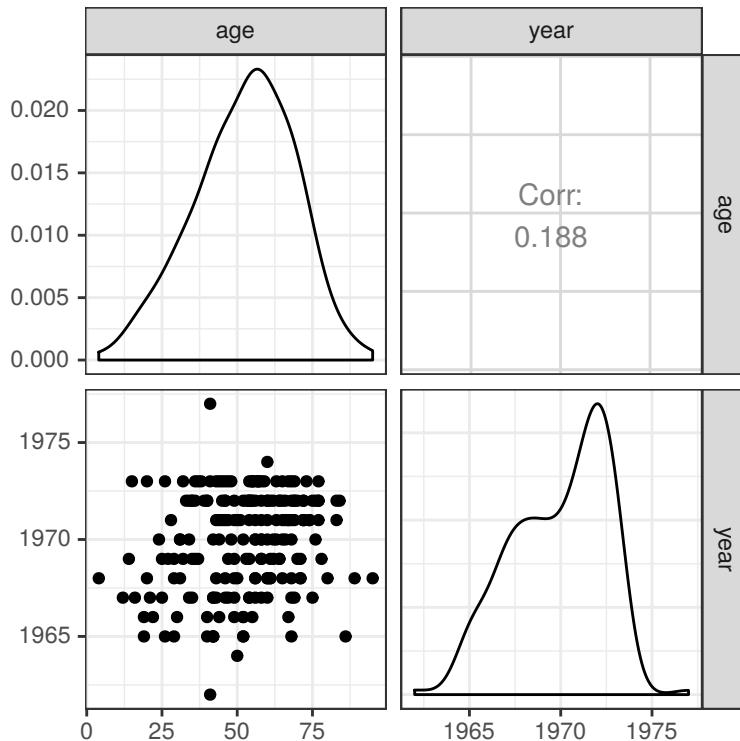


If you have many variables you want to check you can split them up.

Continuous to continuous

```
select_explanatory = c("age", "year")
melanoma %>%
```

```
remove_labels() %>%
  ggpairs(columns = select_explanatory)
```



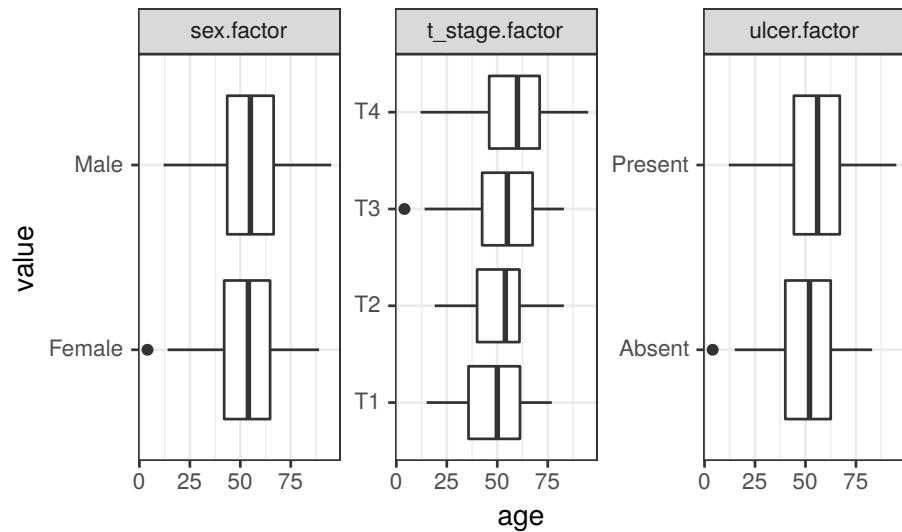
Continuous to categorical

Let's split that up a bit and use a clever `gather()` and `facet_wrap()` combination. We want to compare everything against, for example, `age` so we need to add `-age` to the `gather` call so it doesn't get lumped up with everything else. But because the excluded variable has to be in the third argument of `gather()` we need to type in `key`, `value` as placeholders:

```
select_explanatory = c("age", "ulcer.factor",
                      "sex.factor", "t_stage.factor")

melanoma %>%
  select(one_of(select_explanatory)) %>%
  gather(key, value, -age) %>%
  ggplot(aes(value, age)) +
```

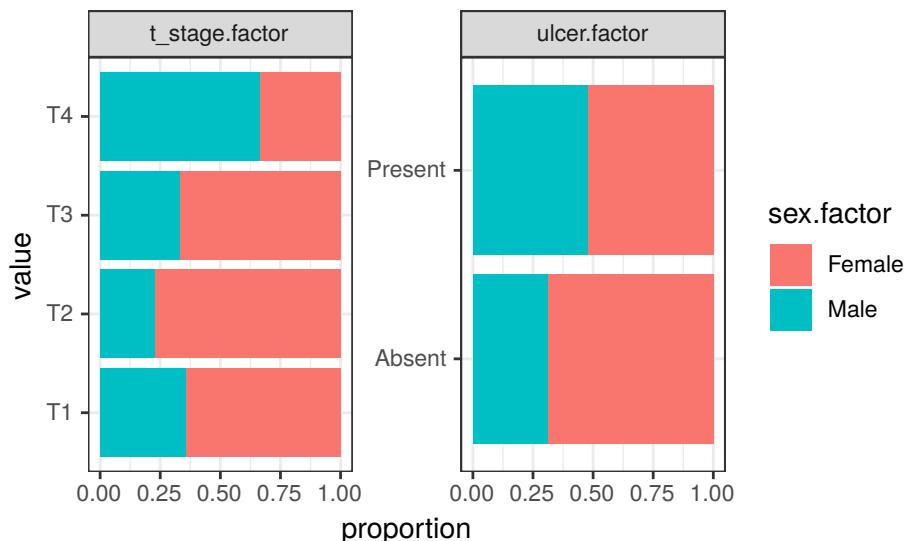
```
geom_boxplot() +
facet_wrap(~key, scale = "free", ncol = 3) +
coord_flip()
```



Categorical to categorical

```
select_explanatory = c("ulcer.factor", "sex.factor", "t_stage.factor")

melanoma %>%
  select(one_of(select_explanatory)) %>%
  gather(key, value, -sex.factor) %>%
  ggplot(aes(value, fill = sex.factor)) +
  geom_bar(position = "fill") +
  ylab("proportion") +
  facet_wrap(~key, scale = "free", ncol = 2) +
  coord_flip()
```



None of the explanatory variables are highly correlated with one another.

We are not trying to over-egg this, but multicollinearity can be important. The message as always is the same. Understand the underlying data using plotting and tables, and you are unlikely to come unstuck.

9.5 Fitting logistic regression models in base R

The `glm()` stands for `generalised linear model` and is the standard base R approach to logistic regression.

The `glm()` function has several options and many different types of model can be run. For instance, ‘Poisson regression’ for count data.

To run binary logistic regression use `family = binomial`. This defaults to `family = binomial(link = 'logit')`. Other link functions exists, such as the `probit` function, but this makes little difference to final conclusions.

Let's start with a simple univariable model using the classical R approach.

```
fit1 = glm(mort_5yr ~ ulcer.factor, data = melanoma, family = binomial)
summary(fit1)

##
## Call:
## glm(formula = mort_5yr ~ ulcer.factor, family = binomial, data = melanoma)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -0.9925 -0.9925 -0.4265 -0.4265  2.2101
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.3514    0.3309 -7.105 1.20e-12 ***
## ulcer.factorPresent 1.8994    0.3953  4.805 1.55e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 215.78 on 204 degrees of freedom
## Residual deviance: 188.24 on 203 degrees of freedom
## AIC: 192.24
##
## Number of Fisher Scoring iterations: 5
```

This is the standard R output which you should become familiar with. It what is included in the previous figures. The estimates of the coefficients (slopes) in this output are on the log-odds scale and always will be.

Easier approaches to use in practice are shown below, but for completeness here is how to extract these results. `str()` shows all the information included in the model object, which is useful for experts but a bit off-putting if you are starting out.

The coefficients and their 95% confidence intervals can be extracted and exponentiated like this.

```
coef(fit1) %>% exp()

##
## (Intercept) ulcer.factorPresent
##          0.0952381          6.6818182
```

```
confint(fit1) %>% exp()

## Waiting for profiling to be done...

##               2.5 %    97.5 %
## (Intercept) 0.04662675 0.1730265
## ulcer.factorPresent 3.18089978 15.1827225
```

Note that the 95% confidence interval is between the 2.5% and 97.5% quantiles of the distribution, hence why the results appear in this way.

A good alternative is the `tidy()` function from the `broom` package.

```
library(broom)
fit1 %>%
  tidy(conf.int = TRUE, exp = TRUE)

## # A tibble: 2 x 7
##   term      estimate std.error statistic p.value conf.low conf.high
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) 0.0952    0.331    -7.11 1.20e-12  0.0466    0.173
## 2 ulcer.factorPre~ 6.68     0.395     4.80 1.55e- 6  3.18     15.2
```

We can see from these results that there is a strong association between tumour ulceration and 5-year mortality (OR 6.68, 95%CI 3.18, 15.18).

Model metrics can be extracted using the `glance()` function.

```
fit1 %>%
  glance()

## # A tibble: 1 x 7
##   null.deviance df.null logLik   AIC   BIC deviance df.residual
##           <dbl>    <int>  <dbl> <dbl> <dbl>     <dbl>        <int>
## 1         216.      204  -94.1  192.  199.     188.        203
```

9.6 Modelling strategy for binary outcomes

A statistical model is a tool to understand the world. The better your model describes your data, the more useful it will be. Fitting a successful statistical model requires decisions around which variables to include in the model. Our advice regarding variable selection follows the same lines as in the linear regression chapter.

1. As few explanatory variables should be used as possible (parsimony);
2. Explanatory variables associated with the outcome variable in previous studies should be accounted for;
3. Demographic variables should be included in model exploration;
4. Population stratification should be incorporated if available;
5. Interactions should be checked and included if influential;
6. Final model selection should be performed using a “criterion-based approach”
 - minimise the Akaike information criterion (AIC)
 - maximise the c-statistic (area under the receiver operator curve).

We will use these principles through the next section.

9.7 Fitting logistic regression models with `Finalfit`

Our preference in model fitting is now to use our own `Finalfit` package. It gets us to our results quicker and more easily, and produces our final model tables which go directly into manuscripts for publication (we hope).

The approach is the same as in linear regression. If the outcome

variable is correctly specified as a factor, the `finalfit()` function will run a logistic regression model directly.

```
library(finalfit)
dependent = "mort_5yr"
explanatory = "ulcer.factor"
melanoma %>%
  finalfit(dependent, explanatory, metrics = TRUE)
```

TABLE 9.2: Univariable logistic regression: 5-year survival from malignant melanoma by tumour ulceration (fit 1).

Dependent: 5-year survival	No	Yes	OR (univariable)	OR (multivariable)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001) 6.68 (3.18-15.18, p<0.001)

TABLE 9.3: Model metrics: 5-year survival from malignant melanoma by tumour ulceration (fit 1).

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 192.2, C-statistic = 0.717, H&L = Chi-sq(8) 0.00 (p=1.000)

9.7.1 Criterion-based model fitting

Passing `metrics = TRUE` to `finalfit()` gives us a useful list of model fitting parameters.

We recommend looking at three metrics:

- Akaike information criterion (AIC), which should be minimised;
- C-statistic (area under the receiver operator curve), which should be maximised;
- Hosmer–Lemeshow test, which should be non-significant.

AIC \index{AIC}

The AIC has been previous described (REF). It provides a measure of model goodness-of-fit - or how well the model fits the available data. It is penalised for each additional variable, so should be somewhat robust against over-fitting (when the model starts to describe noise).

C-statistic

The c-statistic or area under the receiver operator curve (ROC) provides a measure of model ‘discrimination’. It runs from 0.5 to 1.0, with 0.5 being no better than chance, and 1.0 being perfect fit. What the number is can be thought of as this. Take our example of death from melanoma. If you take a random patient who died and a random patient who did not die, then the c-statistic is the probability that the model predicts that patient 1 is more likely to die than patient 2. In our example above, the model should get that correct 72% of the time.

Hosmer-Lemeshow test

If you are interested in using your model for prediction, it is important that it is calibrated correctly. Using our example, calibration means that the model accurately predicts death from melanoma when the risk to the patient is low and also accurately predicts death when the risk is high. The model should work well across the range of probabilities of death. The Hosmer-Lemeshow test assesses this. By default, it assesses the predictive accuracy for death in deciles of risk. If the model predicts equally well (or badly) at low probabilities compared with high probabilities, the null hypothesis of a difference will be rejected (meaning you get a non-significant p-value).

9.8 Model fitting

Engage with the data and the results when model fitting. Do not use automated processes - you have to keep thinking.

Three things are important to keep looking at:

- what is the association between a particular variable and the outcome (OR and 95%CI);
- how much information is a variable bringing to the model (change in AIC and c-statistic);

- how much influence does adding a variable have on the effect size of another variable, and in particular my variable of interest (a rule of thumb is a greater than 10% change in the OR of the variable of interest is important).

We're going to start by including the variables from above which we think are relevant.

```
library(finalfit)
dependent = "mort_5yr"
explanatory = c("ulcer.factor", "age", "sex.factor", "t_stage.factor")
melanoma %>%
  finalfit(dependent, explanatory, metrics = TRUE)
```

TABLE 9.4: Multivariable logistic regression: 5-year survival from malignant melanoma (fit 2).

Dependent: 5-year survival		No	Yes	OR (univariable)	OR (multivariable)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	3.21 (1.32-8.28, p=0.012)
Age (years)	Mean (SD)	51.7 (16.0)	55.3 (18.8)	1.01 (0.99-1.03, p=0.202)	1.00 (0.98-1.02, p=0.948)
Sex	Female	105 (65.6)	21 (46.7)	-	-
	Male	55 (34.4)	24 (53.3)	2.18 (1.12-4.30, p=0.023)	1.26 (0.57-2.76, p=0.558)
T-stage	T1	52 (32.5)	4 (8.9)	-	-
	T2	49 (30.6)	4 (8.9)	1.06 (0.24-4.71, p=0.936)	0.77 (0.16-3.58, p=0.733)
	T3	36 (22.5)	15 (33.3)	5.42 (1.80-20.22, p=0.005)	2.98 (0.86-12.10, p=0.098)
	T4	23 (14.4)	22 (48.9)	12.43 (4.21-46.26, p<0.001)	4.98 (1.34-21.64, p=0.021)

TABLE 9.5: Model metrics: 5-year survival from malignant melanoma (fit 2).

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 188.1, C-statistic = 0.798, H&L = Chi-sq(8) 3.92 (p=0.864)

The model metrics have improved with the AIC decreasing from 192 to 188 and the c-statistic increasing from 0.717 to 0.798.

Let's consider `age`. We may expect age to be associated with the outcome become is so commonly is. But there is weak evidence of an association in the univariable analysis. We have shown above that the relationship of age to the outcome is not linear, therefore we need to act on this.

We can either convert age to a categorical variable or include it with a quadratic term ($x^2 + x$, remember parabolas from school?).

```

melanoma = melanoma %>%
  mutate(
    age.factor = cut(age,
      breaks = c(0, 25, 50, 75, 100)) %>%
    ff_label("Age (years)")

  # Add this to relevel:
  # fct_relevel("(50,75]")

melanoma %>%
  finalfit(dependent, c("ulcer.factor", "age.factor"), metrics = TRUE)

## Warning in chisq.test(tab, correct = FALSE): Chi-squared approximation may
## be incorrect

```

TABLE 9.6: Multivariable logistic regression: using ‘cut’ to convert a continuous variable as a factor (fit 3).

Dependent: 5-year survival		No	Yes	OR (univariable)	OR (multivariable)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	6.28 (2.97-14.35, p<0.001)
Age (years)	(0,25]	10 (6.2)	4 (8.9)	-	-
	(25,50]	62 (38.8)	11 (24.4)	0.44 (0.12-1.84, p=0.229)	0.54 (0.13-2.44, p=0.400)
	(50,75]	79 (49.4)	25 (55.6)	0.79 (0.24-3.08, p=0.712)	0.81 (0.22-3.39, p=0.753)
	(75,100]	9 (5.6)	5 (11.1)	1.39 (0.28-7.23, p=0.686)	1.12 (0.20-6.53, p=0.894)

TABLE 9.7: Model metrics: using ‘cut’ to convert a continuous variable as a factor (fit 3).

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 196.6, C-statistic = 0.742, H&L = Chi-sq(8) 0.20 (p=1.000)

There is no strong relationship between the categorical representation of age and the outcome. Let’s try a quadratic term.

In base R, a quadratic term is added like this.

```

glm(mort_5yr ~ ulcer.factor + I(age^2) + age,
  data = melanoma, family = binomial) %>%
  summary()

## 
## Call:
## glm(formula = mort_5yr ~ ulcer.factor + I(age^2) + age, family = binomial,
##   data = melanoma)
## 
```

```

## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.3253 -0.8973 -0.4082 -0.3889  2.2872
##
## Coefficients:
##                               Estimate Std. Error z value Pr(>|z|)
## (Intercept)           -1.2636638  1.2058471 -1.048   0.295
## ulcer.factorPresent  1.8423431  0.3991559  4.616 3.92e-06 ***
## I(age^2)                0.0006277  0.0004613  1.361   0.174
## age                  -0.0567465  0.0476011 -1.192   0.233
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 215.78 on 204 degrees of freedom
## Residual deviance: 185.98 on 201 degrees of freedom
## AIC: 193.98
##
## Number of Fisher Scoring iterations: 5

```

It can be done in `Finalfit` in a similar manner. Note with default univariable model settings, the quadratic and linear term are considered in separate models, which doesn't make much sense.

```

library(finalfit)
dependent = "mort_5yr"
explanatory = c("ulcer.factor", "I(age^2)", "age")
melanoma %>%
  finalfit(dependent, explanatory, metrics = TRUE)

```

TABLE 9.8: Multivariable logistic regression: including a quadratic term (fit 4).

Dependent: 5-year survival		No	Yes	OR (univariable)	OR (multivariable)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	6.31 (2.98-14.44, p<0.001)
Age (years)	Mean (SD)	2924.7 (1600.8)	3399.8 (2011.9)	1.00 (1.00-1.00, p=0.101)	1.00 (1.00-1.00, p=0.174)
	Mean (SD)	51.7 (16.0)	55.3 (18.8)	1.01 (0.99-1.03, p=0.202)	0.94 (0.86-1.04, p=0.233)

TABLE 9.9: Model metrics: including a quadratic term (fit 4).

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 194, C-statistic = 0.748, H&L = Chi-sq(8) 5.24 (p=0.732)

The AIC is worse when adding age either as a factor or with a quadratic term to the base model.

One final method to visualise the contribution of a particular variable is to remove it from the full model. This is convenient in `Finalfit`.

```
library(finalfit)
dependent = "mort_5yr"
explanatory = c("ulcer.factor", "age.factor", "sex.factor", "t_stage.factor")
explanatory_multi = c("ulcer.factor", "sex.factor", "t_stage.factor")

melanoma %>%
  finalfit(dependent, explanatory, explanatory_multi,
            keep_models = TRUE, metrics = TRUE)

## Warning in chisq.test(tab, correct = FALSE): Chi-squared approximation may
## be incorrect
```

TABLE 9.10: Multivariable logistic regression model: comparing a reduced model in one table (fit 5).

	Dependent: 5-year survival	No	Yes	OR (univariable)	OR (multivariable)	OR (multivariable reduced)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-	-	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	3.06 (1.25-7.93, p=0.017)	3.21 (1.32-8.28, p=0.012)
Age (years)	(0,25]	10 (6.2)	4 (8.9)	-	-	-
	(25,50]	62 (38.8)	11 (24.4)	0.44 (0.12-1.84, p=0.229)	0.37 (0.08-1.80, p=0.197)	-
	(50,75]	79 (49.4)	25 (55.6)	0.79 (0.24-3.08, p=0.712)	0.60 (0.15-2.65, p=0.469)	-
	(75,100]	9 (5.6)	5 (11.1)	1.39 (0.28-7.23, p=0.686)	0.61 (0.09-4.04, p=0.599)	-
Sex	Female	105 (65.6)	21 (46.7)	-	-	-
	Male	55 (34.4)	24 (53.3)	2.18 (1.12-4.30, p=0.023)	1.21 (0.54-2.68, p=0.633)	1.26 (0.57-2.76, p=0.559)
T-stage	T1	52 (32.5)	4 (8.9)	-	-	-
	T2	49 (30.6)	4 (8.9)	1.06 (0.24-4.71, p=0.936)	0.74 (0.15-3.50, p=0.697)	0.77 (0.16-3.58, p=0.733)
	T3	36 (22.5)	15 (33.3)	5.42 (1.80-20.22, p=0.005)	2.91 (0.84-11.82, p=0.106)	2.99 (0.86-12.11, p=0.097)
	T4	23 (14.4)	22 (48.9)	12.43 (4.21-46.26, p<0.001)	5.38 (1.43-23.52, p=0.016)	5.01 (1.37-21.52, p=0.020)

TABLE 9.11: Model metrics: comparing a reduced model in one table (fit 5).

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 190, C-statistic = 0.802, H&L = Chi-sq(8) 13.87 (p=0.085)
 Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 186.1, C-statistic = 0.794, H&L = Chi-sq(8) 1.07 (p=0.998)

The AIC improves when age is removed (186 from 190) at only a small loss in discrimination (0.794 from 0.802). Looking at the model table and comparing the full multivariable with the reduced multivariable, there has been a small change in the OR for ulceration, with some of the variation accounted for by age now being taken up by ulceration. This is to be expected, given the association (albeit weak) that we saw earlier between age and ulceration. Given all this, we will decide not to include age in the model.

Now what about the variable sex. It has a significant association with the outcome in the univariable analysis, but much of this is explained by other variables in multivaribale anlysis. Is it contributing much to the model?

```
library(finalfit)
dependent = "mort_5yr"
explanatory = c("ulcer.factor", "sex.factor", "t_stage.factor")
explanatory_multi = c("ulcer.factor", "t_stage.factor")

melanoma %>%
finalfit(dependent, explanatory, explanatory_multi,
           keep_models = TRUE, metrics = TRUE)
```

TABLE 9.12: Multivariable logistic regression: further reducing the model (fit 6).

	Dependent: 5-year survival	No	Yes	OR (univariable)	OR (multivariable)	OR (multivariable reduced)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-	-	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	3.21 (1.32-8.28, p=0.012)	3.26 (1.35-8.39, p=0.011)
Sex	Female	105 (65.6)	21 (46.7)	-	-	-
	Male	55 (34.4)	24 (53.3)	2.18 (1.12-4.30, p=0.023)	1.26 (0.57-2.76, p=0.559)	-
T-stage	T1	52 (32.5)	4 (8.9)	-	-	-
	T2	49 (30.6)	4 (8.9)	1.06 (0.24-4.71, p=0.936)	0.77 (0.16-3.58, p=0.733)	0.75 (0.16-3.45, p=0.700)
	T3	36 (22.5)	15 (33.3)	5.42 (1.80-20.22, p=0.005)	2.99 (0.86-12.11, p=0.097)	2.96 (0.86-11.96, p=0.098)
	T4	23 (14.4)	22 (48.9)	12.43 (4.21-46.26, p<0.001)	5.01 (1.37-21.52, p=0.020)	5.33 (1.48-22.56, p=0.014)

TABLE 9.13: Model metrics: further reducing the model (fit 6).

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 186.1, C-statistic = 0.794, H&L = Chi-sq(8) 1.07 (p=0.998)
 Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 184.4, C-statistic = 0.791, H&L = Chi-sq(8) 0.43 (p=1.000)

By removing sex we have improved the AIC a little (184.4 from 186.1) with a small change in the c-statistic (0.791 from 0.794).

Looking at the model table, the variation has been taken up mostly by stage 4 disease and a little by ulceration. But there has been little change overall. We will exclude sex from our final model as well.

As a final we can check for a first order interaction between ulceration and T-stage. Just to remind us what this means, a significant interaction would mean the effect of, say, ulceration on 5-year mortality would differ by T-stage. For instance, perhaps the presence

of ulceration confers a much greater risk of death in advanced deep tumours compared with earlier superficial tumours.

```
library(finalfit)
dependent = "mort_5yr"
explanatory = c("ulcer.factor", "t_stage.factor")
explanatory_multi = c("ulcer.factor*t_stage.factor")
melanoma %>%
  finalfit(dependent, explanatory, explanatory_multi,
            keep_models = TRUE, metrics = TRUE)
```

TABLE 9.14: Multivariable logistic regression: including an interaction term (fit 7).

label	levels	No	Yes	OR (univariable)	OR (multivariable)	OR (multivariable reduced)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-	-	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	3.26 (1.35-8.39, p=0.011)	4.00 (0.18-41.34, p=0.274)
T-stage	T1	52 (32.5)	4 (8.9)	-	-	-
	T2	49 (30.6)	4 (8.9)	1.06 (0.24-4.71, p=0.936)	0.75 (0.16-3.45, p=0.700)	0.94 (0.12-5.97, p=0.949)
	T3	36 (22.5)	15 (33.3)	5.42 (1.80-20.22, p=0.005)	2.96 (0.86-11.96, p=0.098)	3.76 (0.76-20.80, p=0.104)
	T4	23 (14.4)	22 (48.9)	12.43 (4.21-46.26, p<0.001)	5.33 (1.48-22.56, p=0.014)	2.67 (0.12-25.11, p=0.426)
UlcerPresent:T2	Interaction	-	-	-	-	0.57 (0.02-21.55, p=0.730)
UlcerPresent:T3	Interaction	-	-	-	-	0.62 (0.04-17.35, p=0.735)
UlcerPresent:T4	Interaction	-	-	-	-	1.85 (0.09-94.20, p=0.716)

There are no significant interaction terms.

Our final model table is therefore:

```
library(finalfit)
dependent = "mort_5yr"
explanatory = c("ulcer.factor", "age.factor",
               "sex.factor", "t_stage.factor")
explanatory_multi = c("ulcer.factor", "t_stage.factor")
melanoma %>%
  finalfit(dependent, explanatory, explanatory_multi, metrics = TRUE)

## Warning in chisq.test(tab, correct = FALSE): Chi-squared approximation may
## be incorrect
```

9.8.1 Odds ratio plot

```
dependent = "mort_5yr"
explanatory_multi = c("ulcer.factor", "t_stage.factor")
melanoma %>%
  or_plot(dependent, explanatory_multi,
```

TABLE 9.15: Multivariable logistic regression: final model (fit 8).

Dependent: 5-year survival		No	Yes	OR (univariable)	OR (multivariable)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)	-	-
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	3.26 (1.35-8.39, p=0.011)
Age (years)	(0,25]	10 (6.2)	4 (8.9)	-	-
	(25,50]	62 (38.8)	11 (24.4)	0.44 (0.12-1.84, p=0.229)	-
	(50,75]	79 (49.4)	25 (55.6)	0.79 (0.24-3.08, p=0.712)	-
	(75,100]	9 (5.6)	5 (11.1)	1.39 (0.28-7.23, p=0.686)	-
Sex	Female	105 (65.6)	21 (46.7)	-	-
	Male	55 (34.4)	24 (53.3)	2.18 (1.12-4.30, p=0.023)	-
T-stage	T1	52 (32.5)	4 (8.9)	-	-
	T2	49 (30.6)	4 (8.9)	1.06 (0.24-4.71, p=0.936)	0.75 (0.16-3.45, p=0.700)
	T3	36 (22.5)	15 (33.3)	5.42 (1.80-20.22, p=0.005)	2.96 (0.86-11.96, p=0.098)
	T4	23 (14.4)	22 (48.9)	12.43 (4.21-46.26, p<0.001)	5.33 (1.48-22.56, p=0.014)

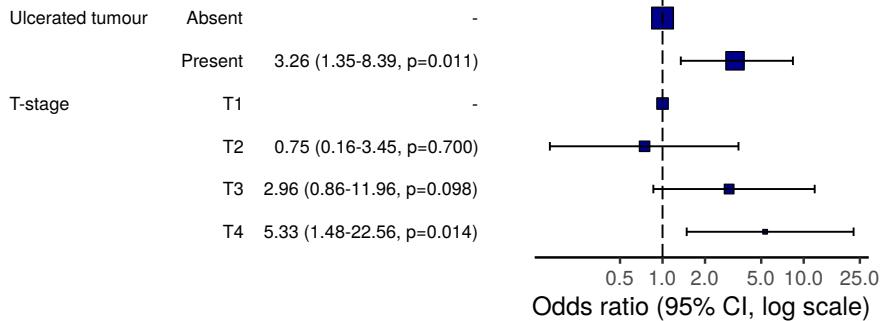
TABLE 9.16: Model metrics: final model (fit 8).

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 184.4, C-statistic = 0.791, H&L = Chi-sq(8) 0.43 (p=1.000)

```
breaks = c(0.5, 1, 2, 5, 10, 25),
table_text_size = 3.5,
title_text_size = 16)
```

```
## Warning: Removed 2 rows containing missing values (geom_errorbarh).
```

5-year survival: OR (95% CI, p-value)

**FIGURE 9.8:** Odds ratio plot

We can conclude that there is evidence of an association between tumour ulceration and 5-year survival which is independent of the tumour depth as captured by T-stage.

9.9 Correlated groups of observations

In our modelling strategy above, we mentioned the incorporation of population stratification if available. What does this mean. Our regression is seeking to capture the characteristics of particular patients. These characteristics are made manifest through the slopes of fitted lines, the estimated coefficients of particular variables. A goal is to estimate these characteristics as precisely as possible. Bias can be introduced when correlations between patients are not accounted for. Correlations may be as simple as being treated within the same hospital. By virtue of this fact, these patients may have commonalities not be captured by the observed variables.

Population characteristics can be incorporated into our models. We may not be interested in capturing and measuring the effects themselves, but ensuring they are accounted for in the analysis.

One approach is to include grouping variables as `random effects`. These may be nested with each other, for example patients within hospitals within countries. These are added in addition to the `fixed effects` we have been dealing with up until now.

These models go under different names including mixed effects model, multilevel model, or hierarchical model.

Other approaches, such as generalized estimating equations are not dealt with here.

9.9.1 Simulate data

Our melanoma dataset doesn't include any higher level structure, so we will simulate this for demonstration purposes. We have just randomly allocated 1 of 4 identifiers to each patient below.

```
# Simulate random hospital identifier
set.seed(1)
melanoma = melanoma %>%
```

```
  mutate(hospital_id = sample(1:4, 205, replace = TRUE))

melanoma = melanoma %>%
  mutate(hospital_id = c(rep(1:3, 50), rep(4, 55)))
```

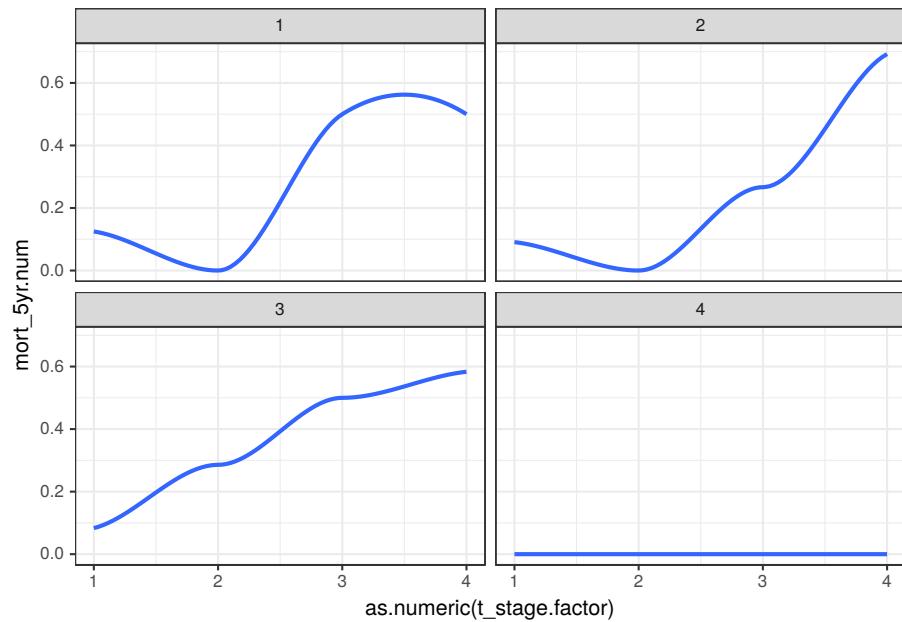
9.9.2 Plot the data

This all might be better in linear regression. Bit difficult to demonstrate with binomial data We will speak in terms of ‘hospitals’ now, but the grouping variable(s) could clearly be anything.

The simplest random effects approach is a ‘random intercept model’. This allows the intercept of fitted lines to vary by hospital. The random intercept model constrains lines to be parallel, in a similar way to the additive models discussed above and in chapter 7.

It is harder to demonstrate with binomial data, but we can stratify the 5 year mortality by t-stage (considered as a continuous variable for this purpose). Note there were no deaths in ‘hospital 4’. We can model this accounting for interhospital variation below.

```
melanoma %>%
  mutate(
    mort_5yr.num = as.numeric(mort_5yr) - 1
  ) %>%
  ggplot(aes(x = as.numeric(t_stage.factor), y = mort_5yr.num)) + #, group = hospital_id
  geom_smooth(method = 'loess', se = FALSE) +
  facet_wrap(~hospital_id)
```



```
# melanoma %>%
#   group_by(hospital_id) %>%
#   count(mort_5yr) %>%
#   spread(mort_5yr, n)
```

9.9.3 Mixed effects models in base R

There are a number of different packages offering mixed effects modelling in R, our preferred is `lme4`.

```
library(lme4)

## Loading required package: Matrix
##
## Attaching package: 'Matrix'
## The following object is masked from 'package:tidyverse':
##     expand
```

```

melanoma %>%
  glmer(mort_5yr ~ t_stage.factor + (1 | hospital_id),
        data = ., family = "binomial") %>%
  summary()

## Generalized linear mixed model fit by maximum likelihood (Laplace
## Approximation) [glmerMod]
## Family: binomial ( logit )
## Formula: mort_5yr ~ t_stage.factor + (1 | hospital_id)
## Data: .
##
##      AIC      BIC  logLik deviance df.resid
##    174.6    191.2   -82.3     164.6     200
##
## Scaled residuals:
##      Min      1Q  Median      3Q     Max
## -1.4507 -0.3930 -0.2891 -0.0640  3.4591
##
## Random effects:
## Groups      Name      Variance Std.Dev.
## hospital_id (Intercept) 2.619     1.618
## Number of obs: 205, groups: hospital_id, 4
##
## Fixed effects:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)      -3.13132   1.01451 -3.087  0.00203 **
## t_stage.factorT2  0.02256   0.74792   0.030  0.97593
## t_stage.factorT3  1.82349   0.62920   2.898  0.00375 **
## t_stage.factorT4  2.61190   0.62806   4.159  3.2e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##          (Intr) t_s.T2 t_s.T3
## t_stg.fctT2 -0.365
## t_stg.fctT3 -0.454  0.591
## t_stg.fctT4 -0.459  0.590  0.718

```

The base R output is similar to `glm()`. It includes the standard deviation on the random effects intercept as well. Meaning the variation between hospitals being captured by the model.

The output can be examined using `tidy()` and `glance()` functions as above.

We find it more straightforward to use `finalfit`.

```

dependent = "mort_5yr"
explanatory = "t_stage.factor"
random_effect = "hospital_id" # Is the same as -
random_effect = "(1 | hospital_id)"
melanoma %>%
  finalfit(dependent, explanatory,
            random_effect = random_effect,
            metrics = TRUE)

```

We can incorporate our (made-up) hospital identifier into our final model from above. Using `keep_models = TRUE`, we can compare univariable, multivariable and mixed effects models.

```

library(finalfit)
dependent = "mort_5yr"
explanatory = c("ulcer.factor", "age.factor",
               "sex.factor", "t_stage.factor")
explanatory_multi = c("ulcer.factor", "t_stage.factor")
random_effect = "hospital_id"
melanoma %>%
  finalfit(dependent, explanatory, explanatory_multi, random_effect,
            keep_models = TRUE,
            metrics = TRUE)

```

```

## Warning in chisq.test(tab, correct = FALSE): Chi-squared approximation may
## be incorrect

```

TABLE 9.17: Multilevel (mixed effects) logistic regression.

Dependent: 5-year survival		No	Yes	OR (univariable)	OR (multivariable)	OR (multivariable reduced)	OR (multilevel)
Ulcerated tumour	Absent	105 (65.6)	10 (22.2)				
	Present	55 (34.4)	35 (77.8)	6.68 (3.18-15.18, p<0.001)	3.06 (1.25-7.93, p=0.017)	3.26 (1.35-8.39, p=0.011)	2.49 (0.94-6.59, p=0.065)
Age (years)	(0,25]	10 (6.2)	4 (8.9)				
	(25,50]	62 (38.8)	11 (24.4)	0.44 (0.12-1.84, p=0.229)	0.37 (0.08-1.80, p=0.197)	-	-
	(50,75]	79 (49.4)	25 (55.6)	0.79 (0.24-3.08, p=0.712)	0.60 (0.15-2.65, p=0.469)	-	-
	(75,100]	9 (5.6)	5 (11.1)	1.39 (0.28-7.23, p=0.686)	0.61 (0.09-4.04, p=0.599)	-	-
Sex	Female	105 (65.6)	21 (46.7)				
	Male	55 (34.4)	24 (53.3)	2.18 (1.12-4.30, p=0.023)	1.21 (0.54-2.68, p=0.633)	-	-
T-stage	T1	52 (32.5)	4 (8.9)				
	T2	49 (30.6)	4 (8.9)	1.06 (0.24-4.71, p=0.936)	0.74 (0.15-3.50, p=0.697)	0.75 (0.16-3.45, p=0.700)	0.83 (0.18-3.73, p=0.807)
	T3	36 (22.5)	15 (33.3)	5.42 (1.80-20.22, p=0.005)	2.91 (0.84-11.82, p=0.106)	2.96 (0.86-11.96, p=0.098)	3.83 (1.00-14.70, p=0.051)
	T4	23 (14.4)	22 (48.9)	12.43 (4.21-46.26, p<0.001)	5.38 (1.43-23.52, p=0.016)	5.33 (1.48-22.56, p=0.014)	7.03 (1.71-28.86, p=0.007)

TABLE 9.18: Model metrics: multilevel (mixed effects) logistic regression.

Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 190, C-statistic = 0.802,
H&L = Chi-sq(8) 13.87 (p=0.085)
Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 184.4, C-statistic = 0.791, H&L = Chi-sq(8) 0.43 (p=1.000)
Number in model = 205, Number of groups = 4, AIC = 173.2, C-statistic = 0.866

As can be seen, incorporating the (made-up) hospital identifier has altered our coefficients. It has also improved the model discrimination with a c-statistic of 0.830 from 0.802. Note that the AIC should not be used to compare mixed effects models estimated in this way with `glm()`' models (the former uses a restricted maximum likelihood [REML] approach by default, while `glmer()`' uses maximum likelihood).

Random slope models are an extension of the random intercept model. Here the gradient of the response to a particular variable is allowed to vary by hospital. For example, this can be included using `random_effect = "(thickness | hospital_id)"` where the gradient of the continuous variable tumour thickness was allow to vary by hospital.

As models get more complex, care has to be taken to ensure the underlying data is understood and assumptions are checked.

Mixed effects modelling is a book in itself and the purpose here is to introduce the concept and provide some approaches its incorporation. Clearly much is written else where for those who are enthusiastic to learn more.

Don't include diagnostic plots and collinearity testing

```
library(ggfortify)
dependent = "mort_5yr"
explanatory_multi = c("ulcer.factor", "t_stage.factor")
melanoma %>%
  glmmulti(dependent, explanatory) %>%
  autoplot(which=1:6)

melanoma %>%
  glmmulti(dependent, explanatory) %>%
  car::vif()
```

Exercises need sorted

9.9.4 Exercise

Repeat this for all the variables contained within the data, particular:

`t_stage.factor`, `age`, `ulcer.factor`, `thickness` and `age.factor`.

Write their odds ratios and 95% confidence intervals down for the next section!

Congratulations on building your first regression model in R!

10

Time-to-event data and survival

The reports of my death have been greatly exaggerated.
Mark Twain

In healthcare, we deal with a lot of binary outcomes. Death yes/no, disease recurrence yes/no, for instance. These outcomes are often easily analysed using binary logistic regression as described in the previous chapter.

When the time taken for the outcome to occur is important, we need a different approach. For instance, in patients with cancer, the time taken until recurrence of the cancer is often just as important as the fact it has recurred.

10.1 The Question

We will again use the classic “Survival from Malignant Melanoma” dataset included in the `boot` package which we have used in REF. The data consist of measurements made on patients with malignant melanoma. Each patient had their tumour removed by surgery at the Department of Plastic Surgery, University Hospital of Odense, Denmark during the period 1962 to 1977.

We are interested in the association between tumour ulceration and survival after surgery.

10.2 Get and check data

```
library(tidyverse)
library(finalfit)
melanoma = boot::melanoma #F1 here for help page with data dictionary
```

```
glimpse(melanoma)
missing_glimpse(melanoma)
ff_glimpse(melanoma)
```

As was seen before, all variables are coded as numeric and some need recoding to factors. This is done below for those we are interested in.

10.3 Death status

`status` is the patients status at the end of the study.

- 1 indicates that they had died from melanoma;
- 2 indicates that they were still alive and;
- 3 indicates that they had died from causes unrelated to their melanoma.

There are three options for coding this.

- Overall survival: considering all-cause mortality, comparing 2 (alive) with 1 (died melanoma)/3 (died other);
- Cause-specific survival: considering disease-specific mortality comparing 2 (alive)/3 (died other) with 1 (died melanoma);
- Competing risks: comparing 2 (alive) with 1 (died melanoma) accounting for 3 (died other); see more below.

10.4 Time and censoring

`time` is the number of days from surgery until either the occurrence of the event (death) or the last time the patient was known to be alive. For instance, if a patient had surgery and was seen to be well in a clinic 30 days later, but there had been no contact since, then the patient's status would be considered 30 days. This patient is censored from the analysis at day 30, an important feature of time-to-event analyses.

10.5 Recode the data

```
library(dplyr)
library(forcats)
melanoma = melanoma %>%
  mutate(
    # Overall survival
    status_os = if_else(status == 2, 0, # "still alive"
                        1), # "died of melanoma" or "died of other causes"

    # Disease-specific survival
    status_dss = if_else(status == 2, 0, # "still alive"
                          if_else(status == 1, 1, # "died of melanoma"
                                 0)), # "died of other causes is censored"

    # Competing risks regression
    status_crr = if_else(status == 2, 0, # "still alive"
                          if_else(status == 1, 1, # "died of melanoma"
                                 2)), # "died of other causes"

    # Label and recode other variables
    age = ff_label(age, "Age (years)'), # ff_label table friendly labels
    thickness = ff_label(thickness, "Tumour thickness (mm)'),
    sex = factor(sex) %>%
      fct_recode("Male" = "1",
                "Female" = "0") %>%
      ff_label("Sex"),
    ulcer = factor(ulcer) %>%
```

```
fct_recode("No" = "0",
           "Yes" = "1") %>%
  ff_label("Ulcerated tumour")
)
```

10.6 Kaplan Meier survival estimator

We will use the excellent `survival` package to produce the Kaplan Meier (KM) survival estimator. This is a non-parametric statistic used to estimate the survival function from time-to-event data.

```
library(survival)

survival_object = melanoma %$%
  Surv(time, status_os)

# Explore:
head(survival_object) # + marks censoring, in this case "Alive"

## [1] 10   30   35+  99   185  204

# Expressing time in years
survival_object = melanoma %$%
  Surv(time/365, status_os)
```

10.6.1 KM analysis for whole cohort

10.6.2 Model

The survival object is the first step to performing univariable and multivariable survival analyses.

If you want to plot survival stratified by a single grouping variable, you can substitute “`survival_object ~ 1`” by “`survival_object ~ factor`”

```
# Overall survival in whole cohort
my_survfit = survfit(survival_object ~ 1, data = melanoma)
my_survfit # 205 patients, 71 events

## Call: survfit(formula = survival_object ~ 1, data = melanoma)
##
##      n  events  median 0.95LCL 0.95UCL
##  205.00    71.00      NA     9.15      NA
```

10.6.3 Life table

A life table is the tabular form of a KM plot, which you may be familiar with. It shows survival as a proportion, together with confidence limits. The whole table is shown with, `summary(my_survfit)`.

```
summary(my_survfit, times = c(0, 1, 2, 3, 4, 5))

## Call: survfit(formula = survival_object ~ 1, data = melanoma)
##
##   time n.risk n.event survival std.err lower 95% CI upper 95% CI
##   0     205      0    1.000  0.0000    1.000    1.000
##   1     193      11   0.946  0.0158    0.916    0.978
##   2     183      10   0.897  0.0213    0.856    0.940
##   3     167      16   0.819  0.0270    0.767    0.873
##   4     160       7   0.784  0.0288    0.730    0.843
##   5     122      10   0.732  0.0313    0.673    0.796

# 5 year overall survival is 73%
```

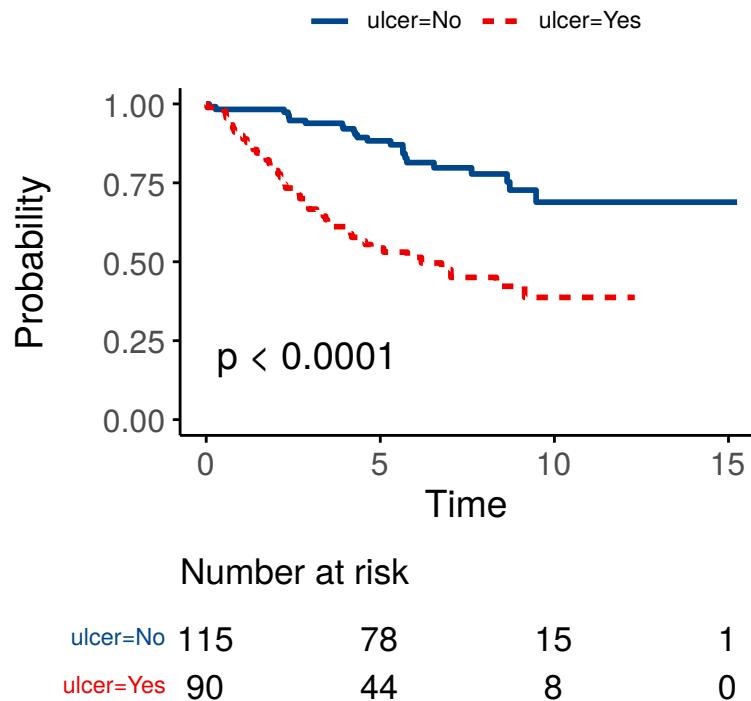
10.7 Kaplan Meier plot

We can plot survival curves using the `finalfit` wrapper for the package `excellent` package `survminer`. There are numerous options available on the help page. You should always include a number-at-risk table under these plots as it is essential for interpretation.

As can be seen, the probability of dying is much greater if the tumour was ulcerated, compared to those that were not ulcerated.

```
dependent_os = "Surv(time/365, status_os)"
explanatory = c("ulcer")

melanoma %>%
  surv_plot(dependent_os, explanatory, pval = TRUE)
```



10.8 Cox-proportional hazards regression

CPH regression can be performed using the all-in-one `finalfit()` function. It produces a table containing counts (proportions) for factors, mean (SD) for continuous variables and a univariable and multivariable CPH regression.

10.8.1 Univariable and multivariable models

```
dependent_os = "Surv(time, status_os)"
dependent_dss = "Surv(time, status_dss)"
dependent_crr = "Surv(time, status_crr)"
explanatory = c("age", "sex", "thickness", "ulcer")

melanoma %>%
  finalfit(dependent_os, explanatory)
```

```
dependent_os = "Surv(time, status_os)"
dependent_dss = "Surv(time, status_dss)"
dependent_crr = "Surv(time, status_crr)"
explanatory = c("age", "sex", "thickness", "ulcer")

melanoma %>%
  finalfit(dependent_os, explanatory) %>%
  mykable(caption = "Univariable and multivariable Cox Proportional Hazards: Overall survival f")
```

TABLE 10.1: Univariable and multivariable Cox Proportional Hazards: Overall survival following surgery for melanoma by patient and tumour variables.

Dependent: Surv(time, status_os)		all	HR (univariable)	HR (multivariable)
Age (years)	Mean (SD)	52.5 (16.7)	1.03 (1.01-1.05, p<0.001)	1.02 (1.01-1.04, p=0.005)
Sex	Female	126 (61.5)	-	-
	Male	79 (38.5)	1.93 (1.21-3.07, p=0.006)	1.51 (0.94-2.42, p=0.085)
Tumour thickness (mm)	Mean (SD)	2.9 (3.0)	1.16 (1.10-1.23, p<0.001)	1.10 (1.03-1.18, p=0.004)
Ulcerated tumour	No	115 (56.1)	-	-
	Yes	90 (43.9)	3.52 (2.14-5.80, p<0.001)	2.59 (1.53-4.38, p<0.001)

The labelling of the final table can be easily adjusted as desired.

```
melanoma %>%
  finalfit(dependent_os, explanatory, add_dependent_label = FALSE) %>%
  rename("Overall survival" = label) %>%
  rename(" " = levels) %>%
  rename(" " = all)
```

TABLE 10.2: Univariable and multivariable Cox Proportional Hazards: Overall survival following surgery for melanoma by patient and tumour variables (tidied).

Overall survival			HR (univariable)	HR (multivariable)
Age (years)	Mean (SD)	52.5 (16.7)	1.03 (1.01-1.05, p<0.001)	1.02 (1.01-1.04, p=0.005)
Sex	Female	126 (61.5)	-	-
	Male	79 (38.5)	1.93 (1.21-3.07, p=0.006)	1.51 (0.94-2.42, p=0.085)
Tumour thickness (mm)	Mean (SD)	2.9 (3.0)	1.16 (1.10-1.23, p<0.001)	1.10 (1.03-1.18, p=0.004)
Ulcerated tumour	No	115 (56.1)	-	-
	Yes	90 (43.9)	3.52 (2.14-5.80, p<0.001)	2.59 (1.53-4.38, p<0.001)

10.8.2 Reduced model

If you are using a backwards selection approach or similar, a reduced model can be directly specified and compared. The full model can be kept or dropped.

```
explanatory_multi = c("age", "thickness", "ulcer")
melanoma %>%
  finalfit(dependent_os, explanatory,
           explanatory_multi, keep_models = TRUE)
```

TABLE 10.3: Cox Proportional Hazards: Overall survival following surgery for melanoma with reduced model.

Dependent: Surv(time, status_os)		all	HR (univariable)	HR (multivariable)	HR (multivariable reduced)
Age (years)	Mean (SD)	52.5 (16.7)	1.03 (1.01-1.05, p<0.001)	1.02 (1.01-1.04, p=0.005)	1.02 (1.01-1.04, p=0.003)
Sex	Female	126 (61.5)	-	-	-
	Male	79 (38.5)	1.93 (1.21-3.07, p=0.006)	1.51 (0.94-2.42, p=0.085)	-
Tumour thickness (mm)	Mean (SD)	2.9 (3.0)	1.16 (1.10-1.23, p<0.001)	1.10 (1.03-1.18, p=0.004)	1.10 (1.03-1.18, p=0.003)
Ulcerated tumour	No	115 (56.1)	-	-	-
	Yes	90 (43.9)	3.52 (2.14-5.80, p<0.001)	2.59 (1.53-4.38, p<0.001)	2.72 (1.61-4.57, p<0.001)

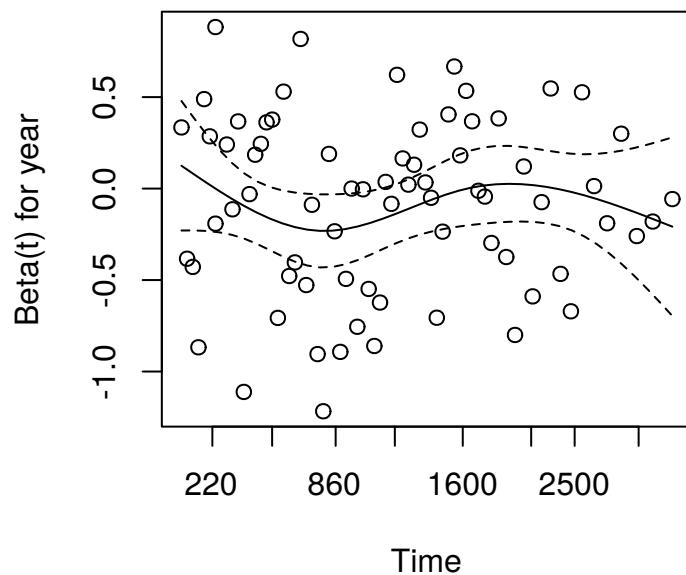
10.8.3 Testing for proportional hazards

An assumption of CPH regression is that the hazard (think risk) associated with a particular variable does not change over time. For example, is the magnitude of the increase in risk of death associated with tumour ulceration the same in the early post-operative period as it is in later years?

The `cox.zph()` function from the survival package allows us to test this assumption for each variable. The plot of scaled Schoenfeld residuals should be a horizontal line. The included hypothesis test

identifies whether the gradient differs from zero for each variable. No variable significantly differs from zero at the 5% significance level.

```
explanatory = c("age", "sex", "thickness", "ulcer", "year")
melanoma %>%
  coxphmulti(dependent_os, explanatory) %>%
  cox.zph() %>%
  {zph_result <- .} %>%
  plot(var=5)
```



```
zph_result
```

	rho	chisq	p
## age	0.1633	2.4544	0.1172
## sexMale	-0.0781	0.4473	0.5036
## thickness	-0.1493	1.3492	0.2454
## ulcerYes	-0.2044	2.8256	0.0928
## year	0.0195	0.0284	0.8663
## GLOBAL	NA	8.4695	0.1322

10.8.4 Stratified models

One approach to dealing with a violation of the proportional hazards assumption is to stratify by that variable. Including a `strata()` term will result in a separate baseline hazard function being fit for each level in the stratification variable. It will be no longer possible to make direct inference on the effect associated with that variable.

This can be incorporated directly into the explanatory variable list.

```
explanatory = c("age", "sex", "ulcer", "thickness",
               "strata(year)")
melanoma %>%
  finalfit(dependent_os, explanatory)
```

TABLE 10.4: Cox Proportional Hazards: Overall survival following surgery for melanoma stratified by year of surgery.

Dependent: Surv(time, status_os)		all	HR (univariable)	HR (multivariable)
Age (years)	Mean (SD)	52.5 (16.7)	1.03 (1.01-1.05, p<0.001)	1.03 (1.01-1.05, p=0.002)
Sex	Female	126 (61.5)	-	-
	Male	79 (38.5)	1.93 (1.21-3.07, p=0.006)	1.75 (1.06-2.87, p=0.027)
Ulcerated tumour	No	115 (56.1)	-	-
	Yes	90 (43.9)	3.52 (2.14-5.80, p<0.001)	2.61 (1.47-4.63, p=0.001)
Tumour thickness (mm)	Mean (SD)	2.9 (3.0)	1.16 (1.10-1.23, p<0.001)	1.08 (1.01-1.16, p=0.027)

10.8.5 Correlated groups of observations

As a general rule, you should always try to account for any higher structure in your data within the model. For instance, patients may be clustered within particular hospitals.

There are two broad approaches to dealing with correlated groups of observations.

A `cluster()` term implies a generalised estimating equations (GEE) approach. Here, a standard CPH model is fitted but the standard errors of the estimated hazard ratios are adjusted to account for correlations.

A `frailty()` term implies a mixed effects model, where specific random effects term(s) are directly incorporated into the model.

Both approaches achieve the same goal in different ways. Volumes have been written on GEE vs mixed effects models. We favour the latter approach because of its flexibility and our preference for mixed effects modelling in generalised linear modelling. Note `cluster()` and `frailty()` terms cannot be combined in the same model.

```
# Simulate random hospital identifier
melanoma = melanoma %>%
  mutate(hospital_id = c(rep(1:10, 20), rep(11, 5)))

# Cluster model
explanatory = c("age", "sex", "thickness", "ulcer",
  "cluster(hospital_id)")
melanoma %>%
  finalfit(dependent_os, explanatory) %>%
  mykable(caption = "Cox Proportional Hazards: Overall survival following surgery for melanoma")

##   Dependent: Surv(time, status_os)           all
## 1                               Age (years) Mean (SD) 52.5 (16.7)
## 2                               Sex     Female 126 (61.5)
## 3                               Male    79 (38.5)
## 4       Tumour thickness (mm) Mean (SD)  2.9 (3.0)
## 5       Ulcerated tumour      No    115 (56.1)
## 6                   Yes    90 (43.9)
##             HR (univariable)      HR (multivariable)
## 1 1.03 (1.01-1.05, p<0.001) 1.02 (1.00-1.04, p=0.016)
## 2          -                  -
## 3 1.93 (1.21-3.07, p=0.006) 1.51 (1.10-2.08, p=0.011)
## 4 1.16 (1.10-1.23, p<0.001) 1.10 (1.04-1.17, p<0.001)
## 5          -                  -
## 6 3.52 (2.14-5.80, p<0.001) 2.59 (1.61-4.16, p<0.001)

# Frailty model
explanatory = c("age", "sex", "thickness", "ulcer",
  "frailty(hospital_id)")
melanoma %>%
  finalfit(dependent_os, explanatory)
```

The `frailty()` method here is being superseded by the `coxme` package, and we look forward to incorporating this in the future.

TABLE 10.5: Cox Proportional Hazards: Overall survival following surgery for melanoma (frailty model)

Dependent: Surv(time, status_os)		all	HR (univariable)	HR (multivariable)
Age (years)	Mean (SD)	52.5 (16.7)	1.03 (1.01-1.05, p<0.001)	1.02 (1.01-1.04, p=0.005)
Sex	Female	126 (61.5)	-	-
	Male	79 (38.5)	1.93 (1.21-3.07, p=0.006)	1.51 (0.94-2.42, p=0.085)
Tumour thickness (mm)	Mean (SD)	2.9 (3.0)	1.16 (1.10-1.23, p<0.001)	1.10 (1.03-1.18, p=0.004)
Ulcerated tumour	No	115 (56.1)	-	-
	Yes	90 (43.9)	3.52 (2.14-5.80, p<0.001)	2.59 (1.53-4.38, p<0.001)

10.8.6 Hazard ratio plot

A plot of any of the above models can be easily produced.

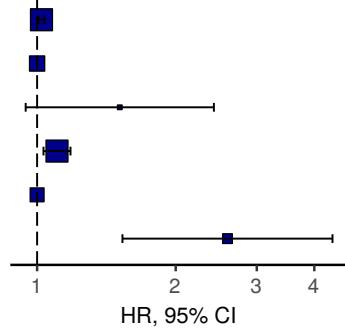
```
melanoma %>%
  hr_plot(dependent_os, explanatory)
```

```
library(ggplot2)
melanoma %>%
  hr_plot(dependent_os, explanatory, table_text_size = 3.5,
         title_text_size = 16,
         plot_opts=list(xlab("HR, 95% CI"), theme(axis.title = element_text(size=12))))
```

```
## Warning: Removed 2 rows containing missing values (geom_errorbarh).
```

Survival: HR (95% CI, p-value)

Age (years)	-	1.02 (1.01-1.04, p=0.005)
Sex	Female	-
	Male	1.51 (0.94-2.42, p=0.085)
Tumour thickness (mm)	-	1.10 (1.03-1.18, p=0.004)
Ulcerated tumour	No	-
	Yes	2.59 (1.53-4.38, p<0.001)

**FIGURE 10.1:** Hazard ratio plot

10.9 Competing risks regression

Competing-risks regression is an alternative to CPH regression. It can be useful if the outcome of interest may not be able to occur simply because something else (like death) has happened first. For instance, in our example it is obviously not possible for a patient to die from melanoma if they have died from another disease first. By simply looking at cause-specific mortality (deaths from melanoma) and considering other deaths as censored, bias may result in estimates of the influence of predictors.

The approach by Fine and Gray is one option for dealing with this. It is implemented in the package `cprsk`. The `crr()` syntax differs from `survival::coxph()` but `finalfit` brings these together.

It uses the `finalfit::ff_merge()` function, which can join any number of models together.

```
explanatory = c("age", "sex", "thickness", "ulcer")
dependent_dss = "Surv(time, status_dss)"
dependent_crr = "Surv(time, status_crr)"

melanoma %>%
  # Summary table
  summary_factorlist(dependent_dss, explanatory,
                     column = TRUE, fit_id = TRUE) %>%
  # CPH univariable
  ff_merge(
    melanoma %>%
      coxphmulti(dependent_dss, explanatory) %>%
      fit2df(estimate_suffix = " (DSS CPH univariable)")
  ) %>%
  # CPH multivariable
  ff_merge(
    melanoma %>%
      coxphmulti(dependent_dss, explanatory) %>%
      fit2df(estimate_suffix = " (DSS CPH multivariable)")
  ) %>%
  # Fine and Gray competing risks regression
```

```

ff_merge(
  melanoma %>%
    crrmulti(dependent_crr, explanatory) %>%
    fit2df(estimate_suffix = " (competing risks multivariable)")
) %>%

select(-fit_id, -index) %>%
dependent_label(melanoma, "Survival")

```

Dependent variable is a survival object

TABLE 10.6: Cox Proportional Hazards and competing risks regression combined

Dependent: Survival		all	HR (DSS CPH univariable)	HR (DSS CPH multivariable)	HR (competing risks multivariable)
Age (years)	Mean (SD)	52.5 (16.7)	1.01 (1.00-1.03, p=0.141)	1.01 (1.00-1.03, p=0.141)	1.01 (0.99-1.02, p=0.520)
Sex	Female	126 (61.5)	-	-	-
	Male	79 (38.5)	1.54 (0.91-2.60, p=0.106)	1.54 (0.91-2.60, p=0.106)	1.50 (0.87-2.57, p=0.140)
Tumour thickness (mm)	Mean (SD)	2.9 (3.0)	1.12 (1.04-1.20, p=0.004)	1.12 (1.04-1.20, p=0.004)	1.09 (1.01-1.18, p=0.019)
Ulcerated tumour	No	115 (56.1)	-	-	-
	Yes	90 (43.9)	3.20 (1.75-5.88, p<0.001)	3.20 (1.75-5.88, p<0.001)	3.09 (1.71-5.60, p<0.001)

10.10 Summary

So here we have various aspects of time-to-event analysis which is commonly used when looking at survival. There are many other applications, some which may not be obvious: for instance we use CPH for modelling length of stay in hospital.

Stratification can be used to deal with non-proportional hazards in a particular variable.

Hierarchical structure in your data can be accommodated with cluster or frailty (random effects) terms.

Competing risks regression may be useful if your outcome is in competition with another, such as all-cause death, but is currently limited in its ability to accommodate hierarchical structures.

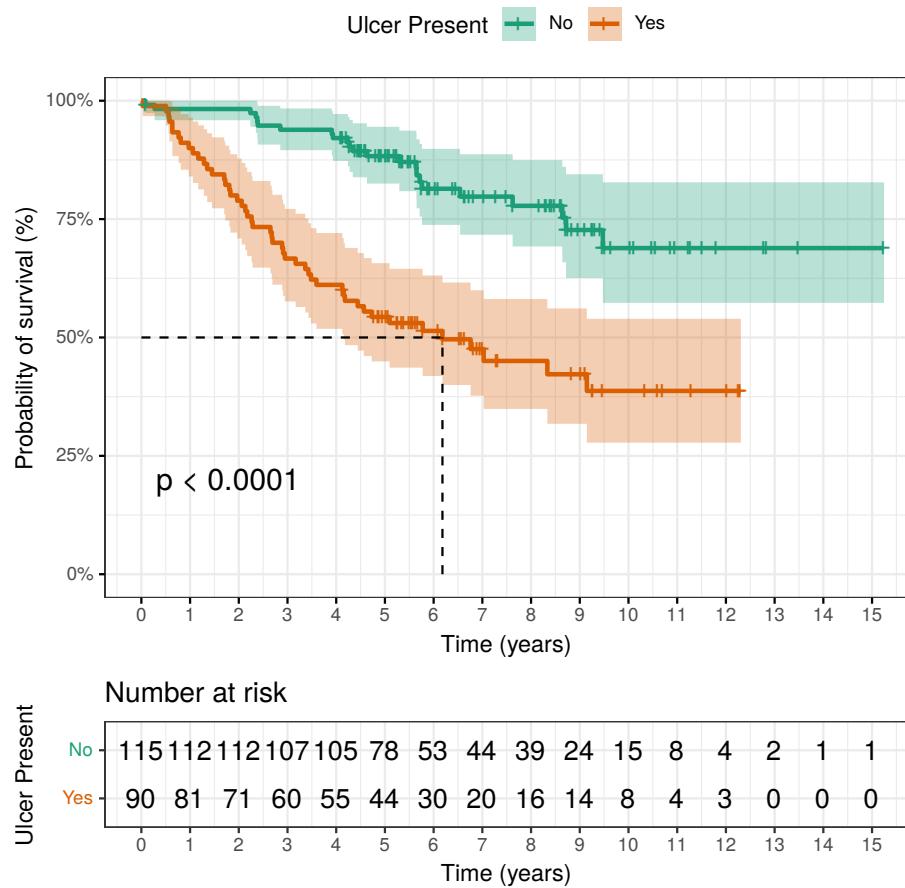
10.11 Exercise

Using the above scripts, perform a univariable Kaplan Meier analysis to determine if `ulcer` influences overall survival. Hint: `survival_object ~ ulcer`.

Try modifying the plot produced (see Help for `ggsurvplot`). For example:

- Add in a medial survival lines: `surv.median.line="hv"`
- Alter the plot legend: `legend.title = "Ulcer Present", legend.labs = c("No", "Yes")`
- Change the y-axis to a percentage: `ylab = "Probability of survival (%)", surv.scale = "percent"`
- Display follow-up up to 10 years, and change the scale to 1 year: `xlim = c(0,10), break.time.by = 1`

```
## Loading required package: ggpubr
## Loading required package: magrittr
##
## Attaching package: 'magrittr'
## The following object is masked from 'package:purrr':
##     set_names
## The following object is masked from 'package:tidyverse':
##     extract
```



10.11.1 Exercise

Create a new CPH model, but now include the variable `thickness` as a variable. How would you interpret the output? Is it an independent predictor of overall survival in this model? Are CPH assumptions maintained?

10.12 Dates in R

10.12.1 Converting dates to survival time

In the melanoma example dataset, we already had the time in a convenient format for survival analysis - survival time in days since the operation. This section shows how to convert dates into “days from event”. First we will generate a dummy operation date and censoring date based on the melanoma data.

```
library(lubridate)
first_date = ymd("1966-01-01")           # let's create made-up dates for the operations
last_date = first_date + days(nrow(melanoma)-1) # assume tone every day from 1-Jan 1966
operation_date = seq(from = first_date, to = last_date, by = "1 day") # create dates

melanoma$operation_date = operation_date # add the created sequence to melanoma dataset

# interval(anaesthetic_start_time, into_theatre_time) %>%
#   time_length(unit="minutes"),
```

Now we will to create a ‘censoring’ date by adding `time` from the melanoma dataset to our made up operation date.

Remember the censoring date is either when an event occurred (e.g. death) or the last known alive status of the patient.

```
melanoma = melanoma %>%
  mutate(censoring_date = operation_date + days(time))

# (Same as doing):
melanoma$censoring_date = melanoma$operation_date + days(melanoma$time)
```

Now consider if we only had the `operation` date and `censoring` date. We want to create the `time` variable.

```
melanoma = melanoma %>%
  mutate(time_days = censoring_date - operation_date)
```

The `Surv()` function expects a number (`numeric` variable), rather than a `date` object, so we'll convert it:

```
# Surv(melanoma$time_days, melanoma$status==1) # this doesn't work

melanoma %>%
  mutate(time_days_numeric = as.numeric(time_days)) ->
  melanoma

#survival_object = Surv(melanoma$time_days_numeric, melanoma$status.factor == "Died") # this works
```

10.13 Solutions

9.2.2

```
# Fit survival model
my_survfit.solution = survfit(survival_object ~ ulcer, data = melanoma)

# Show results
my_survfit.solution
summary(my_survfit.solution, times=c(0,1,2,3,4,5))

# Plot results
my_survplot.solution = ggsurvplot(my_survfit.solution,
                                   data = melanoma,
                                   palette = 'Dark2',
                                   risk.table = TRUE,
                                   ggtheme = theme_bw(),
                                   conf.int = TRUE,
                                   pval=TRUE,

                                   # Add in a medial survival line.
                                   surv.median.line="hv",

                                   # Alter the plot legend (change the names)
                                   legend.title = "Ulcer Present",
                                   legend.labs = c("No", "Yes"),

                                   # Change the y-axis to a percentage
                                   ylab = "Probability of survival (%)",
                                   surv.scale = "percent",

                                   # Display follow-up up to 10 years, and change the scale to 1 year
```

```
xlab = "Time (years)",  
# present narrower X axis, but not affect survival estimates.  
xlim = c(0,10),  
# break X axis in time intervals by 1 year  
break.time.by = 1)  
  
my_survplot.solution
```

9.3.3

```
# Fit model  
# my_hazard = coxph(survival_object~sex.factor+ulcer+age.factor+thickness, data=melanoma)  
# summary(my_hazard)  
  
# Melanoma thickness has a HR 1.12 (1.04 to 1.21).  
# This is interpreted as a 12% increase in the  
# risk of death at any time for each 1 mm increase in thickness.  
  
# Check assumptions  
# ph = cox.zph(my_hazard)  
# ph  
# # GLOBAL shows no overall violation of assumptions.  
# Plot Schoenfield residuals to evaluate PH  
# plot(ph, var=6)
```



Part III

Workflow



Throughout this book we have tried to provide the most efficient approaches data analysis using R. In this section, we will provide workflows, or ways-of-working, which maximise efficiency, incorporate reporting of results within analyses, make exporting of tables and plots easy, and keep data safe, secured and backed up.

We also include a section on dealing with missing data in R. Something that we both feel strongly about and which is often poorly described and dealt with in academic publishing.



11

Notebooks and Markdown

11.1 What is a Notebook?

R is all-powerful for the manipulation, visualisation and analysis of data. What is often be under-appreciated is the flexibility in which analyses can be exported or reported.

For instance, a full scientific paper, industry report, or monthly update can be easily written to accommodate a varying underlying dataset, and all tables and plots will be updated automatically.

This idea can be extended to a workflow in which all analyses are performed primarily within a document which doubles as the final report.

Enter “Data Notebooks”! Notebooks are documents which combine code and rich text elements, such as headings, paragraphs and links, in one document. They combine analysis and reporting in one human-readable document to provide an intuitive interface between the researcher and their analysis (Figure 11.1). This sometimes gets called “literate programming”, given the resulting logical structure of information can be easily read in the manner a human would read a book.

In our own work, we have now moved to doing most of our analyses in a Notebook file, rather than using a “script” file.

You may have guessed, but this whole book is written in this way.

Some of the advantages of the Notebook format are:

- code and output are adjacent to each other, so you are not constantly switching between “panes”;

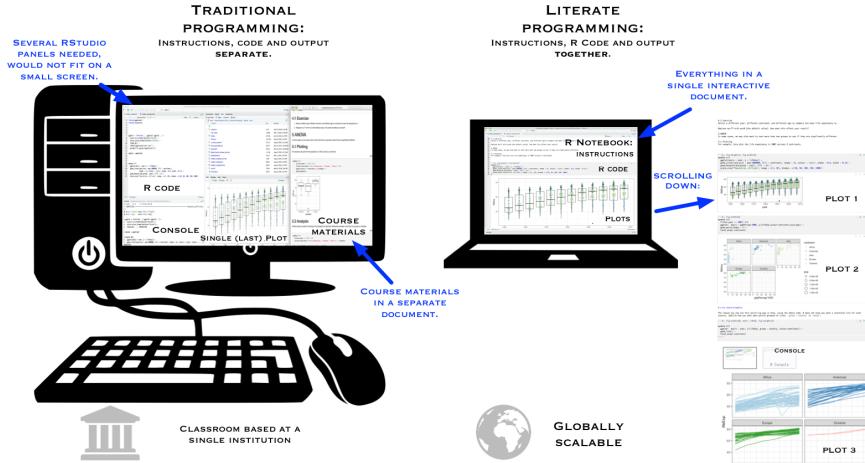


FIGURE 11.1: Traditional versus literate programming using Notebooks.

- easier to work on smaller screen;
- fully formatted text elements can be included in documents;
- documentation and reporting can be done beside the code;
- the code itself can be outputted or hidden;
- the code is not limited to R - you can include Python, SQL etc.;
- facilitate collaboration by easily sharing human-readable analysis documents;
- can be outputted in a number of formats including HTML (web page), PDF, and Microsoft Word;
- output can be extended to other formats such as presentations;
- training/learning may be easier as course materials, examples, and student notes are all in same document.

11.2 What is Markdown?

Markdown is a lightweight language that can be used to write fully-formated documents. It is plain-text and uses a simple set of rules to produce rather sophisticated output - we love it!

It is easy to format headings, bold text, italics etc. Within RStudio there is a Quick Reference guide (Figure 11.2) and links to the RStudio cheatsheets¹ can be found in the Help drop-down menu.

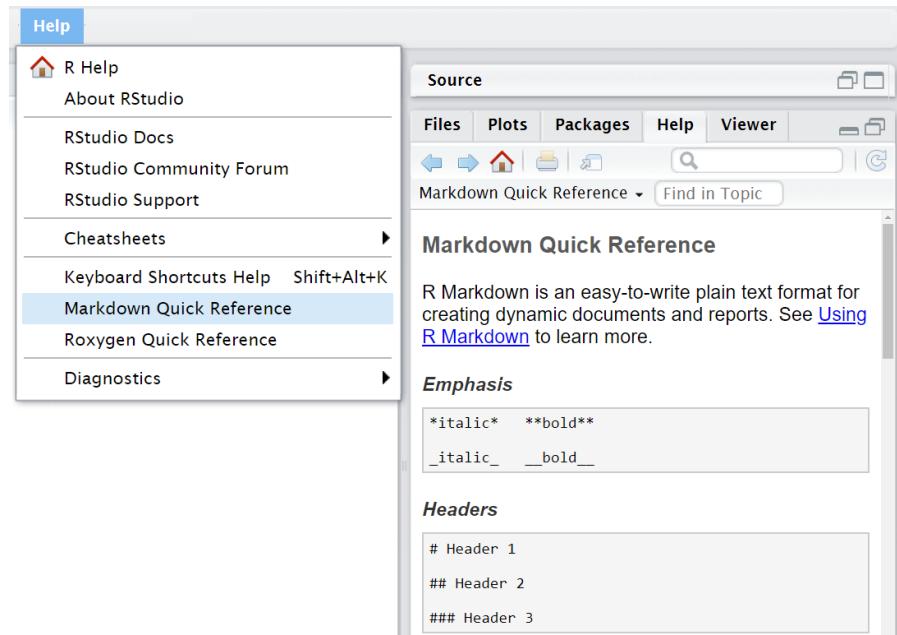


FIGURE 11.2: RStudio Markdown quick reference guide.

11.3 What is the difference between a Notebook and a Markdown file?

A Notebook and an R Markdown file are essentially the same. They are documents in which markdown is combined with code to produce a fully-formatted final document.

An important difference is in the execution of code. In R Markdown, when the file is `Run`, all the elements (chunks) are also run. In a Notebook, when the file is `Previewed`, no code is re-run, only

¹<https://www.rstudio.com/resources/cheatsheets>

that which has already been run in and is present in the document is included in the output. Also, in the Notebook behind-the-scenes file (`.nb`) all the code is always included. Something to watch out for if your code contains sensitive information, such as a password (which it never should!). This is not the case in the R Markdown `.tex` file.

11.4 Notebook vs HTML vs PDF vs Word

In RStudio, a Notebook can be created by going to File -> New File -> R Notebook.

Alternatively, you can create a Markdown file File -> New File -> R Markdown... .

Don't worry which you choose. As mentioned above, they are essentially the same thing but just come with different options. It is easy to switch from a Notebook to a Markdown file if you wish to create a PDF or Word document for instance.

If you are primarily doing analysis in the Notebook environment, choose Notebook. If you are primarily creating a PDF or Word document, choose R Markdown file.

11.5 The anatomy of a Notebook / R Markdown file

When you create a file, a helpful template is provided to get you started. Figure 11.3 shows the essential elements of a Notebook file and how these translate to what is seen in the `HTML` preview.

11.5.1 YAML header

Every Notebook and Markdown files requires a “YAML header”. Where do they get these terms you ask? Originally YAML was said to mean Yet Another Markup Language, referencing its purpose as a markup language. It was later repurposed as YAML Ain’t Markup Language, a recursive acronym, to distinguish its purpose as data-oriented rather than document markup (thank you Wikipedia).

This is simply where many of the settings/options for file creation are placed. In RStudio, these often update automatically as different settings are invoked in the Options menu.

11.5.2 R code chunks

R code within a Notebook or Markdown file can included in two ways:

- in-line: e.g. The total number of oranges was `R sum(fruit$oranges);`
- as a “chunk”.

R chunks are flexible, come with lots of options, and you will soon get into the way of using them.

Figure 11.3 shows how a chunk fits into the document.

```
```{r}
This is basic chunk.
It always starts with ```{r}
And ends with ``
Code goes here
sum(fruit$oranges)
```
```

This is off-putting, but just go with it for now. You can type it manually, or use the `Insert` button and hit `r`. You will also notice that chunks are not limited to R code. It is particularly helpful that Python can also be run in this way.

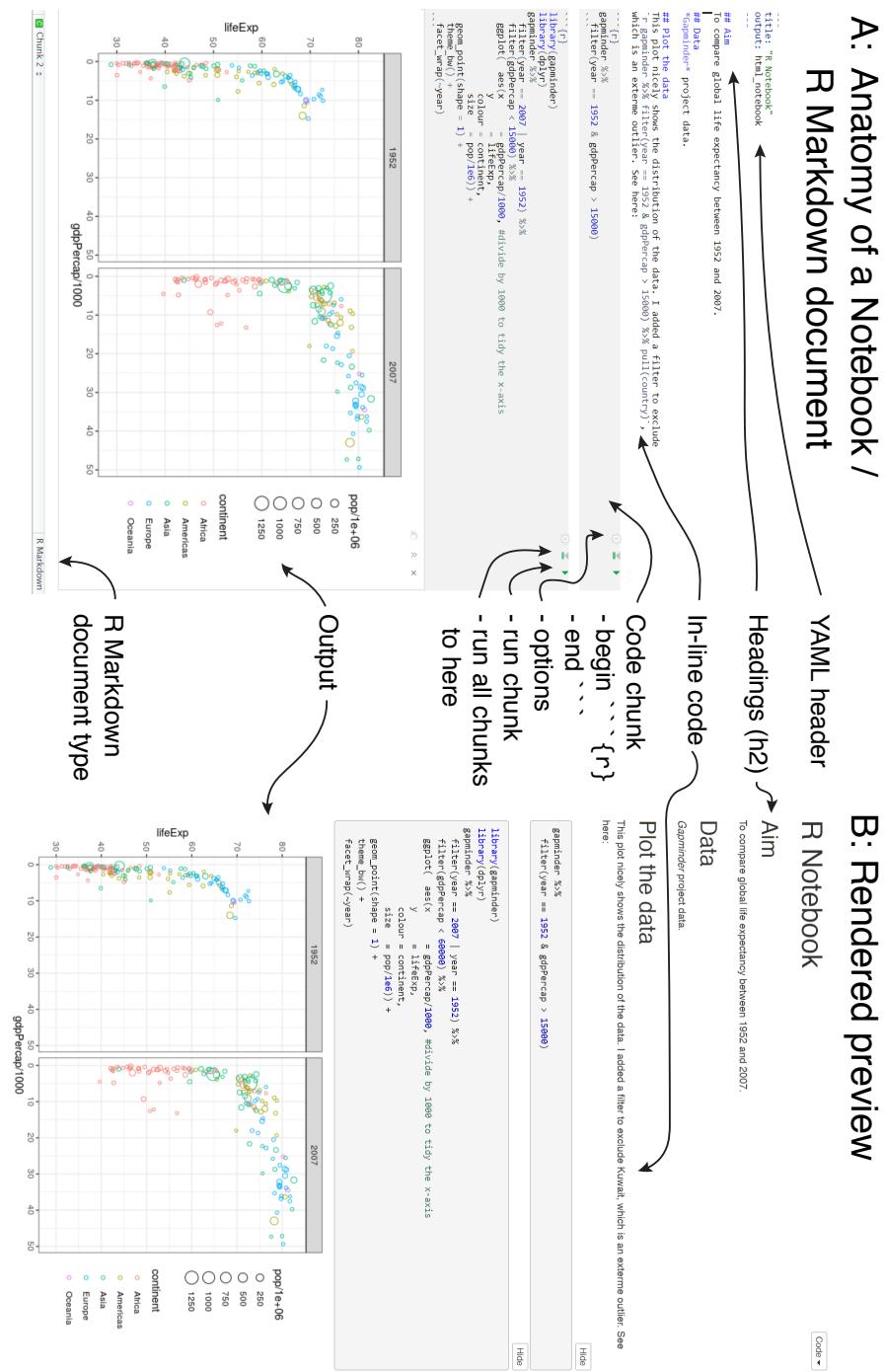


FIGURE 11.3: The Anatomy of a Notebook/Markdown file. Input (left) and output (right)

TABLE 11.1: Chunk output options when knitting an R Markdown file.

| Option | Code |
|-------------------------------|---------------------------|
| Show output only | echo=FALSE |
| Show code and output | echo=TRUE |
| Show code (don't run code) | eval=FALSE |
| Show nothing (run code) | include=FALSE |
| Show nothing (don't run code) | include=FALSE, eval=FALSE |
| Hide warnings | warnings=FALSE |
| Hide messages | messages=FALSE |

Chunks can be named which is sometimes useful for debugging, but is not essential for the code to run, e.g. “`{r table1-demographics}`”.

When doing an analysis in a Notebook you will almost always want to see the code and the output. When you are creating a final document you may wish to hide code. Chunk behaviour can be controlled via the `chunk cog` on the right of the chunk (Figure 11.3) which makes this very easy.

Table 11.1 shows the various permutations of code and output options that are available. The code is placed in the chunk header but the options fly-out now does this automatically, e.g.

```
```{r echo=FALSE}
```

### 11.5.3 Setting default chunk options

We can set default options for all our chunks at the top of our document by adding and editing `knitr::opts_chunk$set(echo = TRUE)` at the top of the document.

```
```{r}
knitr::opts_chunk$set(echo = TRUE,
                      warning = FALSE)
...```

```

11.5.4 Setting default figure options

It is possible to set different default sizes for different output types by including these in the YAML header (or using the document cog):

```
---
title: "R Notebook"
output:
  pdf_document:
    fig_height: 3
    fig_width: 4
  html_document:
    fig_height: 6
    fig_width: 9
---
```

The YAML header is very sensitive to the spaces/tabs, so make sure these are correct.

11.5.5 Markdown elements

Markdown text can be included as you wish around your chunks. Figure 11.3 shows an example of how this can be done.

This is a great way of getting into the habit of explicitly documenting your analysis. When you come back to a file in 6 months time, all of your thinking is there in front of you. Rather than having to work out what on Earth you were on about from a collection of random code!

11.6 Output

11.6.1 Running code and chunks

Figure 11.4 shows the various controls for running chunks and producing an output document. Code can be run line line-by-line using `Ctrl+Enter` as you are perhaps used to. There are options for

running all the chunks above the current chunk you are working on. This is useful as a chunk you are working on will often rely on objects created in preceding chunks.

It is good practice to use the `Restart R` and `Run All Chunks` option in the `Run` menu every so often. This ensures that all the code in your document is self-contained and is not relying on an object in the environment which you have created elsewhere. If this was the case, then it will fail when rendering a Markdown document.

11.6.2 Knitting

Probably the most important engine behind the RStudio Notebooks functionality is the `knitr` package by Yihui Xie.

Not knitting like your Granny does, but rendering a Markdown document into an output file, such as HTML, PDF or Word.

There are many options which can be applied in order to achieve the desired output. Some of these have been specifically coded into RStudio (Figure 11.4).

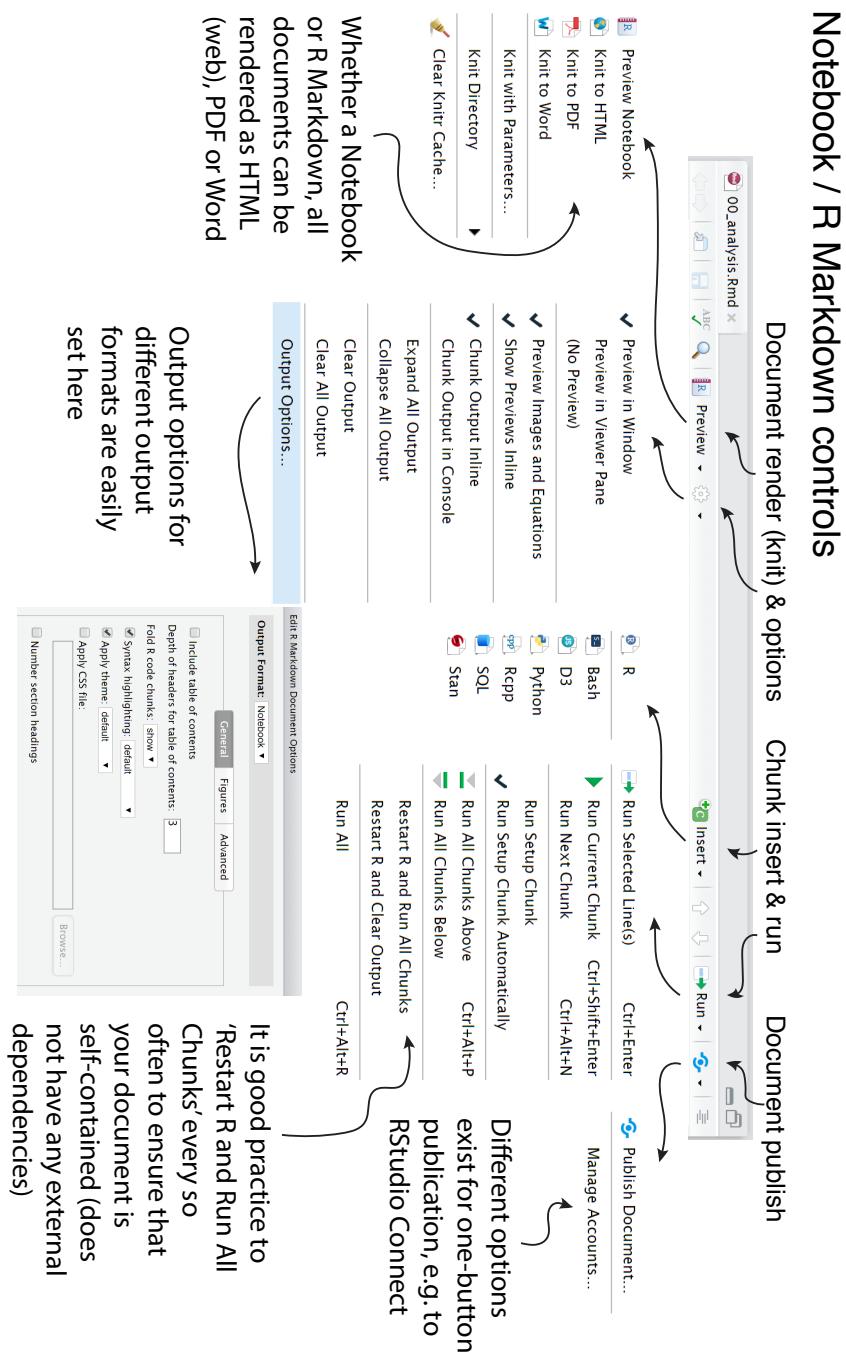
PDF and Word document creation require a `LaTeX` distribution to be installed on your computer. Depending on what system you are using, this may be set-up already.

An easy way to do this is using the `tinytex` package. We run courses on using Notebooks and this approach works on >95% of Windows and Mac machines.

```
```{r}
install.packages("tinytex")
```

This should allow you to knit all the different file options.

In the next chapter we will focus on the details of producing a polished final document.



**FIGURE 11.4:** Chunk and document options in Notebook/Markdown files.

## 11.7 File structure and workflow

As projects get bigger, it is really important that they are well organised. This will avoid errors and make collaboration easier.

Here is our suggested approach.

All projects must reside within an RStudio Project that has a meaningful name (not MyProject!). Never work within the (Home) or root directory. Projects should be initiated with a Git repository for version control (see Chapter ??).

Structure your directory sensibly.

```
proj/
- R/
- data/
- doc/
- figs/
- 00_analysis.Rmd
```

R/ contains all the .R script files used for data cleaning/preparation.  
data/ contains all raw data, such as .csv files.

doc/ contains all output documents and reports, such as .doc and .PDF files.

figs/ contains all figures and plots.

00\_analysis.Rmd or 00\_analysis.R is the actual main working file, and we keep this in the main project directory. We prefix this with 00 so it is always obvious which is the main file.

### 11.7.1 Data cleaning

This is all done in specific .R files, which are kept in the R/ folder, and are clearly labelled, e.g.

```
R/0_source_all.R
R/01_data_upload.R
R/02_make_factors.R
R/03_duplicate_records.R
```

For instance, 01\_data\_upload.R may look like this.

```
Melanoma project
Data upload

Get data
library(readr)
melanoma = read_csv(
 here::here("data", "melanoma.csv")
)

Other arguments here

Save
save(melanoma, file =
 here::here("data", "melanoma_working.rda")
)
```

Note use of the `here::here()`. This is a very useful package which “just works” when it comes to finding your files. Do not change the working directory (using `setwd()`) from the project root - there is never a reason to do this. `here::here()` is particularly useful when sharing projects between Linux, Mac and Windows machines, which have different conventions for file paths.

`02_make_factors.R` is our example second file, but it could be anything you want. It could look something like this.

```
Melanoma project
Create factors
library(tidyverse)

load(
 here::here("data", "melanoma_working.rda")
)

Recode variables
melanoma = melanoma %>%
 mutate(
 sex = factor(sex) %>%
 fct_recode("Male" = "1",
 "Female" = "0")
)

Save
save(melanoma, file =
 here::here("data", "melanoma_working.rda")
)
```

All these files can then be brought together in a single file to `source()`. This function is used to run code from a file.

`0_source_all.R` might look like this:

```
Melanoma project
Source all

source(here::here("R", "01_data_upload.R"))
source(here::here("R", "02_make_factors.R"))
source(here::here("R", "03_duplicate_records.R"))

Save
save(melanoma, file =
 here::here("data", "melanoma_all.rda")
)
```

You can now bring your robustly prepared data into your analysis file, which can be `.R` or `.Rmd` if you are working in a Notebook. We call this `00_analysis.Rmd` and it always sits in the project root director. You have two options in bringing in the data.

1. `source()` the data again
  - this is useful if the data is changing
  - may take a long time if it is a large dataset with lots of manipulations
2. `load()` from the `data/` folder
  - usually quicker, but loads the static dataset which was created the last time you ran `0_soruce_all.R`

The two options look like this:

```

title: "Melanoma analysis"
output: html_notebook

```{r get-data-option-1, echo=FALSE}
load(
  here::here("data", "melanoma_all.rda")
)
```

```
```{r get-data-option-2, echo=FALSE}
source(
 here::here("R", "0_source_all.R")
)
```

Why go to all this bother?

It comes from many years of finding errors due to badly organised projects. It is clearly not needed for a small quick project, but is essential for any major work.

At the very start of an analysis (as in the first day), we will start working in a single file. We will quickly move chunks of data cleaning / preparation code into single files as we go.

Compartmentalising the data cleaning helps in debugging. Sourced files can be ‘commented out’ (adding a `#` to a line in the `0_source_all.R` file) if you wish to exclude the manipulations in that particular file.

Most important, it helps with collaboration. When multiple people are working on a project, it is essential that communication is good and everybody is working to the same overall plan.

# 12

---

## *Exporting and reporting*

In chapter 11 we emphasised one of the less well known strengths of R - the ease with which reports can be written in HTML (web page), PDF, or Word documents.

In this chapter we will demonstrate how tables and figures can be easily exported to PDF and Word documents. To save space, we will not reproduce the whole analysis, but will demonstrate how a report document can be assembled.

Our report will contain two tables - a demographics table and a regression table - and a plot. We will use the `colon_s` data from the **finalfit** package. What follows is for demonstration purposes and is not meant to illustrate model building. We will ask, does a particular characteristic of a colon cancer (differentiation) predict 5-year survival?

---

### 12.1 Working in a `.R` file

We will demonstrate this in two ways. First, we will show what you might do if you were working in standard R script file, then exporting certain objects only.

Second, we will talk about the approach if you were primarily working in a Notebook, which makes things easier.

We presume that the data has been cleaned and carefully the `Get the data`, `Check the data`, `Data exploration` and `Model building` steps have already been completed.

## 12.2 Demographics table

Look at associations between our explanatory variable of interest (exposure) and other explanatory variables.

```
library(tidyverse)
library(finalfit)

Specify explanatory variables of interest
explanatory = c("age", "sex.factor",
 "extent.factor", "obstruct.factor",
 "nodes")

colon_s %>%
 summary_factorlist("differ.factor", explanatory,
 p=TRUE, na_include=TRUE)
```

**TABLE 12.1:** Exporting 'table 1': Tumour differentiation by patient and disease factors.

label	levels	Moderate	Poor	Well	p
Age (years)	Mean (SD)	59.9 (11.7)	59.0 (12.8)	60.2 (12.8)	0.788
Sex	Female	314 (71.7)	73 (16.7)	51 (11.6)	0.400
	Male	349 (74.6)	77 (16.5)	42 (9.0)	
Extent of spread	Submucosa	12 (60.0)	3 (15.0)	5 (25.0)	0.081
	Muscle	78 (76.5)	12 (11.8)	12 (11.8)	
	Serosa	542 (72.8)	127 (17.0)	76 (10.2)	
Obstruction	Adjacent structures	31 (79.5)	8 (20.5)	0 (0.0)	
	No	531 (74.4)	114 (16.0)	69 (9.7)	0.110
	Yes	122 (70.9)	31 (18.0)	19 (11.0)	
nodes	Missing	10 (50.0)	5 (25.0)	5 (25.0)	
	Mean (SD)	3.6 (3.4)	4.7 (4.4)	2.7 (2.2)	<0.001

Note that we include missing data in this table (see Chapter 14).

Also note that `nodes` has not been labelled properly. There are small numbers in some variables generating `chisq.test()` warnings (expected fewer than 5 in any cell). Generate final table.

```
colon_s = colon_s %>%
 mutate(
 nodes = ff_label(nodes, "Lymph nodes involved")
```

```

)
summary_factorlist("differ.factor", explanatory,
 p=TRUE, na_include=TRUE,
 add_dependent_label=TRUE,
 dependent_label_prefix = "Exposure: "
)
table1

```

**TABLE 12.2:** Exporting 'table 1': adjusting labels and output.

Exposure: Differentiation		Moderate	Poor	Well	p
Age (years)	Mean (SD)	59.9 (11.7)	59.0 (12.8)	60.2 (12.8)	0.788
Sex	Female	314 (71.7)	73 (16.7)	51 (11.6)	0.400
	Male	349 (74.6)	77 (16.5)	42 (9.0)	
Extent of spread	Submucosa	12 (60.0)	3 (15.0)	5 (25.0)	0.081
	Muscle	78 (76.5)	12 (11.8)	12 (11.8)	
	Serosa	542 (72.8)	127 (17.0)	76 (10.2)	
Obstruction	Adjacent structures	31 (79.5)	8 (20.5)	0 (0.0)	
	No	531 (74.4)	114 (16.0)	69 (9.7)	0.110
	Yes	122 (70.9)	31 (18.0)	19 (11.0)	
Lymph nodes involved	Missing	10 (50.0)	5 (25.0)	5 (25.0)	
	Mean (SD)	3.6 (3.4)	4.7 (4.4)	2.7 (2.2)	<0.001

### 12.3 Logistic regression table

Now examine explanatory variables against outcome and check the plots run ok.

```

explanatory = c("differ.factor", "age", "sex.factor",
 "extent.factor", "obstruct.factor",
 "nodes")
dependent = "mort_5yr"
table2 = colon_s %>%
 finalfit(dependent, explanatory,
 dependent_label_prefix = "")
table2

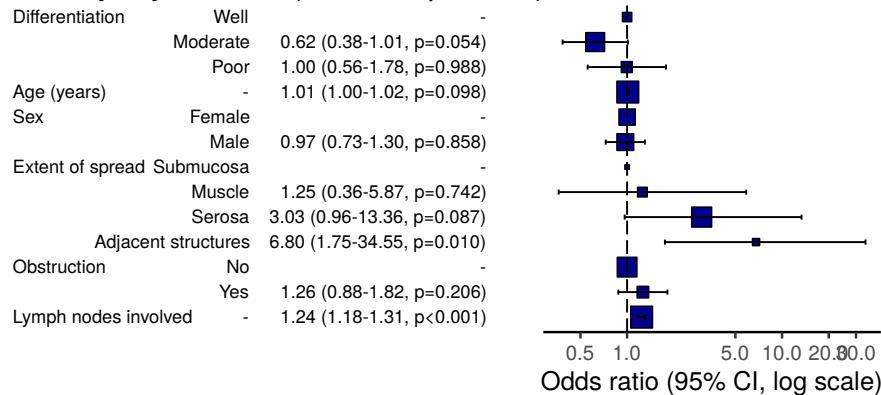
```

**TABLE 12.3:** Exporting a regression results table.

		Alive	Died	OR (univariable)	OR (multivariable)
Differentiation	Well	52 (10.5)	40 (10.1)	-	-
	Moderate	382 (76.9)	269 (68.1)	0.92 (0.59-1.43, p=0.694)	0.62 (0.38-1.01, p=0.054)
	Poor	63 (12.7)	86 (21.8)	1.77 (1.05-3.01, p=0.032)	1.00 (0.56-1.78, p=0.988)
Age (years)	Mean (SD)	59.8 (11.4)	59.9 (12.5)	1.00 (0.99-1.01, p=0.986)	1.01 (1.00-1.02, p=0.098)
	Female	243 (47.6)	194 (48.0)	-	-
Sex	Male	268 (52.4)	210 (52.0)	0.98 (0.76-1.27, p=0.889)	0.97 (0.73-1.30, p=0.858)
	Submucosa	16 (3.1)	4 (1.0)	-	-
	Muscle	78 (15.3)	25 (6.2)	1.28 (0.42-4.79, p=0.681)	1.25 (0.36-5.87, p=0.742)
Extent of spread	Serosa	401 (78.5)	349 (86.4)	3.48 (1.26-12.24, p=0.027)	3.03 (0.96-13.36, p=0.087)
	Adjacent structures	16 (3.1)	26 (6.4)	6.50 (1.98-25.93, p=0.004)	6.80 (1.75-34.55, p=0.010)
	No	408 (82.1)	312 (78.6)	-	-
Obstruction	Yes	89 (17.9)	85 (21.4)	1.25 (0.90-1.74, p=0.189)	1.26 (0.88-1.82, p=0.206)
	Lymph nodes involved	Mean (SD)	2.7 (2.4)	4.9 (4.4)	1.24 (1.18-1.30, p<0.001)
				1.24 (1.18-1.31, p<0.001)	

## 12.4 Odds ratio plot

```
colon_s %>%
 or_plot(dependent, explanatory,
 breaks = c(0.5, 1, 5, 10, 20, 30),
 table_text_size = 3.5)
```

**Mortality 5 year: OR (95% CI, p-value)****FIGURE 12.1:** Odds ratio plot.

## 12.5 MS Word via knitr/R Markdown

When moving from a .R file to a Markdown (.Rmd) file, environment objects such as tables or data frames / tibbles usually require to be saved and loaded to R Markdown document.

```
Save objects for knitr/markdown
save(table1, table2, dependent, explanatory,
 file = here::here("data", "out.rda"))
```

In RStudio, select File > New File > R Markdown.

A useful template file is produced by default. Try hitting knit to Word on the knitr button at the top of the .Rmd script window. If you have difficulties at this stage, refer to chapter 11.

Now paste this into the file:

```

title: "Example knitr/R Markdown document"
author: "Your name"
date: "22/5/2020"
output:
 word_document: default

```{r setup, include=FALSE}
# Load data into global environment.
library(finalfit)
library(dplyr)
library(knitr)
load(here::here("data", "out.rda"))
```

Table 1 - Demographics
```{r table1, echo = FALSE}
kable(table1, row.names=FALSE, align=c("l", "l", "r", "r", "r", "r"))
```

Table 2 - Association between tumour factors and 5 year mortality
```{r table2, echo = FALSE}
kable(table2, row.names=FALSE, align=c("l", "l", "r", "r", "r", "r"))
```
```

```
Figure 1 - Association between tumour factors and 5 year mortality
```{r figure1, echo = FALSE}
explanatory = c( "differ.factor", "age", "sex.factor",
                 "extent.factor", "obstruct.factor",
                 "nodes")
dependent = "mort_5yr"
colon_s %>%
  or_plot(dependent, explanatory)
```

```

The result is ok, but not great (Figure 12.2A).

## 12.6 Create Word template file

Now, edit your Word file to create a new template. Click on a table. The style should be `compact`. Right click > Modify... > font size = 9. Alter heading and text styles in the same way as desired. Save this as `colonTemplate.docx` (avoid underscores in the name). Upload the file to your project folder. Add this reference to the `.Rmd` YAML heading, as below. Make sure you get the spacing correct.

Also, the plot look correct and the warning messages is printed. Experiment with `fig.width` to get it looking right.

Now paste this into your `.Rmd` file and run:

```

title: "Example knitr/R Markdown document"
author: "Your name"
date: "22/5/2020"
output:
 word_document:
 reference_docx: colonTemplate.docx

```{r setup, include=FALSE}
# Load data into global environment.
library(finalfit)
library(dplyr)
library(knitr)
```

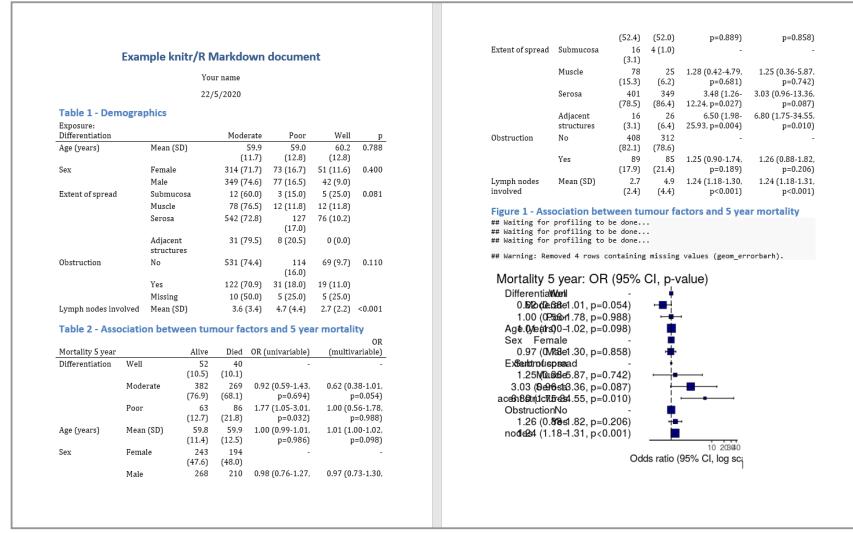
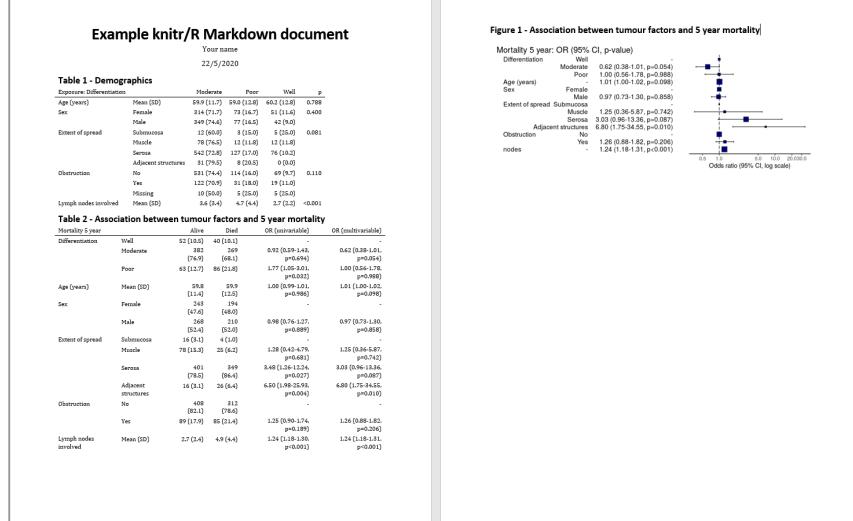
A**B**

FIGURE 12.2: Knitting to Microsoft Word from R Markdown. Before (A) and after (B) adjustment.

```

load(here::here("data", "out.rda"))
```

Table 1 - Demographics
```{r table1, echo = FALSE}
kable(table1, row.names=FALSE, align=c("l", "l", "r", "r", "r", "r"))
```

Table 2 - Association between tumour factors and 5 year mortality
```{r table2, echo = FALSE}
kable(table2, row.names=FALSE, align=c("l", "l", "r", "r", "r", "r"))
```

Figure 1 - Association between tumour factors and 5 year mortality
```{r figure1, echo=FALSE, message=FALSE, warning=FALSE, fig.width=10}
explanatory = c( "differ.factor", "age", "sex.factor",
                 "extent.factor", "obstruct.factor",
                 "nodes")
dependent = "mort_5yr"
colon_s %>%
  or_plot(dependent, explanatory,
         breaks = c(0.5, 1, 5, 10, 20, 30))
```

```

This is now looking good, and further tweaks can be made if you wish (Figure 12.2B).

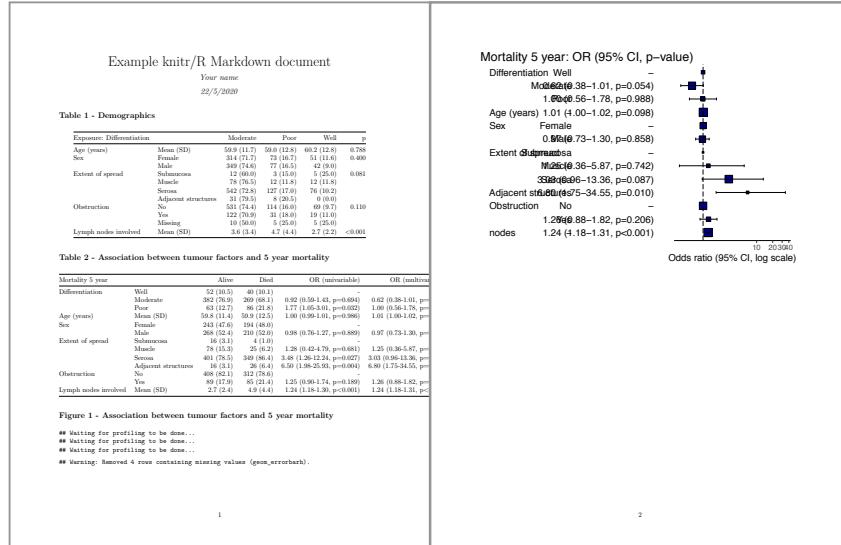
## 12.7 PDF via knitr/R Markdown

Using the default settings and the unmodified Markdown file above gets us close.

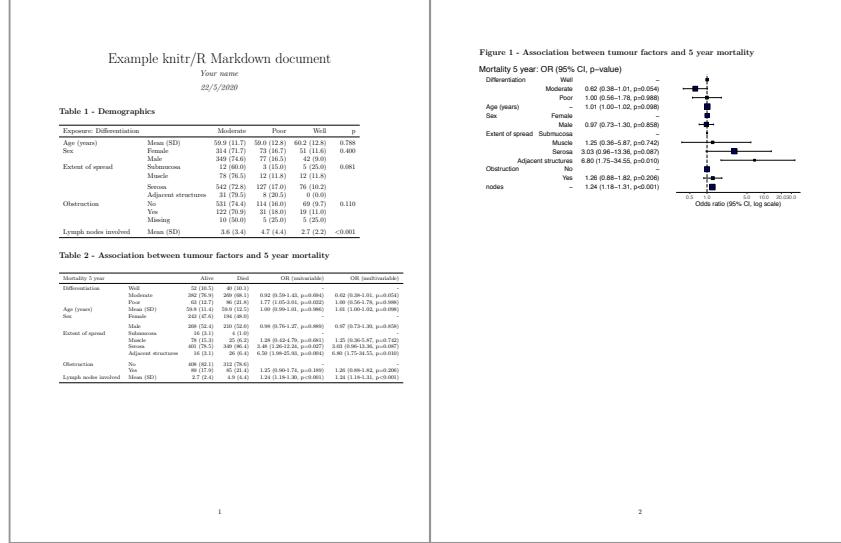
Again, this is not bad, but has issues (Figure 12.3A).

We can fix the plot in exactly the same way, but the table is off the side of the page. For this we use the **kableExtra** package. Install this in the normal manner. You may also want to alter the margins of your page using geometry in the preamble.

A



B



**FIGURE 12.3:** Knitting to Microsoft Word from R Markdown. Before (A) and after (B) adjustment.

```

```

```
title: "Example knitr/R Markdown document"
author: "Your name"
date: "22/5/2020"
output:
 pdf_document: default
geometry: margin=0.75in

```

```
```{r setup, include=FALSE}
# Load data into global environment.
library(finalfit)
library(dplyr)
library(knitr)
library(kableExtra)
load(here::here("data", "out.rda"))
```
```

```
Table 1 - Demographics
```{r table1, echo = FALSE}
kable(table1, row.names=FALSE, align=c("l", "l", "r", "r", "r", "r"),
      booktabs = TRUE)
```
```

```
Table 2 - Association between tumour factors and 5 year mortality
```{r table2, echo = FALSE}
kable(table2, row.names=FALSE, align=c("l", "l", "r", "r", "r", "r"),
      booktabs=TRUE) %>%
kable_styling(font_size=8)
```
```

```
Figure 1 - Association between tumour factors and 5 year mortality
```{r figure1, echo=FALSE, fig.width=10, message=FALSE, warning=FALSE}
explanatory = c( "differ.factor", "age", "sex.factor",
                "extent.factor", "obstruct.factor",
                "nodes")
dependent = "mort_5yr"
colon_s %>%
  or_plot(dependent, explanatory,
         breaks = c(0.5, 1, 5, 10, 20, 30))
```
```

This now looks better (Figure 12.3B).

## 12.8 Working in a `.Rmd` file

We now perform almost all our analyses in a Notebook / Markdown file as described in the previous chapter. This means running all analyses within the document, without the requirement to save and reload table or plot objects.

As mentioned earlier, A Notebook document can be rendered as a PDF or a Word document. Some refining is usually needed to move from an ‘analysis’ document to a final ‘report’ document, but it is often minimal.

Figure 12.4 demonstrates a report-type document rendered as a PDF. All the code is run within the document, but not included in the output (`echo=FALSE`).

## Outcomes after resection for colorectal cancer

*The Colorectal Cancer Collaborative*

### Introduction

Colorectal cancer is the third most common cancer worldwide. In this study, we have re-analysed a classic dataset to determine the influence of tumour differentiation on 5-year survival prior to the introduction of modern chemotherapeutic regimes.

### Methods

Data were generated within a randomised trial of adjuvant chemotherapy for colon cancer. Levamisole is a low-toxicity compound etc. etc.

### Results

#### Patient characteristics

Table 1 shows the characteristics of patients in the study.

Table 1: Demographics of patients entered into randomised controlled trial of adjuvant chemotherapy after surgery for colon cancer.

| Exposure: Differentiation |                     | Moderate    | Poor        | Well        | p      |
|---------------------------|---------------------|-------------|-------------|-------------|--------|
| Age (years)               | Mean (SD)           | 59.9 (11.7) | 59.0 (12.8) | 60.2 (12.8) | 0.788  |
| Sex                       |                     |             |             |             |        |
| Female                    |                     | 314 (71.7)  | 73 (16.7)   | 51 (11.6)   | 0.400  |
| Male                      |                     | 349 (74.6)  | 77 (16.5)   | 42 (9.0)    |        |
| Extent of spread          | Submucosa           | 12 (60.0)   | 3 (15.0)    | 5 (25.0)    | 0.081  |
|                           | Muscle              | 78 (76.5)   | 12 (11.8)   | 12 (11.8)   |        |
|                           | Serosa              | 542 (72.8)  | 127 (17.0)  | 76 (10.2)   |        |
|                           | Adjacent structures | 31 (79.5)   | 8 (20.5)    | 0 (0.0)     |        |
| Lymph nodes involved      | Mean (SD)           | 3.6 (3.4)   | 4.7 (4.4)   | 2.7 (2.2)   | <0.001 |

#### 5-year outcome by tumour differentiation

Table 2 shows a univariable and multivariable regression analysis of 5-year mortality by patient and disease characteristics.

Table 2: 5-year survival after colon cancer (logistic regression)

| Mortality 5 year     |                     | Alive       | Died        | OR (univariable)           | OR (multivariable)         |
|----------------------|---------------------|-------------|-------------|----------------------------|----------------------------|
| Differentiation      | Well                | 52 (10.5)   | 40 (10.1)   | -                          | -                          |
|                      | Moderate            | 382 (76.9)  | 269 (68.1)  | 0.92 (0.59-1.43, p=0.694)  | 0.70 (0.44-1.12, p=0.132)  |
|                      | Poor                | 63 (12.7)   | 86 (21.8)   | 1.77 (1.05-3.01, p=0.032)  | 1.08 (0.61-1.90, p=0.796)  |
| Age (years)          | Mean (SD)           | 59.8 (11.4) | 59.9 (12.5) | 1.00 (0.99-1.01, p=0.986)  | 1.01 (1.00-1.02, p=0.195)  |
| Sex                  | Female              | 243 (47.6)  | 194 (48.0)  | -                          | -                          |
|                      | Male                | 268 (52.4)  | 210 (52.0)  | 0.98 (0.76-1.27, p=0.889)  | 0.98 (0.74-1.30, p=0.885)  |
| Extent of spread     | Submucosa           | 16 (3.1)    | 4 (1.0)     | -                          | -                          |
|                      | Muscle              | 78 (15.3)   | 25 (6.2)    | 1.28 (0.42-4.79, p=0.681)  | 1.28 (0.37-5.92, p=0.722)  |
|                      | Serosa              | 401 (78.5)  | 349 (86.4)  | 3.48 (1.26-12.24, p=0.027) | 3.13 (1.01-13.76, p=0.076) |
|                      | Adjacent structures | 16 (3.1)    | 26 (6.4)    | 6.50 (1.98-25.93, p=0.004) | 6.04 (1.58-30.41, p=0.015) |
| Lymph nodes involved | Mean (SD)           | 2.7 (2.4)   | 4.9 (4.4)   | 1.24 (1.18-1.30, p<0.001)  | 1.23 (1.17-1.30, p<0.001)  |

### Discussion

In this study etc.

**FIGURE 12.4:** Writing a final report in a Markdown document.

# **13**

---

## *Version control*

Version control is essential for keeping track of data analysis projects, as well as collaborating. It allows backup of scripts and easy collaboration on complex projects. RStudio works really well with Git, an open source open source distributed version control system, and GitHub, a web-based Git repository hosting service.

This example is done on RStudio Server, but the same procedure can be used for RStudio desktop. Git is piece of software which runs locally. It may need to be installed first. Ensure you are clear about the difference between Git (local version control software) and Github (a web-based repository store).

---

### **13.1 Setup Git on RStudio and Associate with GitHub**

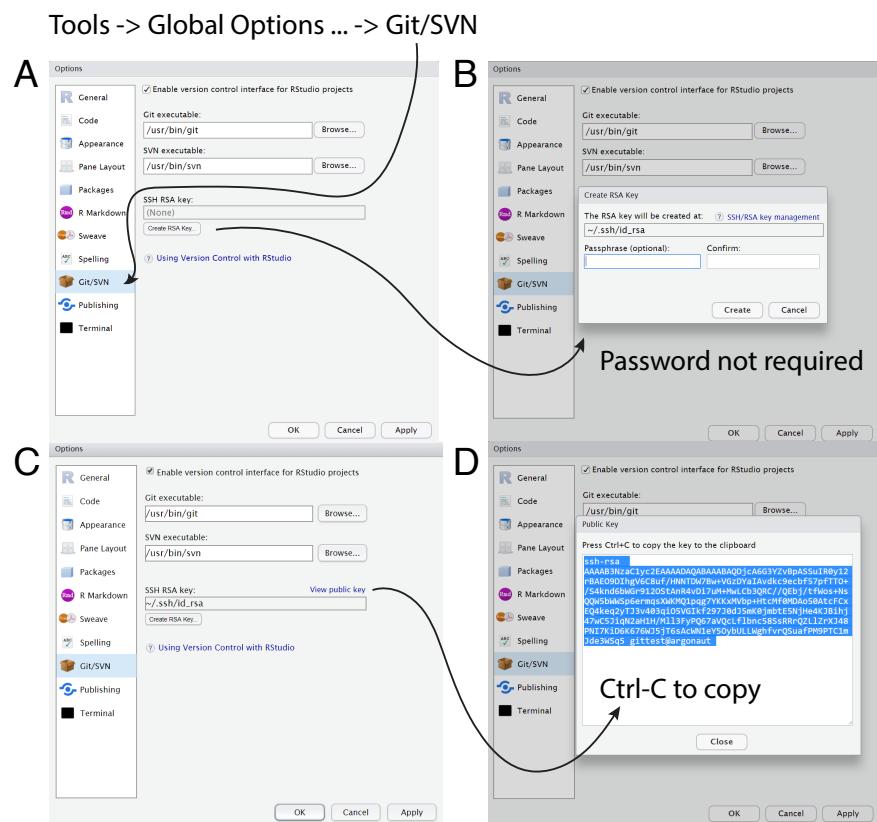
In RStudio, Tools -> Global Options, select Git//SVN tab. Ensure the path to the Git executable is correct. This is particularly important in Windows where it may not default correctly (e.g. C:/Program Files (x86)/Git/bin/git.exe).

---

### **13.2 Create an SSH RSA key and add to your GitHub account**

Fig 1, 2

Now hit, Create RSA Key. Close this window. Click, View public key, and copy the displayed public key. If you haven't already, create a GitHub account. Open your account settings and click the SSH keys tab. Click Add SSH key. Paste in the public key you have copied from RStudio.

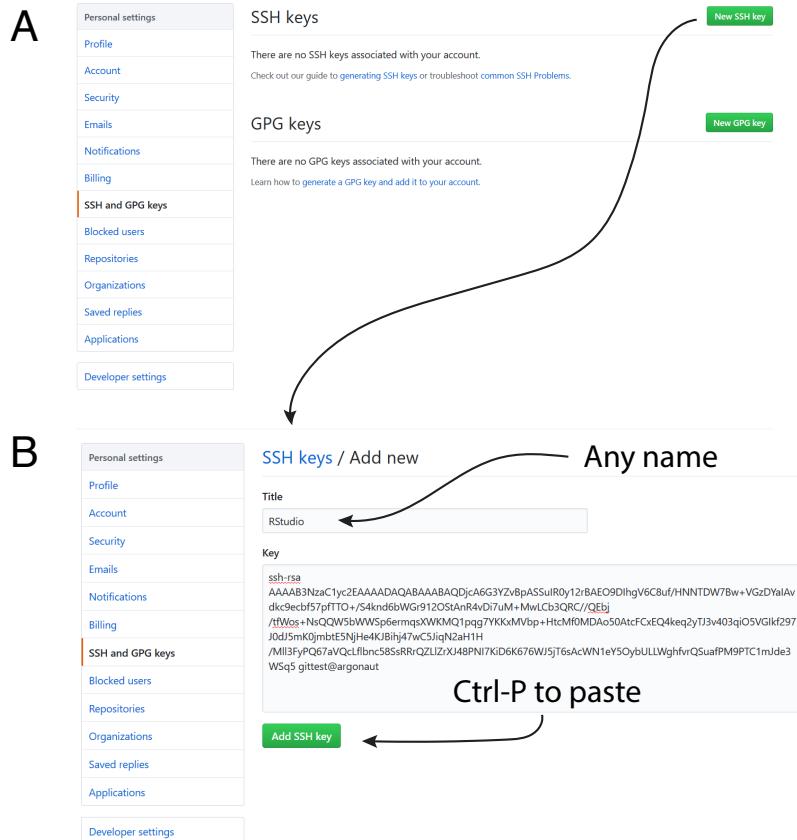


**FIGURE 13.1: SSH.**

### 13.3 Create a project in RStudio and commit a file

Fig 3, 4

## github.com -&gt; Settings -&gt; SSH and GPG keys

**FIGURE 13.2: CAPTION**

Tell Git who you are. Remember Git is a piece of software running on your own computer. This is distinct to GitHub, which is the repository website.

In RStudio, click New project as normal. Click New Directory. Name the project and check Create a git repository. Now in RStudio, create a new script which you will add to your repository.

After saving your new script (test.R), it should appear in the Git tab on the Environment / history panel.

Click the file you wish to add, and the status should turn to a

green ‘A’. Now click Commit and enter an identifying message in Commit message.

It is important to commit prior to linking the project and the GitHub repository.

You have now committed the current version of this file to your repository on your computer/server. In the future you may wish to create branches to organise your work and help when collaborating.

---

#### 13.4 Create a new repository on GitHub and link to RStudio project

Fig 5, 6

Now you want to push the contents of this commit to GitHub, so it is also backed-up off site and available to collaborators. In GitHub, create a New repository, called here myproject.

You have now pushed your commit to GitHub, and should be able to see your files in your GitHub account. The Pull Push buttons in RStudio will now also work. Remember, after each Commit, you have to Push to GitHub, this doesn’t happen automatically.

---

#### 13.5 Clone an existing GitHub project to new RStudio project

In RStudio, click New project as normal. Click Version Control.

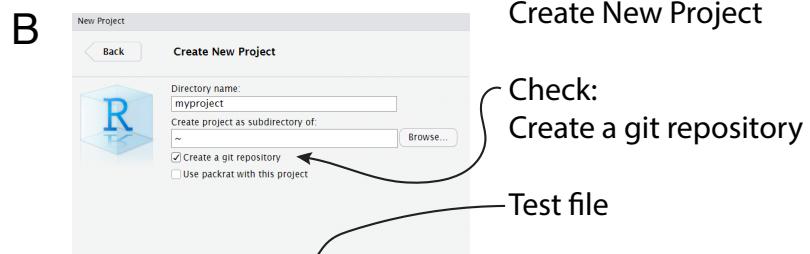
In Clone Git Repository, enter the GitHub repository URL as per below. Change the project directory name if necessary.

remote origin

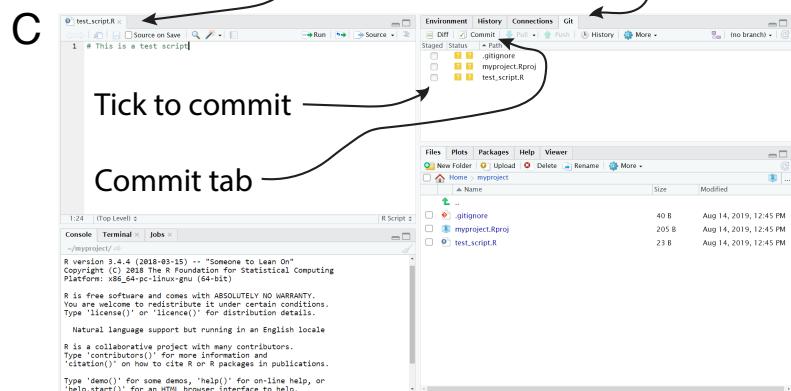
### A RStudio terminal to configure global settings

```
gittest@argonaut:~$ git config --global user.email "email@email.com"
gittest@argonaut:~$ git config --global user.name "testgit-healthyR"
```

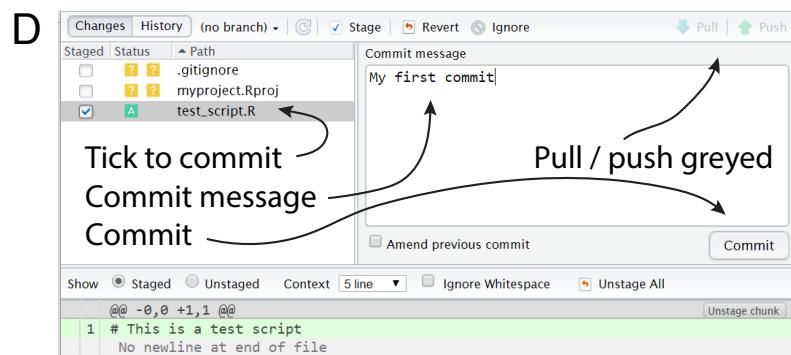
B



C



D



**FIGURE 13.3: CAPTION**

## A github.com -> + New repository

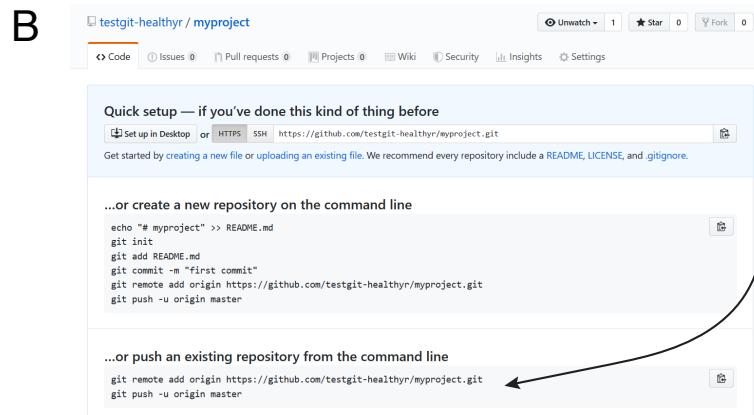
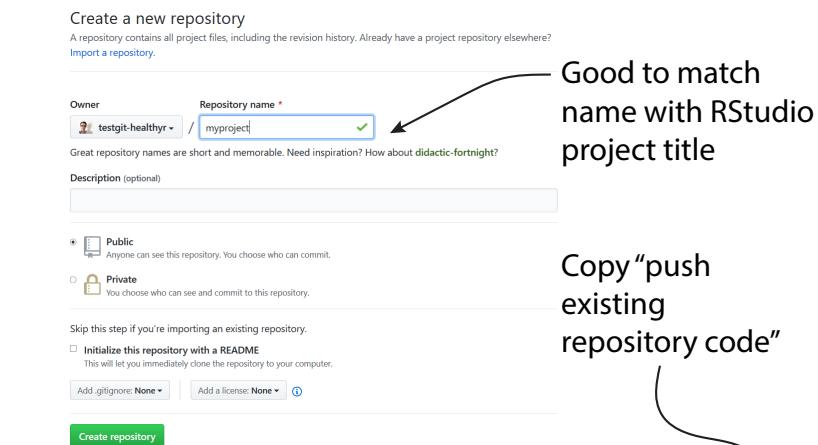


FIGURE 13.4: CAPTION

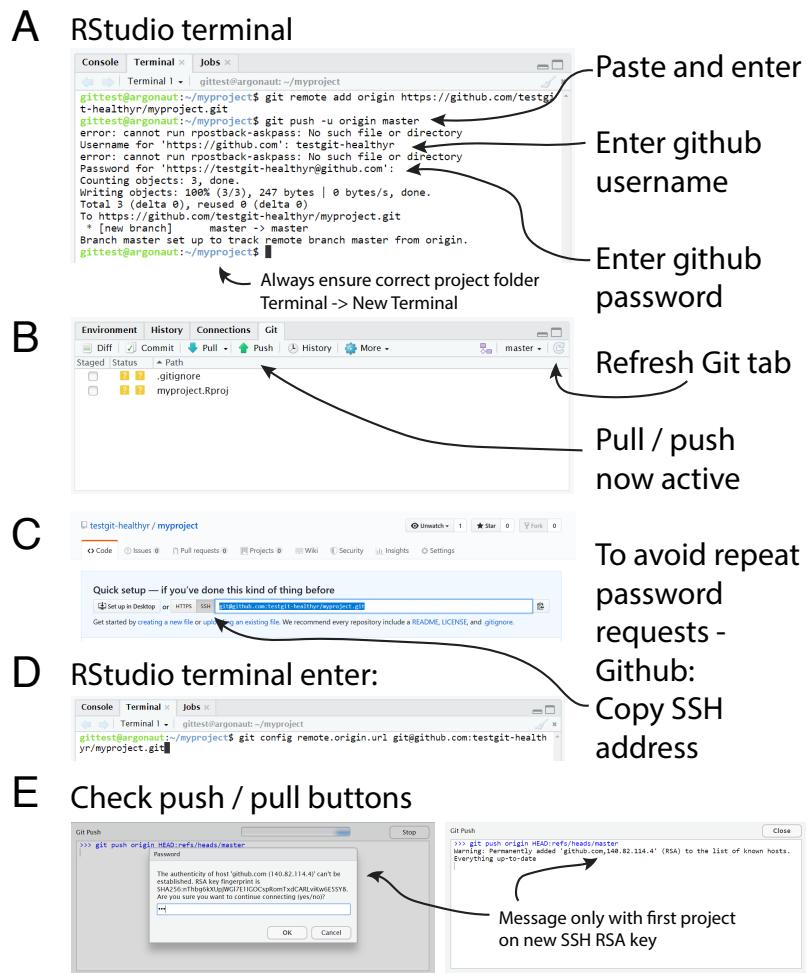


FIGURE 13.5: CAPTION

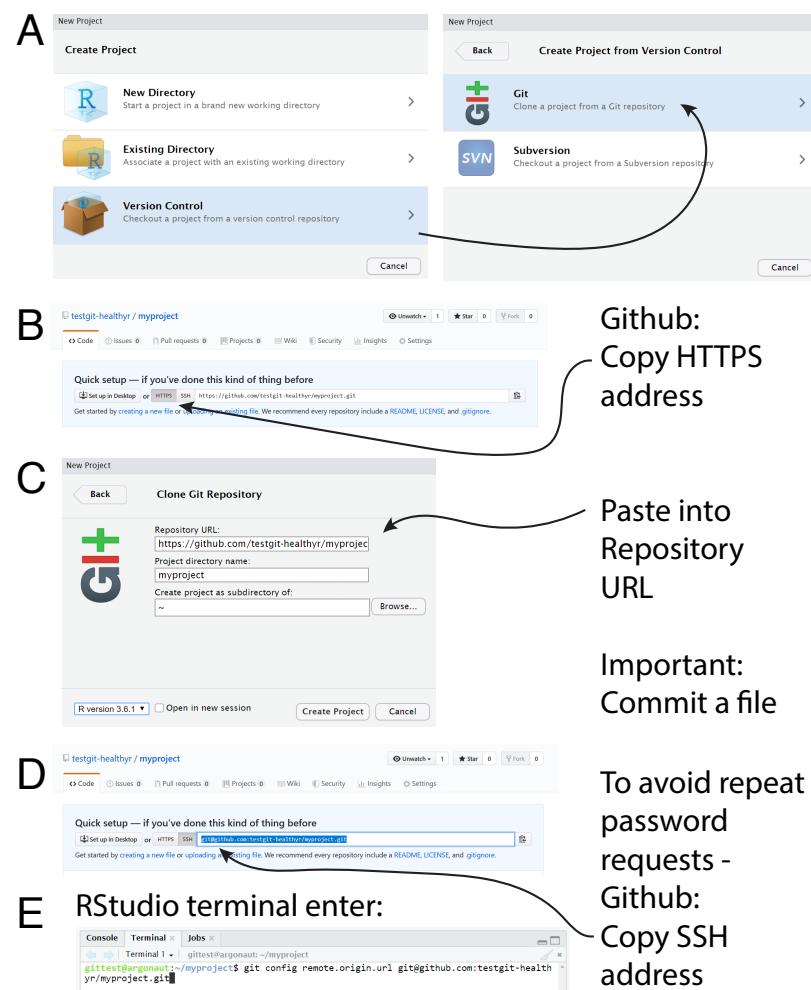


FIGURE 13.6: CAPTION

# 14

---

## *Missing data*

---

### 14.1 The problem of missing data

As journal editors, we often receive studies in which the investigators fail to describe, analyse, or even acknowledge missing data. This is frustrating, as it is often of the utmost importance. Conclusions may (and do) change when missing data is accounted for. Some folk seem to not even appreciate that in conventional regression, only rows with complete data are included. By reading this, you will not be one of them!

These are the five steps to ensuring missing data are correctly identified and appropriately dealt with:

1. Ensure your data are coded correctly.
2. Identify missing values within each variable.
3. Look for patterns of missingness.
4. Check for associations between missing and observed data.
5. Decide how to handle missing data.

We will work through a number of functions that will help with each of these.

## 14.2 Some confusing terminology

But first there are some terms which are easy to mix up. These are important as they describe the mechanism of missingness and this determines how you can handle the missing data.

### 14.2.1 Missing completely at random (MCAR)

As it says, values are randomly missing from your dataset. Missing data values do not relate to any other data in the dataset and there is no pattern to the actual values of the missing data themselves.

For instance, when smoking status is not recorded in a random subset of patients.

This is easy to handle, but unfortunately, data are almost never missing completely at random.

### 14.2.2 Missing at random (MAR)

This is confusing and would be better stated as missing conditionally at random. Here, missing data do have a relationship with other variables in the dataset. However, the actual values that are missing are random.

For example, smoking status is not documented in female patients because the doctor was too shy to ask. Yes ok, not that realistic!

### 14.2.3 Missing not at random (MNAR)

The pattern of missingness is related to other variables in the dataset, but in addition, the values of the missing data are not random.

For example, when smoking status is not recorded in patients ad-

mitted as an emergency, who are also more likely to have worse outcomes from surgery.

Missing not at random data are important, can alter your conclusions, and are the most difficult to diagnose and handle. They can only be detected by collecting and examining some of the missing data. This is often difficult or impossible to do.

How you deal with missing data is dependent on the type of missingness. Once you know this, then you can sort it.

More on this below.

---

### 14.3 Ensure your data are coded correctly: `ff_glimpse`

While clearly obvious, this step is often ignored in the rush to get results. The first step in any analysis is robust data cleaning and coding. Lots of packages have a glimpse-type function and our own Finalfit is no different. This function has three specific goals:

1. Ensure all factors and numerics are correctly assigned. That is the commonest reason to get an error with a Finalfit function. You think you're using a factor variable, but in fact it is incorrectly coded as a continuous numeric.
2. Ensure you know which variables have missing data. This presumes missing values are correctly assigned `NA`.
3. Ensure factor levels and variable labels are assigned correctly.

---

### 14.4 The Question

Using the `colon_s` colon cancer dataset, we are interested in exploring the association between a cancer obstructing the bowel and

5-year survival, accounting for other patient and disease characteristics.

For demonstration purposes, we will create random MCAR and MAR smoking variables to the dataset.

```
Create some extra missing data
library(finalfit)
library(dplyr)
data(colon_s)
set.seed(1)
colon_s = colon_s %>%
 mutate(
 ## Smoking missing completely at random
 smoking_mcarr = sample(c("Smoker", "Non-smoker", NA),
 dim(colon_s)[1], replace=TRUE,
 prob = c(0.2, 0.7, 0.1)) %>%
 factor() %>%
 ff_label("Smoking (MCAR)"),

 ## Smoking missing conditional on patient sex
 smoking_mar = ifelse(sex.factor == "Female",
 sample(c("Smoker", "Non-smoker", NA),
 sum(sex.factor == "Female"),
 replace = TRUE,
 prob = c(0.1, 0.5, 0.4)),
 sample(c("Smoker", "Non-smoker", NA),
 sum(sex.factor == "Male"),
 replace=TRUE, prob = c(0.15, 0.75, 0.1)))
) %>%
 factor() %>%
 ff_label("Smoking (MAR)")
)

Examine with ff_glimpse
explanatory = c("age", "sex.factor",
 "nodes", "obstruct.factor",
 "smoking_mcarr", "smoking_mar")
dependent = "mort_5yr"

colon_s %>%
 ff_glimpse(dependent, explanatory)

Continuous
label var_type n missing_n missing_percent mean sd min
age Age (years) <dbl> 929 0 0.0 59.8 11.9 18.0
nodes nodes <dbl> 911 18 2.0 3.7 3.6 0.0
quartile_25 median quartile_75 max
age 53.0 61.0 69.0 85.0
```

```

nodes 1.0 2.0 5.0 33.0
##
Categorical
label var_type n missing_n missing_percent
sex.factor Sex <fct> 929 0 0.0
obstruct.factor Obstruction <fct> 908 21 2.3
mort_5yr Mortality 5 year <fct> 915 14 1.5
smoking_mcar Smoking (MCAR) <fct> 828 101 10.9
smoking_mar Smoking (MAR) <fct> 726 203 21.9
levels_n levels levels_count
sex.factor 2 "Female", "Male" 445, 484
obstruct.factor 2 "No", "Yes", "(Missing)" 732, 176, 21
mort_5yr 2 "Alive", "Died", "(Missing)" 511, 404, 14
smoking_mcar 2 "Non-smoker", "Smoker", "(Missing)" 645, 183, 101
smoking_mar 2 "Non-smoker", "Smoker", "(Missing)" 585, 141, 203
levels_percent
sex.factor 48, 52
obstruct.factor 78.8, 18.9, 2.3
mort_5yr 55.0, 43.5, 1.5
smoking_mcar 69, 20, 11
smoking_mar 63, 15, 22

```

The function summarises a data frame or tibble by numeric (continuous) variables and factor (discrete) variables. The dependent and explanatory are for convenience. Pass either or neither e.g. to summarise data frame or tibble:

Use this to check that the variables are all assigned and behaving as expected. The proportion of missing data can be seen, e.g. `smoking_mar` has 22% missing data.

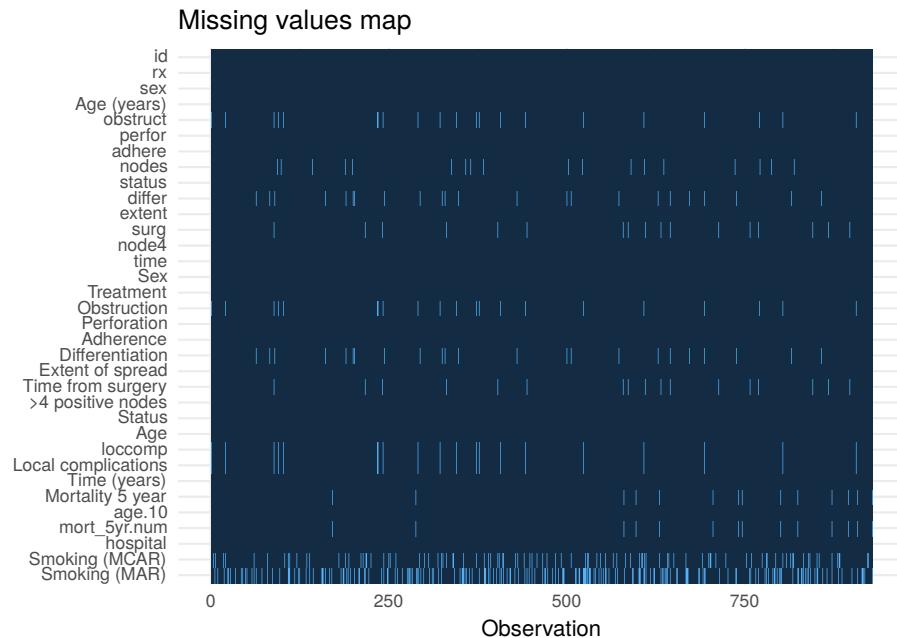
## 14.5 2. Identify missing values in each variable: *missing\_plot*

In detecting patterns of missingness, this plot is useful. Row number is on the x-axis and all included variables are on the y-axis. Associations between missingness and observations can be easily seen, as can relationships of missingness between variables.

```

colon_s %>%
 missing_plot()

```



Further visualisations of missingness are available via the `naniar`<sup>1</sup> package.

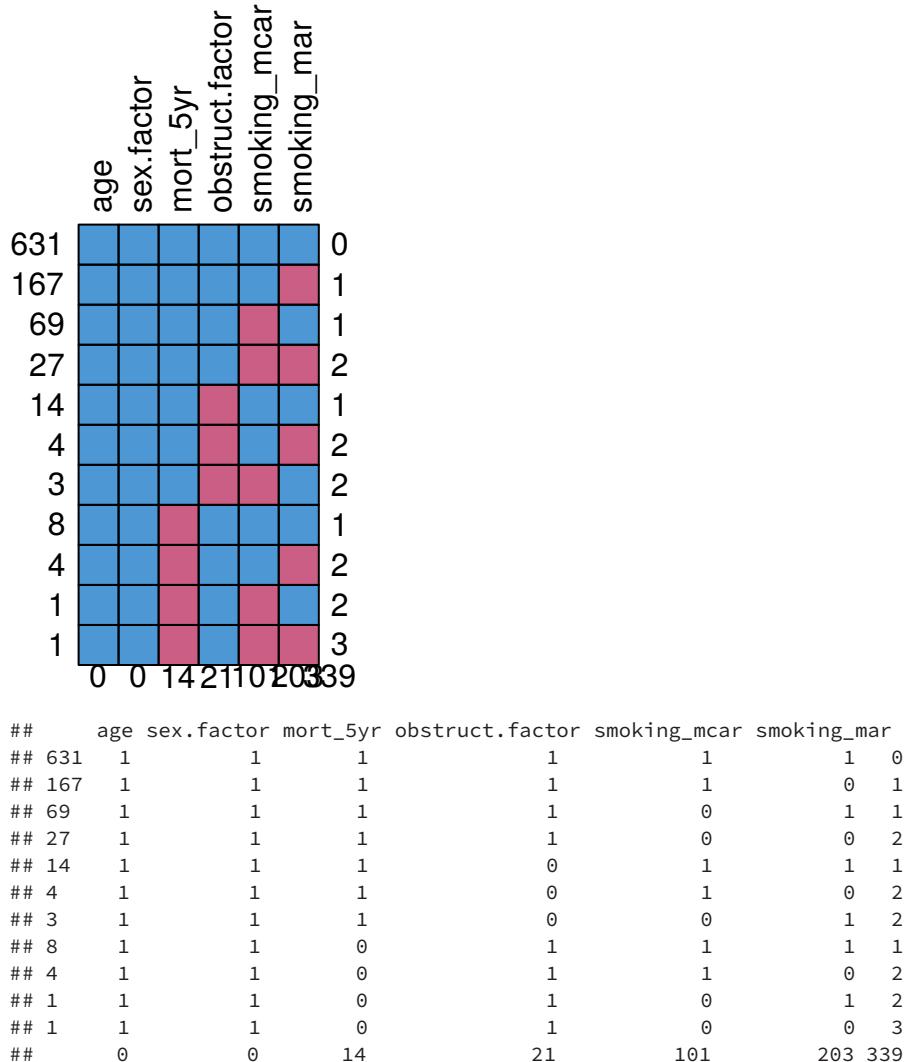
#### 14.6 3. Look for patterns of missingness: `missing_pattern`

Using `finalfit`, `missing_pattern()` wraps a function from the `mice` package, `md.pattern()`. This produces a table and a plot showing the pattern of missingness between variables.

```
explanatory = c("age", "sex.factor",
 "obstruct.factor",
 "smoking_mcar", "smoking_mar")
dependent = "mort_5yr"

colon_s %>%
 missing_pattern(dependent, explanatory)
```

<sup>1</sup><http://naniar.njtierney.com>



This allows us to look for patterns of missingness between variables. There are 11 patterns in these data. The number and pattern of missingness help us to determine the likelihood of it being random rather than systematic.

#### 14.6.1 Make sure you include missing data in demographics tables

Table 1 in a healthcare study is often a demographics table of an “explanatory variable of interest” against other explanatory variables/confounders. Do not silently drop missing values in this table. It is easy to do this correctly with `summary_factorlist()`. This function provides a useful summary of a dependent variable against explanatory variables. Despite its name, continuous variables are handled nicely.

`na_include=TRUE` ensures missing data from the explanatory variables (but not dependent) are included. Note that any p-values are generated across missing groups as well, so run a second time with `na_include=FALSE` if you wish a hypothesis test only over observed data.

```
Explanatory or confounding variables
explanatory = c("age", "sex.factor",
 "nodes",
 "smoking_mcar", "smoking_mar")

Explanatory variable of interest
dependent = "obstruct.factor" # Bowel obstruction

table1 = colon_s %>%
 summary_factorlist(dependent, explanatory,
 na_include=TRUE, p=TRUE)
```

---

#### 14.7 4. Check for associations between missing and observed data: `missing_pairs` | `missing_compare`

In deciding whether data is MCAR or MAR, one approach is to explore patterns of missingness between levels of included variables. This is particularly important (we would say absolutely required) for a primary outcome measure / dependent variable.

Take for example “death”. When that outcome is missing it is often

**TABLE 14.1:** Simulated missing completely at random (MCAR) and missing at random (MAR) dataset.

| label          | levels     | No          | Yes         | p     |
|----------------|------------|-------------|-------------|-------|
| Age (years)    | Mean (SD)  | 60.2 (11.5) | 57.3 (13.3) | 0.014 |
| Sex            | Female     | 346 (79.2)  | 91 (20.8)   | 0.290 |
|                | Male       | 386 (82.0)  | 85 (18.0)   |       |
| nodes          | Mean (SD)  | 3.7 (3.7)   | 3.5 (3.2)   | 0.774 |
| Smoking (MCAR) | Non-smoker | 500 (79.4)  | 130 (20.6)  | 0.173 |
|                | Smoker     | 154 (85.6)  | 26 (14.4)   |       |
| Smoking (MAR)  | Missing    | 78 (79.6)   | 20 (20.4)   |       |
|                | Non-smoker | 456 (79.9)  | 115 (20.1)  | 0.724 |
|                | Smoker     | 112 (81.2)  | 26 (18.8)   |       |
|                | Missing    | 164 (82.4)  | 35 (17.6)   |       |

for a particular reason. For example, perhaps patients undergoing emergency surgery were less likely to have complete records compared with those undergoing planned surgery. And of course, death is more likely after emergency surgery.

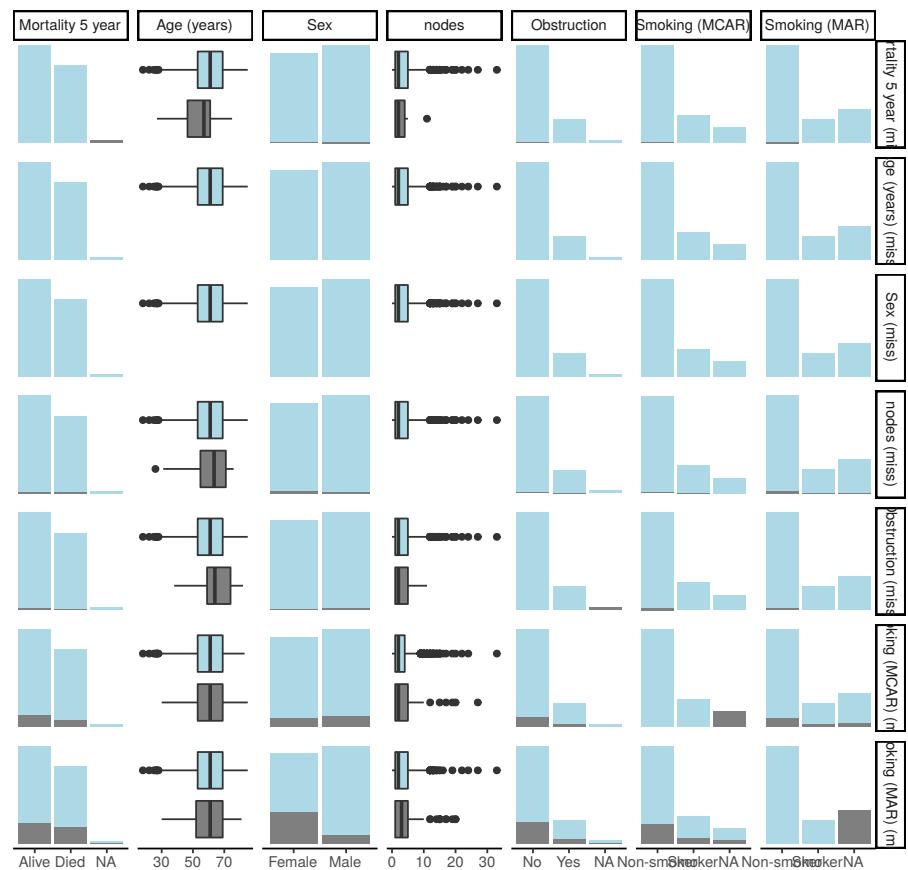
`missing_pairs()` uses functions from the excellent `ggally` package. It produces pairs plots to show relationships between missing values and observed values in all variables.

```
explanatory = c("age", "sex.factor",
 "nodes", "obstruct.factor",
 "smoking_mcar", "smoking_mar")
dependent = "mort_5yr"
colon_s %>%
 missing_pairs(dependent, explanatory)
```

For continuous variables (age and nodes), the distributions of observed and missing data can be visually compared. Is there a difference between age and mortality above?

For discrete, data, counts are presented by default. It is often easier to compare proportions:

Missing data matrix

**FIGURE 14.1:** Missing data matrix with `missing_pairs()`.

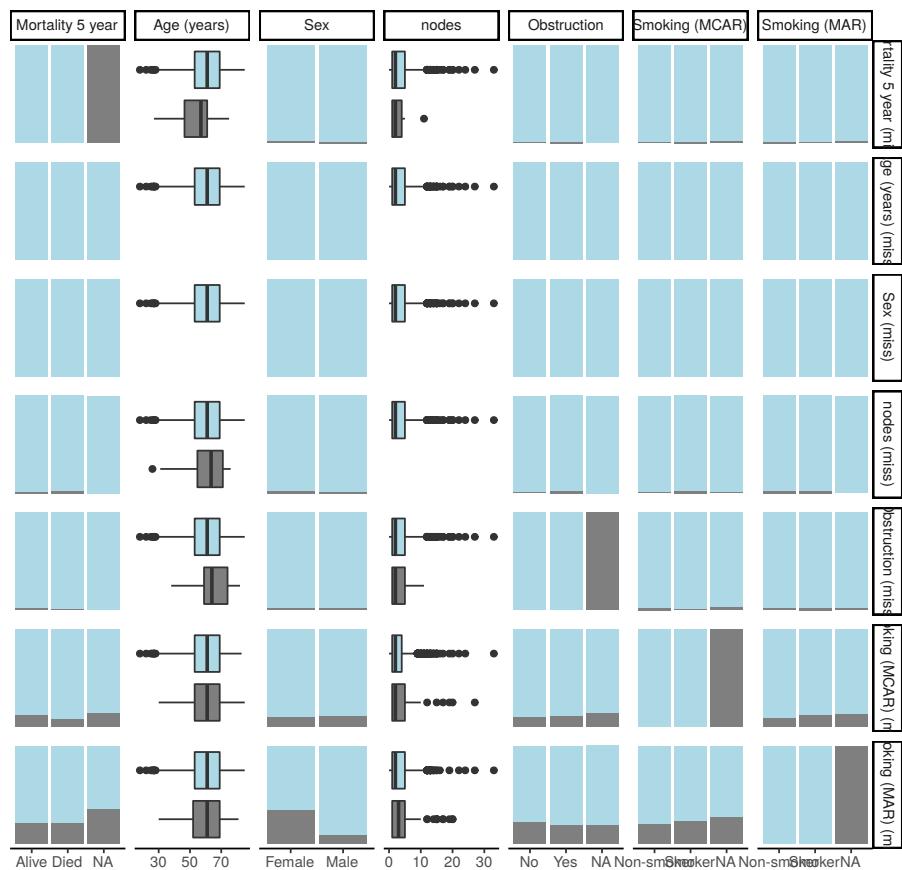
```
colon_s %>%
 missing_pairs(dependent, explanatory, position = "fill")
```

It should be obvious that missingness in Smoking (MCAR) does not relate to sex (row 6, column 3). But missingness in Smoking (MAR) does differ by sex (last row, column 3) as was designed above when the missing data were created.

We can confirm this using `missing_compare()`.

14.7 4. Check for associations between missing and observed data: `missing_pairs / missing_compare` 329

Missing data matrix



**FIGURE 14.2:** Missing data matrix with `missing_pairs(position = 'fill')`.

```
explanatory = c("age", "sex.factor",
 "nodes", "obstruct.factor")
dependent = "smoking_mcarr"

missing_mcarr = colon_s %>%
 missing_compare(dependent, explanatory)
```

```
dependent = "smoking_mar"

missing_mar = colon_s %>%
 missing_compare(dependent, explanatory)
```

**TABLE 14.2:** Missing data comparison: Smoking (MCAR).

| Missing data analysis: Smoking (MCAR) |           | Missing     | Not missing | p     |
|---------------------------------------|-----------|-------------|-------------|-------|
| Age (years)                           | Mean (SD) | 59.9 (12.6) | 59.7 (11.9) | 0.867 |
| Sex                                   | Female    | 46 (10.3)   | 399 (89.7)  | 0.616 |
|                                       | Male      | 55 (11.4)   | 429 (88.6)  |       |
| nodes                                 | Mean (SD) | 4.0 (4.5)   | 3.6 (3.4)   | 0.990 |
| Obstruction                           | No        | 78 (10.7)   | 654 (89.3)  | 0.786 |
|                                       | Yes       | 20 (11.4)   | 156 (88.6)  |       |

**TABLE 14.3:** Missing data comparison: Smoking (MAR).

| Missing data analysis: Smoking (MAR) |           | Missing     | Not missing | p      |
|--------------------------------------|-----------|-------------|-------------|--------|
| Age (years)                          | Mean (SD) | 59.4 (12.6) | 59.9 (11.8) | 0.667  |
| Sex                                  | Female    | 157 (35.3)  | 288 (64.7)  | <0.001 |
|                                      | Male      | 46 (9.5)    | 438 (90.5)  |        |
| nodes                                | Mean (SD) | 3.9 (3.9)   | 3.6 (3.5)   | 0.827  |
| Obstruction                          | No        | 164 (22.4)  | 568 (77.6)  | 0.468  |
|                                      | Yes       | 35 (19.9)   | 141 (80.1)  |        |

It takes dependent and explanatory variables, but in this context dependent just refers to the variable being tested for missingness against the explanatory variables.

Comparisons for continuous data use a Kruskal Wallis and for discrete data a chi-squared test.

As expected, a relationship is seen between Sex and Smoking (MAR) but not Smoking (MCAR).

## 14.8 For those who like an omnibus test

If you are work predominately with continuous rather than categorical data, you may find these tests from the `MissMech` package useful. The package and output is well documented, and provides two tests which can be used to determine whether data are MCAR.

```
library(MissMech)
explanatory = c("age", "nodes")
```

```
dependent = "mort_5yr"

colon_s %>%
 select(explanatory) %>%
 MissMech::TestMCARNormality()

Call:
MissMech::TestMCARNormality(data = .)
##
Number of Patterns: 2
##
Total number of cases used in the analysis: 929
##
Pattern(s) used:
age nodes Number of cases
group.1 1 1 911
group.2 1 NA 18
##
Test of normality and Homoscedasticity:

Hawkins Test:
##
P-value for the Hawkins test of normality and homoscedasticity: 7.607252e-14
##
Either the test of multivariate normality or homoscedasticity (or both) is rejected.
Provided that normality can be assumed, the hypothesis of MCAR is
rejected at 0.05 significance level.
##
Non-Parametric Test:
##
P-value for the non-parametric test of homoscedasticity: 0.6171955
##
Reject Normality at 0.05 significance level.
There is not sufficient evidence to reject MCAR at 0.05 significance level.
```

---

## 14.9 5. Decide how to handle missing data

Prior to a standard regression analysis, we can either:

- Delete the variable with the missing data
- Delete the cases with the missing data
- Impute (fill in) the missing data

- Model the missing data

## 14.10 MCAR, MAR, or MNAR

### 14.10.1 MCAR vs MAR

Using the examples, we identify that Smoking (MCAR) is missing completely at random.

We know nothing about the missing values themselves, but we know of no plausible reason that the values of the missing data, for say, people who died should be different to the values of the missing data for those who survived. The pattern of missingness is therefore felt to be MNAR.

**Common solution** Depending on the number of data points that are missing, we may have sufficient power with complete cases to examine the relationships of interest.

We therefore elect to simply omit the patients in whom smoking is missing. This is known as list-wise deletion and will be performed by default in standard regression analyses in R.

```
explanatory = c("age", "sex.factor",
 "nodes", "obstruct.factor",
 "smoking_mcar")
dependent = "mort_5yr"
fit = colon_s %>%
 finalfit(dependent, explanatory)
```

**TABLE 14.4:** Regression analysis with missing data: listwise deletion.

| Dependent: Mortality 5 year |            | Alive       | Died        | OR (univariable)          | OR (multivariable)        |
|-----------------------------|------------|-------------|-------------|---------------------------|---------------------------|
| Age (years)                 | Mean (SD)  | 59.8 (11.4) | 59.9 (12.5) | 1.00 (0.99-1.01, p=0.986) | 1.01 (1.00-1.02, p=0.200) |
| Sex                         | Female     | 243 (47.6)  | 194 (48.0)  | -                         | -                         |
|                             | Male       | 268 (52.4)  | 210 (52.0)  | 0.98 (0.76-1.27, p=0.889) | 1.02 (0.76-1.38, p=0.872) |
| nodes                       | Mean (SD)  | 2.7 (2.4)   | 4.9 (4.4)   | 1.24 (1.18-1.30, p<0.001) | 1.25 (1.18-1.33, p<0.001) |
| Obstruction                 | No         | 408 (82.1)  | 312 (78.6)  | -                         | -                         |
|                             | Yes        | 89 (17.9)   | 85 (21.4)   | 1.25 (0.90-1.74, p=0.189) | 1.53 (1.05-2.22, p=0.027) |
| Smoking (MCAR)              | Non-smoker | 358 (79.9)  | 277 (75.3)  | -                         | -                         |
|                             | Smoker     | 90 (20.1)   | 91 (24.7)   | 1.31 (0.94-1.82, p=0.113) | 1.37 (0.96-1.96, p=0.083) |

### Other considerations

- Sensitivity analysis
- Omit the variable
- Imputation
- Model the missing data

If the variable in question is thought to be particularly important, you may wish to perform a sensitivity analysis. A sensitivity analysis in this context aims to capture the effect of uncertainty on the conclusions drawn from the model. Thus, you may choose to re-label all missing smoking values as “smoker”, and see if that changes the conclusions of your analysis. The same procedure can be performed labeling with “non-smoker”.

If smoking is not associated with the explanatory variable of interest (bowel obstruction) or the outcome, it may be considered not to be a confounder and so could be omitted. That neatly deals with the missing data issue, but of course may not be appropriate.

Imputation and modelling are considered below.

#### 14.10.2 MCAR vs MAR

But life is rarely that simple.

Consider that the smoking variable is more likely to be missing if the patient is female (missing compares shows a relationship). But, say, that the missing values are not different from the observed values. Missingness is then MAR.

If we simply drop all the cases (patients) in which smoking is missing (list-wise deletion), then proportionally we drop more females than men. This may have consequences for our conclusions if sex is associated with our explanatory variable of interest or outcome.

**Common solution** `mice` is our go to package for multiple imputation. That's the process of filling in missing data using a best-estimate from all the other data that exists. When first encountered, this may not sound like a good idea.

However, taking our simple example, if missingness in smoking is predicted strongly by sex (and other observed variables), and the values of the missing data are random, then we can impute (best-guess) the missing smoking values using sex and other variables in the dataset.

Imputation is not usually appropriate for the explanatory variable of interest or the outcome variable. In both case, the hypothesis is that there is an meaningful association with other variables in the dataset, therefore it doesn't make sense to use these variables to impute them.

Here is some code to run mice. The package is well documented, and there are a number of checks and considerations that should be made to inform the imputation process. Read the documentation carefully prior to doing this yourself.

Note also `finalfit::missing_predictorMatrix()`. This provides an easy way to include or exclude variables to be imputed or to be used for imputation.

```
Multivariate Imputation by Chained Equations (mice)
library(finalfit)
library(dplyr)
library(mice)
explanatory = c("age", "sex.factor",
 "nodes", "obstruct.factor", "smoking_mar")
dependent = "mort_5yr"

Choose not to impute missing values
for explanatory variable of interest and
outcome variable.
But include in algorithm for imputation.
colon_s %>%
 select(dependent, explanatory) %>%
 missing_predictorMatrix(
 drop_from_imputed = c("obstruct.factor", "mort_5yr")
) -> predM

colon_s %>%
 select(dependent, explanatory) %>%
 # Usually run imputation with 10 imputed sets, 4 here for demonstration
 mice(m = 4, predictorMatrix = predM) %>%
 # Run logistic regression on each imputed set
 with(glm(formula(ff_formula(dependent, explanatory)),
```

```

family="binomial")) %>%
Pool and summarise results
pool() %>%
summary(conf.int = TRUE, exponentiate = TRUE) %>%
Jiggle into finalfit format
mutate(explanatory_name = rownames(.)) %>%
select(explanatory_name, estimate, `2.5 %`, `97.5 %`, p.value) %>%
condense_fit(estimate_name = "OR (multiple imputation)") %>%
remove_intercept() -> fit_imputed

iter imp variable
1 1 mort_5yr nodes obstruct.factor smoking_mar
1 2 mort_5yr nodes obstruct.factor smoking_mar
1 3 mort_5yr nodes obstruct.factor smoking_mar
1 4 mort_5yr nodes obstruct.factor smoking_mar
2 1 mort_5yr nodes obstruct.factor smoking_mar
2 2 mort_5yr nodes obstruct.factor smoking_mar
2 3 mort_5yr nodes obstruct.factor smoking_mar
2 4 mort_5yr nodes obstruct.factor smoking_mar
3 1 mort_5yr nodes obstruct.factor smoking_mar
3 2 mort_5yr nodes obstruct.factor smoking_mar
3 3 mort_5yr nodes obstruct.factor smoking_mar
3 4 mort_5yr nodes obstruct.factor smoking_mar
4 1 mort_5yr nodes obstruct.factor smoking_mar
4 2 mort_5yr nodes obstruct.factor smoking_mar
4 3 mort_5yr nodes obstruct.factor smoking_mar
4 4 mort_5yr nodes obstruct.factor smoking_mar
5 1 mort_5yr nodes obstruct.factor smoking_mar
5 2 mort_5yr nodes obstruct.factor smoking_mar
5 3 mort_5yr nodes obstruct.factor smoking_mar
5 4 mort_5yr nodes obstruct.factor smoking_mar

Use finalfit merge methods to create and compare results
colon_s %>%
summary_factorlist(dependent, explanatory, fit_id = TRUE) -> summary1

colon_s %>%
glmmuni(dependent, explanatory) %>%
fit2df(estimate_suffix = " (univariable)") -> fit_uni

colon_s %>%
glmmmulti(dependent, explanatory) %>%
fit2df(estimate_suffix = " (multivariable inc. smoking)") -> fit_multi

explanatory = c("age", "sex.factor",
"nodes", "obstruct.factor")
colon_s %>%
glmmulti(dependent, explanatory) %>%

```

```
fit2df(estimate_suffix = " (multivariable)") -> fit_multi_r

Combine to final table
fit_impute = summary1 %>%
 ff_merge(fit_uni) %>%
 ff_merge(fit_multi_r) %>%
 ff_merge(fit_multi) %>%
 ff_merge(fit_imputed) %>%
 select(-fit_id, -index)
```

**TABLE 14.5:** Regression analysis with missing data: multiple imputation using ‘mice()’.

| label         | levels     | Alive       | Died        | OR (univariable)          | OR (multivariable)        | OR (multivariable inc. smoking) | OR (multiple imputation)  |
|---------------|------------|-------------|-------------|---------------------------|---------------------------|---------------------------------|---------------------------|
| Age (years)   | Mean (SD)  | 59.8 (11.4) | 59.9 (12.5) | 1.00 (0.99-1.01, p=0.986) | 1.01 (1.00-1.02, p=0.122) | 1.02 (1.01-1.04, p=0.004)       | 1.01 (1.00-1.02, p=0.213) |
| Sex           | Female     | 243 (55.6)  | 194 (44.4)  |                           |                           |                                 |                           |
|               | Male       | 268 (56.1)  | 210 (43.9)  | 0.98 (0.76-1.27, p=0.880) | 0.98 (0.74-1.30, p=0.890) | 0.97 (0.69-1.34, p=0.836)       | 1.01 (0.77-1.34, p=0.924) |
| nodes         | Mean (SD)  | 2.7 (2.4)   | 4.9 (4.4)   | 1.24 (1.18-1.30, p<0.001) | 1.25 (1.19-1.32, p<0.001) | 1.28 (1.21-1.37, p<0.001)       | 1.23 (1.17-1.29, p<0.001) |
| Obstruction   | No         | 312 (54.0)  | 310 (45.9)  |                           |                           |                                 |                           |
|               | Yes        | 89 (51.1)   | 85 (48.9)   | 1.25 (0.90-1.74, p=0.189) | 1.36 (0.95-1.93, p=0.089) | 1.49 (1.00-2.22, p=0.052)       | 1.34 (0.95-1.90, p=0.098) |
| Smoking (MAR) | Non-smoker | 312 (54.0)  | 266 (46.0)  |                           |                           |                                 |                           |
|               | Smoker     | 87 (62.6)   | 52 (37.4)   | 0.70 (0.48-1.02, p=0.067) | -                         | 0.77 (0.51-1.16, p=0.221)       | 0.75 (0.50-1.14, p=0.178) |

By examining the coefficients, the effect of the imputation compared with the complete case analysis can be clearly seen.

### Other considerations

- Omit the variable
- Imputing factors with new level for missing data
- Model the missing data

As above, if the variable does not appear to be important, it may be omitted from the analysis. A sensitivity analysis in this context is another form of imputation. But rather than using all other available information to best-guess the missing data, we simply assign the value as above. Imputation is therefore likely to be more appropriate.

There is an alternative method to model the missing data for the categorical in this setting – just consider the missing data as a factor level. This has the advantage of simplicity, with the disadvantage of increasing the number of terms in the model. Multiple imputation is generally preferred.

```
library(dplyr)
explanatory = c("age", "sex.factor",
 "nodes", "obstruct.factor", "smoking_mar")
fit_explicit_na = colon_s %>%
 mutate(
 smoking_mar = forcats::fct_explicit_na(smoking_mar)
) %>%
 finalfit(dependent, explanatory)
```

**TABLE 14.6:** Regression analysis with missing data: explicitly modelling missing data.

| Dependent: Mortality 5 year |            | Alive       | Died        | OR (univariable)          | OR (multivariable)        |
|-----------------------------|------------|-------------|-------------|---------------------------|---------------------------|
| Age (years)                 | Mean (SD)  | 59.8 (11.4) | 59.9 (12.5) | 1.00 (0.99-1.01, p=0.986) | 1.01 (1.00-1.02, p=0.114) |
| Sex                         | Female     | 243 (47.6)  | 194 (48.0)  | -                         | -                         |
|                             | Male       | 268 (52.4)  | 210 (52.0)  | 0.98 (0.76-1.27, p=0.889) | 0.95 (0.71-1.28, p=0.743) |
| nodes                       | Mean (SD)  | 2.7 (2.4)   | 4.9 (4.4)   | 1.24 (1.18-1.30, p<0.001) | 1.25 (1.19-1.32, p<0.001) |
| Obstruction                 | No         | 408 (82.1)  | 312 (78.6)  | -                         | -                         |
|                             | Yes        | 89 (17.9)   | 85 (21.4)   | 1.25 (0.90-1.74, p=0.189) | 1.35 (0.95-1.92, p=0.099) |
| Smoking (MAR)               | Non-smoker | 312 (61.1)  | 266 (65.8)  | -                         | -                         |
|                             | Smoker     | 87 (17.0)   | 52 (12.9)   | 0.70 (0.48-1.02, p=0.067) | 0.78 (0.52-1.17, p=0.233) |
|                             | (Missing)  | 112 (21.9)  | 86 (21.3)   | 0.90 (0.65-1.25, p=0.528) | 0.85 (0.59-1.23, p=0.390) |

### 14.10.3 MNAR vs MAR

Missing not at random data is tough in healthcare. To determine if data are MNAR for definite, we need to know their value in a subset of observations (patients).

Using our example above. Say smoking status is poorly recorded in patients admitted to hospital as an emergency with an obstructing cancer. Obstructing bowel cancers may be larger or their position may make the prognosis worse. Smoking may relate to the aggressiveness of the cancer and may be an independent predictor of prognosis. The missing values for smoking may therefore not be random. Smoking may be more common in the emergency patients and may be more common in those that die.

There is no easy way to handle this. If at all possible, try to get the missing data. Otherwise, take care when drawing conclusions from analyses where data are thought to be missing not at random.



# 15

---

## *Encryption*

Health data is precious and often sensitive. Datasets may contain information patient identifiable information. Information may be clearly disclosive, such as a patient's data of birth, post/zip code, or social security number.

Other datasets may have been processed to remove the most obviously confidential information. These still require great care, as the data is usually only 'pseudoanonymised'. This may mean that the data of an individual patient is disclosive when considered as a whole - perhaps the patient had a particularly rare diagnosis. Or it may mean that the data can be combined with other datasets and in combination, individual patients can be identified.

The governance around safe data handling is one of the greatest issues facing health data scientists today. It needs to be taken very seriously and robust practices developed to ensure public confidence.

---

### 15.1 Safe practice

Storing sensitive information as raw values leaves the data vulnerable to confidentiality breaches. This is true even when you are working in a 'safe' environment, such as a secure server.

It is best to simply remove as much confidential information from records whenever possible. If the data is not present, then it cannot be compromised.

This may not be possible if the data can never be re-associated with an individual. This may be a problem if, for example, auditors of a clinical trial need to re-identify an individual from the trial data. A study ID can be used, but that still requires the confidential data to be stored and available in a lookup table in another file.

- A formal short section on data governance best practice here?\*
- 

## 15.2 `encryptr` package

The `encryptr` package allows users to store confidential data in a pseudoanonymised form, which is far less likely to result in re-identification.

Either columns in data frames/tibbles or whole files can be directly encrypted from R using strong RSA encryption.

The basis of RSA encryption is a public/private key pair and is the method used of many modern encryption applications. The public key can be shared and is used to encrypt the information.

The private key is sensitive and should not be shared. The private key requires a password to be set, which should follow modern rules on password complexity. You know what you should do! If lost, it cannot be recovered.

---

## 15.3 Get the package

The `encryptr` package can be installed in the standard manner or the development version can be obtained from Github.

Full documentation is maintained separately at [encrypt-r.org](https://encrypt-r.org)<sup>1</sup>.

---

<sup>1</sup><https://encrypt-r.org>

```
install.packages("encryptr")

Or the development version from Github
devtools::install_github("SurgicalInformatics/encryptr")
```

## 15.4 Get the data

An example dataset containing the addresses general practitioners (family doctors) in Scotland is included in the package.

```
library(encryptr)
gp
#> # A tibble: 1,212 x 12
#> organisation_code name address1 address2 address3 city postcode
#> <chr> <chr> <chr> <chr> <chr> <chr>
#> 1 S10002 MUIRHE... LIFF RO... MUIRHEAD NA DUND... DD2 5NH
#> 2 S10017 THE BL... CRIEFF ... KING ST... NA CRIE... PH7 3SA
```

## 15.5 Generate private/public keys

The `genkeys()` function generates a public and private key pair. A password is required to be set in the dialogue box for the private key. Two files are written to the active directory.

The default name for the private key is:

- `id_rsa`

And for the public key name is generated by default:

- `id_rsa.pub`

If the private key file is lost, nothing encrypted with the public

key can be recovered. Keep this safe and secure. Do not share it without a lot of thought on the implications.

```
genkeys()
#> Private key written with name 'id_rsa'
#> Public key written with name 'id_rsa.pub'
```

## 15.6 Encrypt columns of data

Once the keys are created, it is possible to encrypt one or more columns of data in a data frame/tibble using the public key. Every time RSA encryption is used it will generate a unique output. Even if the same information is encrypted more than once, the output will always be different. It is therefore not possible to match two encrypted values.

These outputs are also secure from decryption without the private key. This may allow sharing of data within or between research teams without sharing confidential data.

Encrypting columns to a ciphertext is straightforward. As stated above, an important principle is dropping sensitive data which is never going to be required.

```
library(dplyr)
gp_encrypt = gp %>%
 select(-c(name, address1, address2, address3)) %>%
 encrypt(postcode)
gp_encrypt

#> A tibble: 1,212 x 8
#> organisation_code city county postcode
#> <chr> <chr> <chr> <chr>
#> 1 S10002 DUNDEE ANGUS 796284eb46ca...
#> 2 S10017 CRIEFF PERTHSHIRE 639dfc076ae3...
```

## 15.7 Decrypt specific information only

Decryption requires the private key generated using `genkeys()` and the password set at the time. The password and file are not replaceable so need to be kept safe and secure. It is important to only decrypt the specific pieces of information that are required. The beauty of this system is that when decrypting a specific cell, the rest of the data remain secure.

```
gp_encrypt %>%
 slice(1:2) %>% # Only decrypt the rows and columns necessary
 decrypt(postcode)

#> A tibble: 1,212 x 8
#> organisation_code city county postcode
#> <chr> <chr> <chr> <chr>
#> 1 S10002 DUNDEE ANGUS DD2 5NH
#> 2 S10017 CRIEFF PERTHSHIRE PH7 3SA
```

## 15.8 Using a lookup table

Rather than storing the ciphertext in the working dataframe, a lookup table can be used as an alternative. Using `lookup = TRUE` has the following effects:

- returns the dataframe / tibble with encrypted columns removed and a `key` column included;
- returns the lookup table as an object in the R environment;
- creates a lookup table `.csv` file in the active directory. file of the lookup

```
gp_encrypt = gp %>%
 select(-c(name, address1, address2, address3)) %>%
 encrypt(postcode, telephone, lookup = TRUE)
```

```
#> Lookup table object created with name 'lookup'
#> Lookup table written to file with name 'lookup.csv'

gp_encrypt

#> A tibble: 1,212 x 7
#> key organisation_code city county opendate
#> <int> <chr> <chr> <chr> <date>
#> 1 1 S10002 DUNDEE ANGUS 1995-05-01
#> 2 2 S10017 CRIEFF PERTHSHIRE 1996-04-06
```

The file creation can be turned off with `write_lookup = FALSE` and the name of the lookup can be changed with `lookup_name = "anyNameHere"`. The created lookup file should be itself encrypted using the method below.

Decryption is performed by passing the lookup object or file to the `decrypt()` function.

```
gp_encrypt %>%
 decrypt(postcode, telephone, lookup_object = lookup)

Or
gp_encrypt %>%
 decrypt(postcode, telephone, lookup_path = "lookup.csv")

#> A tibble: 1,212 x 8
#> postcode telephone organisation_code city county opendate
#> <chr> <chr> <chr> <chr> <chr> <date>
#> 1 DD2 5NH 01382 580264 S10002 DUNDEE ANGUS 1995-05-01
#> 2 PH7 3SA 01764 652283 S10017 CRIEFF PERTHSHIRE 1996-04-06
```

## 15.9 Encrypting a file

Encrypting the object within R has little point if a file with the disclosive information is still present on the system. Files can be encrypted and decrypted using the same set of keys.

To demonstrate, the included dataset is written as a .csv file.

```
write.csv(gp, "gp.csv")
encrypt_file("gp.csv")
#> Encrypted file written with name 'gp.csv.encryptr.bin'
```

Check that the file can be decrypted prior to removing the original file from your system.

Warning: it is strongly suggested that the original unencrypted data file is stored as a back-up in case unencryption is not possible, e.g., the private key file or password is lost

The `decrypt_file` function will not allow the original file to be overwritten, therefore use the option to specify a new name for the unencrypted file.

```
decrypt_file("gp.csv.encryptr.bin", file_name = "gp2.csv")
#> Decrypted file written with name 'gp2.csv'
```

---

## 15.10 Ciphertexts are no matchable

The ciphertext produced for a given input will change with each encryption. This is a feature of the RSA algorithm. Ciphertexts should not therefore be attempted to be matched between datasets encrypted using the same public key. This is a conscious decision given the risks associated with sharing the necessary details.

---

## 15.11 Providing a public key

In collaborative projects where data may be pooled, a public key can be made available by you via a link to enable collaborators to

encrypt sensitive data. This provides a robust method for sharing potentially disclosive data points.

```
gp_encrypt = gp %>%
 select(-c(name, address1, address2, address3)) %>%
 encrypt(postcode, telephone, public_key_path = "https://argonaut.is.ed.ac.uk/public/id_rsa.pub")
```

---

### 15.12 Use in clinical trials

Another potential application is maintaining blinding / allocation concealment in randomised controlled clinical trials. Using the same method of encryption, it is possible to encrypt the participant allocation group, allowing the sharing of data without compromising blinding. If other members of the trial team are permitted to see treatment allocation (unblinded), then the decryption process can be followed to reveal the group allocation.

---

### 15.13 Caution

All confidential information must be treated with the utmost care. Data should never be carried on removable devices or portable computers. Data should never be sent by open email. Encrypting data provides some protection against disclosure. But particularly in healthcare, data often remains potentially disclosive (or only pseudonymised) even after encryption of identifiable variables. Treat it with great care and respect.

# 16

---

## *RStudio settings, good practise*

---

### 16.1 Installation

#### 16.1.1 R

Download and install the latest version of R from:

<https://www.stats.bris.ac.uk/R/>

*Windows:* Download R for Windows - base - Download R 3.6.0 for Windows

*Mac:* Download R for (Mac) OS X - R-3.6.0.pkg

#### 16.1.2 RStudio

Then download and install RStudio:

<https://www.rstudio.com/products/rstudio/download/#download>

*Windows:* RStudio 1.2.1335 - Windows 7+

*Mac:* RStudio 1.2.1335 - Mac OS X 10.12+ (64-bit)

#### 16.1.3 R packages

Open RStudio and copy-paste the lines below into the Console and press Enter:

```
healthyr_packages = c("tidyverse", "boot", "finalfit", "flexdashboard", "gapminder", "here", "kableExtra")
install.packages(healthyr_packages)
```

**Then Restart R** before running the next two lines (same as before, copy-paste to the Console):

```
remotes:::install_github("thomasp85/patchwork")
tinytex:::install_tinytex()
```

If it asks you “Do you want to restart your R session”, press **Yes** the first time. If it immediately asks again, press **No**.

Code to check tinyTex and Tex working as expected:

```
tinytex:::::is_tinytex()

If installed should return `TRUE`.
Notice the triple colon, this is because it's an internal variable name.
```

#### 16.1.4 Troubleshooting 3.3

1. ~“can’t find remotes”: try doing step 3.2 (restart) again. We only just installed `remotes::` in 3.1 so RStudio needs a restart before we can start using it.
2. ~“TeX failed, you already have a TeX distribution...”. If you already have LaTex installed on your computer, the `tinytex::install_tinytex()` one is not necessary and might not work. This is fine, there is no harm in copy-pasting in anyway if you are not sure.

When working with data, don’t copy or type code directly into the Console. We will only be using the Console for viewing output, warnings, and errors (and installing packages as in the previous section). All code should be in a script and executed (=Run) using Control+Enter (line or section) or Control+Shift+Enter (whole script). Make sure you are always working in a project (the right-top corner of your RStudio interface should say “HealthyR”).

## 16.2 Script vs Console

Throughout this course, don't copy or type code directly into the Console. We will only be using the Console for viewing output, warnings, and errors. All code should be in a script and executed (=run) using Ctrl+Enter (line or section) or Ctrl+Shift+Enter (whole script). Make sure you are always working in a project (the right-top corner of your RStudio interface should say "HealthyR").

---

## 16.3 Starting with a blank canvas

In the first session we loaded some data that we then plotted. When we import data, R stores it and displays it in the Environment tab.

It's good practice to restart R before commencing new work. This is to avoid accidentally using the wrong data or functions stored in the environment.

Restarting R only takes a second!

- Restart R (Ctrl+Shift+F10 or select it from Session -> Restart R).

RStudio has a default setting that is no longer considered best practice. You should do this once:

- Go to Tools -> Global Options -> General and set "Save .RData on exit" to Never. This does not mean you can't or shouldn't save your work in .RData files. But it is best to do it consciously and load exactly what you need to load, rather than letting R always save and load everything for you, as this could also include broken data or objects.



---

## **Bibliography**

Bryan, J. (2017). *gapminder: Data from Gapminder*. R package version 0.3.0.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.



## **Index**

- AIC, 187  
analysis of variance (ANOVA), 144  
binary data, 219  
Bonferroni correction, 148  
C-statistic, 246  
**categorical data**  
    chi-squared test, 210  
    finalfit, 209  
    Fisher's exact test, 210  
    quantiles, 203  
    summarising, 209  
**categorical data**, 197  
    convert from continuous, 201  
chi-squared test, 210  
chunks, 289  
confounding, 171  
**continuous data**, 129  
Cox proportional hazards regression, 266  
effect modification, 166  
**exporting**, 299  
    factors, 197  
    false discovery rate, 148  
**file structure**, 295  
Fisher's exact test, 210  
**functions**  
    aov, 145  
    chisq.test, 212  
    coef, 243  
    confint, 243  
    cox.zph, 269  
    cut, 204  
    factor, 200  
    fct\_collapse, 208  
    fct\_recode, 200  
    fct\_relevel, 208  
    ff\_glimpse, 131  
    ff\_label, 200  
    finalfit, 190  
    fisher.test, 213  
    ggpairs, 238  
    glance, 178  
    glimpse, 131  
    glm, 242, 249  
    glmer, 257  
    hr\_plot, 272  
    kruskal.test, 152  
    lm, 175  
    missing\_glimpse, 131  
    or\_plot, 253  
    pairwise.t.test, 147, 148  
    quantile, 204  
    strata, 270  
    summary\_factorlist, 153, 214  
    Surv, 264  
    surv\_plot, 266  
    survfit, 265

- t.test, 138, 141
- tidy, 178
- wilcox.test, 152
- Hosmer-Lemeshow test, 246
- HTML, 288
- interaction terms, 168
- Kaplan Meier estimator, 264
- knitr, 293, 303
- linear regression**, 159
  - AIC, 187
  - assumptions, 162
  - coefficients, 177
  - confounding, 171
  - effect modification, 166
  - finalfit, 189
  - fitted line, 160
  - interactions, 168
  - model fitting principles, 185
  - multivariable, 179
  - r-squared, 171
- log-rank test, 264
- logistic regression**, 219
  - AIC, 245
  - assumptions, 236
  - binary data, 219
  - C-statistic, 246
  - confounding, 226
  - correlated groups, 254
  - effect modification, 226
  - finalfit, 246
  - fitted line, 223
  - hierarchical, 254
- Hosmer-Lemeshow test, 246
- interactions, 226
- loess, 236
- mixed effects, 254
- model fitting, 241, 246
- model fitting principles, 244
- multicollinearity, 237
- multilevel, 254
- odds and probabilities, 221
- odds ratio, 222
- odds ratio plot, 252
- random effects, 254
- markdown**, 285
- Microsoft Word, 288, 303
- non-parametric tests**, 149
  - Mann-Whitney U, 151
  - Wilcoxon rank sum, 151
- notebooks**, 285
  - odds ratio, 222
- pairwise testing, 146
- PDF, 288, 303
- plotting**
  - boxplot, 135
  - geom\_boxplot, 135
  - geom\_line, 140
  - geom\_qq, 134
  - geom\_qq\_line, 134
  - hazard ratio plot, 272
  - hr\_plot, 272
  - odds ratio, 252
  - or\_plot, 253
  - patchwork, 151
  - surv\_plot, 266
- symbols**
  - AND &, 24
  - assignment =, 14
  - comment #, 27

- equal =, 24
- greater or equal >=, 24
- greater than >, 24
- less or equal <=, 24
- less than <, 24
- not !, 24
- OR |, 24
- pipe %>%:, 16
- select column \$, 11
- t-test**, 137
  - one-sample, 142
  - paired, 139
  - two-sample, 137
- time-to-event / survival**,  
261
  - assumptions, 268
  - censoring, 263
  - cluster, 270
  - competing risks regression,  
273
  - correlated groups, 270
  - Cox proportional hazards  
regression, 266
  - frailty, 270
  - hazard ratio plot, 272
  - Kaplan Meier estimator,  
264
  - Kaplan Meier plot, 265
  - life table, 265
  - log-rank test, 264
  - mixed effects, 270
  - multilevel, 270
  - random effects, 270
  - stratified models, 270
  - testing for proportional  
hazards, 268
- transformations, 149
- workflow**, 295
- YAML header, 289