

*Ewen Harrison and Riinu Ots*

---

# ***HealthyR: R for healthcare data analysis***

Never trust a data scientist - they are always plotting.

---

---

## *Contents*

---

<b>List of Tables</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xv</b>
<b>Preface</b>	<b>xix</b>
<b>I Data wrangling and visualisation</b>	<b>1</b>
<b>1 Your first R plots</b>	<b>3</b>
1.1 Data . . . . .	3
1.2 First plot . . . . .	4
1.2.1 Question . . . . .	4
1.2.2 Exercise . . . . .	5
1.3 Comparing bars of different height . . . . .	6
1.3.1 Stretch each bar to 100% . . . . .	6
1.3.2 Plot each bar next to each other . . . . .	6
1.4 Facets (panels) . . . . .	7
1.5 Extra: using aesthetics outside of the aes() . . . . .	8
1.5.1 Setting a constant fill . . . . .	8
1.5.2 Exercise . . . . .	9
1.5.3 Exercise . . . . .	10
1.6 Two geoms for barplots: <code>geom_bar()</code> or <code>geom_col()</code> . . . . .	10
1.7 Solutions . . . . .	11
1.8 Tidyverse packages: <code>ggplot2</code> , <code>dplyr</code> , <code>tidyverse</code> , etc. . . . .	11
<b>2 R Basics</b>	<b>13</b>
2.1 Getting help . . . . .	13
2.2 Objects and functions . . . . .	13
2.3 Working with Objects . . . . .	18
2.4 Pipe - <code>%&gt;%</code> . . . . .	20

2.4.1	When pipe sends data to the wrong place: <code>use , data = .</code> to direct it . . . . .	21
2.5	Reading data into R . . . . .	22
2.5.1	Reading in the Global Burden of Disease example dataset (short version) . . . . .	24
2.6	Operators for filtering data . . . . .	27
2.6.1	Worked examples . . . . .	30
2.7	The combine function: <code>c()</code> . . . . .	31
2.8	Missing values (NAs) and filters . . . . .	32
2.9	Variable types and why we care . . . . .	36
2.10	Numeric variables (continuous) . . . . .	38
2.11	Character variables (categorical, IDs, free text) . .	40
2.12	Factor variables (categorical) . . . . .	42
2.13	Date/time variables . . . . .	43
2.14	Creating new columns - <code>mutate()</code> . . . . .	47
2.14.1	Worked example/exercise . . . . .	50
2.15	Conditional calculations - <code>if_else()</code> . . . . .	51
2.16	Create labels - <code>paste()</code> . . . . .	52
2.17	Joining multiple datasets . . . . .	53
2.17.1	Further notes about the joins . . . . .	55
<b>3</b>	<b>Summarising data</b>	<b>59</b>
3.1	Dataset: Global Burden of Disease (year, cause, sex, income, deaths) . . . . .	59
3.2	Aggregating: <code>group_by()</code> , <code>summarise()</code> . . . . .	62
3.3	Add new columns: <code>mutate()</code> . . . . .	63
3.3.1	percentages formatting: <code>percent()</code> . . . . .	64
3.4	<code>summarise()</code> vs <code>mutate()</code> . . . . .	65
3.5	Common arithmetic functions - <code>sum()</code> , <code>mean()</code> , <code>median()</code> , etc. . . . .	68
3.6	<code>select()</code> columns . . . . .	69
3.7	Reshaping data - long vs wide format . . . . .	69
3.7.1	<code>spread()</code> values from rows into columns . .	71
3.7.2	<code>gather()</code> values from columns to rows . . .	74
3.8	Sorting: <code>arrange()</code> . . . . .	76
3.9	Factor handling . . . . .	77
3.9.1	Exercise . . . . .	77

3.9.2 <code>fct_collapse()</code> - grouping levels together . . . . .	77
3.9.3 <code>fct_relevel()</code> - change the order of levels . . . . .	78
3.9.4 <code>fct_recode()</code> - rename levels . . . . .	78
3.9.5 Converting factors to numbers . . . . .	78
3.9.6 Exercise . . . . .	78
3.10 Long Exercise . . . . .	80
3.11 Extra: formatting a table for publication . . . . .	80
3.12 Solution: Long Exercise . . . . .	81
<b>4 Different types of plots</b>	<b>83</b>
4.1 Data . . . . .	83
4.2 Scatter plots/bubble plots - <code>geom_point()</code> . . . . .	84
4.2.1 Exercise . . . . .	84
4.3 Line chart/timeplot - <code>geom_line()</code> . . . . .	85
4.3.1 Exercise . . . . .	87
4.3.2 Advanced example . . . . .	88
4.3.3 Advanced Exercise . . . . .	88
4.4 Box-plot - <code>geom_boxplot()</code> . . . . .	89
4.4.1 Exercise . . . . .	89
4.4.2 Dot-plot - <code>geom_dotplot()</code> . . . . .	91
4.5 Barplot - <code>geom_bar()</code> and <code>geom_col()</code> . . . . .	91
4.5.1 Exercise . . . . .	92
4.6 All other types of plots . . . . .	93
4.7 Specifying <code>aes()</code> variables . . . . .	94
4.8 Extra: Optional exercises . . . . .	94
4.8.1 Exercise . . . . .	94
4.8.2 Exercise . . . . .	97
4.9 Solutions . . . . .	98
<b>5 Fine tuning plots</b>	<b>101</b>
5.1 Data and initial plot . . . . .	101
5.2 Scales . . . . .	102
5.2.1 Logarithmic . . . . .	102
5.2.2 Expand limits . . . . .	103
5.2.3 Zoom in . . . . .	105
5.2.4 Exercise . . . . .	106
5.2.5 Axis ticks . . . . .	106

5.2.6 Swap the axes . . . . .	107
5.3 Colours . . . . .	108
5.3.1 Using the Brewer palettes: . . . . .	108
5.3.2 Legend title . . . . .	108
5.3.3 Choosing colours manually . . . . .	109
5.4 Titles and labels . . . . .	111
5.4.1 Annotation . . . . .	111
5.4.2 Annotation with a superscript and a variable . . . . .	113
5.5 Text size . . . . .	114
5.5.1 Legend position . . . . .	115
5.6 Saving your plot . . . . .	117
<b>II Data analysis</b>	<b>119</b>
<b>6 Working with continuous outcome variables</b>	<b>123</b>
6.1 Continuous data . . . . .	123
6.2 The Question . . . . .	124
6.3 Get the data . . . . .	124
6.4 Check the data . . . . .	124
6.5 Plot the data . . . . .	126
6.5.1 Histogram . . . . .	126
6.5.2 Q-Q plot . . . . .	127
6.5.3 Boxplot . . . . .	128
6.6 Compare the means of two groups . . . . .	130
6.6.1 T-test . . . . .	130
6.6.2 Two-sample <i>t</i> -tests . . . . .	130
6.6.3 When pipe sends data to the wrong place: use <code>, data = .</code> to direct it . . . . .	132
6.6.4 Paired <i>t</i> -tests . . . . .	133
6.7 Compare the mean of one group . . . . .	135
6.7.1 One sample <i>t</i> -tests . . . . .	135
6.8 Compare the means of more than two groups . . . . .	137
6.8.1 Plot the data . . . . .	137
6.8.2 ANOVA . . . . .	137
6.8.3 Assumptions . . . . .	139
6.8.4 Pairwise testing and multiple comparisons	139

6.9	Non-parametric data . . . . .	142
6.9.1	Transforming data . . . . .	142
6.9.2	Non-parametric test for comparing two groups . . . . .	144
6.9.3	Non-parametric test for comparing more than two groups . . . . .	145
6.10	Finalfit approach . . . . .	146
6.11	Conclusions . . . . .	147
6.12	Exercises . . . . .	147
6.12.1	Exercise 1 . . . . .	147
6.12.2	Exercise 2 . . . . .	147
6.12.3	Exercise 3 . . . . .	148
6.12.4	Exercise 4 . . . . .	148
6.13	Exercise solutions . . . . .	148
<b>7</b>	<b>Linear regression</b>	<b>153</b>
7.1	Regression . . . . .	153
7.1.1	The Question (1) . . . . .	154
7.1.2	Fitting a regression line . . . . .	154
7.1.3	When the line fits well . . . . .	156
7.1.4	The fitted line and the linear equation . . . . .	160
7.1.5	Effect modification . . . . .	162
7.1.6	R-squared and model fit . . . . .	164
7.1.7	Confounding . . . . .	166
7.1.8	Summary . . . . .	166
7.2	Fitting simple models . . . . .	168
7.2.1	The Question (2) . . . . .	168
7.2.2	Get the data . . . . .	168
7.2.3	Check the data . . . . .	168
7.2.4	Plot the data . . . . .	168
7.2.5	Simple linear regression . . . . .	169
7.2.6	Multivariable linear regression . . . . .	174
7.2.7	Check assumptions . . . . .	178
7.3	Fitting more complex models . . . . .	179
7.3.1	The Question (3) . . . . .	179
7.3.2	Model fitting principles . . . . .	179
7.3.3	AIC . . . . .	181

7.3.4	Get the data . . . . .	181
7.3.5	Check the data . . . . .	182
7.3.6	Plot the data . . . . .	182
7.3.7	Linear regression with <code>finalfit</code> . . . . .	183
7.3.8	Summary . . . . .	189
<b>8</b>	<b>Working with categorical outcome variables</b>	<b>191</b>
8.1	Factors . . . . .	191
8.2	The Question . . . . .	192
8.3	Get the data . . . . .	192
8.4	Check the data . . . . .	192
8.5	Recode the data . . . . .	194
8.6	Should I convert a continuous variable to a categorical variable? . . . . .	195
8.6.1	Equal intervals vs quantiles . . . . .	197
8.7	Plot the data . . . . .	199
8.8	Group factor levels together - <code>fct_collapse()</code> . . . . .	201
8.9	Change the order of values within a factor - <code>fct_relevel()</code> . . . . .	202
8.10	Summarising factors with <code>Finalfit</code> . . . . .	203
8.11	Pearson's chi-squared and Fisher's exact tests . . . . .	204
8.11.1	Base R . . . . .	205
8.12	Fisher's exact test . . . . .	207
8.13	Chi-squared / Fisher's exact test using <code>Finalfit</code> . . . . .	208
8.14	Exercise . . . . .	210
8.15	Exercise . . . . .	211
8.16	Exercise . . . . .	211
8.17	Exercise . . . . .	211
<b>9</b>	<b>Logistic regression</b>	<b>213</b>
9.1	What is Logistic Regression? . . . . .	213
9.2	Definitions . . . . .	216
9.3	Odds and probabilities . . . . .	216
9.3.1	Odds ratios . . . . .	217
9.4	Melanoma dataset . . . . .	219
9.4.1	Doing logistic regression in R . . . . .	219
9.5	Setting up your data . . . . .	220

<i>Contents</i>	ix
9.5.1 Worked Example . . . . .	220
9.6 Creating categories . . . . .	221
9.6.1 Exercise . . . . .	221
9.6.2 Always plot your data first! . . . . .	222
9.7 Basic: One explanatory variable (predictor) . . . . .	224
9.7.1 Worked example . . . . .	224
9.7.2 Exercise . . . . .	226
9.8 Finalfit package . . . . .	227
9.9 Summarise a list of variables by another variable	227
9.10 <code>finalfit</code> function for logistic regression . . . . .	228
9.11 Adjusting for multiple variables in R . . . . .	229
9.11.1 Worked Example . . . . .	229
9.11.2 Exercise . . . . .	230
9.12 Advanced: Fitting the best model . . . . .	231
9.12.1 Extra material: Diagnostics plots . . . . .	232
<b>10 Time-to-event data and survival</b>	<b>235</b>
10.1 The Question . . . . .	235
10.2 Get and check data . . . . .	236
10.3 Death status . . . . .	236
10.4 Time and censoring . . . . .	237
10.5 Recode the data . . . . .	237
10.6 Kaplan-Meier survival estimator . . . . .	238
10.6.1 KM analysis for whole cohort . . . . .	238
10.6.2 Model . . . . .	238
10.6.3 Life table . . . . .	239
10.7 Kaplan Meier plot . . . . .	239
10.8 Cox-proportional hazards regression . . . . .	240
10.8.1 Univariable and multivariable models . . . . .	241
10.8.2 Reduced model . . . . .	242
10.8.3 Testing for proportional hazards . . . . .	242
10.8.4 Stratified models . . . . .	243
10.8.5 Correlated groups of observations . . . . .	244
10.8.6 Hazard ratio plot . . . . .	245
10.9 Competing risks regression . . . . .	246
10.10 Summary . . . . .	248
10.11 Exercise . . . . .	248

10.11.1 Exercise . . . . .	249
10.12 Dates in R . . . . .	250
10.12.1 Converting dates to survival time . . . . .	250
10.13 Solutions . . . . .	251
<b>III Workflow</b>	<b>253</b>
<b>11 Notebooks and markdown</b>	<b>255</b>
<b>12 Missing data</b>	<b>257</b>
12.1 The problem of missing data . . . . .	257
12.2 Some confusing terminology . . . . .	258
12.2.1 Missing completely at random (MCAR) .	258
12.2.2 Missing at random (MAR) . . . . .	258
12.2.3 Missing not at random (MNAR) . . . . .	258
12.3 Ensure your data are coded correctly: ff_glimpse	259
12.4 The Question . . . . .	259
12.5 2. Identify missing values in each variable: missing_plot . . . . .	261
12.6 3. Look for patterns of missingness: missing_pattern . . . . .	262
12.6.1 Make sure you include missing data in demographics tables . . . . .	264
12.7 4. Check for associations between missing and observed data: missing_pairs   missing_compare .	264
12.8 For those who like an omnibus test . . . . .	268
12.9 5. Decide how to handle missing data . . . . .	269
12.10 MCAR, MAR, or MNAR . . . . .	270
12.10.1 <b>MCAR</b> vs MAR . . . . .	270
12.10.2 MCAR vs <b>MAR</b> . . . . .	271
12.10.3 <b>MNAR</b> vs MAR . . . . .	275
<b>13 Encryption</b>	<b>277</b>
13.1 Safe practice . . . . .	277
13.2 <b>Encryptr</b> package . . . . .	278
13.3 Get the package . . . . .	278
13.4 Get the data . . . . .	279
13.5 Generate private/public keys . . . . .	279

<i>Contents</i>	xi
13.6 Encrypt columns of data . . . . .	280
13.7 Decrypt specific information only . . . . .	281
13.8 Using a lookup table . . . . .	281
13.9 Encrypting a file . . . . .	282
13.10 Ciphertexts are no matchable . . . . .	283
13.11 Providing a public key . . . . .	283
13.12 Use in clinical trials . . . . .	284
13.13 Caution . . . . .	284
<b>14 Exporting tables and plots</b>	<b>285</b>
<b>15 RStudio settings, good practise</b>	<b>287</b>
15.1 Script vs Console . . . . .	287
15.2 Starting with a blank canvas . . . . .	287
<b>Bibliography</b>	<b>289</b>
<b>Index</b>	<b>291</b>



---

## ***List of Tables***

---

2.1	Example of a table (=tibble once read into R), including missing values denoted NA (Not applicable/Not available). . . . .	14
2.2	Deaths per year from three broad disease categories (short version of the Global Burden of Disease example dataset). . . . .	25
2.3	Filtering operators. . . . .	30
3.1	Deaths per year from three broad disease categories, sex, and World Bank county-level income groups. . . . .	60
3.2	Global Burden of Disease data in human-readable wide format. This is not tidy data. . . . .	69
3.3	Global Burden of Disease data in analysis-friendly long format. This is tidy data. . . . .	70
3.4	Exercise: putting the cause variable into the wide format using spread. . . . .	74
6.1	Life expectancy, population and GDPperCap in Africa 1982 v 2007 . . . . .	147
7.1	WCGS data, ff_glimpse: continuous . . . . .	182
7.2	WCGS data, ff_glimpse: categorical . . . . .	182
7.3	Linear regression: Systolic blood pressure by personality type. . . . .	184
7.4	Model metrics: Systolic blood pressure by personality type. . . . .	184
7.5	Multivariable linear regression: Systolic blood pressure by personality type and weight. . . . .	185
7.6	Multivariable linear regression metrics: Systolic blood pressure by personality type and weight. . . . .	185

7.7	Multivariable linear regression: Systolic blood pressure by available explanatory variables. . . . .	186
7.8	Model metrics: Systolic blood pressure by available explanatory variables. . . . .	187
7.9	Multivariable linear regression: Systolic blood pressure using BMI. . . . .	188
7.10	Model metrics: Systolic blood pressure using BMI. . . . .	188
7.11	Multivariable linear regression: Systolic blood pressure by available explanatory variables and reduced model. . . . .	188
7.12	Model metrics: Systolic blood pressure by available explanatory variables (top) with reduced model (bottom). . . . .	189
8.1	CAPTION . . . . .	203
8.2	CAPTION . . . . .	204
8.3	CAPTION . . . . .	208
8.4	CAPTION . . . . .	209
8.5	CAPTION . . . . .	209
8.6	CAPTION . . . . .	210
10.1	CAPTION . . . . .	241
10.2	CAPTION . . . . .	241
10.3	CAPTION . . . . .	242
10.4	CAPTION . . . . .	244
10.5	CAPTION . . . . .	245
10.6	CAPTION . . . . .	247
12.1	CAPTION . . . . .	265
12.2	CAPTION . . . . .	268
12.3	CAPTION . . . . .	268
12.4	CAPTION . . . . .	271
12.5	CAPTION . . . . .	274
12.6	CAPTION . . . . .	275

---

## ***List of Figures***

---

2.1	This is not a pipe. René Magritte inspired artwork by Stefan Milton Bache (creator of %>% in R). Image source: <a href="https://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html">https://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html</a> . . . . .	21
2.2	View or import a data file. . . . .	22
2.3	Import: Some of the special settings your data file might have. . . . .	23
2.4	After using the Import Dataset window, copy-paste the resulting code into your script. . . . .	23
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.	26
3.1	Global Burden of Disease data with subgroups: cause, sex, World Bank income group. . . . .	61
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. . . . .	70
6.1	Histogram: country life expectancy by continent and year . . . . .	127
6.2	Q-Q plot: country life expectancy by continent and year . . . . .	128
6.3	Boxplot: country life expectancy by continent and year . . . . .	129
6.4	Boxplot with jitter points: country life expectancy by continent and year . . . . .	130
6.5	Line plot: Change in life expectancy in Asian countries from 2002 to 2007 . . . . .	133

6.6	Boxplot: Life expectancy in selected continents for 2007 . . . . .	138
6.7	Diagnostic plots: ANOVA model of life expectancy by continent for 2007 . . . . .	140
6.8	Histogram: Log transformation of life expectancy for countries in Africa 2002 . . . . .	143
6.9	Panels plots: histogram, Q-Q, boxplot for life expectancy in Africa 1992 v 2007 . . . . .	145
7.1	The anatomy of a regression plot. . . . .	155
7.2	How a regression line is fitted. . . . .	157
7.3	Regression diagnostics. Does this also appear in the contents. What about this? . . . . .	159
7.4	Linking the fitted line, regression equation and R output. . . . .	161
7.5	Causal pathways, effect modification and confounding. . . . .	163
7.6	Multivariable linear regression with additive and multiplicative effect modification. . . . .	165
7.7	Multivariable linear regression with confounding of coffee drinking by smoking. . . . .	167
7.8	Scatterplot with fitted line plot: Life expectancy by year in European countries . . . . .	169
7.9	Scatterplot: Life expectancy by year Turkey and Europe. . . . .	170
7.10	Scatter and line plot. Life expectancy in Turkey and the UK - univariable fit. . . . .	175
7.11	Scatter and line plot. Life expectancy in Turkey and the UK - multivariable additive fit. . . . .	176
7.12	Scatter and line plot. Life expectancy in Turkey and the UK - multivariable multiplicative fit. . . . .	177
7.13	Scatter and line plot. Systolic blood pressure by weight and personality type. . . . .	183
8.1	cut a continuous variable into a categorical variable. . . . .	197
8.2	Bar chart: outcome after surgery for patients with ulcerated melanoma. . . . .	199

## *List of Figures*

xvii

8.3	Bar chart: outcome after surgery for patients with ulcerated melanoma, reversed levels.	200
8.4	Facetted bar plot: outcome after surgery for patients with ulcerated melanoma aggregated by sex and age.	202



---

## **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 ...

## Software information and conventions

I used the **knitr** package (Xie, 2015) and the **bookdown** package (Xie, 2018) to compile my book. My R session information is shown below:

```
xfun::session_info()

## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.5 LTS
##
## Locale:
##   LC_CTYPE=en_GB.UTF-8
##   LC_NUMERIC=C
##   LC_TIME=en_GB.UTF-8
##   LC_COLLATE=en_GB.UTF-8
##   LC_MONETARY=en_GB.UTF-8
##   LC_MESSAGES=en_GB.UTF-8
##   LC_PAPER=en_GB.UTF-8
##   LC_NAME=C
##   LC_ADDRESS=C
##   LC_TELEPHONE=C
##   LC_MEASUREMENT=en_GB.UTF-8
##   LC_IDENTIFICATION=C
##
## Package version:
##   base64enc_0.1.3 bookdown_0.12 compiler_3.6.1
##   digest_0.6.20   evaluate_0.14 glue_1.3.1
##   graphics_3.6.1 grDevices_3.6.1 highr_0.8
##   htmltools_0.3.6 jsonlite_1.6 knitr_1.23
##   magrittr_1.5    markdown_1.0 methods_3.6.1
##   mime_0.7       Rcpp_1.0.2 rmarkdown_1.14
##   stats_3.6.1    stringi_1.4.3 stringr_1.4.0
##   tinytex_0.14   tools_3.6.1 utils_3.6.1
##   xfun_0.8       yaml_2.2.0
```

Package names are in bold text (e.g., **rmarkdown**), and inline code and filenames are formatted in a typewriter font (e.g., `knitr::knit('foo.Rmd')`). Function names are followed by parentheses (e.g., `bookdown::render_book()`).

---

## Acknowledgments

A lot of people helped me when I was writing the book.

Frida Gomam  
on the Mars

---

## Installation

- Download R

<https://www.r-project.org/>

- Install RStudio

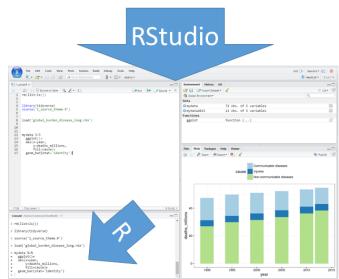
<https://www.rstudio.com/products/rstudio/>

- Install packages (copy these lines into the Console in RStudio):

```
install.packages("tidyverse")
install.packages("gapminder")
install.packages("gmodels")
install.packages("Hmisc")
install.packages("devtools")
devtools::install_github("ewenharrison/finalfit")
install.packages("pROC")
install.packages("survminer")
```

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

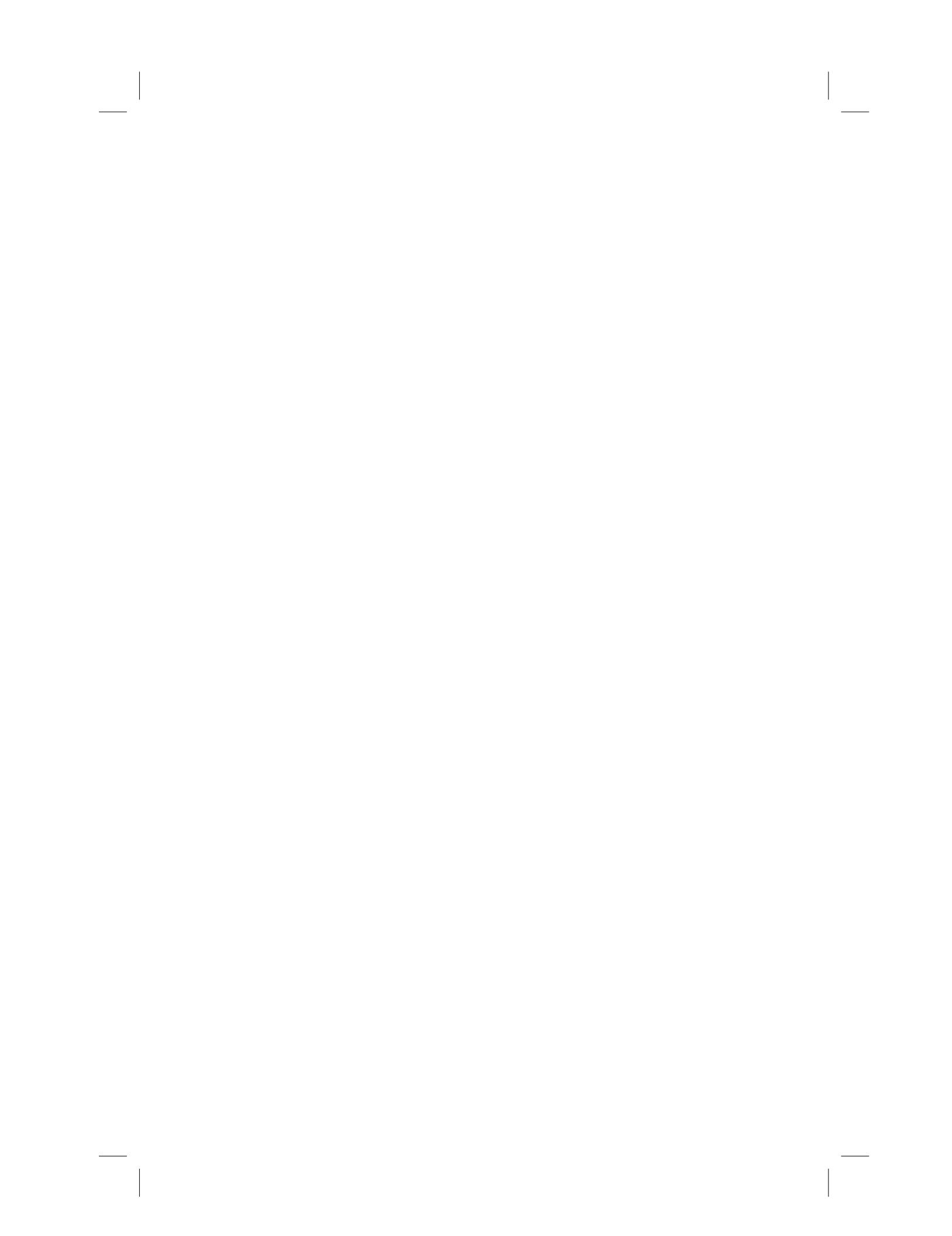
Control+Enter (line or section) or Control+Shift+Enter (whole script). Make sure you are always working in a project (the right-top corner of your RStudio interface should say “HealthyR”).





# **Part I**

## **Data wrangling and visualisation**



# 1

---

## *Your first R plots*

In this session, we will create five beautiful and colourful barplots in less than an hour. Do not worry about understanding every single word or symbol (e.g. the pipe - `%>%`) in the R code you are about to see. The purpose of this session is merely to

- gain familiarity with the RStudio interface:
    - to know what a script looks like,
    - what is the Environment tab,
    - where do your plots appear.
- 

### 1.1 Data

Load the example dataset which is already saved as an R-Data file (recognisable by the file extension .rda or .RData):

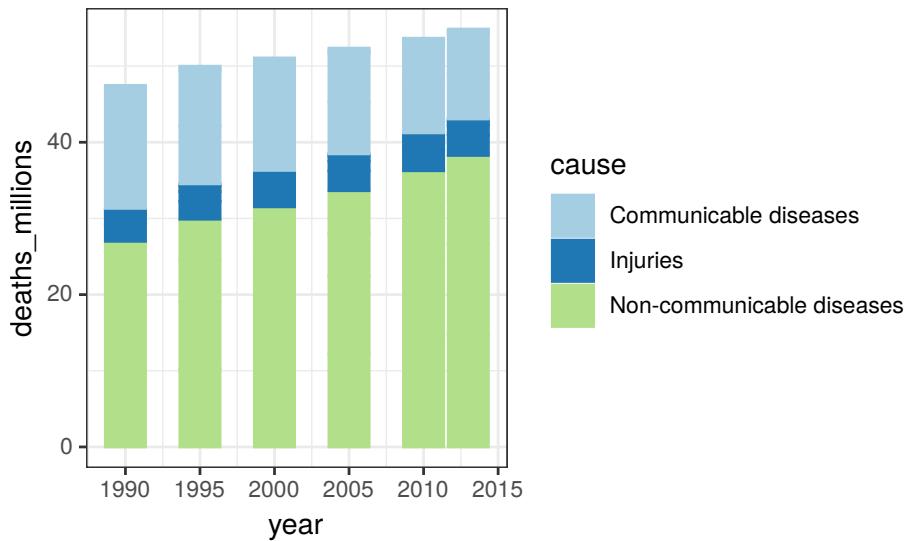
```
library(ggplot2)
source("1_source_theme.R")
load("global_burden_disease_long.rda")
```

After loading the datasets, investigate your Environment tab (top-right). You will see two things listed: `mydata` and `mydata2013`, which is a subset of `mydata`.

Click on the name `mydata` and it will pop up next to where your script is. Clicking on the blue button is not as useful (in this session), but it doesn't do any harm either. Try it.

## 1.2 First plot

```
mydata %>% #press Control-Shift-M to insert this symbol (pipe)
  ggplot(aes(x      = year,
             y      = deaths_millions,
             fill   = cause,
             colour = cause)) +
  geom_col()
```



`ggplot()` stands for **grammar of graphics plot** - a user friendly yet flexible alternative to `plot()`.

`aes()` stands for **aesthetics** - things we can see.

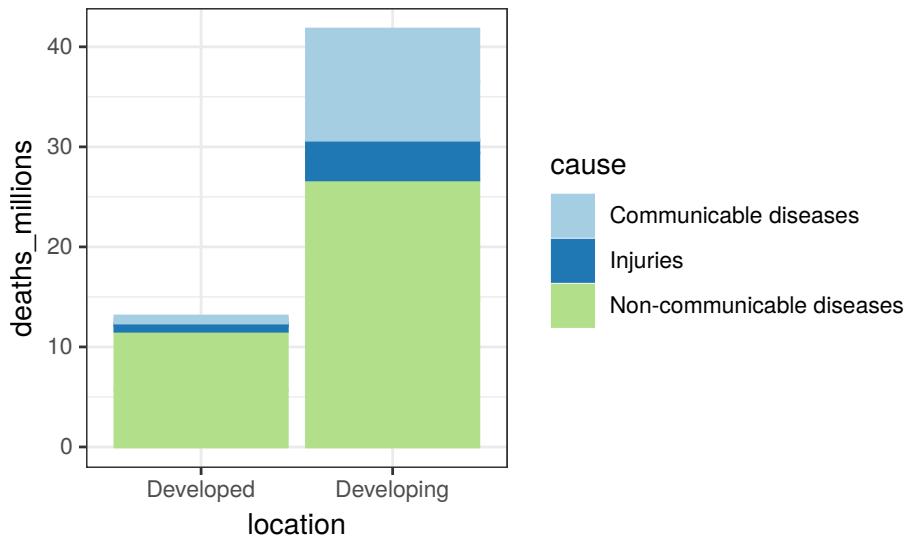
`geom_()` stands for **geometric**.

### 1.2.1 Question

Why are there two closing brackets - `)` - after the last aesthetic (`colour`)?

### 1.2.2 Exercise

Plot the number of deaths in Developed and Developing countries for the year 2013:

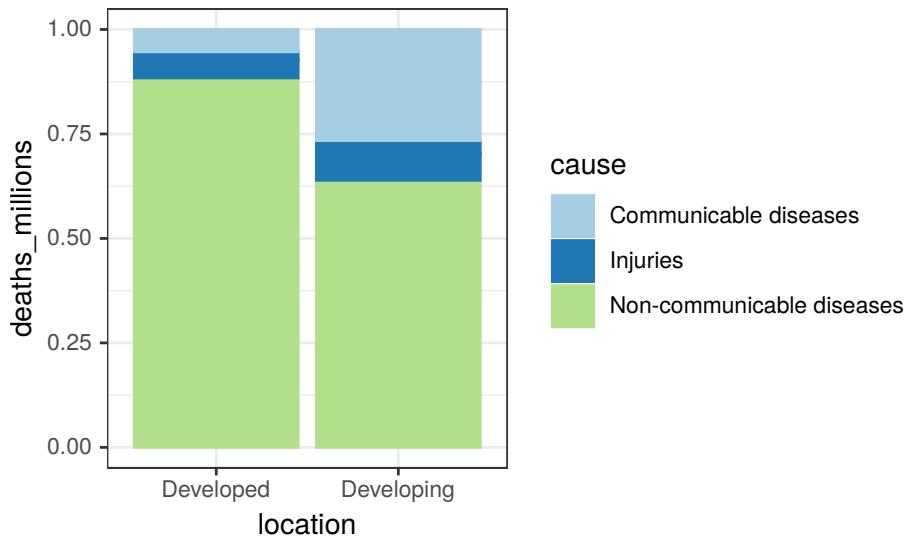


### 1.3 Comparing bars of different height

#### 1.3.1 Stretch each bar to 100%

`position="fill"` stretches the bars to show relative contributions:

```
mydata2013 %>%
  ggplot(aes(x      = location,
             y      = deaths_millions,
             fill   = cause,
             colour = cause)) +
  geom_col(position = "fill")
```

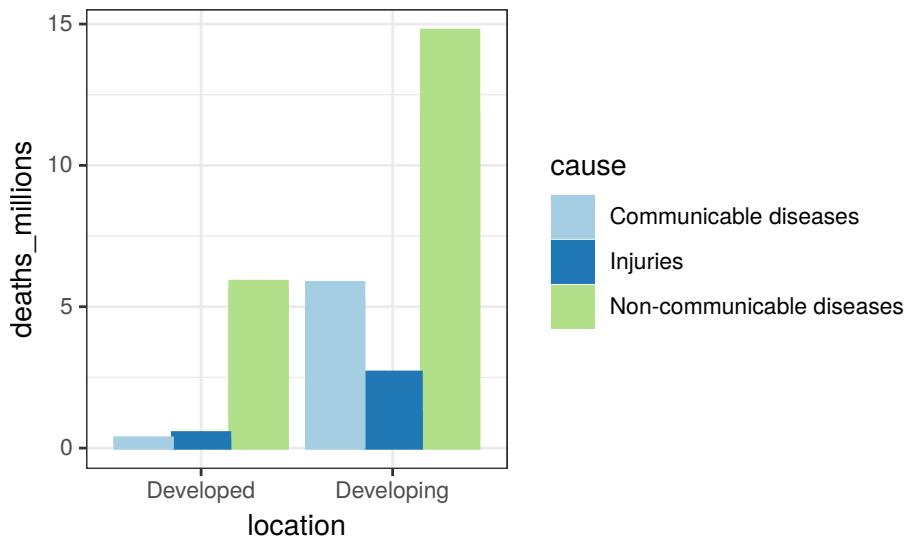


#### 1.3.2 Plot each bar next to each other

`position="dodge"` puts the different causes next to each rather (the default is `position="stack"`):

```
mydata2013 %>%
  ggplot(aes(x      = location,
             y      = deaths_millions,
             fill   = cause,
```

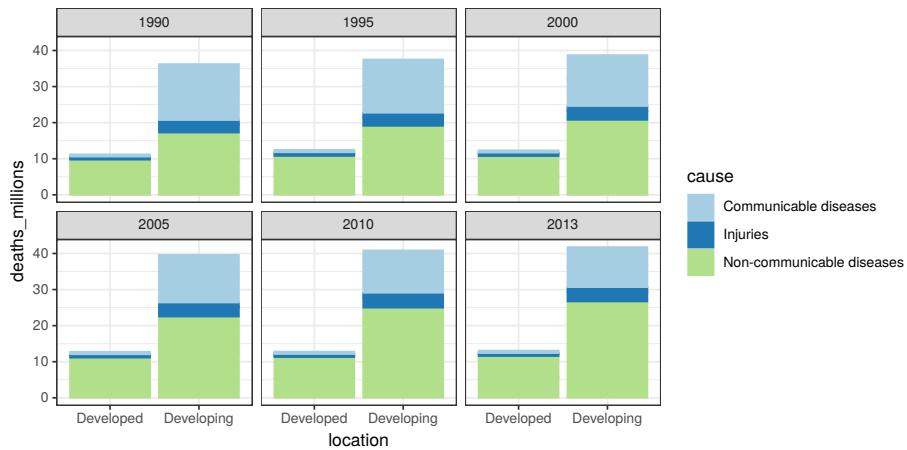
```
    colour = cause)) +  
  geom_col(position = "dodge")
```



## 1.4 Facets (panels)

Going back to the dataframe with all years (1990 – 2015), add `facet_wrap(~year)` to plot all years at once:

```
mydata %>%  
  ggplot(aes(x      = location,  
            y      = deaths_millions,  
            fill   = cause,  
            colour = cause)) +  
  geom_col() +  
  facet_wrap(~year)
```

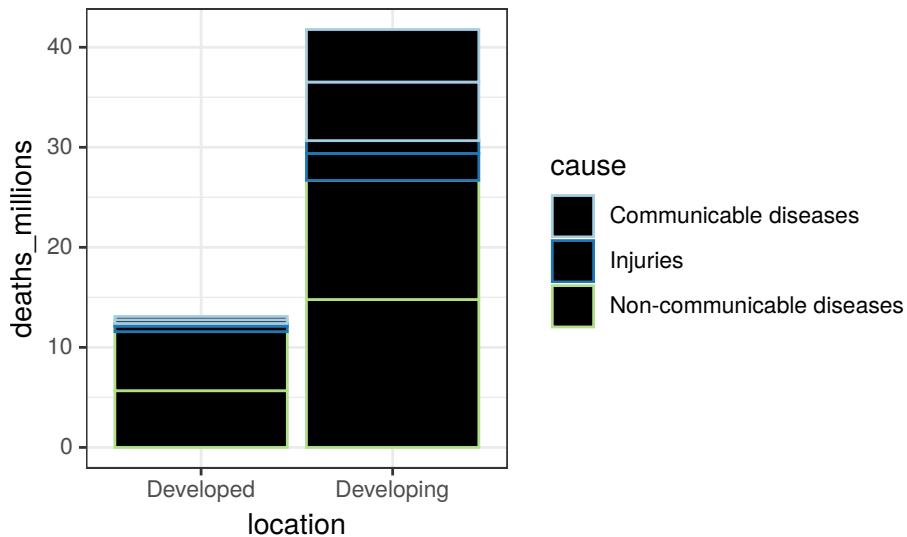


## 1.5 Extra: using aesthetics outside of the aes()

### 1.5.1 Setting a constant fill

Using the `mydata2013` example again, what does the addition of `fill = "black"` in this code do? Note that putting the `ggplot(aes())` code all on one line does not affect the result.

```
mydata2013 %>%
  ggplot(aes(x = location, y = deaths_millions, fill = cause, colour = cause)) +
  geom_col(fill = "black")
```



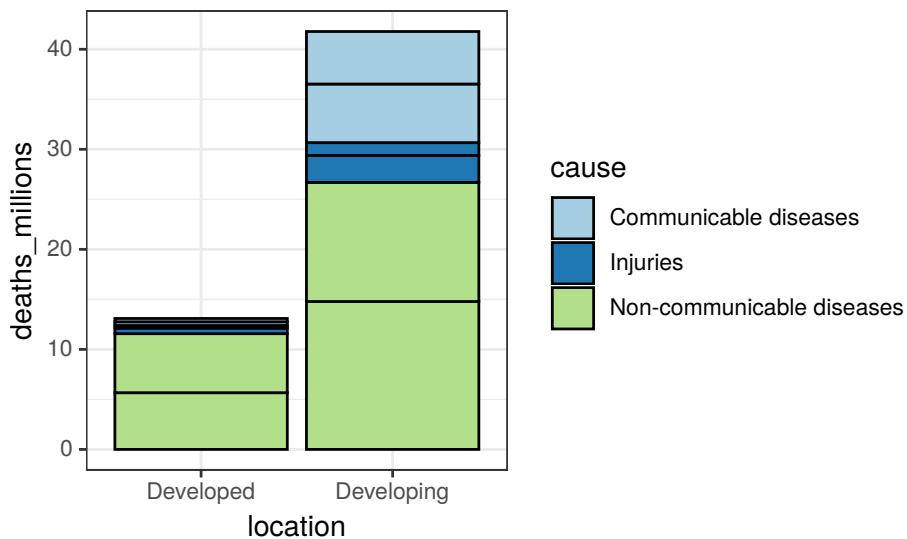
Setting aesthetics (x, y, fill, colour, etc.) outside of `aes()` sets them to a constant value. R can recognise a lot of colour names, e.g., try “cornflowerblue”, “firebrick”, or just “red”, “green”, “blue”, etc. For a full list, search Google for “Colours in R”. R also knows HEX codes, e.g. `fill = "#fec3fc"` is pink.

### 1.5.2 Exercise

What is the difference between colour and fill in the context of a barplot?

Hint: Use `colour = "black"` instead of `fill = "black"` to investigate what `ggplot()` thinks a colour is.

```
mydata2013 %>%
  ggplot(aes(x = location, y = deaths_millions, fill = cause, colour = cause)) +
  geom_col(colour = "black")
```



### 1.5.3 Exercise

Why are some of the words in our code quoted (e.g. `fill = "black"`) whereas others are not (e.g. `x = location`)?

---

## 1.6 Two geoms for barplots: `geom_bar()` or `geom_col()`

Both `geom_bar()` and `geom_col()` create barplots. If you:

- Want to visualise the count of different lines in a dataset - use `geom_bar()`
  - For example, if you are using a patient-level dataset (each line is a patient record): `mydata %>% ggplot(aes(x = sex)) + geom_bar()`
- Your dataset is already summarised - use `geom_col()`
  - For example, in the GBD dataset we use here, each line already includes a summarised value (`deaths_millions`)

If you have used R before you might have come across `geom_bar(stat = "identity")` which is the same as `geom_col()`.

## 1.7 Solutions

**1.2.1:** There is a double closing bracket because `aes()` is wrapped inside `ggplot()` - `ggplot(aes())`.

**1.2.2:**

```
mydata2013 %>%
  ggplot(aes(x      = location,
             y      = deaths_millions,
             fill   = cause,
             colour = cause)) +
  geom_col()
```

**1.5.2:**

On a barplot, the colour aesthetic outlines the fill. In a later session we will see, however, that for points and lines, colour is the main aesthetic to define.

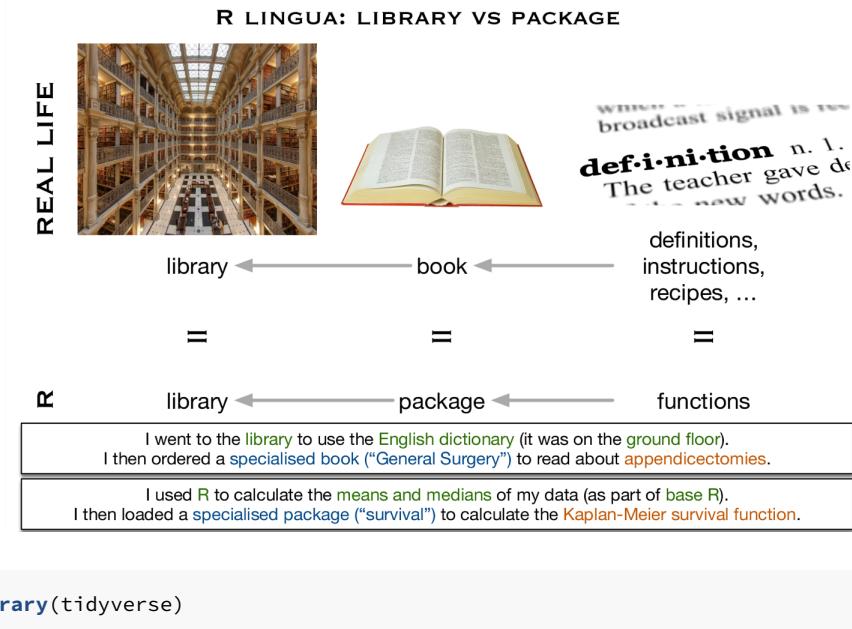
**1.5.3:**

Words in quotes are generally something set to a constant value (e.g. make all outlines black, rather than colour them based on the cause they are representing). Unquoted words are generally variables (or functions). If the word “function” just threw you, Google ”Jesse Maegan: What the h\*ck is a function”

---

## 1.8 Tidyverse packages: `ggplot2`, `dplyr`, `tidyR`, etc.

Most of the functions introduced in this book come from the tidyverse family (<http://tidyverse.org/>), rather than Base R. Including `library(tidyverse)` in your script loads a list of packages: `ggplot2`, `dplyr`, `tidyR`, `forcats`, etc.



# **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. We will be working with a shorter version of the Global Burden of Disease dataset that we met earlier.

---

### **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**

The two fundamental concepts to understand about statistical programming are objects and functions. As usual, in this book, we prefer introducing new concepts using specific examples first. And then define things in general terms after examples.

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

- so something with rows and columns. It should be regular, e.g., the made-up example in Table 2.1.<sup>1</sup>

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, e.g., 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 table when it lives outside R, but once you read it into R (import it), we call it a tibble.<sup>2</sup> 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 as you can have multiple tibbles going on at the same time (`tibble` is not a name, it is the class of an object). To keep our code examples easy to follow, we call our example tibble `mydata`. In real analysis, you should give your tibbles meaningful names, e.g., `patient_data`, `lab_results`, `annual_totals`, etc.

<sup>1</sup>Regular does not mean it can’t have missing values. Missing values are denoted `NA` which stands for either `Not available` or `Not applicable`. In some contexts, these things can have a different meaning. For example, since `var2` is `NA` for all male subjects, it may mean “Not applicable”, i.e. something that can only be measured in females. Whereas in `var3`, `NA` is more likely to mean “Not available” so real missing data, e.g. lost to follow-up.

<sup>2</sup>There used to be an older version of tables in R - they are called data frames. In most cases, `data frames` and `tibbles` work interchangeably (and both are R objects), but `tibbles` are newer and better. Another great alternative to base R’s `data frames` are `data tables`. In this book, and for most of our day-to-day work these days, we use `tibbles` though.

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
```

An example of a function that can be applied on numeric data is `mean()`. R functions always have round brackets after their name. This is for two reasons. First, to easily differentiate them from objects - which don't have round brackets after their name. Second, and more important, we can put arguments in these brackets. Arguments can also be thought of as input, and 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 that has 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 it is neither.<sup>3</sup>

So `mean(mydata$var2)` does not return an Error, but it also doesn't return the mean of the numeric values included in this column. That is because the column 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 size. Which might be fine if the sample sizes are sufficiently representative and the values are missing at random. Therefore, if you decide to ignore the NAs and want to calculate the mean anyway, you can do so by adding another argument to `mean()`:

---

<sup>3</sup>Logical is a data type with two potential values: TRUE or FALSE. We will come back to data types shortly.

```
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 it has look exactly how the function expects it, 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 (press `F1` when the cursor is on the name of the function) can remind you of. Functions’ 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] "2019-08-07 16:11:34 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). The data values input into a function are usually its first argument, further arguments can be used to specify a function’s 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()`

examples 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 into our Environment we use the equals sign:

```
a = 103
```

This reads: the variable `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 a variable, 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 a variable, 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 a variable, it does not get printed. If you run a function without the assignment (=), its results get printed, but not saved in a variable.

---

You can call your variables (where you assigns new objects or the output of functions in) 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 variable `sequence_15_to_30`. Spaces in variable 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 variable, 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 the variable 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 variable (so it shows up in the Environment), you need to add the name of the new variable 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

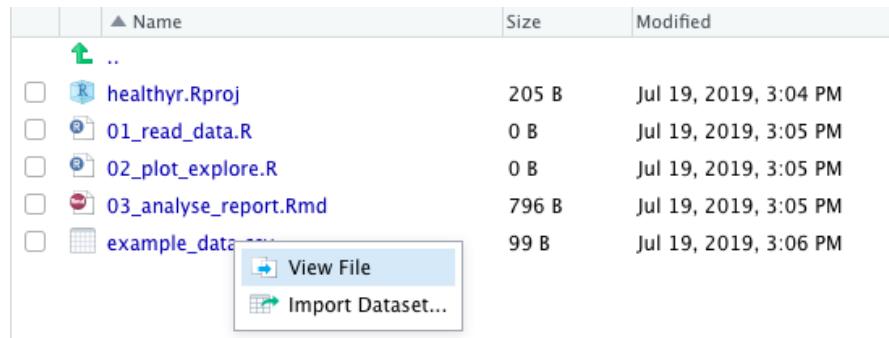
The pipe usually sends data to the beginning of function brackets (as most of the functions we use expect a tibble as the first argument). So `mydata %>% lm(dependent~explanatory)` is equivalent to `lm(mydata, dependent~explanatory)` (`lm()` stands for linear model introduced in detail in Chapter 7: linear regression). However, tibble first is not the order the `lm()` function expects. `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 or objects other than 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).



	Name	Size	Modified
...	..		
□	healthyr.Rproj	205 B	Jul 19, 2019, 3:04 PM
□	01_read_data.R	0 B	Jul 19, 2019, 3:05 PM
□	02_plot_explore.R	0 B	Jul 19, 2019, 3:05 PM
□	03_analyse_report.Rmd	796 B	Jul 19, 2019, 3:05 PM
□	example_data.csv	99 B	Jul 19, 2019, 3:06 PM

FIGURE 2.2: View or import a data file.

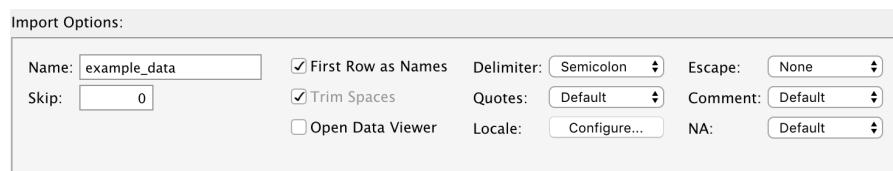
Clicking on a data file gives us two options: `View File` or `Import Dataset`. For very standard CSV files, we don't usually bother with the Import interface and just type in (or copy from a previous script):

```
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”)
- 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 as it will give you the relevant `read_()` syntax with all the extra arguments filled in for you.



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

```
library(readr)
example_data <- read_delim("example_data.csv",
",",
escape_double = FALSE,
locale = locale(decimal_mark = ","),
trim_ws = TRUE)
```

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

After selecting the specific options for your import file (there is a friendly preview window too, so you can immediately see whether R understands the format of the your data file), DO NOT BE tempted to press the `Import` button. Yes, this will read in your dataset once, but means you have to redo the selections every time you come back to RStudio. Do copy-paste the code it gives you (e.g., Figure 2.4) into your R script - this way you can use it over and over again. Making sure all steps are recorded in scripts makes your workflow reproducible by your future self, colleagues, supervisors, 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 trying to make sense of legacy scripts passed on to you by colleagues, you might see `read.csv()` rather than `read_csv()`. In short: `read_csv()` is faster and better, and in all new scripts that's what you should use. But in existing scripts that work and are tested, do not just start replacing `read.csv()` with `read_csv()`. The thing is, `read_csv()` handles categorical variables slightly differently <sup>4</sup>. This means that 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.

---

### 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. <sup>5</sup>

---

<sup>4</sup>It 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.

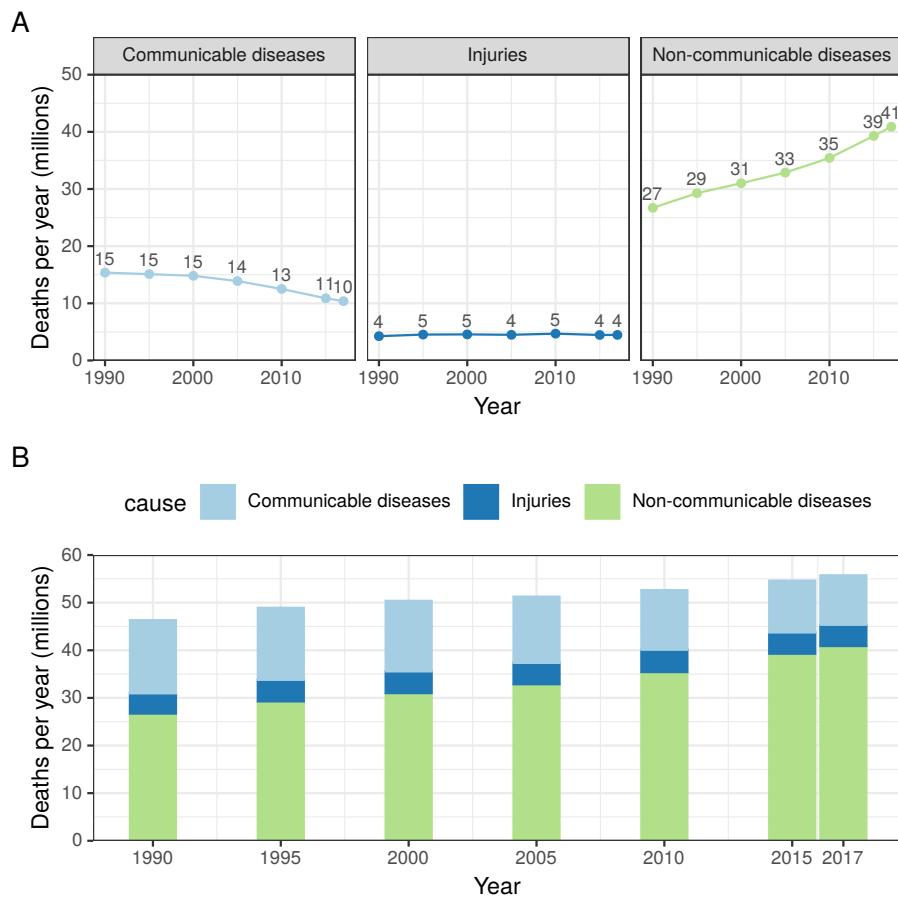
<sup>5</sup>Global Burden of Disease Collaborative Network. Global Burden of Disease Study 2017 (GBD 2017) Results. Seattle, United States: Institute for

GBD data are publicly available from their website. Table 2.2 and Figure 2.5 show a very high level version of the project’s 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")
```

**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     15.4
## 5 1995 Injuries                 4.25
## 6 1995 Non-communicable diseases 26.7
```

```
## 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. For example, the first filtering example was actually equivalent to:

```
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
```

Accidentally using the single equals (= so the assignment operator) is a very common mistake and still occasionally happens to the best of us. In fact, 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 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 whenever checking for equality of variables but the result is not what you expect (you'll get an Error, but not necessary with the same wording as above), remember to 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 to keep rows where year is equal to 1995 or year is equal to 2017. Using specific values like we've done there (1995/2017) is called "hard-coding", which is fine if we know for sure we don't want to apply 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>&lt;</code>	Less than
<code>&gt;</code>	Greater than
<code>&lt;=</code>	Less than or equal to
<code>&gt;=</code>	Greater than or equal to
<code>&amp;</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 - these are notes for your future self on what and why you did.

## 2.7 The combine function: c()

The combine function is used to list several values. It is especially useful together with the `%in%` operator which can be used 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.<sup>6</sup> 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 even more time if we had more different values we wanted to include). This 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>
## 1 1990 Communicable diseases     15.4
```

<sup>6</sup>In this example, it would just be easier to `filter(cause != "Injuries")` but imagine your column had more than just three different values in it.

```
## # 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
```

26.7

## 2.8 Missing values (NAs) and filters

Filtering for missing values (NAs) needs your 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 very common source of (minor, easily fixed) mistakes, and it still happens to the best of us.

If filtering for rows where `var2` is not missing, we do this<sup>7</sup>

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

## # A tibble: 2 × 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” (`!=`)<sup>8</sup>:

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

## # A tibble: 1 × 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:

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

---

<sup>7</sup>In this simple example, `mydata %>% filter(! is.na(var2))` could be replace 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.

<sup>8</sup>`filter(var2 != 5)` is equivalent to `filter(! var2 == 5)`

```
## # 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-time<sup>9</sup>.

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 the type your data is (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 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

---

<sup>9</sup>Portable 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 and POSIXlt that” at you, check your date and time variables

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 × 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 now 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 will also get parsed as a categorical variable because of a small typo, e.g., if entered as “3..7” instead of “3.7”. And the reason R didn’t automatically make sense of the date column is that it can’t know whether which one is the month and which one 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, but it is important to understand what the different types are to be ready to deal with them when issues arise.

Furthermore, since health data is generally full of categorical data, it is crucial that you do understand the difference between characters and factors (both are types of categorical variables in R with their 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`. <sup>10</sup> 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. 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:

---

<sup>10</sup>Integers 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`).

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

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

Let's save that in a new object:

```
measurement_mean = typesdata$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 intend 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 unuseful 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] "2019-08-07 16:11:37 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 481.8669 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] "2022-04-29 08:48:22 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] 1.162147
```

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](https://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
%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

Notation	Meaning	Example
%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] "2019-08-07 16:11:37 BST"
```

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

```
## [1] "16:11 on August-07 (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 16:11 August-07 (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 × 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 × 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 × 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 error<sup>11</sup>:

```
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
```

---

<sup>11</sup>If 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 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 x 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 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     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 x 3
##       id sex     age
##   <dbl> <chr>  <dbl>
## 1     7 Female  38
## 2     8 Male    29
```

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

```
## # A tibble: 8 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
## 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 x 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 we will get to know our three best friends for summarising data: `group_by()`, `summarise()`, and `mutate()`.

---

### 3.1 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")

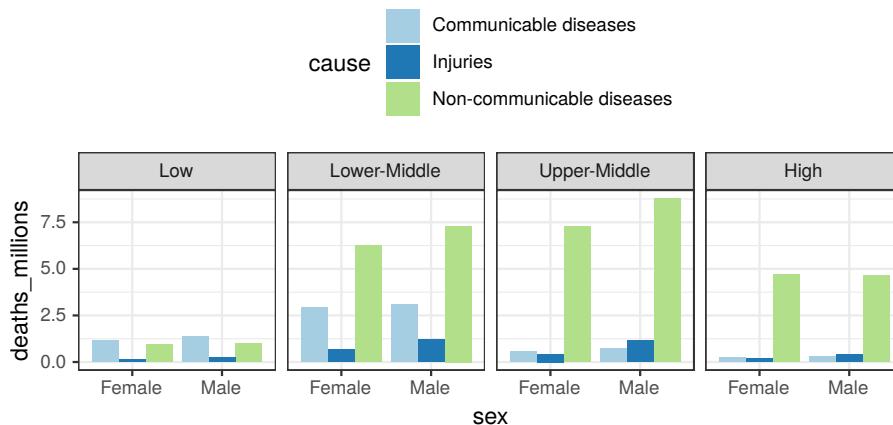
# Creating a single-year tibble for printing and simple examples:
gbd2017 = gbd_full %>%
  filter(year == 2017)
```

**TABLE 3.1:** Deaths per year 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.

### 3.2 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 Non-communicable diseases 40.89
## 3 Injuries        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
```

### 3.3 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>                 <chr>           <dbl>        <dbl>
```

```
## 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
```

### 3.3.1 percentages formatting: `percent()`

So `summarise()` condenses a tibble, whereas `mutate()` retains it's 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_pergroups/deaths_total)
```

and convert to nicely formatted percentages later:

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

---

### 3.4 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_pergroups = sum(deaths_millions)) %>%
  arrange(sex)
```

```
gbd_summarised
```

```
## # A tibble: 6 x 3
## # Groups:   cause [3]
##   cause                  sex    deaths_pergroups
##   <chr>                 <chr>          <dbl>
## 1 Road injury             Males      100.0
## 2 Road injury             Females     99.0
## 3 Road injury             Both       199.0
## 4 Noncommunicable diseases Males      100.0
## 5 Noncommunicable diseases Females     99.0
## 6 Noncommunicable diseases Both       199.0
```

```
## 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%
```

### 3.5 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 12.

```
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.

---

### 3.6 `select()` columns

---

## 3.7 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-sex-year.csv")
```

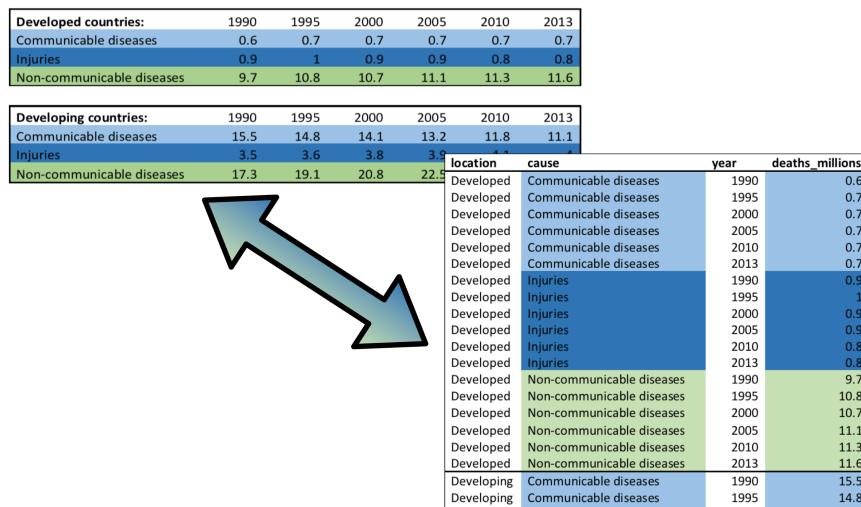
**TABLE 3.2:** Global Burden of Disease data in human-readable wide format. This is not tidy data.

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

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

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

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



**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.

### 3.7.1 `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 × 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 × 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 x 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 x 5
##   cause           sex     year   deaths_millions sex_year
##   <chr>          <chr>  <dbl>            <dbl> <chr>
## 1 Communicable diseases Female  1990             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).

## Exercise

Using this dataset (Table 3.3):

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

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

### 3.7.2 `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

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.

---

### 3.8 Sorting: `arrange()`

To reorder data ascendingly or descendingly, use `arrange()`:

```
mydata %>%
  group_by(year) %>%
  summarise(total = sum(deaths_millions)) %>%
  arrange(-year) # reorder after summarise()
```

## 3.9 Factor handling

We talked about the pros and cons of working with factors in Session 2. Overall, they are extremely useful for the type of analyses done in medical research.

### 3.9.1 Exercise

Explain how and why these two plots are different.

```
mydata %>%
  ggplot(aes(x = year, y = deaths_millions, fill = cause)) +
  geom_col()

mydata %>%
  ggplot(aes(x = factor(year), y = deaths_millions, fill = cause, colour = cause)) +
  geom_col()
```

What about these?

These illustrate why it might sometimes be useful to use numbers as factors - on the second one we have used `fill = factor(year)` as the fill, so each year gets a distinct colour, rather than a gradual palette.

### 3.9.2 `fct_collapse()` - grouping levels together

```
mydata$cause %>%
  fct_collapse("Non-communicable and injuries" = c("Non-communicable diseases", "Injuries")) ->
  mydata$cause2

mydata$cause %>% levels()
mydata$cause2 %>% levels()
```

### 3.9.3 `fct_relevel()` - change the order of levels

Another reason to sometimes make a numeric variable into a factor is that we can then reorder it for the plot:

```
mydata$year %>%
  factor() %>%
  fct_relevel("2013") -> #brings 2013 to the front
  mydata$year.factor

source("1_source_theme.R")

mydata %>%
  ggplot(aes(x=year.factor, y=deaths_millions, fill=cause))+
  geom_col()
```

### 3.9.4 `fct_recode()` - rename levels

```
mydata$cause %>%
  levels() # levels() lists the factor levels of a column

mydata$cause %>%
  fct_recode("Deaths from injury" = "Injuries") %>%
  levels()
```

### 3.9.5 Converting factors to numbers

MUST REMEMBER: factor needs to become `as.character()` before converting to numeric or date! Factors are actually stored as labelled integers (so like number codes), only the function `as.character()` will turn a factor back into a collated format which can then be converted into a number or date.

### 3.9.6 Exercise

Investigate the two examples converting the `year.factor` variable back to a number.

```
mydata$year.factor  
mydata$year.factor %>%  
  as.numeric()  
  
mydata$year.factor %>%  
  as.character() %>%  
  as.numeric()
```

---

### 3.10 Long Exercise

This exercise includes multiple steps, combining all of the above.

First, create a new script called “2\_long\_exercise.R”. Then Restart your R session, add `library(tidyverse)` and load `"global_burden_disease_long.rda"`.

- Calculate the total number of deaths in Developed and Developing countries. Hint: use `group_by(location)` and `summarise(new-column-name = sum(variable-to-sum))`.
- Calculate the total number of deaths in Developed and Developing countries and for men and women. Hint: this is as easy as adding `, sex` to `group_by()`.
- Filter for 1990.
- `spread()` the `location` column.

---

### 3.11 Extra: formatting a table for publication

Creating a publication table with both the total numbers and percentages (in brackets) + using `formatC()` to retain trailing zeros:

```
# Let's use alldata from Exercise 5.2:

mydata %>%
  group_by(year, cause) %>%
  summarise(total_per_cause = sum(deaths_millions)) %>%
  group_by(year) %>%
  mutate(total_per_year = sum(total_per_cause)) %>%
  mutate(percentage = 100*total_per_cause/total_per_year) -> alldata

alldata %>%
  mutate(total_percentage =
    paste0(round(total_per_cause, 1) %>% formatC(1, format = "f"),
          " (",
          round(percentage, 1) %>% formatC(1, format = "f"),
          "%)"
        )
```

```
) %>%  
  select(year, cause, total_percentage) %>%  
  spread(cause, total_percentage)
```

### 3.12 Solution: Long Exercise

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



# 4

---

## Different types of plots

---

### 4.1 Data

We will be using the gapminder dataset:

```
library(tidyverse)
library(gapminder)

mydata = gapminder

summary(mydata)

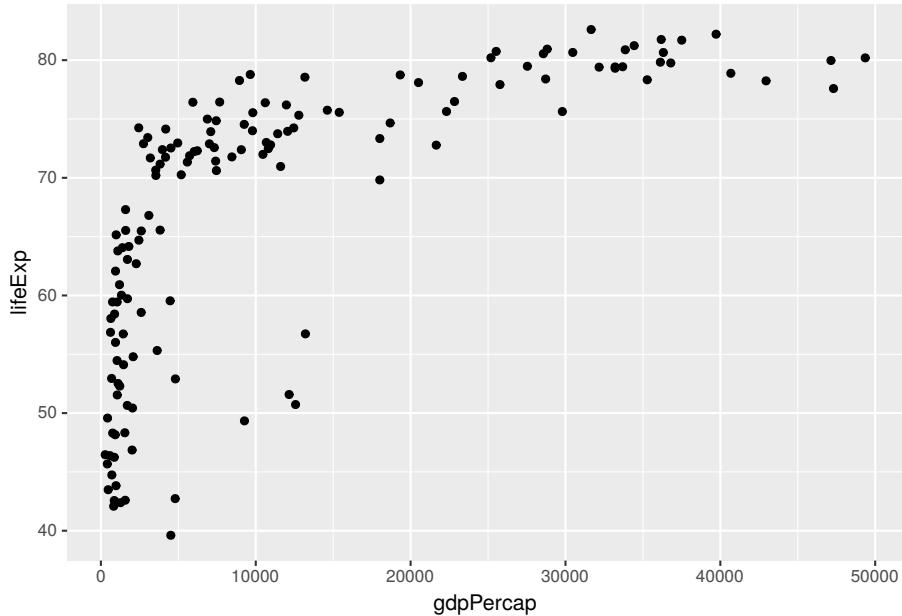
##          country      continent       year     lifeExp
##  Afghanistan: 12   Africa :624   Min.  :1952   Min.   :23.60
##  Albania     : 12   Americas:300   1st Qu.:1966   1st Qu.:48.20
##  Algeria     : 12   Asia   :396   Median :1980   Median :60.71
##  Angola      : 12   Europe :360   Mean   :1980   Mean   :59.47
##  Argentina   : 12   Oceania: 24   3rd Qu.:1993   3rd Qu.:70.85
##  Australia   : 12                           Max.   :2007   Max.   :82.60
##  (Other)     :1632
##          pop           gdpPercap
##  Min.   :6.001e+04   Min.   : 241.2
##  1st Qu.:2.794e+06   1st Qu.: 1202.1
##  Median :7.024e+06   Median : 3531.8
##  Mean   :2.960e+07   Mean   : 7215.3
##  3rd Qu.:1.959e+07   3rd Qu.: 9325.5
##  Max.   :1.319e+09   Max.   :113523.1
##
##  mydata$year %>% unique()

##  [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

## 4.2 Scatter plots/bubble plots - `geom_point()`

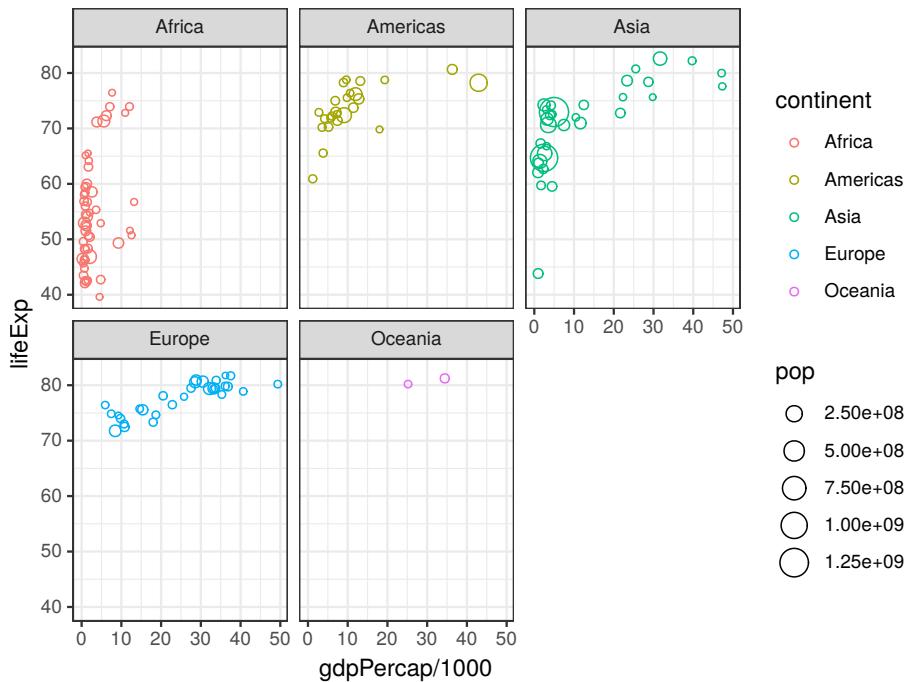
Plot life expectancy against GDP per capita (`x = gdpPercap`, `y=lifeExp`) at year 2007:

```
mydata %>%
  filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y=lifeExp)) +
  geom_point()
```



### 4.2.1 Exercise

Follow the step-by-step instructions to transform the grey plot just above into this:

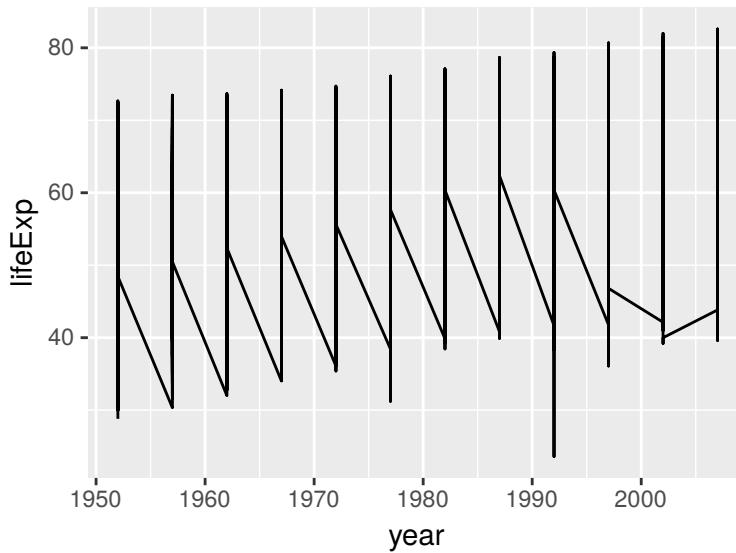


- Add points: `geom_point()`
  - Change point type: `shape = 1` (or any number from your Quickstart Sheet) inside the `geom_point()`
- Colour each country point by its continent: `colour=continent` to `aes()`
- Size each country point by its population: `size=pop` to `aes()`
- Put the country points of each continent on a separate panel: `+ facet_wrap(~continent)`
- Make the background white: `+ theme_bw()`

### 4.3 Line chart/timeplot - `geom_line()`

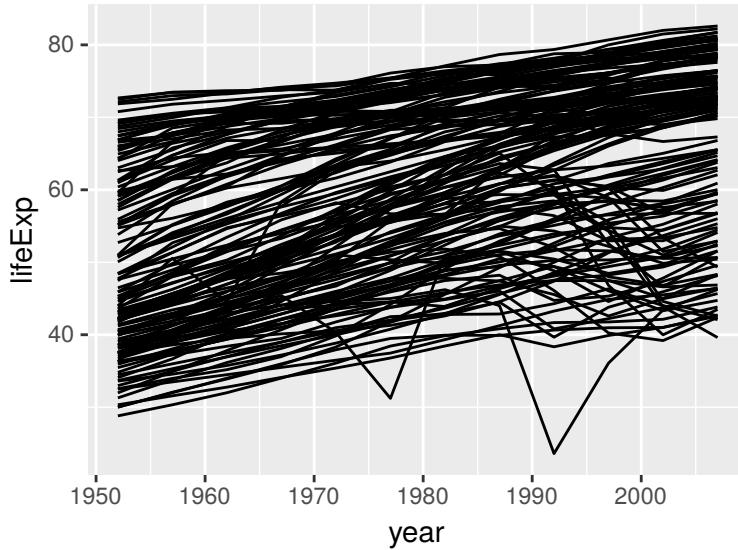
Plot life expectancy against year (`x = year`, `y=lifeExp`), add `geom_line()`:

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



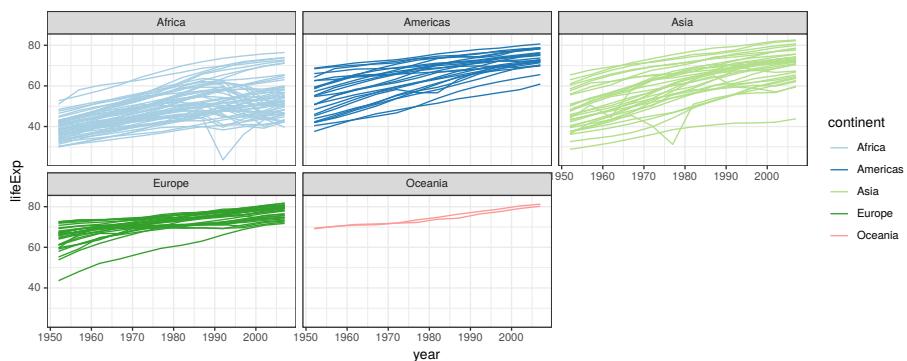
The reason you now see this weird zig-zag is that, using the above code, R does not know you want a connected line for each country. Specify how you want data points grouped to lines: `group = country` in `aes()`:

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



#### 4.3.1 Exercise

Follow the step-by-step instructions to transform the grey plot just above into this:

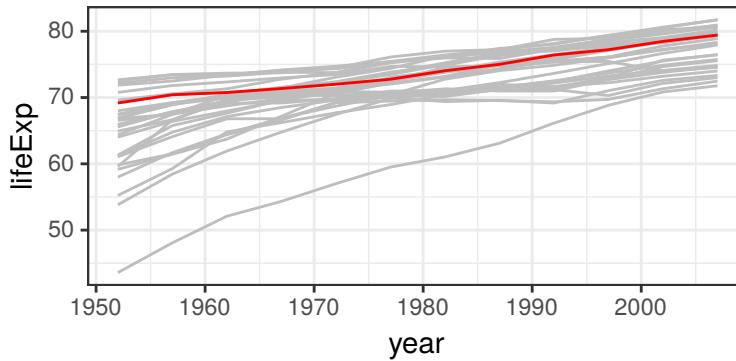


- Colour lines by continents: `colour=continent` to `aes()`
- *Similarly to what we did in `geom_point()`, you can even size the line thicknesses by each country's population: `size=pop` to `aes()`*
- Continents on separate panels: `+ facet_wrap(~continent)`
- Make the background white: `+ theme_bw()`
- Use a nicer colour scheme: `+ scale_colour_brewer(palette = "Paired")`

### 4.3.2 Advanced example

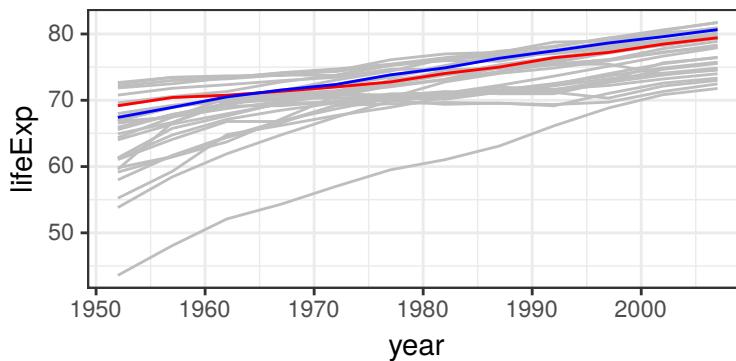
For European countries only (`filter(continent == "Europe") %>%`), plot life expectancy over time in grey colour for all countries, then add United Kingdom as a red line:

```
mydata %>%
  filter(continent == "Europe") %>% #Europe only
  ggplot(aes(x = year, y=lifeExp, group = country)) +
  geom_line(colour = "grey") +
  theme_bw() +
  geom_line(data = filter(mydata, country == "United Kingdom"), colour = "red")
```



### 4.3.3 Advanced Exercise

As previous, but add a line for France in blue:

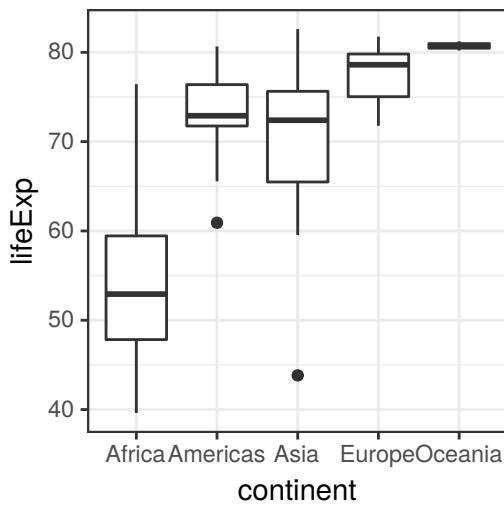


## 4.4 Box-plot - `geom_boxplot()`

Plot the distribution of life expectancies within each continent at year 2007:

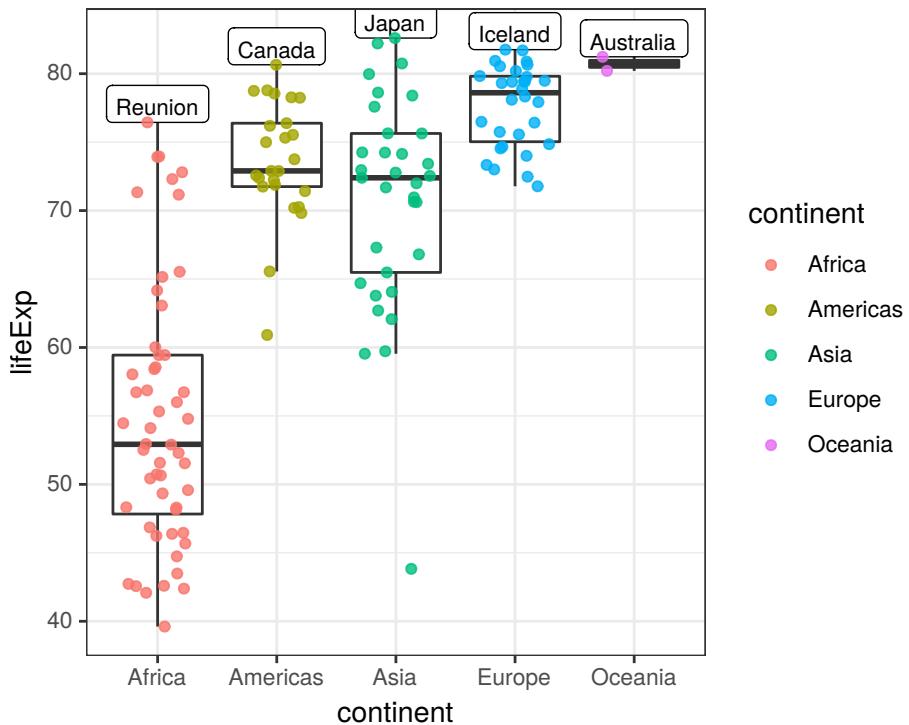
- `filter(year == 2007) %>%`
- `x = continent, y = lifeExp`
- `+ geom_boxplot()`

```
mydata %>%
  filter(year == 2007) %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  theme_bw()
```



### 4.4.1 Exercise

Add individual (country) points on top of the box plot:



Hint: Use `geom_jitter()` instead of `geom_point()` to reduce overlap by spreading the points horizontally. Include the `width=0.3` option to reduce the width of the jitter.

### Optional:

Include text labels for the highest life expectancy country of each continent.

**Hint 1** Create a separate dataframe called `label_data` with the maximum countries for each continent:

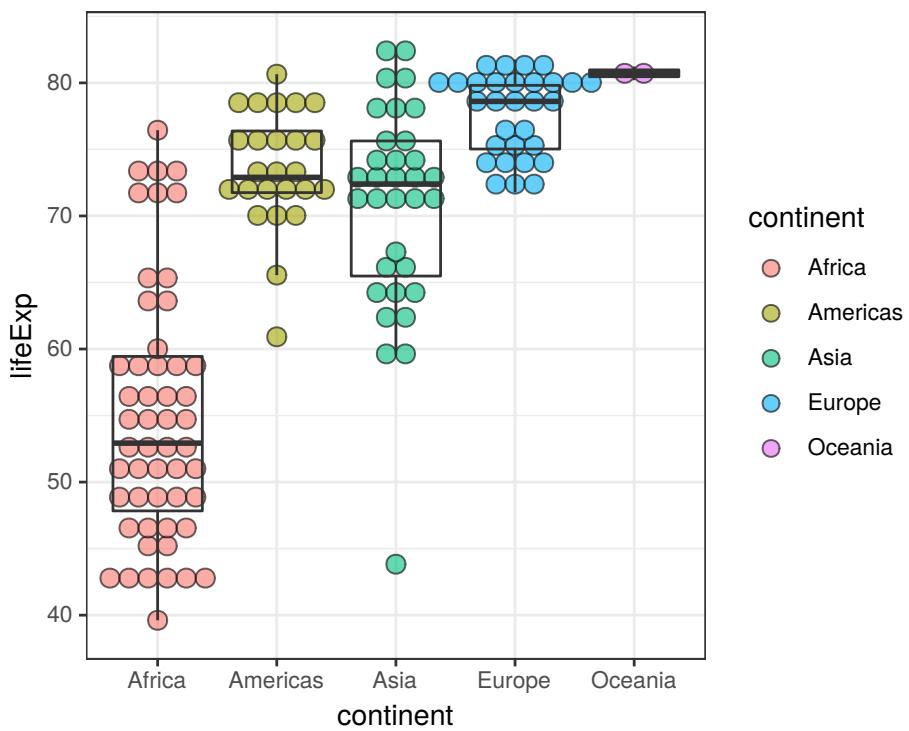
```
label_data = mydata %>%
  filter(year == max(year)) %>% # same as year == 2007
  group_by(continent) %>%
  filter(lifeExp == max(lifeExp) )
```

**Hint 2** Add `geom_label()` with appropriate `aes()`:

```
+ geom_label(data = label_data, aes(label=country), vjust = 0)
```

#### 4.4.2 Dot-plot - `geom_dotplot()`

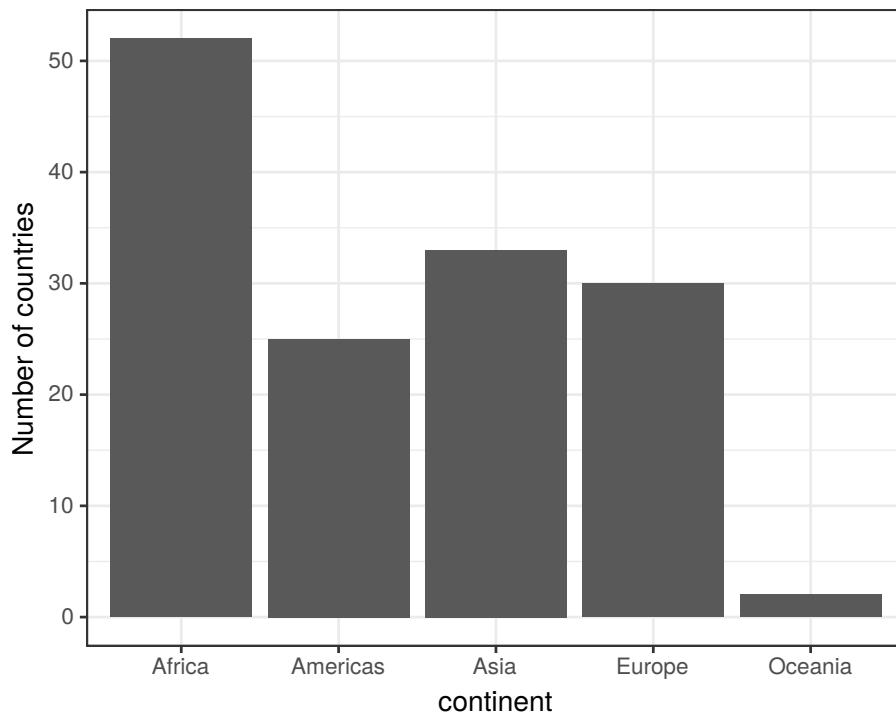
```
geom_dotplot(aes(fill=continent), binaxis = 'y', stackdir = 'center',
alpha=0.6)
```



## 4.5 Barplot - `geom_bar()` and `geom_col()`

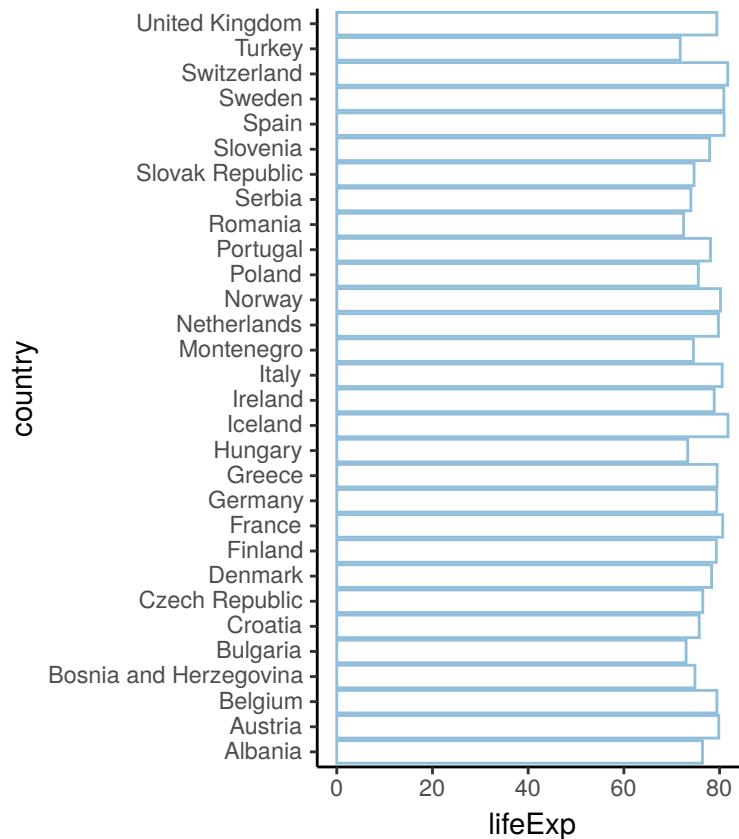
In the first module, we plotted barplots from already summarised data (using the `geom_col()`), but `geom_bar()` is perfectly happy to count up data for you. For example, we can plot the number of countries in each continent without summarising the data beforehand:

```
mydata %>%
  filter(year == 2007) %>%
  ggplot(aes(x = continent)) +
  geom_bar() +
  ylab("Number of countries") +
  theme_bw()
```



#### 4.5.1 Exercise

Create this barplot of life expectancies in European countries (year 2007). Hint: `coord_flip()` makes the bars horizontal, `fill = NA` makes them empty, have a look at your QuickStar sheet for different themes.



## 4.6 All other types of plots

These are just some of the main ones, see this gallery for more options: <http://www.r-graph-gallery.com/portfolio/ggplot2-package/>

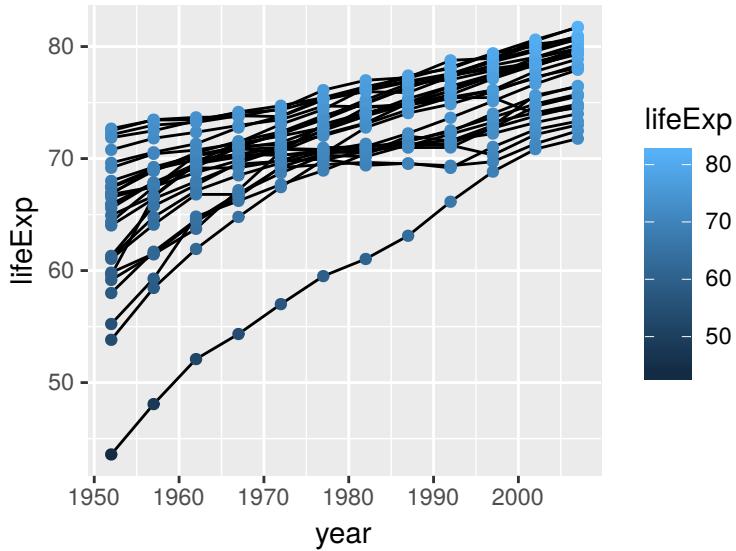
And the `ggplot()` documentation: <http://docs.ggplot2.org/>

Remember that you can always combine different types of plots - i.e. add lines or points on bars, etc.

## 4.7 Specifying `aes()` variables

The `aes()` variables wrapped inside `ggplot()` will be taken into account by all geoms. If you put `aes(colour = lifeExp)` inside `geom_point()`, only points will be coloured:

```
mydata %>%
  filter(continent == "Europe") %>%
  ggplot(aes(x = year, y = lifeExp, group = country)) +
  geom_line() +
  geom_point(aes(colour = lifeExp))
```



---

## 4.8 Extra: Optional exercises

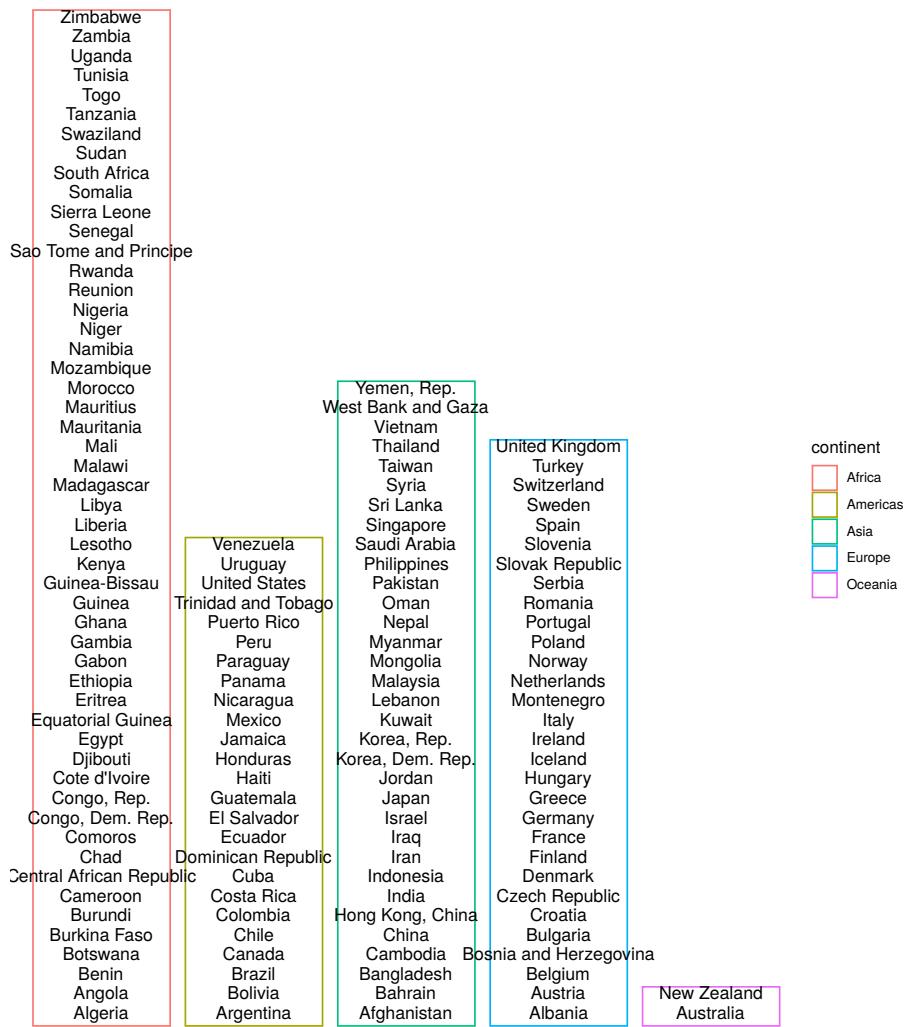
### 4.8.1 Exercise

Make this:

```
mydata$dummy = 1 # create a column called "dummy" that includes number 1 for each country

mydata2007 = mydata %>%
  filter(year==max(year)) %>%
  group_by(continent) %>%
  mutate(country_number = cumsum(dummy)) # create a column called "country_number" that
# is a cumulative sum of the number of countries before it - basically indexing

mydata2007 %>%
  ggplot(aes(x = continent)) +
  geom_bar(aes(colour=continent), fill = NA) +
  geom_text(aes(y = country_number, label=country), size=4, vjust=1, colour='black')+
  theme_void()
```

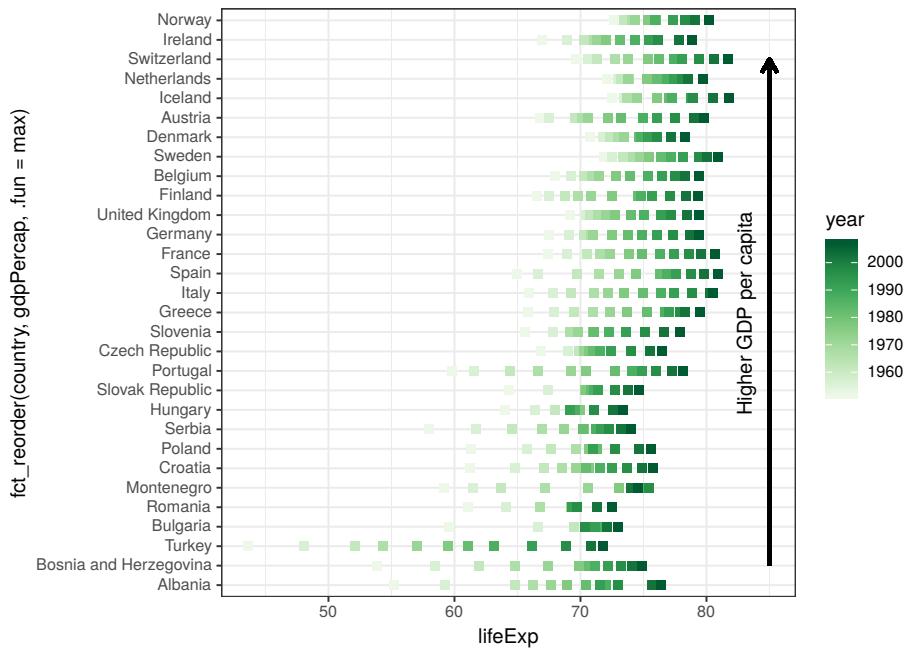


### 4.8.2 Exercise

Make this:

Hints: `coord_flip()`, `scale_color_gradient(...)`, `geom_segment(...)`, `annotate("text", ...)`

```
mydata %>%
  filter(continent == "Europe") %>%
  ggplot(aes(y = fct_reorder(country, gdpPercap, .fun=max), x=lifeExp, colour=year)) +
  geom_point(shape = 15, size = 2) +
  theme_bw() +
  scale_colour_distiller(palette = "Greens", direction = 1) +
  geom_segment(aes(yend = "Switzerland", x = 85, y = "Bosnia and Herzegovina", xend = 85),
               colour = "black", size=1,
               arrow = arrow(length = unit(0.3, "cm")))) +
  annotate("text", y = "Greece", x=83, label = "Higher GDP per capita", angle = 90)
```



## 4.9 Solutions

### 4.2.1

```
mydata %>%
  filter(year == 2007) %>%
  ggplot( aes(x = gdpPercap/1000, #divide by 1000 to tidy the x-axis
              y=lifeExp,
              colour=continent,
              size=pop)) +
  geom_point(shape = 1) +
  facet_wrap(~continent) +
  theme_bw()
```

### 4.3.1

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

which

Add + geom\_line(data = filter(mydata, country == "France"), colour = "blue")

### 4.4.1

```
mydata %>%
  filter(year == 2007) %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  geom_boxplot(outlier.shape = NA) +
  geom_jitter(aes(colour=continent), width=0.3, alpha=0.8) + #width defaults to 0.8 of box width
  theme_bw()
```

```
mydata %>%
  filter(year == 2007) %>%
  ggplot(aes(x = continent, y = lifeExp)) +
  geom_boxplot(outlier.shape = NA) +
```

```
geom_jitter(aes(colour=continent), width=0.3, alpha=0.8)
theme_bw()
```

#### 4.5.1

```
mydata %>%
  filter(year == 2007) %>%
  filter(continent == "Europe") %>%
  ggplot(aes(x = country, y = lifeExp)) +
  geom_col(colour = "#91bfdb", fill = NA) +
  coord_flip() +
  theme_classic()
```



# 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)

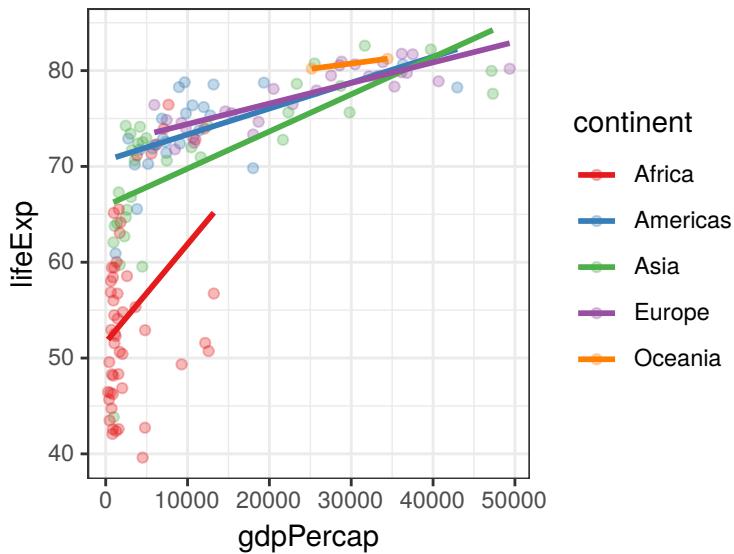
mydata = gapminder

mydata$year %>% unique()

## [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007

p = mydata %>%
  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")

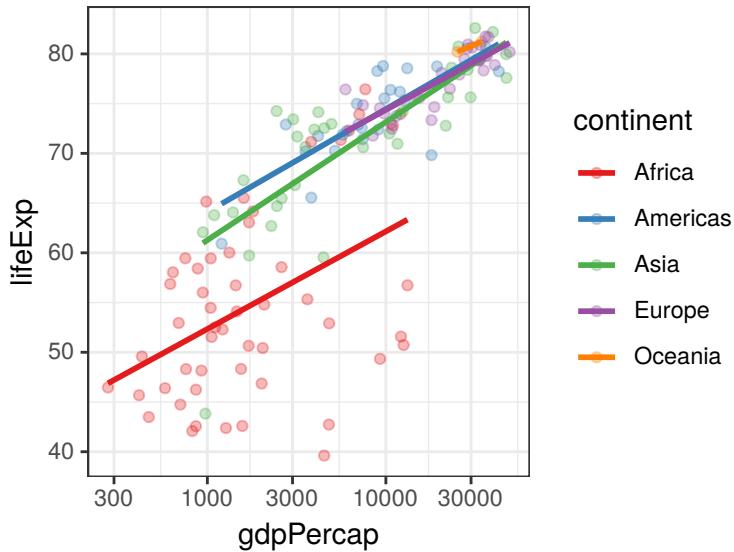
p
```



## 5.2 Scales

### 5.2.1 Logarithmic

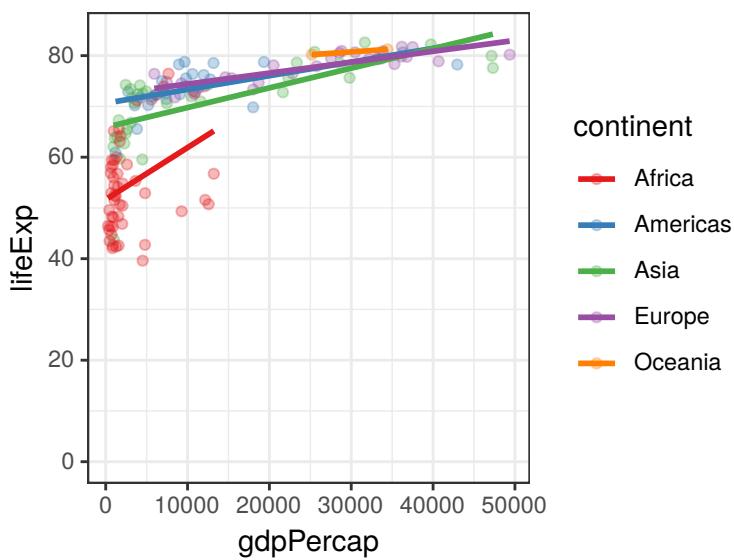
```
p +  
  scale_x_log10()
```



### 5.2.2 Expand limits

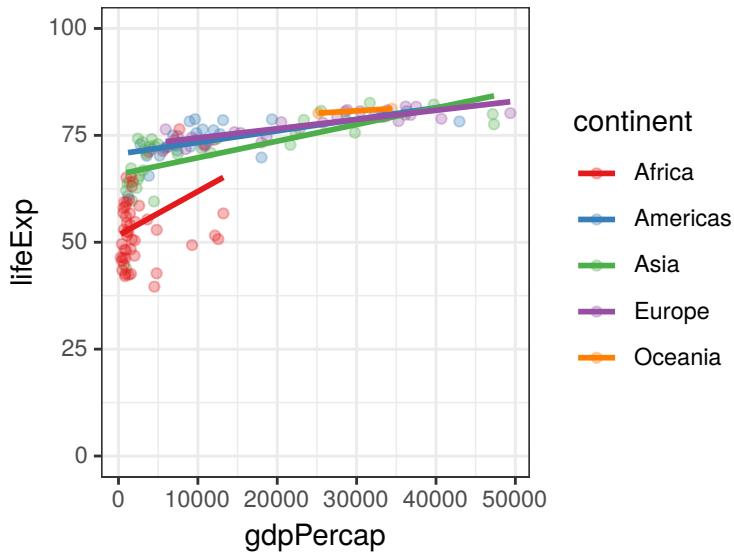
Specify the value you want to be included:

```
p +  
  expand_limits(y = 0)
```



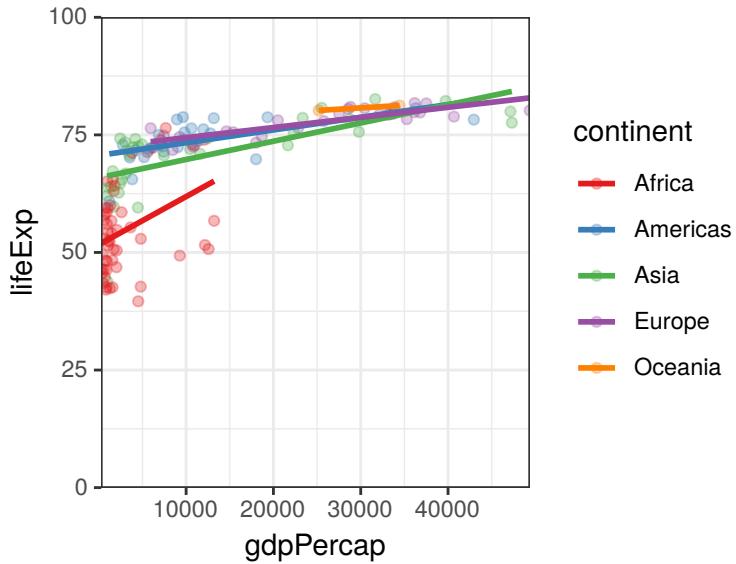
Or two:

```
p +
  expand_limits(y = c(0, 100))
```



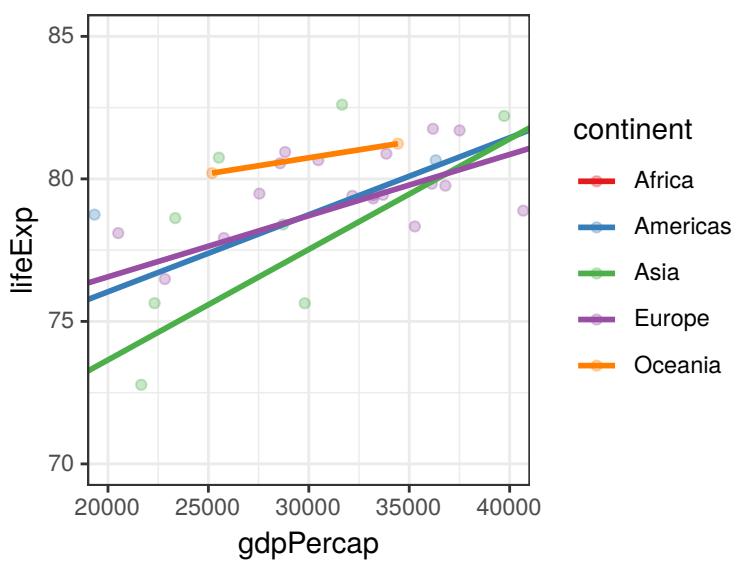
By default, `ggplot()` adds some padding around the included area (see how the scale doesn't start from 0, but slightly before). You can remove this padding with the `expand` option:

```
p +
  expand_limits(y = c(0, 100)) +
  coord_cartesian(expand = FALSE)
```



### 5.2.3 Zoom in

```
p +  
  coord_cartesian(ylim = c(70, 85), xlim = c(20000, 40000))
```

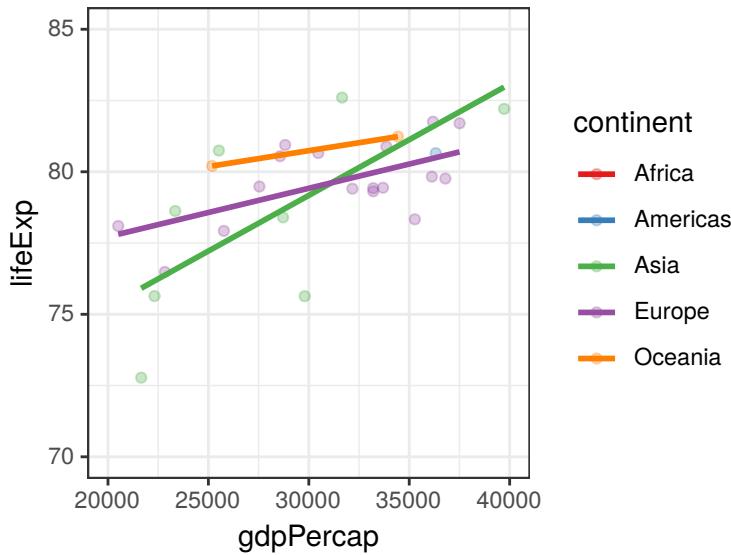


### 5.2.4 Exercise

How is this one different to the previous?

```
p +
  scale_y_continuous(limits = c(70, 85)) +
  scale_x_continuous(limits = c(20000, 40000))

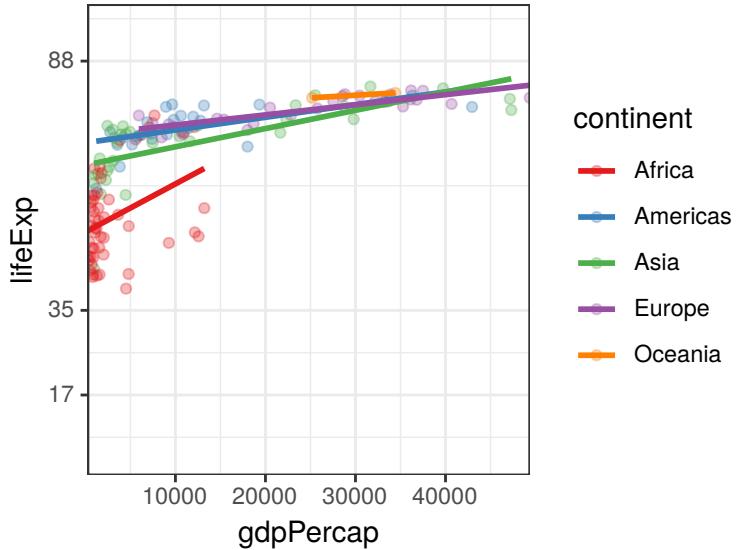
## Warning: Removed 114 rows containing non-finite values (stat_smooth).
## Warning: Removed 114 rows containing missing values (geom_point).
```



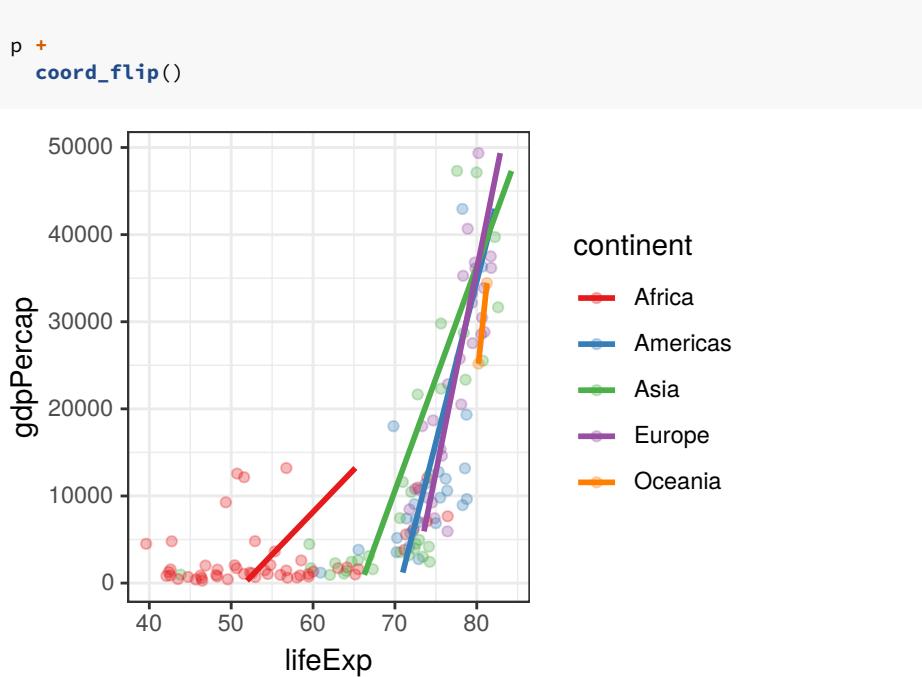
Answer: the first one zooms in, still retaining information about the excluded points when calculating the linear regression lines. The second one removes the data (as the warnings say), calculating the linear regression lines only for the visible points.

### 5.2.5 Axis ticks

```
p +
  coord_cartesian(ylim = c(0, 100), expand = 0) +
  scale_y_continuous(breaks = c(17, 35, 88))
```



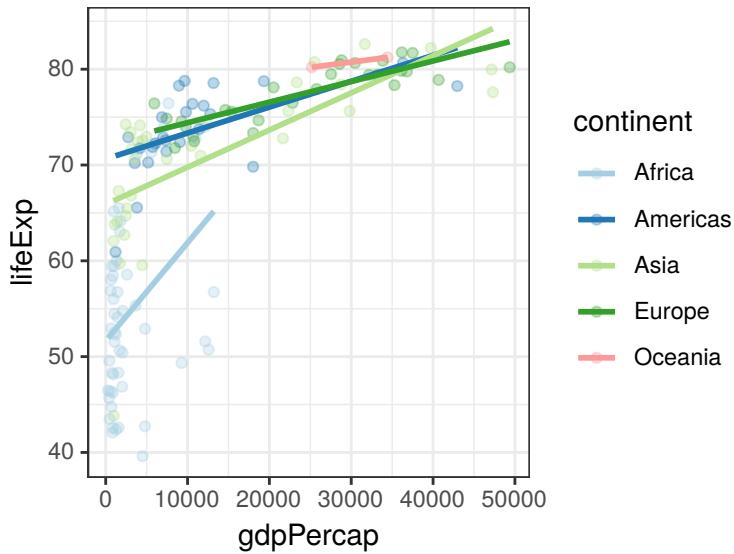
#### 5.2.6 Swap the axes



### 5.3 Colours

#### 5.3.1 Using the Brewer palettes:

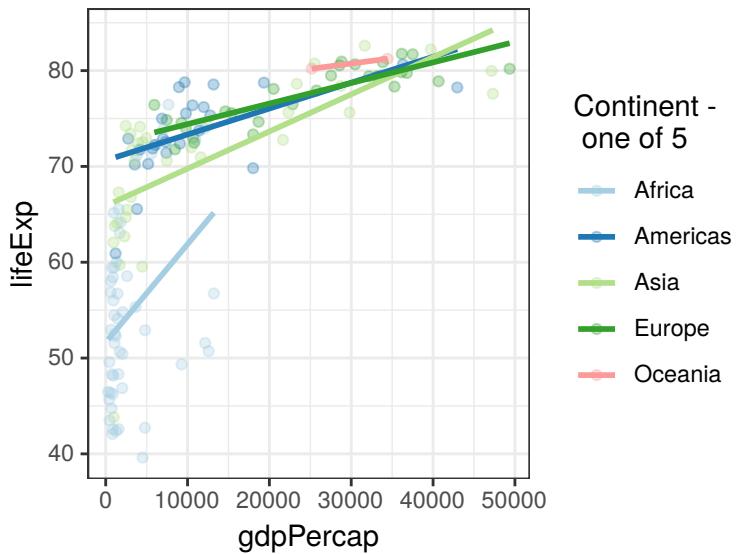
```
p +  
  scale_color_brewer(palette = "Paired")
```



#### 5.3.2 Legend title

`scale_colour_brewer()` is also a convenient place to change the legend title:

```
p +  
  scale_color_brewer("Continent - \n one of 5", palette = "Paired")
```

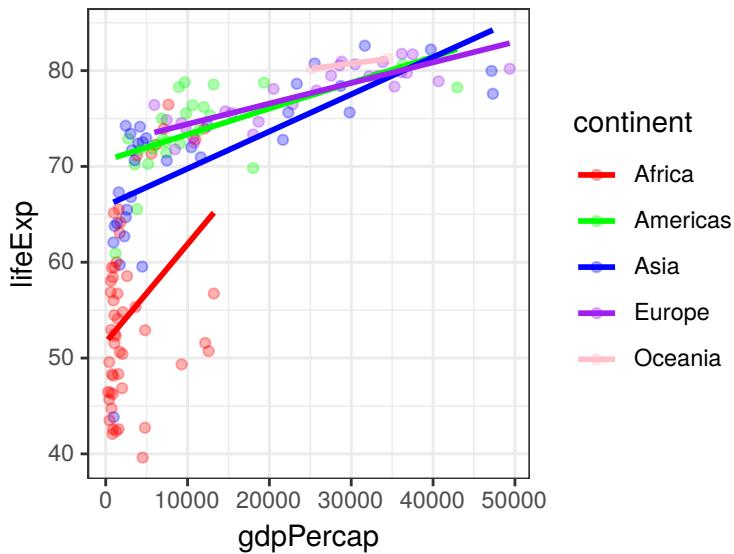


Note the \n inside the new legend title - new line.

### 5.3.3 Choosing colours manually

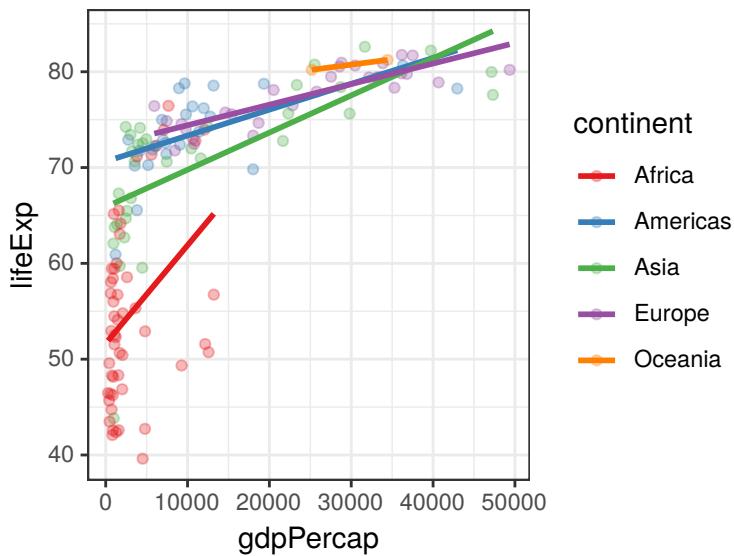
Use words:

```
p +  
  scale_color_manual(values = c("red", "green", "blue", "purple", "pink"))
```



Or HEX codes (either from <http://colorbrewer2.org/> or any other resource):

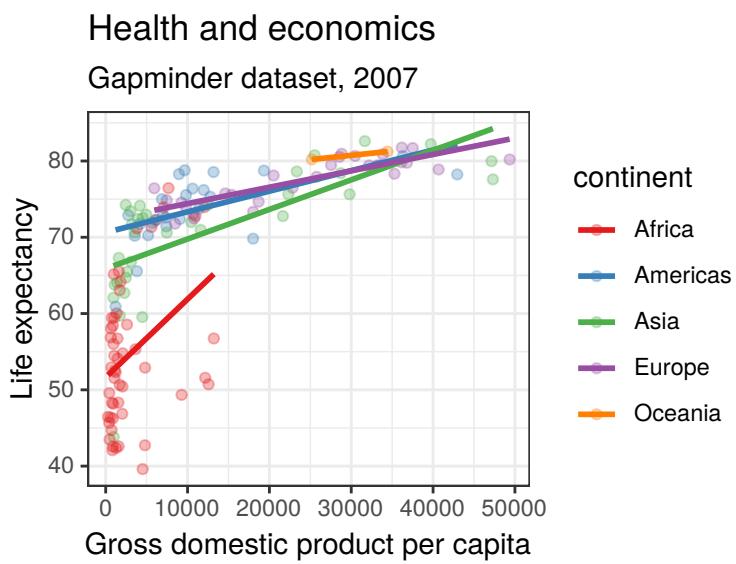
```
p +
  scale_color_manual(values = c("#e41alc", "#377eb8", "#4daf4a", "#984ea3", "#ff7f00"))
```



Note that <http://colorbrewer2.org/> also has options for *Colourblind safe* and *Print friendly*.

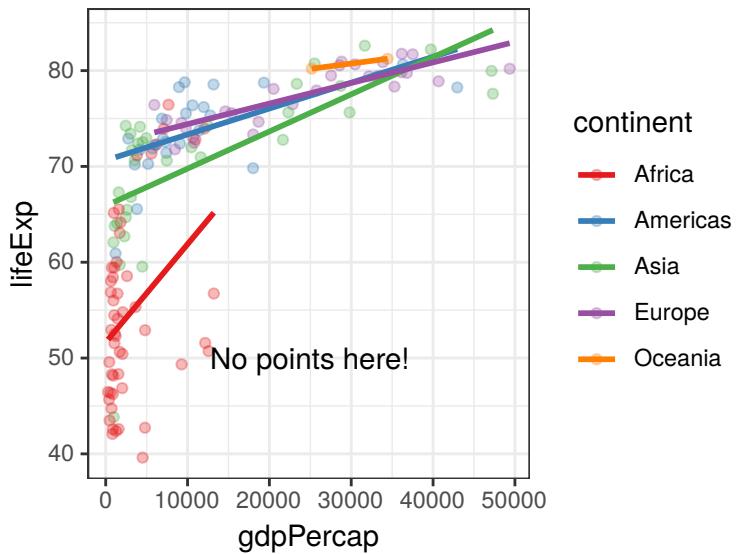
## 5.4 Titles and labels

```
p +
  labs(x = "Gross domestic product per capita",
       y = "Life expectancy",
       title = "Health and economics",
       subtitle = "Gapminder dataset, 2007")
```

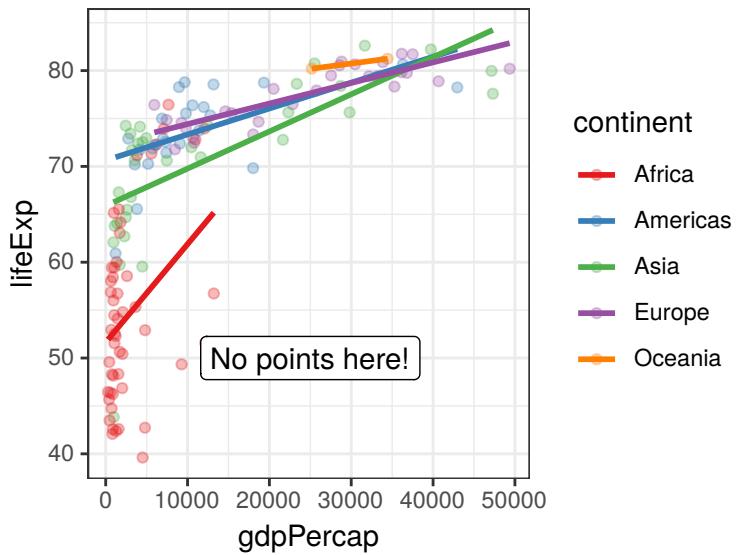


### 5.4.1 Annotation

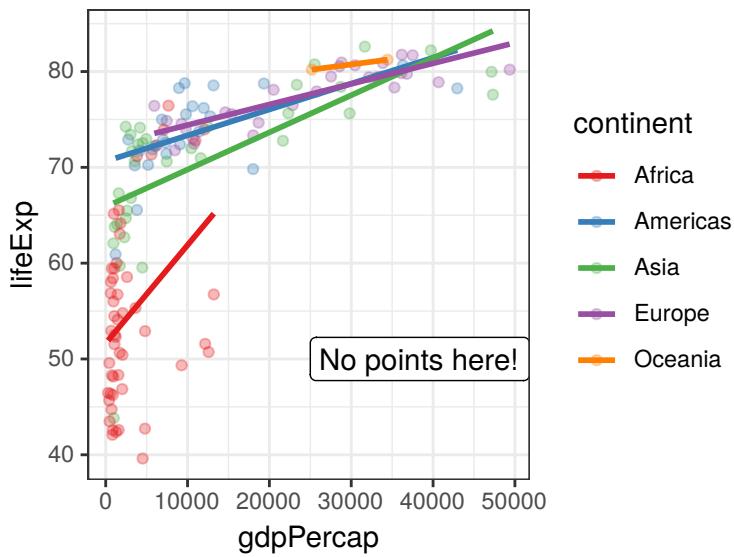
```
p +
  annotate("text",
           x = 25000,
           y = 50,
           label = "No points here!")
```



```
p +  
  annotate("label",  
          x = 25000,  
          y = 50,  
          label = "No points here!")
```



```
p +
  annotate("label",
    x = 25000,
    y = 50,
    label = "No points here!",
    hjust = 0)
```



`hjust` stand 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.

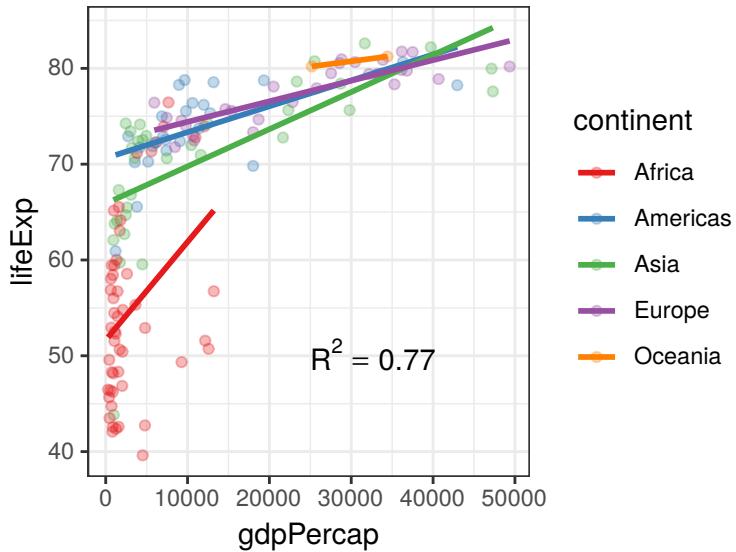
#### 5.4.2 Annotation with a superscript and a variable

```
fit_glance = data.frame(r.squared = 0.7693465)

plot_rsquared = paste0(
  "R^2 == ",
  fit_glance$r.squared %>% round(2))

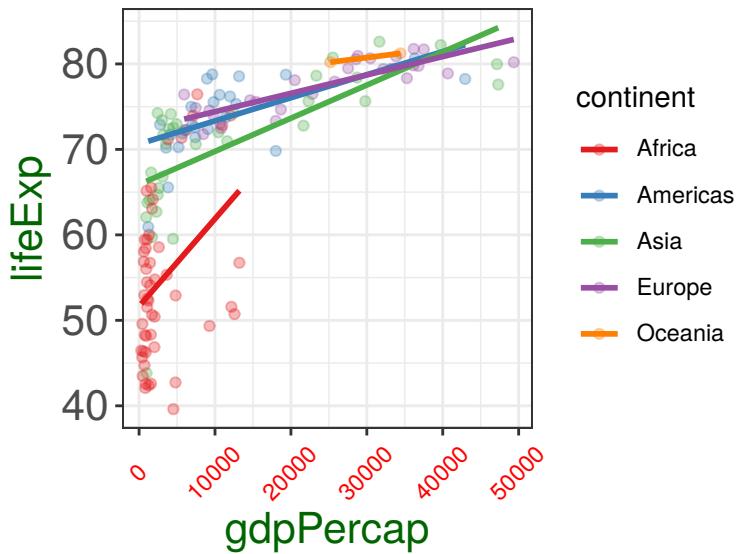
p +
  annotate("text",
```

```
x = 25000,
y = 50,
label = plot_rsquared, parse = TRUE,
hjust = 0)
```



## 5.5 Text size

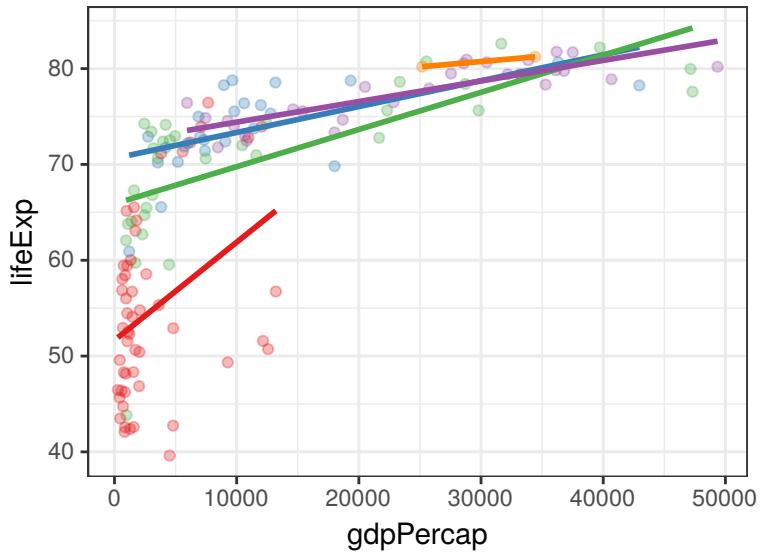
```
p +
  theme(axis.text.y = element_text(size = 16),
        axis.text.x = element_text(colour = "red", angle = 45, vjust = 0.5),
        axis.title = element_text(size = 16, colour = "darkgreen")
      )
```



### 5.5.1 Legend position

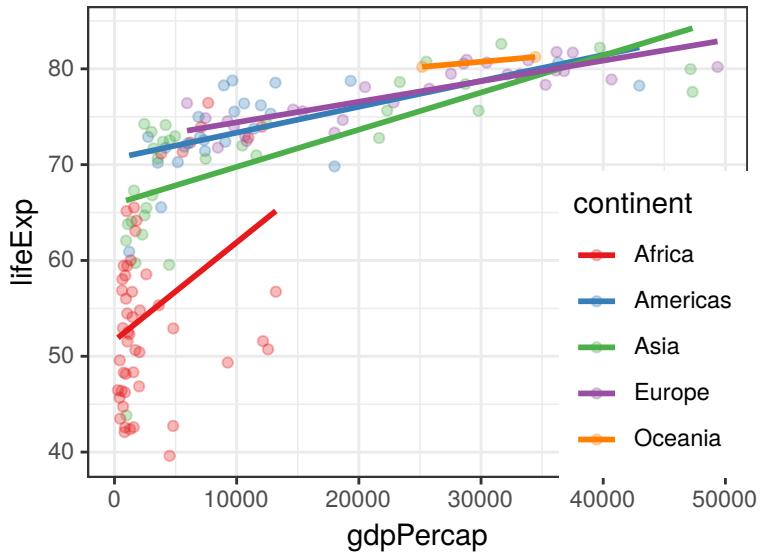
Use the following words: "right", "left", "top", "bottom", OR "none" to remove the legend.

```
p +  
  theme(legend.position = "none")
```

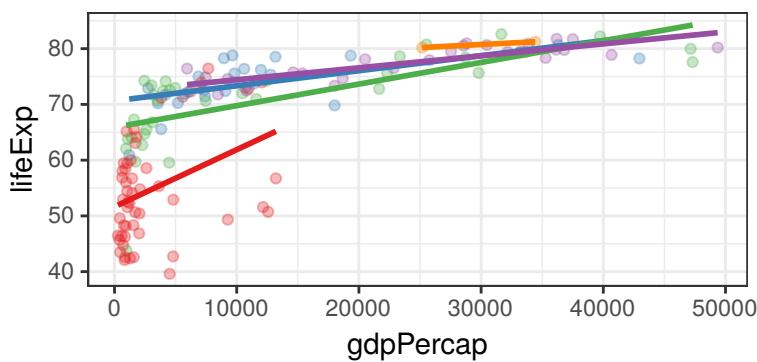


Or use relative coordinates (0–1) to give it an -y location:

```
p +
  theme(legend.position      = c(1,0),
        legend.justification = c(1,0)) #bottom-right corner
```



```
p +  
  theme(legend.position = "top") +  
  guides(colour = guide_legend(ncol = 2))
```



## 5.6 Saving your plot

```
ggsave(p, 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 life expectancy between the year 2002 and 2007 or between the Africa and Europe, is usually more useful than simply describing the average life expectancy across the entire world 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 project<sup>1</sup>. 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. 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. 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.

---

<sup>1</sup><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
```

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?

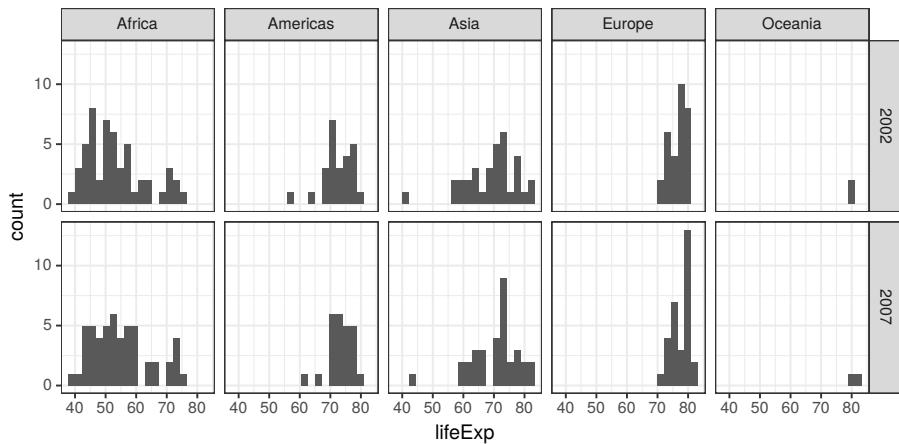
There are three useful plots which can help here:

- 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
```

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.



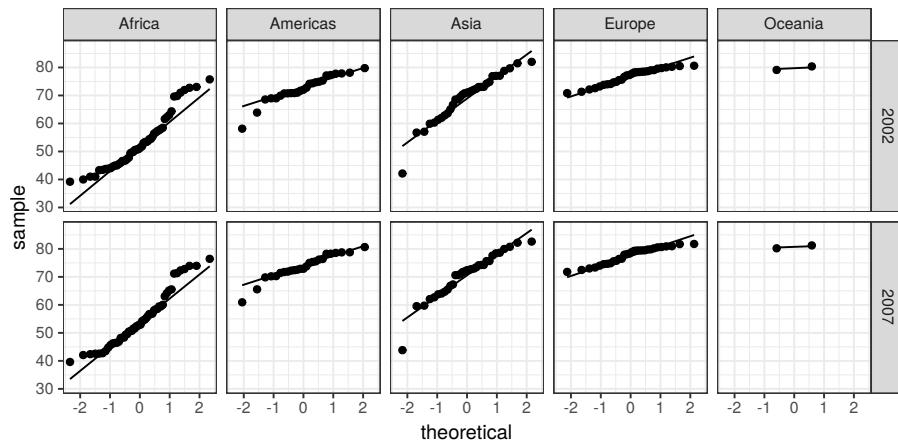
**FIGURE 6.1:** Histogram: country life expectancy by continent and year

### 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 observations. 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 (default below is normal). 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` # defaults to normal distribution
    geom_qq_line() + # add 45 degree line
    facet_grid(year ~ continent)
```

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,



**FIGURE 6.2:** Q-Q plot: country life expectancy by continent and year

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.

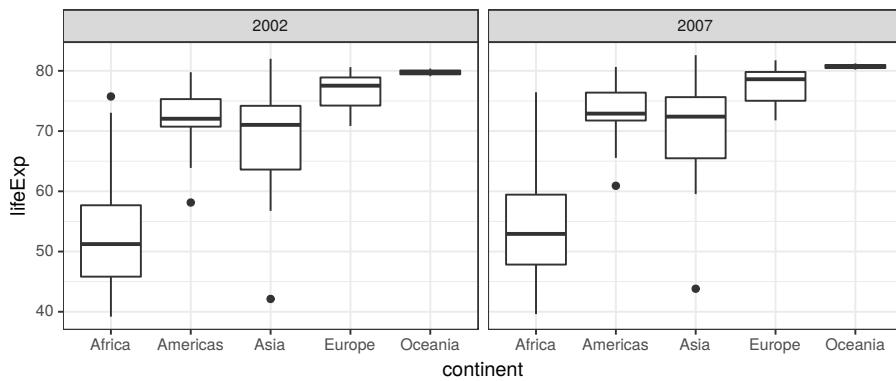
### 6.5.3 Boxplot

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 tendency (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 `.`
```



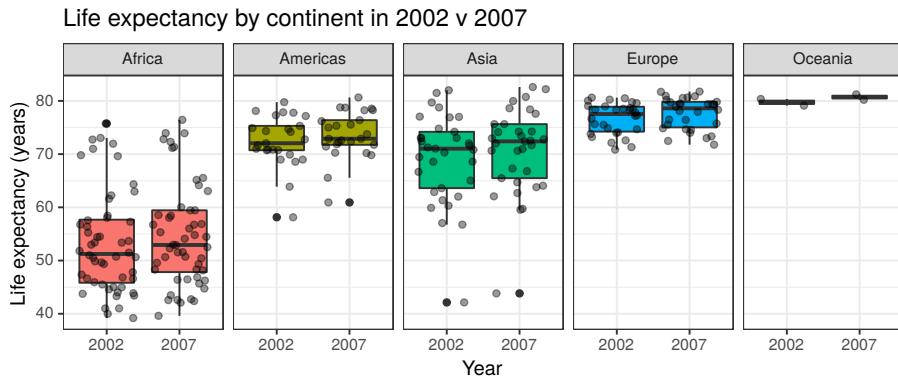
**FIGURE 6.3:** Boxplot: country life expectancy by continent and year

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
  facet_grid(. ~ continent) + # spread by year, note `.` 
  theme(legend.position = "none") + # remove legend
  xlab("Year") + # label x-axis
  ylab("Life expectancy (years)") + # label y-axis
```

```
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 if independent groups (first figure). Repeated-measures data such as comparing the same countries between years can be analysed using a paired *t*-test (second figure).

### 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 %>%
  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) %>%
                                # 2007 only
  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-05    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.

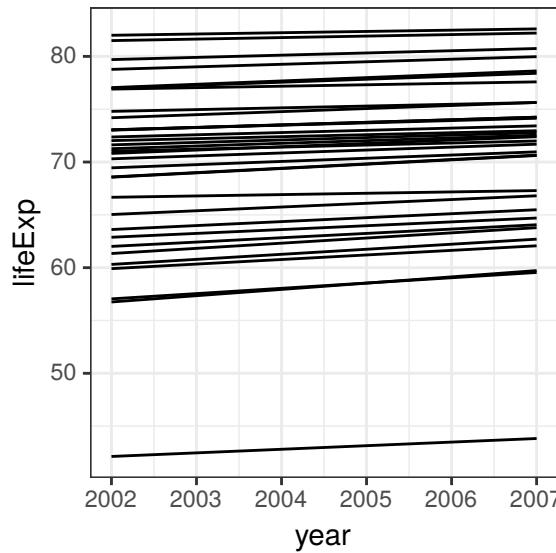
#### 6.6.4 Paired *t*-tests

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)) %>%
  filter(continent == "Asia")           # Asia only

paired_data %>%
  ggplot(aes(x = year, y = lifeExp,
             group = country)) +
  geom_line()                         # for individual country lines
```



**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 %>%
  select(country, year, lifeExp) %>%
  spread(year, lifeExp) %>%
  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 x 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 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) %>%
  group_by(continent) %>%
  do(
    t.test(.lifeExp, mu = 77) %>%
    tidy()
  )

## # 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    -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>

mydata %>%
  filter(year == 2007) %>%
  group_by(continent) %>%
  group_modify(
    ~ t.test(.lifeExp, mu = 77) %>%
    tidy()
  )

## # 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    -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 comparing life expectancy across 3 continents such as the Americas, Europe and Asia.

### 6.8.1 Plot the data

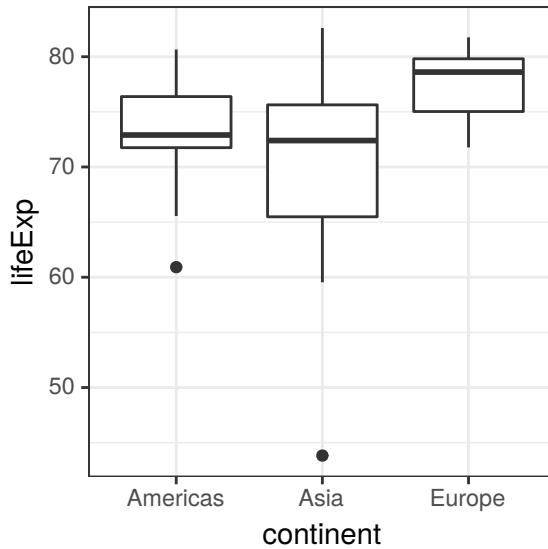
```
mydata %>%
  filter(year == 2007) %>%
  filter(continent %in%
         c("Americas", "Europe", "Asia")) %>%
  ggplot(aes(x = continent, y=lifeExp)) +
  geom_boxplot()
```

### 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"))
```



**FIGURE 6.6:** Boxplot: Life expectancy in selected continents for 2007

```
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 <NA>       2    755.6  377.8    11.63 3.42e-05
```

```
## 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)
```

```
par(mfrow=c(1,1))
```

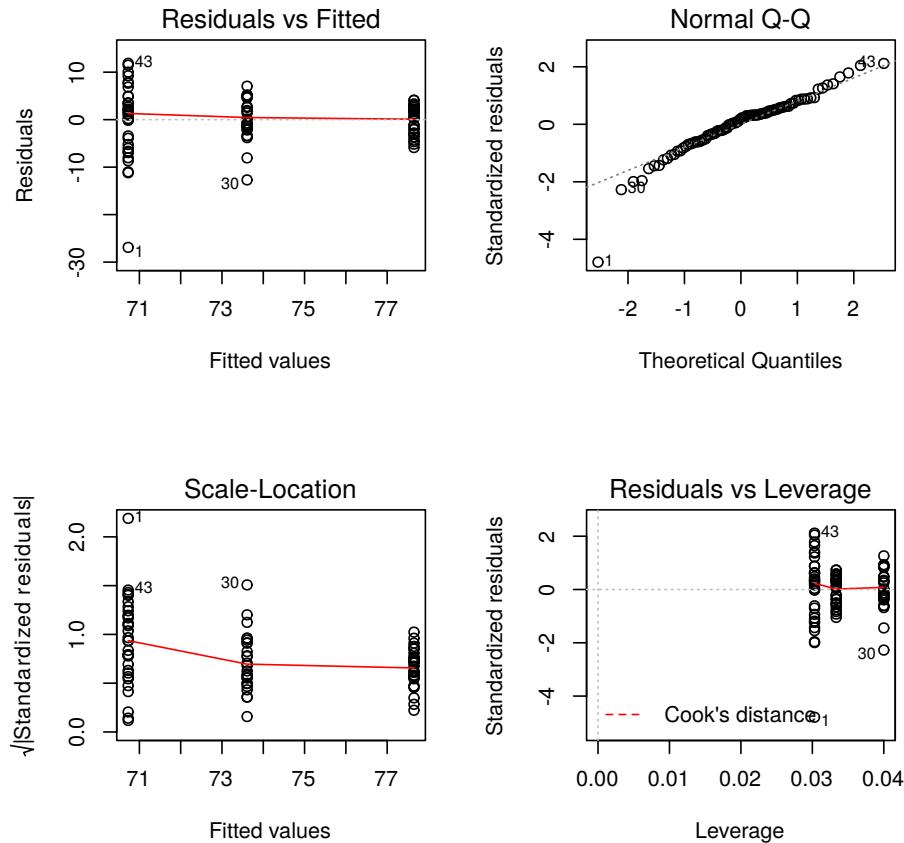
### 6.8.4 Pairwise testing and multiple comparisons

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.



**FIGURE 6.7:** Diagnostic plots: ANOVA model of life expectancy by continent for 2007

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 rejection of 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 to no 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 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 particularly 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,
  p.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, 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 x).

If a comparison of means approach is reasonable, but the normality assumption are 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 most common.

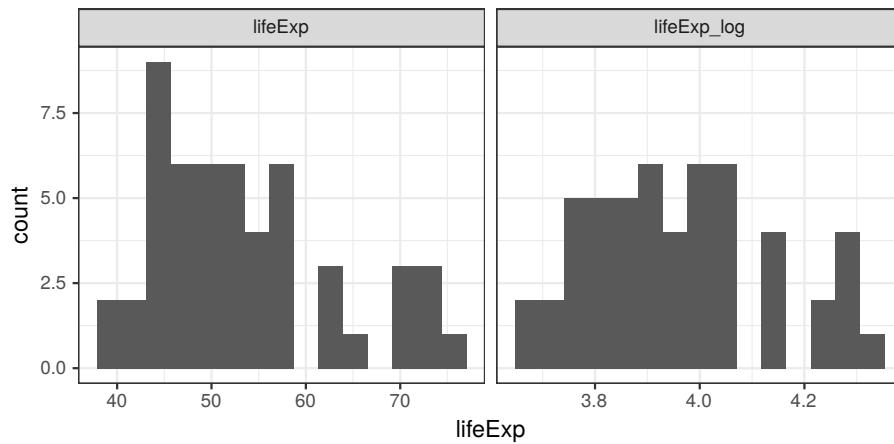
```

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)) %>%          # only 1982 and 2007
  filter(continent %in% c("Africa"))               # 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)

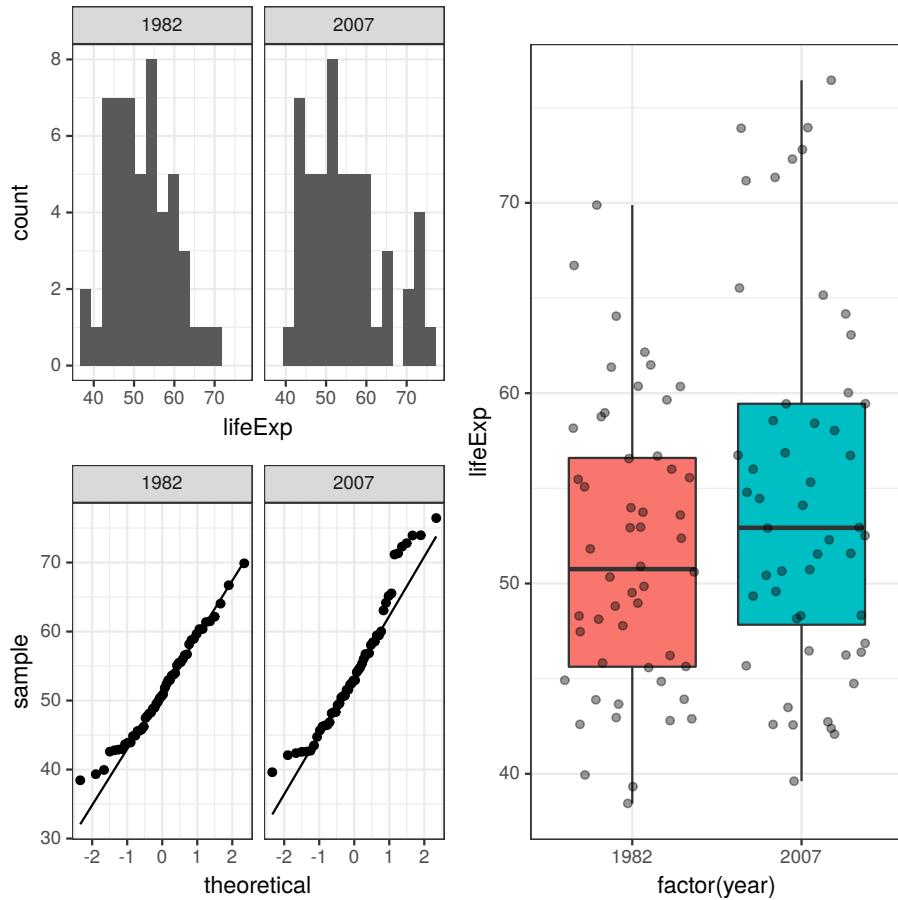
p3 = africa_plot %>%                            # save plot as p3
  ggplot(aes(x = factor(year),
             y = lifeExp)) +
  geom_boxplot(aes(fill = factor(year))) +          # change year to factor
  geom_jitter(alpha = 0.4) +                         # colour boxplot
  theme(legend.position = "none")                   # add data points
                                                 # 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) %>%
knitr::kable(row.names = FALSE, booktabs = TRUE,
            align = c("l", "l", "r", "r", "r", "r"),
            caption = "Life expectancy, population and GDPperCap in Africa 1982 v 2007")

```

**TABLE 6.1:** 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?

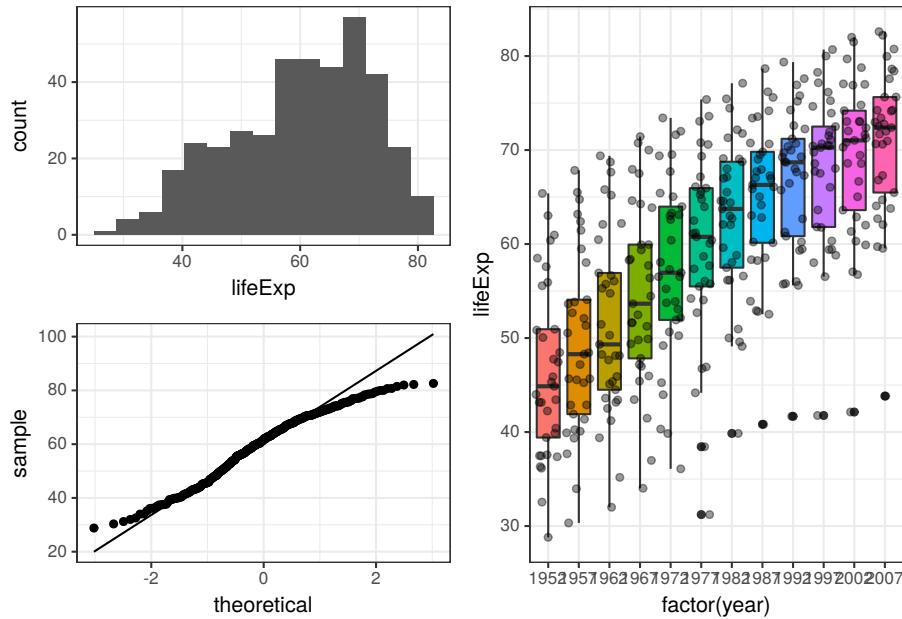
asia_plot = mydata %>%
  filter(continent %in% c("Asia"))

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.

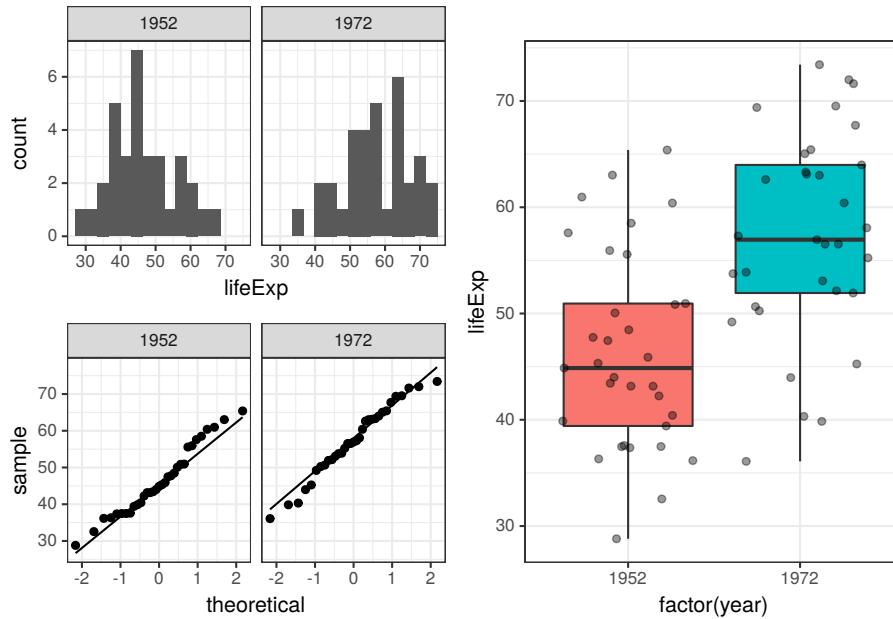
asia_years = mydata %>%
  filter(continent %in% c("Asia")) %>%
  filter(year %in% c(1952, 1972))

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 = .)
```

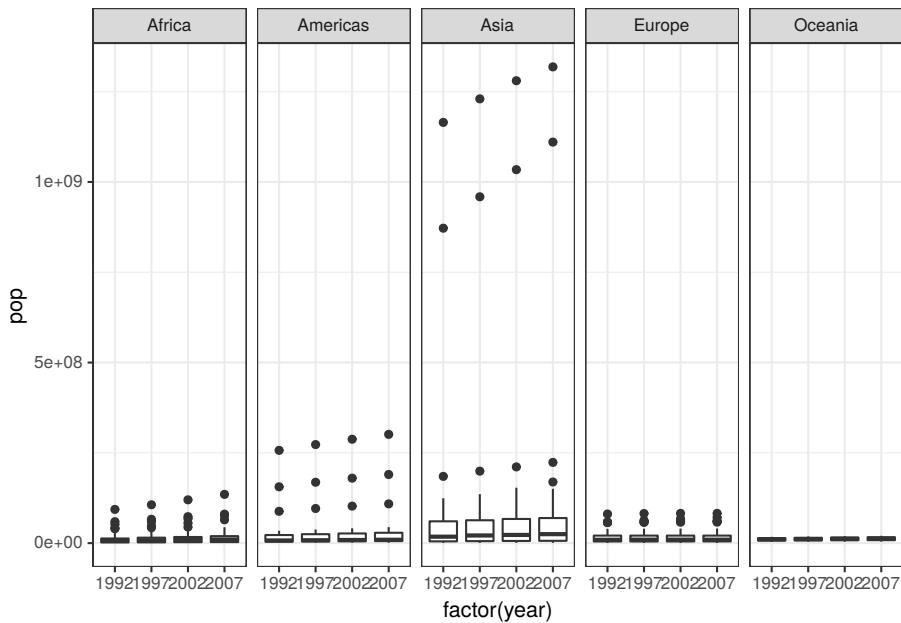
```
##
## Welch Two Sample t-test
##
## data: lifeExp by year
## t = -4.7007, df = 63.869, p-value = 1.428e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -15.681981 -6.327769
## sample estimates:
## mean in group 1952 mean in group 1972
##           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>    <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  
## # Groups: continent [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 with 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.

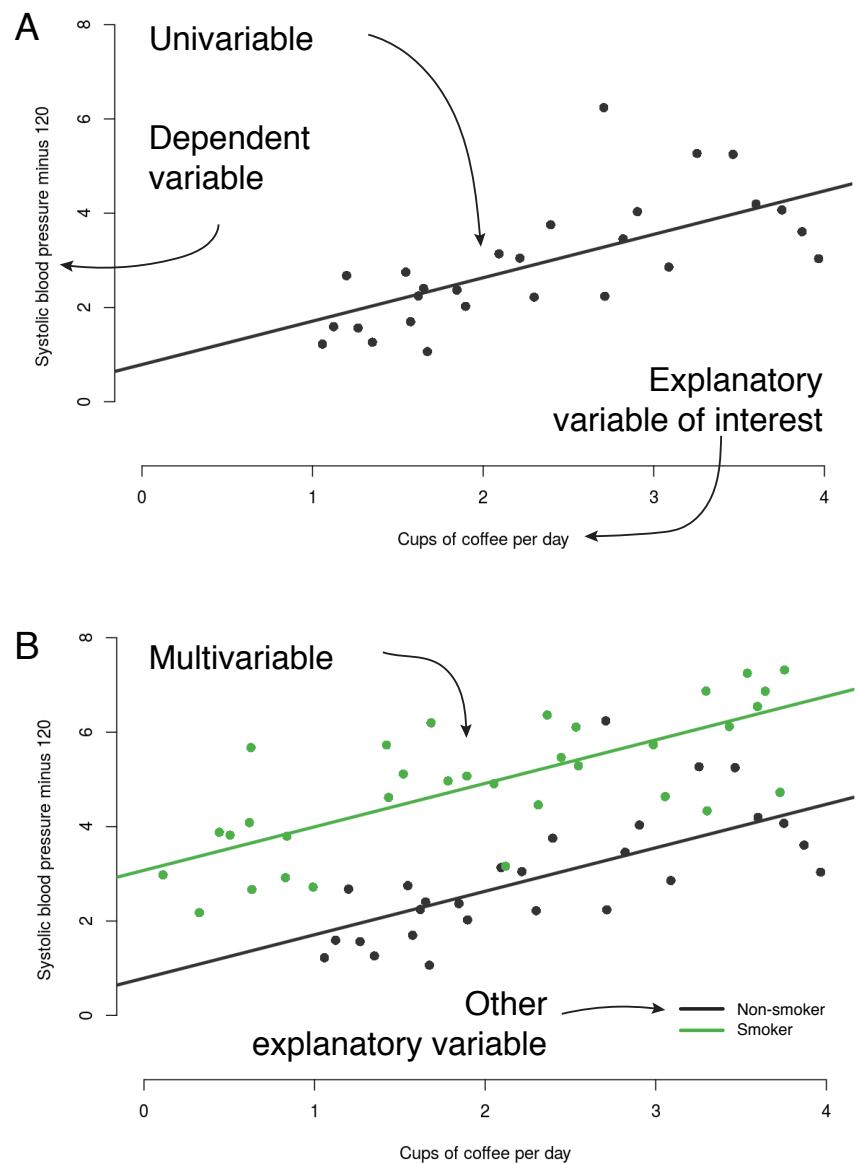
### 7.1.1 The Question (1)

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 and average number of cups of coffee per day is the explanatory variable of interest ( $x$ ) 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 of this 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 app<sup>1</sup> 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 penalise further away points) 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 app<sup>2</sup> to get a handle on these.

Figure 7.3 shows diagnostic plots from the app, which we will run ourselves below.

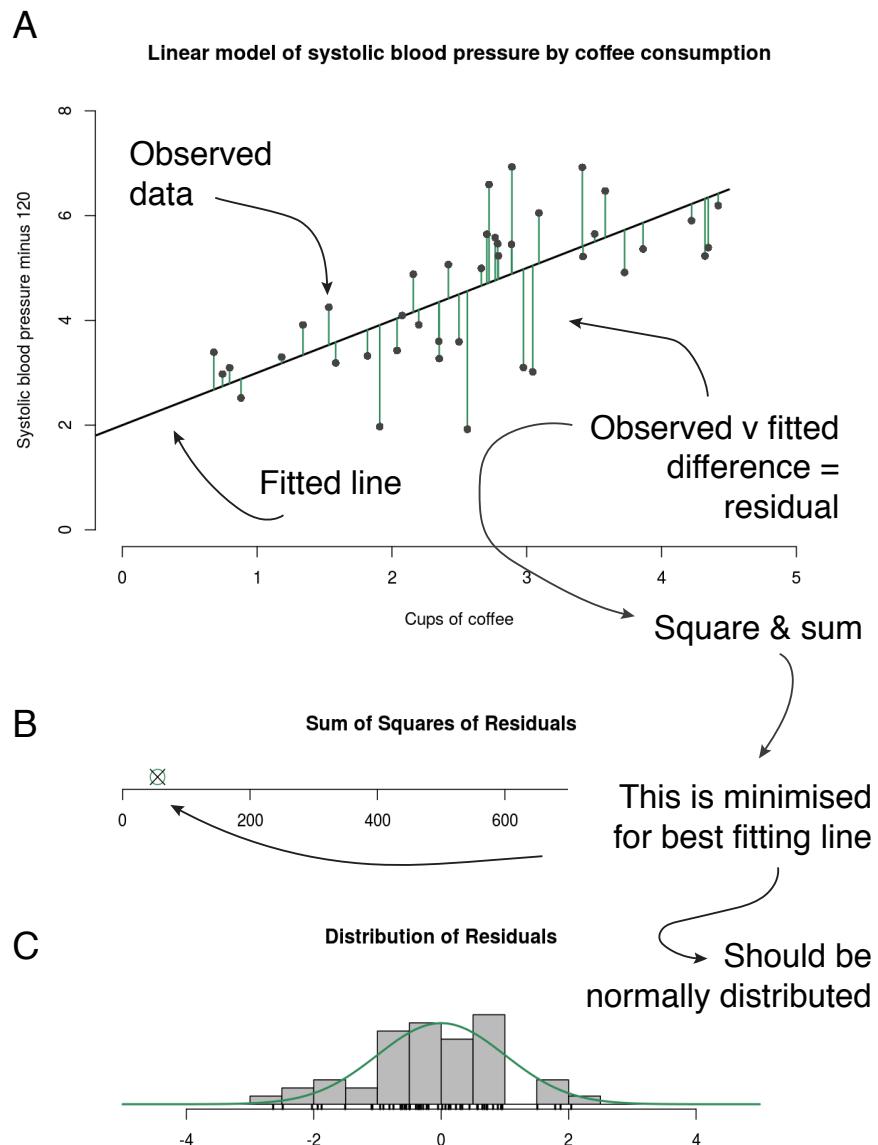
#### *Linear relationship*

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

---

<sup>1</sup>[https://argoshare.is.ed.ac.uk/simple\\_regression](https://argoshare.is.ed.ac.uk/simple_regression)

<sup>2</sup>[https://argoshare.is.ed.ac.uk/simple\\_regression\\_diagnostics](https://argoshare.is.ed.ac.uk/simple_regression_diagnostics)



**FIGURE 7.2:** How a regression line is fitted.

model should be considered, such as including a quadratic ( $x^2$ ) or polynomial term.

#### *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 *Q-Q plot ref*.

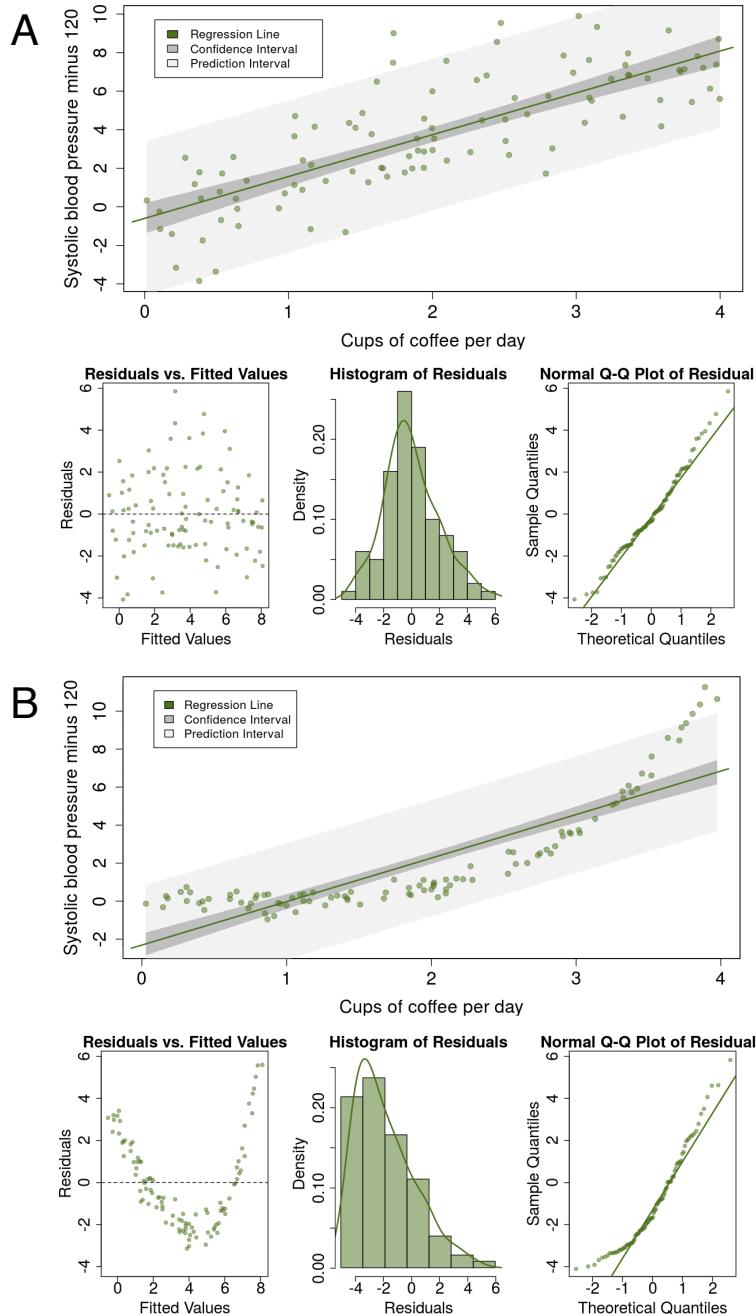
#### *Equal variance of residuals*

The distribution of the observations around the fitted line should be the same on the left side of the scatter plot as they are on the right side. Look at the fan-shaped data on the simple regression diagnostics shiny app<sup>3</sup>. 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 valid across the range of the explanatory variable and dependent variable. It is really about understanding the underlying data, rather than relying on a fancy statistical test that gives you a p-value.

---

<sup>3</sup>[https://argosshare.is.ed.ac.uk/simple\\_regression\\_diagnostics](https://argosshare.is.ed.ac.uk/simple_regression_diagnostics)



**FIGURE 7.3:** Regression diagnostics. Does this also appear in the contents. What about this?

### 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.

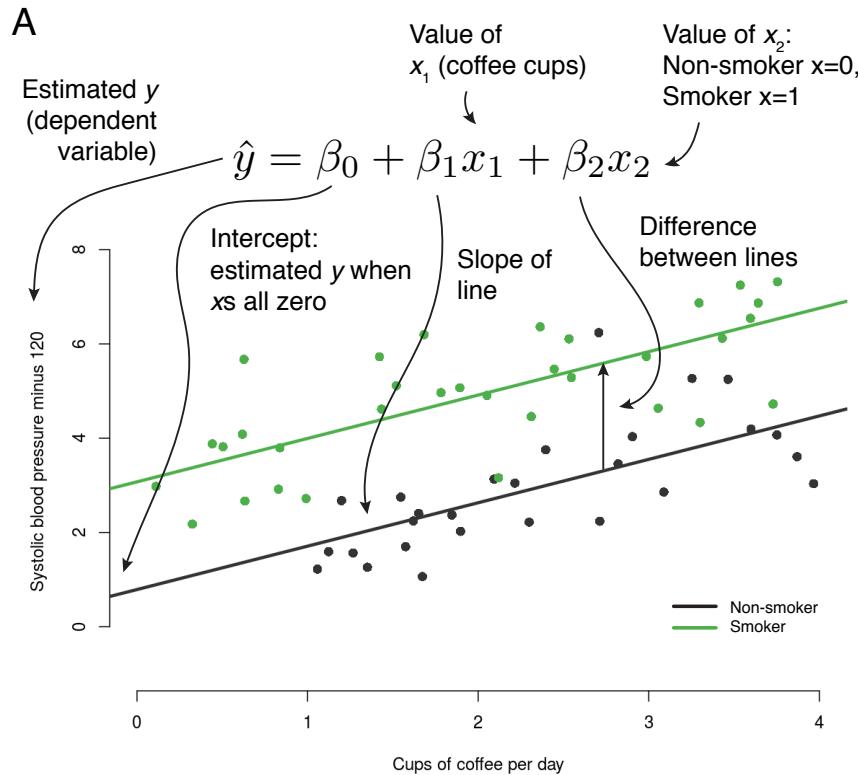
Figure 7.4A shows a scatter plot with fitted lines from a multivariable linear regression model. The plot is taken from the multivariable regression shiny app<sup>4</sup>. Remember, these data are simulated and are not real. This app will really help you understand different regression models, more on this below. The app allows us to specify “the truth” with the sliders on the left hand side. For instance, we can set the  $\text{intercept}=1$ , meaning that when all  $x = 0$ , the value of the dependent variable,  $\hat{y} = 1$ .

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  $\text{intercept} = 1$ ,  $\beta_1 = 1$  (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  $\text{intercept} = 0.67$ ,  $\beta_1 = 1.00$  (coffee), and  $\beta_2 = 2.48$  (smoking). A  $p$ -value is provided ( $\text{Pr}(> |t|)$ ), which is the result of a null hypothesis significance test for the gradient of the line being equal to zero. Said another way, this is the probability that the gradient of the particular line is equal to zero.

---

<sup>4</sup>[https://argoshare.is.ed.ac.uk/multi\\_regression/](https://argoshare.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 ← Distribution of residuals

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 ← Adjusted R2

```

$s\hat{B}P = \beta_0 + \beta_{coffee}x_{coffee} + \beta_{smoking}x_{smoking}$

**FIGURE 7.4:** Linking the fitted line, regression equation and R output.

### 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 app<sup>5</sup>.

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.

#### *Additive vs. multiplicative effect modification (interaction)*

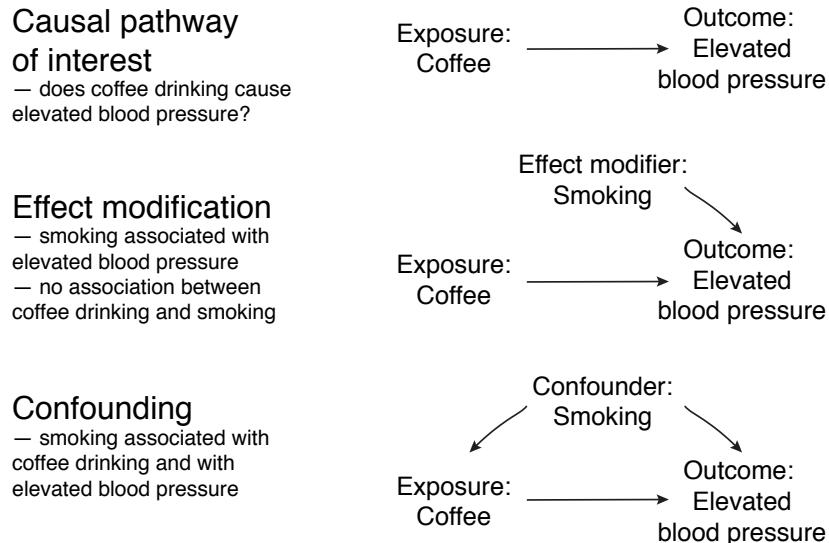
Depending on the field you work, will depend on which set of terms you use. Effect modification can be additive or multiplicative. We refer to multiplicative effect modification as simply including a statistical interaction.

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  $x_2 = 0$ );

---

<sup>5</sup>[https://argosshare.is.ed.ac.uk/multi\\_regression/](https://argosshare.is.ed.ac.uk/multi_regression/)



**FIGURE 7.5:** Causal pathways, effect modification and confounding.

- $\beta_2 = 1$  (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  $\beta_1$  and  $\beta_2$ , 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.

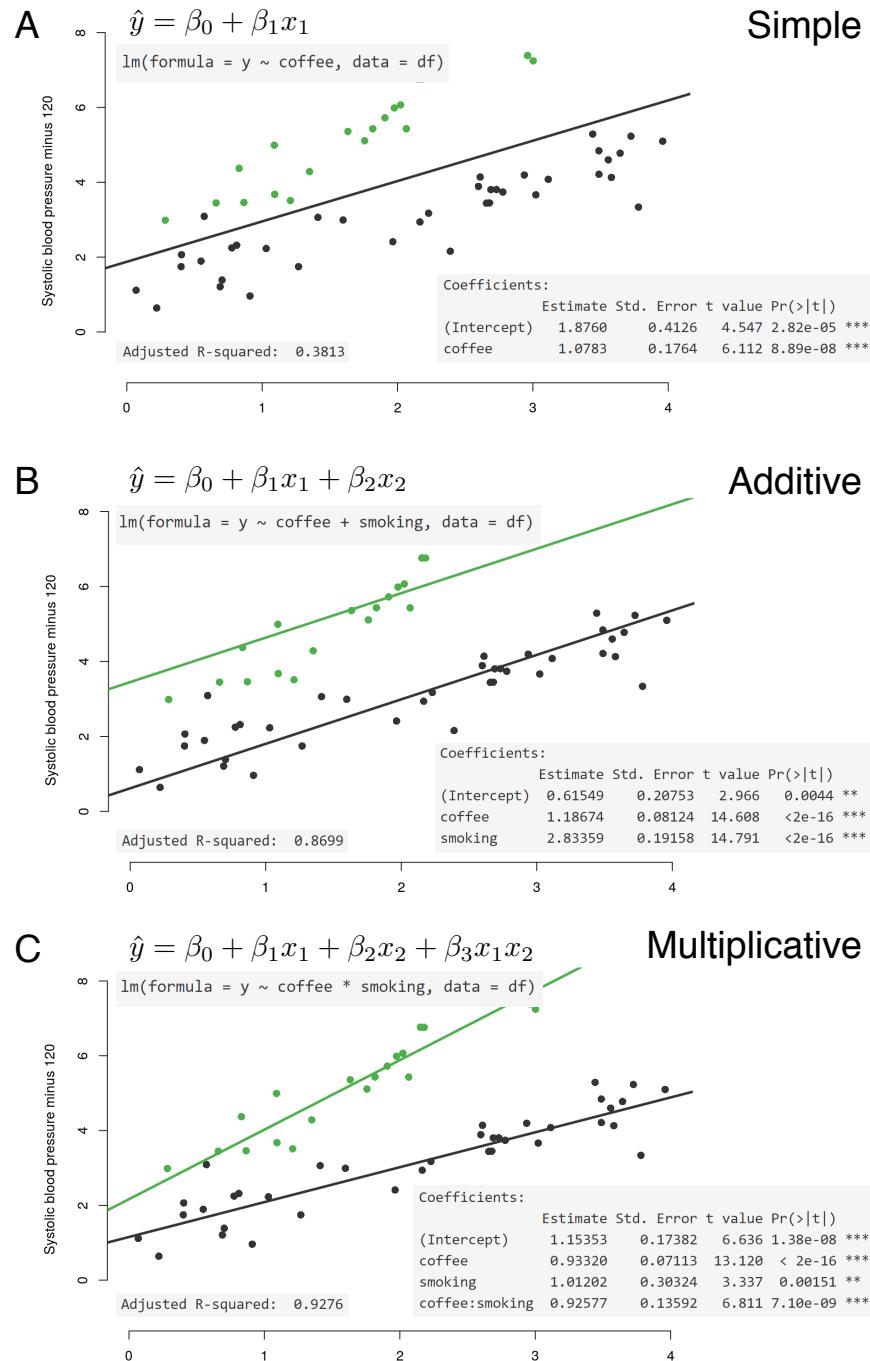
### 7.1.6 R-squared and model fit

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).



**FIGURE 7.6:** Multivariable linear regression with additive and multiplicative effect modification.

### 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 consumption (Figure 7.5C). This exists simply if smokers drink more coffee than non-smokers.

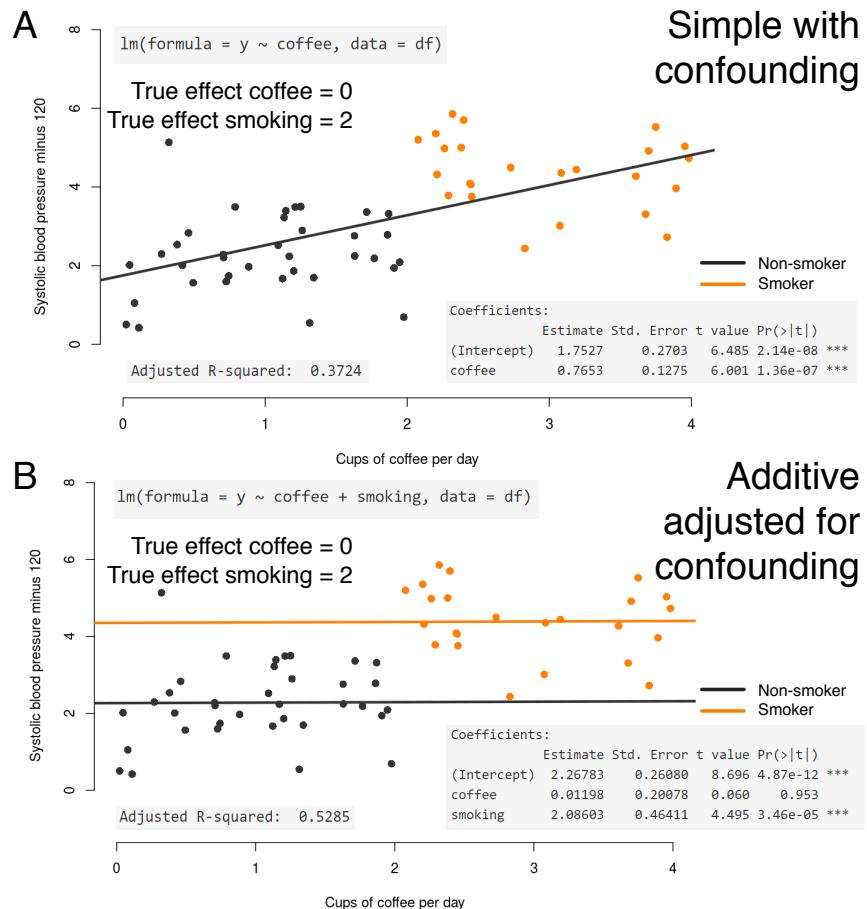
Figure 7.7 shows this really clearly. The underlying data have been altered so that those who drink more than two cups of coffee per day also smoke and those who drink fewer than two cups per day do not smoke. A true effect of smoking on blood pressure is entered, but NO effect of coffee on blood pressure.

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.



**FIGURE 7.7:** Multivariable linear regression with confounding of coffee drinking by smoking.

We will now perform all this ourselves in R using a gapminder dataset which you are familiar with from preceding chapters.

## 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 06-1

```
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") %>% # save as object p1
  # Europe only
```

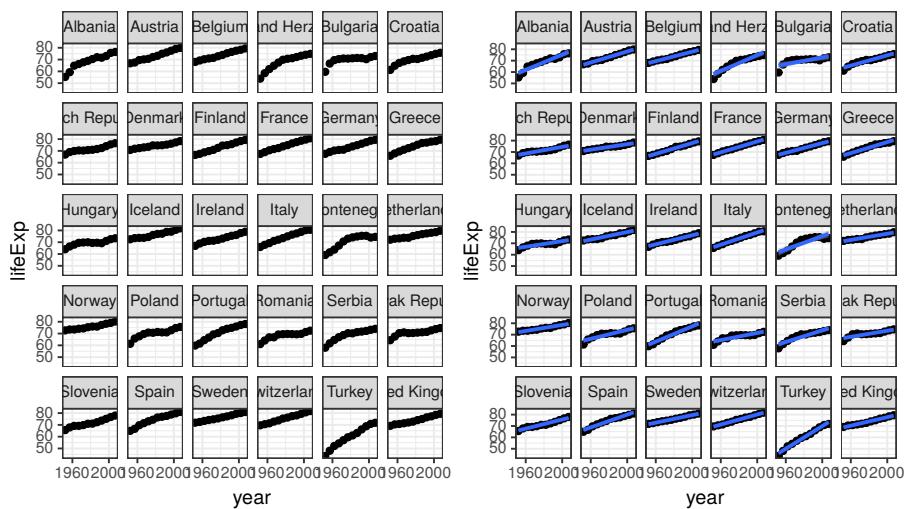
```

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

```



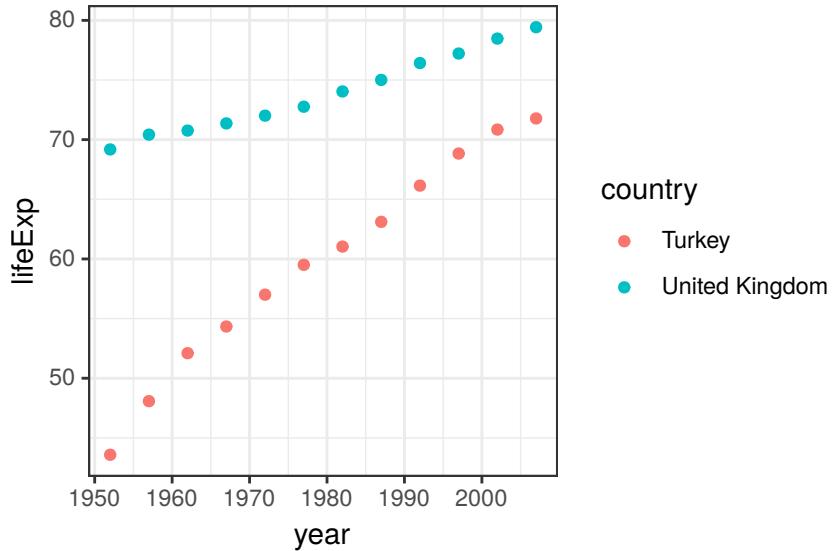
**FIGURE 7.8:** Scatterplot with fitted line plot: Life expectancy by year in European countries

### 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()
```



**FIGURE 7.9:** Scatterplot: Life expectancy by year Turkey and Europe.

The two non-parallel lines may make you think of what has been discussed above.

First, let's model the two countries separately.

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

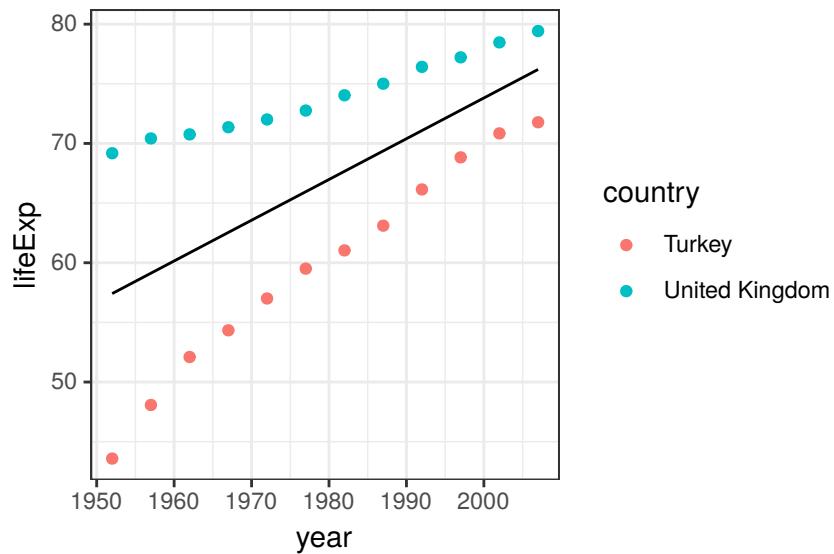
## 
## Call:
## lm(formula = lifeExp ~ year_from1952, data = .)
## 
## Coefficients:
## (Intercept)  year_from1952
##      57.4154        0.3416
```

```

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))

```



**FIGURE 7.10:** Scatter and line plot. Life expectancy in Turkey and the UK - univariable fit.

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*

```

fit_both2 = mydata_UK_T %>%
  lm(lifeExp ~ year_from1952 + country, data = .)
fit_both2

##
## Call:
## lm(formula = lifeExp ~ year_from1952 + country, data = .)

```

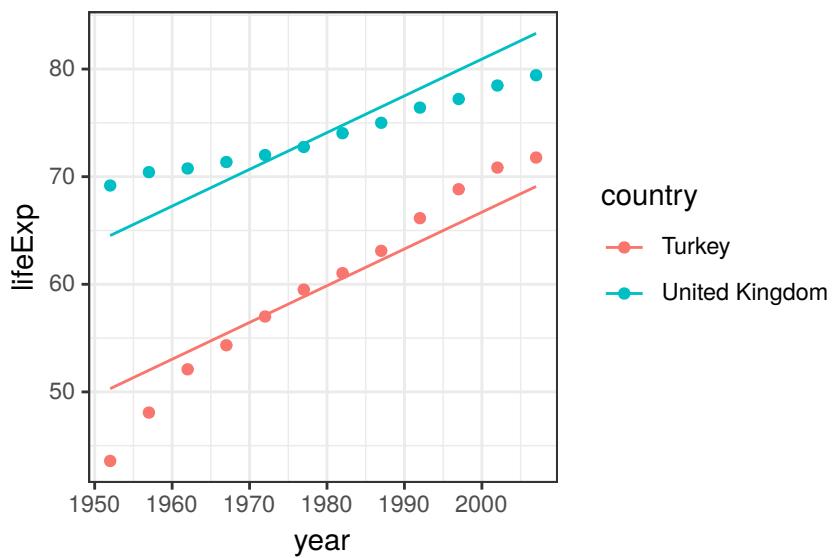
```

## 
## 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))

```



**FIGURE 7.11:** Scatter and line plot. Life expectancy in Turkey and the UK - multivariable additive fit.

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 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*

```

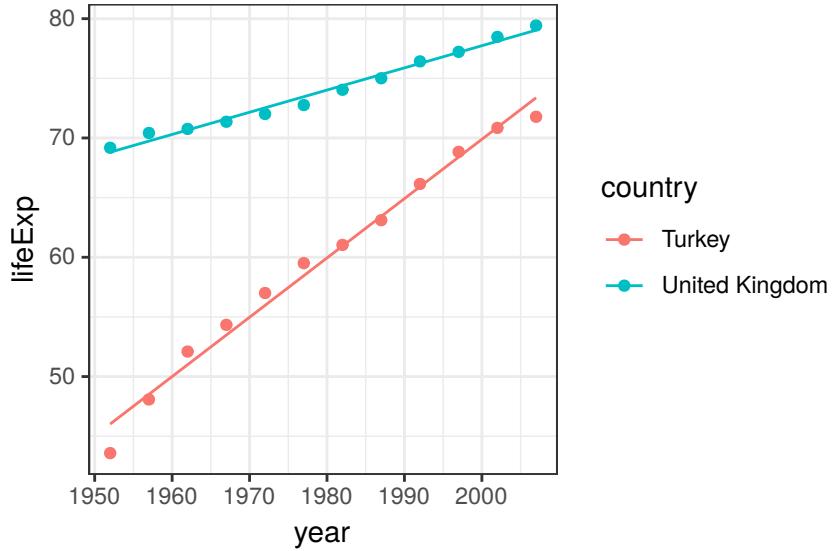
fit_both3 = mydata_UK_T %>%
  lm(lifeExp ~ year_from1952 * country, data = .)
fit_both3

## 
## 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*

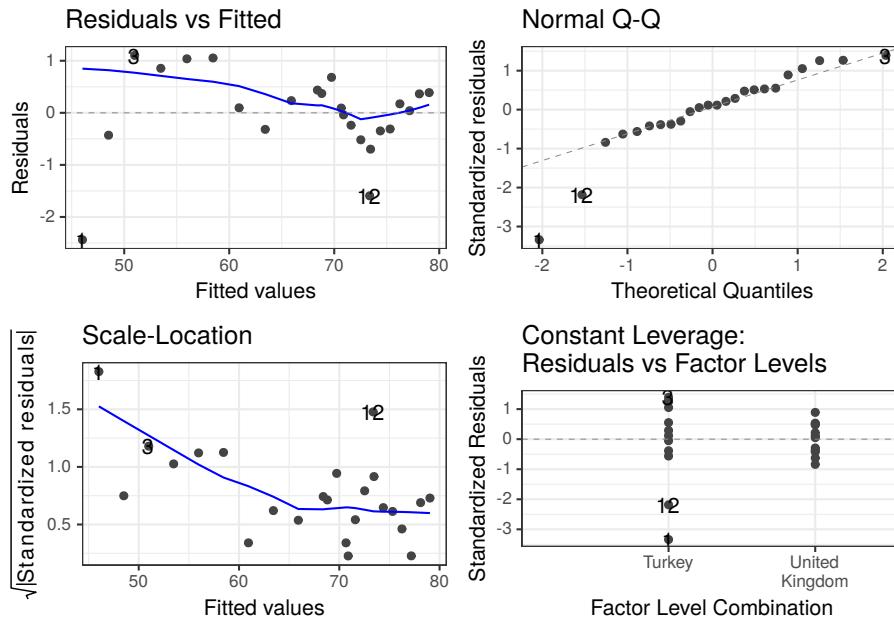
```
library(purrr)
list(fit_both1, fit_both2, fit_both3) %>%
  map_df(glance, .id = "fit")

## # 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 five 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 enough, 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 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 (ref: 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. Its use is as a comparison between 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.

```
## 
## Attaching package: 'kableExtra'
## 
## The following object is masked from 'package:dplyr':
## 
##     group_rows
```

**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

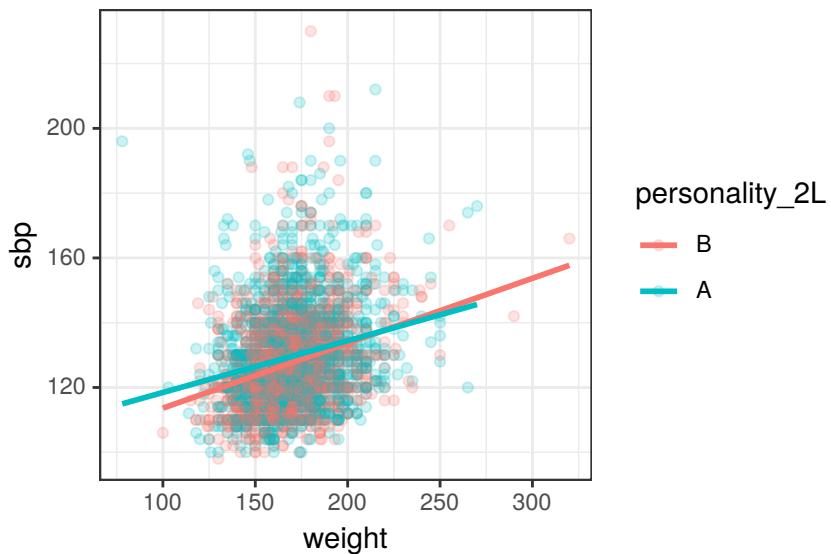
**TABLE 7.2:** WCGS data, ff\_glimpse: categorical

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)
```



**FIGURE 7.13:** Scatter and line plot. Systolic blood pressure by weight and personality type.

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.

### 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 website<sup>6</sup>.

---

<sup>6</sup><https://finalfit.org>

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.

```
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 (REF Ch8 `fct_relevel`). 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 missing data section. 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 may hypothesise also influences blood pressure.

```
dependent = "sbp"
explanatory = c("weight", "personality_2L")
fit_sb2 = 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 an

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 (REF SEE smoker/bp plots).

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 A	127.5 (14.4) 129.8 (15.7)	2.32 (1.26 to 3.37, p<0.001) 0.18 (0.16 to 0.21, p<0.001)	1.44 (0.44 to 2.43, p=0.005) 0.24 (0.21 to 0.27, p<0.001)
Weight (pounds)	[78,320]	128.6 (15.1)		
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 Smoker	128.6 (15.6) 128.7 (14.6)	0.08 (-0.98 to 1.14, p=0.883)	0.95 (-0.05 to 1.96, p=0.063)

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.

**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

---

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.

```
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)
```

On the principle of parsimony, we may want to remove variables 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.

**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)

**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

---

```
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)	-

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.

**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

---

An important message here 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 models<sup>7</sup>. This great post demonstrates beautifully how the statisti-

---

<sup>7</sup><https://lindeloev.github.io/tests-as-linear>

cal 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.

# 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, 1, 1, 1, 1, 1, 1...
## $ sex        <dbl> 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 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, 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.

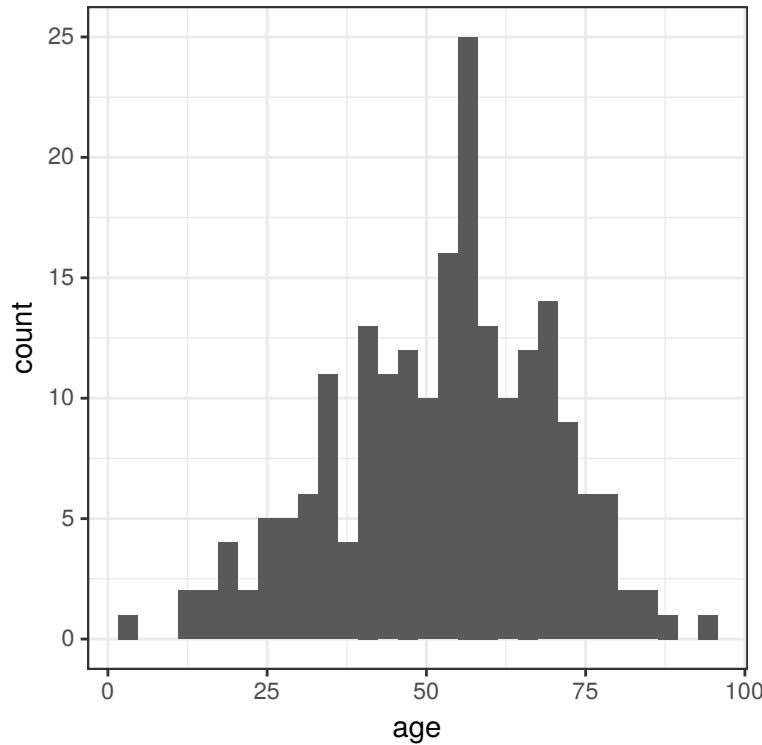
So the answer. Do not do it unless you have to. Plot and understand the continuous variable first. If you do it, try not to throw away too much information.

```
# 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`.`



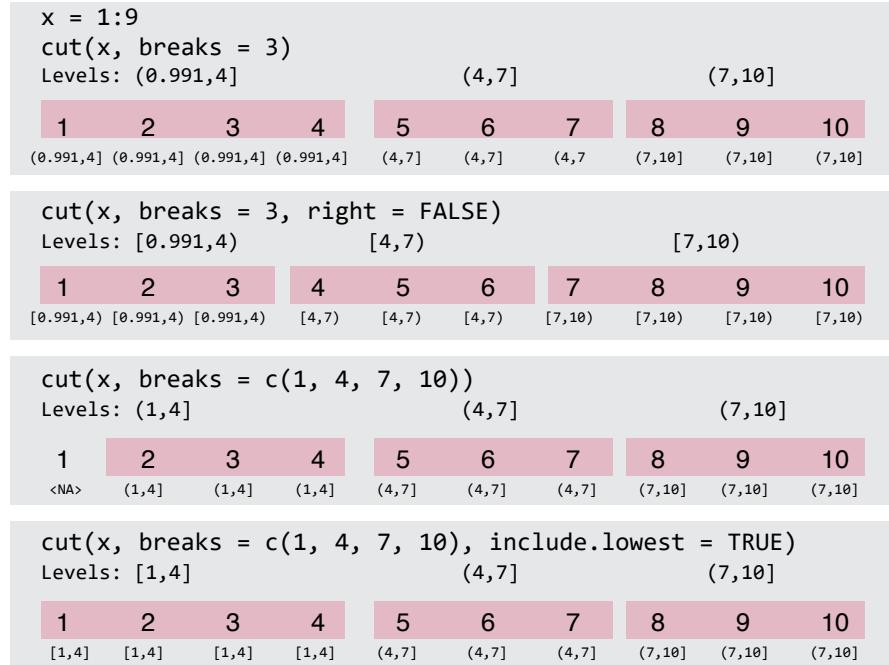
There are different ways in which a continuous variable can be converted to a factor. You may wish to create a number of intervals of equal length. The `cut()` function can be used for this.

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.



**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
```

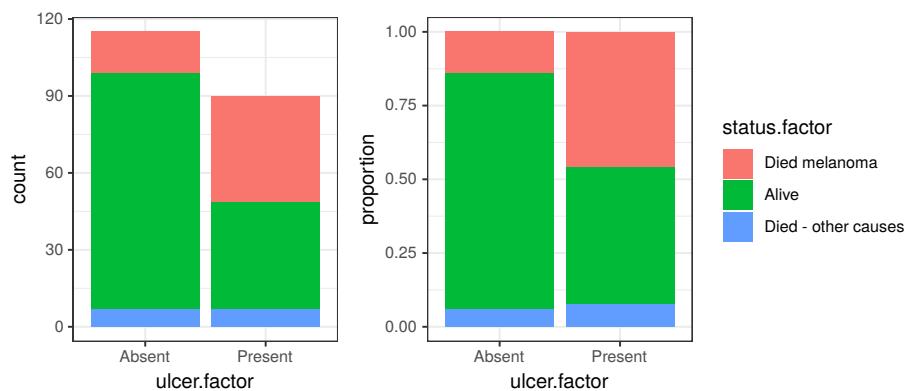
## 8.7 Plot the data

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
```



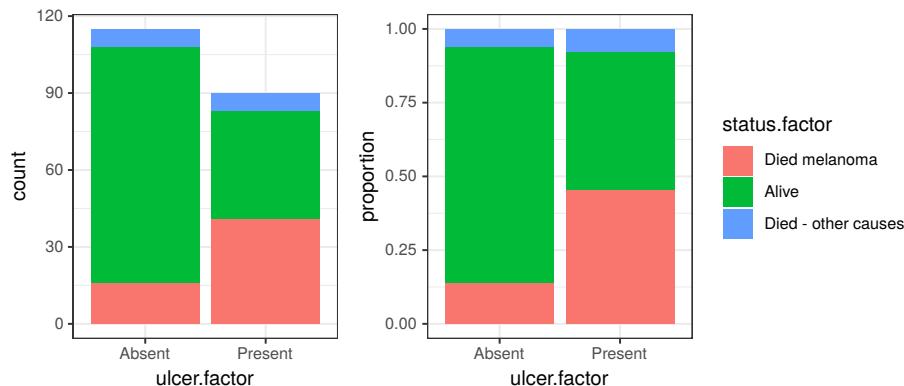
**FIGURE 8.2:** Bar chart: outcome after surgery for patients with ulcerated melanoma.

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")

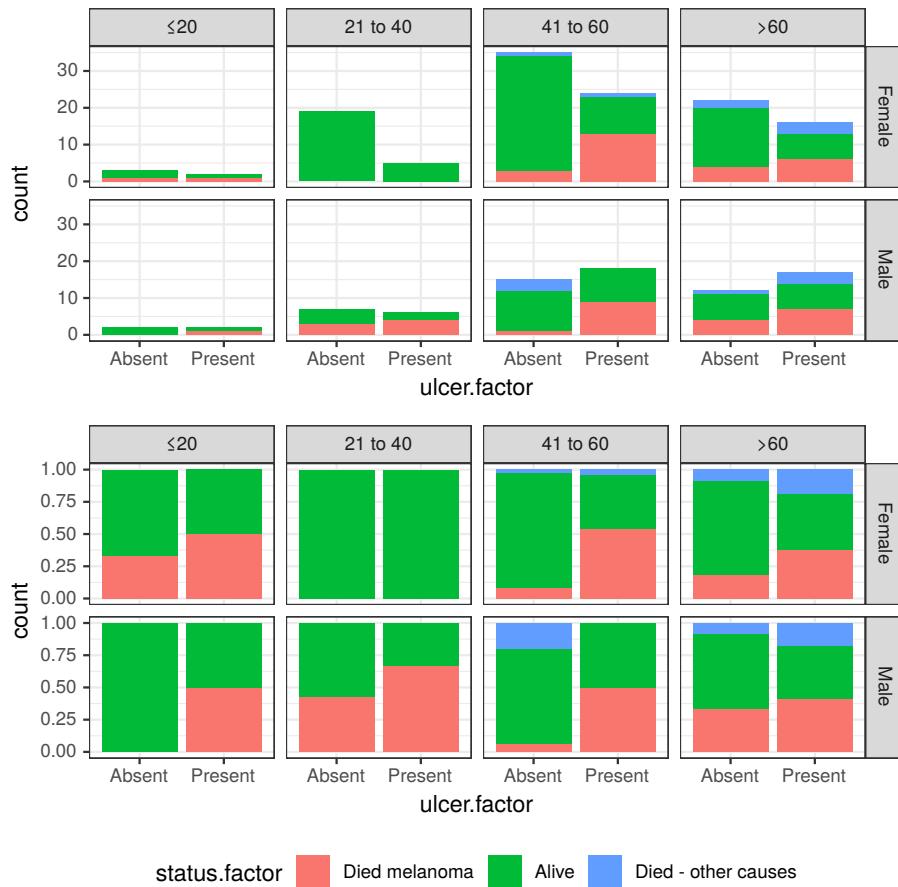
p1 / p2
```

---

## 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"))
  )
```



**FIGURE 8.4:** Facetted bar plot: outcome after surgery for patients with ulcerated melanoma aggregated by sex and age.

## 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: CAPTION**

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:** CAPTION

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)
Sex	>60	46 (68.7)	21 (31.3)
	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 ( $\chi^2$ ) 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 categorise variables each with two levels, such as is show 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 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 par-

ticularly 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
```



**TABLE 8.4:** CAPTION

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)		
	>60	46 (68.7)	21 (31.3)		
Sex	Female	98 (77.8)	28 (22.2)	0.024	
	Male	50 (63.3)	29 (36.7)		
thickness	Mean (SD)	2.4 (2.5)	4.3 (3.6)	<0.001	

```
p = TRUE,
catTest = catTestfisher)
```

```
mydata %>%
  summary_factorlist(dependent = "status_dss",
                     explanatory =
                       c("ulcer.factor", "age.factor",
                         "sex.factor", "thickness"),
                     p = TRUE,
                     catTest = catTestfisher) %>%
  mykable()
```

**TABLE 8.5:** CAPTION

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	

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
    add_dependent_label = TRUE
  )
```

**TABLE 8.6:** CAPTION

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*

---

---

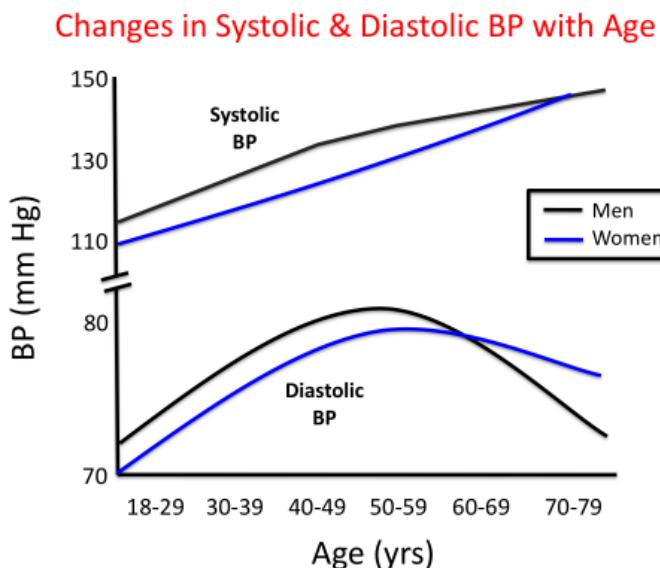
All generalizations are false, including this one.  
Mark Twain

---

---

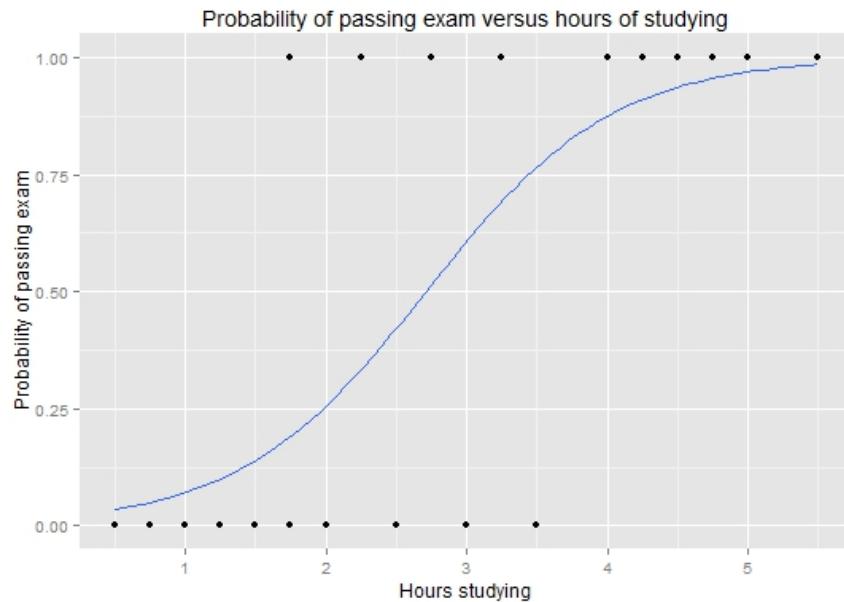
### 9.1 What is Logistic Regression?

As we have seen in previously, regression analysis is a statistical process for estimating the relationships between variables. For instance, we may try to predict the blood pressure of a group of patients based on their age. As age and blood pressure are on a continuous scale, this is an example of linear regression.



Adapted from: JNC7 & Burt et al (1995) Hypertension 23:305-313

Logistic regression is an extension of this, where the variable being predicted is *categorical*. We will deal with binary logistic regression, where the variable being predicted has two levels, e.g. yes or no, 0 or 1. In healthcare, this is usually done for an event (like death) occurring or not occurring. Logistic regression can tell us the probability of the outcome occurring.



Logistic regression lets you adjust for the effects of confounding factors on an outcome. When you read a paper that says it has adjusted for confounding factors, this is the usual method which is used.

Adjusting for confounding factors allows us to isolate the true effect of a variable upon an outcome. For example, if we wanted to know the effects of smoking on deaths from heart attacks, we would need to also control for things like sex and diabetes, as we know they contribute towards heart attacks too.

Although in binary logistic regression the outcome must have two levels, the predictor variables (also known as the explanatory variables) can be either continuous or categorical.

Logistic regression can be performed to examine the influence of one predictor variable, which is known as a univariable analysis. Or multiple predictor variables, known as a multivariable analysis.

## 9.2 Definitions

**Dependent** variable (in clinical research usually synonymous to **outcome**) - is what we are trying to explain, i.e. we are trying to identify the factors associated with a particular outcome. In binomial logistic regression, the dependent variable has exactly two levels (e.g. “Died” or “Alive”, “Yes - Complications” or “No Complications”, “Cured” or “Not Cured”, etc.).

**Explanatory** variables (also known as **predictors**, **confounding** variables, or “**adjusted for**”) - patient-level information, usually including demographics (age, gender) as well as clinical information (disease stage, tumour type). Explanatory variables can be categorical as well as continuous, and categorical variables can have more than two levels.

**Univariable** - analysis with only one Explanatory variable.

**Multivariable** - analysis with more than one Explanatory variable. Synonymous to “adjusted”.

(**Multivariate** - technically means more than one **Dependent variable** (we will not discuss this type of analysis), but very often used interchangeably with **Multivariable**.)

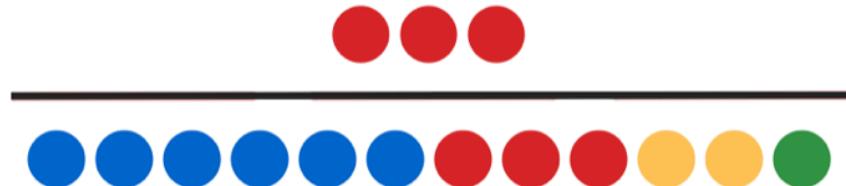
---

## 9.3 Odds and probabilities

Odds and probabilities can get confusing so let's get them straight:

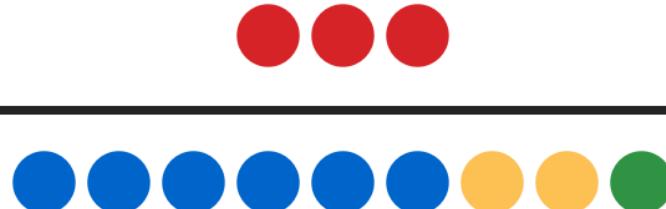
Probability of Red

$$3/12 = 1/4$$



Odds For Red

$$3/9 = 1/3$$



Odds and probabilities can be interconverted. For example, if the odds of a patient dying from a disease are 9 to 1 then the probability of death (also known as risk) is 10%. Odds of 1 to 1 equal 50%.

$Odds = \frac{p}{1-p}$ , where  $p$  is the probability of the outcome occurring (or the circle being red).

Look at the numbers and convince yourself that this works.

### 9.3.1 Odds ratios

For a given categorical explanatory variable (e.g. gender), the likelihood of an outcome/dependent occurring (e.g cancer) can be expressed in a ratio of odds or odds ratio, e.g. the odds of men developing cancer is 2-times that of females, odds ratio = 2.0.

	<b>a</b>	<b>b</b>
Cancer: Yes		
	<b>c</b>	<b>d</b>
Cancer: No		
	Sex: Male	Sex: Female

Odds cancer | Male  
 $= a / c$

Odds cancer | Female  
 $= b/d$

Odds of cancer  
 male vs. female

$$\frac{a/c}{b/d}$$

←  
Odds ratio

An alternative is a ratio of probabilities, called a risk ratio or relative risk. Odds ratios have useful mathematical characteristics and are the main expression of results in logistic regression analysis.

## 9.4 Melanoma dataset

Malignant melanoma is a cancer of the skin. It is aggressive and highly invasive, making it difficult to treat.

It's classically divided into 4 stages of severity, based upon the depth of the tumour:

- Stage I: <0.5 mm depth
- Stage II: 0.5 to 1.0 mm depth
- Stage III: 1.0 to 4.0 mm depth
- Stage IV: > 4.0 mm depth

This will be important in our analysis as we will creating a new variable based upon this.

Using logistic regression, we will investigate factors associated with death from malignant melanoma.

### 9.4.1 Doing logistic regression in R

There are a few different ways of doing logistic regression in R. The `glm()` function is probably the most common and most flexible one to use. (`glm` stands for `generalised linear model`.)

Within the `glm()` function there are several `options` in the function we must define to make R run a logistic regression.

`data` - you must define the dataframe to be used in the regression.

`family` - this tells R to treat the analysis as a logistic regression. For our purposes, `family` will always be "`binomial`" (as binary data follow this distribution).

`x ~ a + b + c` - this is the formula for the logistic regression, with `x` being the outcome and `a`, `b` and `c` being predictor variables.

Note the outcome is separated from the rest of the formula and

sits on the left hand side of a  $\sim$ . The confounding variables are on the right side, separated by a  $+$  sign.

The final `glm()` function takes the following form:

```
glm(x ~ a + b + c + d, data = data, family = "binomial")
```

## 9.5 Setting up your data

The most important step to ensure a good basis to start from is to ensure your variables are well structured and your outcome variable has exactly two outcomes.

We will need to make sure our outcome variables and predictor variables (the ones we want to adjust for) are suitably prepared.

In this example, the outcome variable called `status.factor` describes whether patients died or not and will be our (dependent) variable of interest.

### 9.5.1 Worked Example

```
library(tidyverse)
load("melanoma_factored.rda")
#Load in data from the previous session
```

Here `status.factor` has three levels: `Died`, `Died - other causes` and `Alive`. This is not useful for us, as logistic regression requires outcomes to be binary (exactly two levels).

We want to find out which variables predict death from melanoma. So we should create a new factor variable, `died_melanoma.factor`. This will have two outcomes, `yes` (did die from melanoma) or `no` (did not die from melanoma).

```
mydata$status.factor %>%
  fct_collapse("Yes" = c("Died"),
              "No"   = c("Alive", "Died - other causes")) ->
  mydata$died_melanoma.factor

mydata$died_melanoma.factor %>% levels()

## [1] "No"  "Yes"
```

---

## 9.6 Creating categories

Now that we have set up our outcome variable, we should ensure our predictor variables are prepared too.

Remember the stages of melanoma? This is an important predictor of melanoma Mortality based upon the scientific literature.

We should take this into account in our model.

### 9.6.1 Exercise

Create a new variable called `stage.factor` to encompass the stages of melanoma based upon the thickness. In this data, the `thickness` variable is measured in millimetres too.

```
#the cut() function makes a continuous variable into a categorical variable
mydata$thickness %>%
  cut(breaks = c(0,0.5,1,4, max(mydata$thickness, na.rm=T)),
       include.lowest = T) ->
  mydata$stage.factor

mydata$stage.factor %>% levels()

## [1] "[0,0.5]"  "(0.5,1]"  "(1,4]"    "(4,17.4]"

mydata$stage.factor %>%
  fct_recode("Stage I"   = "[0,0.5]",
```

```

    "Stage II" = "(0.5,1]",
    "Stage III" = "(1,4]",
    "Stage IV" = "(4,17.4]"
) -> mydata$stage.factor

mydata$stage.factor %>% levels()

## [1] "Stage I"   "Stage II"  "Stage III" "Stage IV"

```

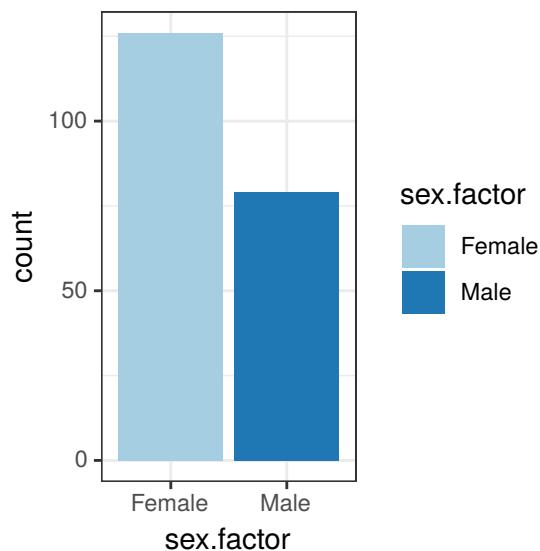
### 9.6.2 Always plot your data first!

```

source("1_source_theme.R")

mydata %>%
  ggplot(aes(x = sex.factor)) +
  geom_bar(aes(fill = sex.factor))

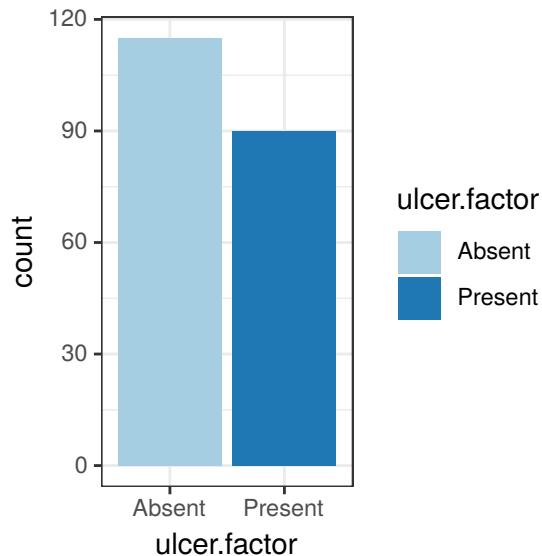
```



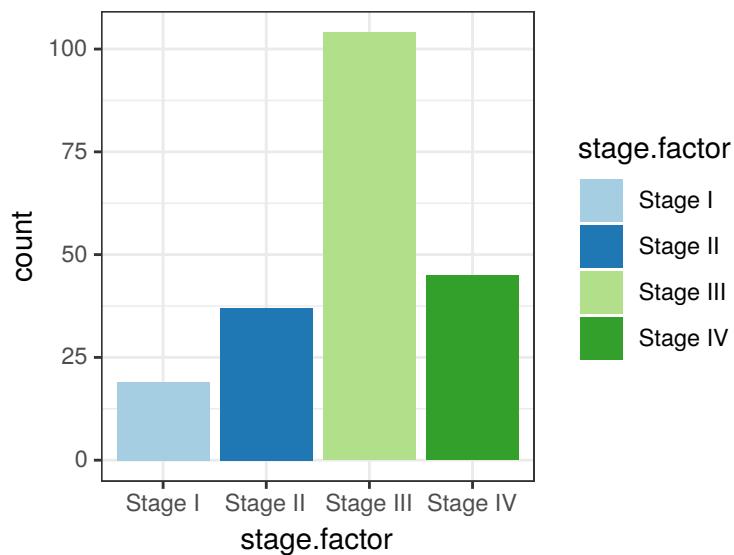
```

mydata %>%
  ggplot(aes(x = ulcer.factor)) +
  geom_bar(aes(fill = ulcer.factor))

```

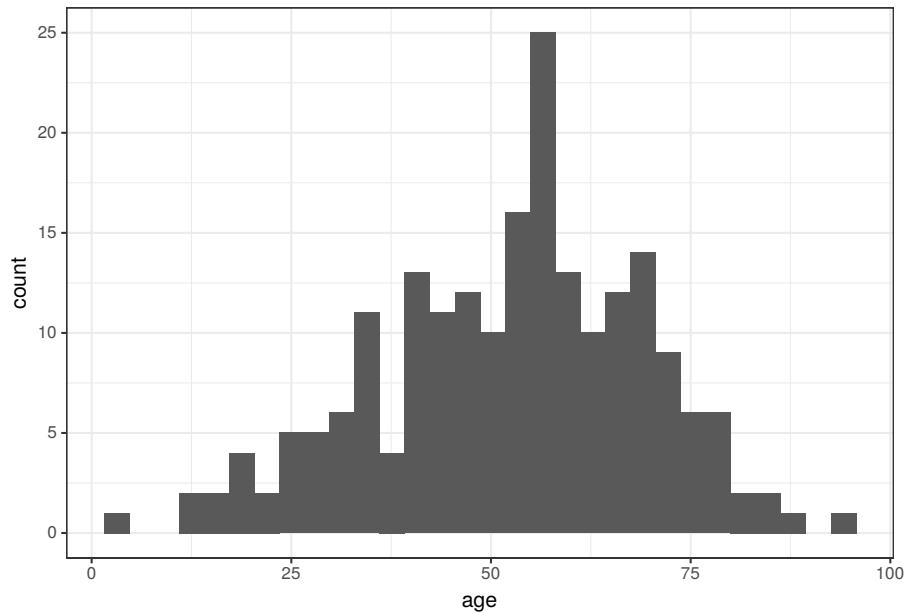


```
mydata %>%
  ggplot(aes(x = stage.factor)) +
  geom_bar(aes(fill = stage.factor))
```



```
mydata %>%
  ggplot(aes(x = age)) +
  geom_histogram(aes(fill = age))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Now we are ready for some modelling!

## 9.7 Basic: One explanatory variable (predictor)

Lets find out what the influence of each predictor/confounding variable is on mortality from melanoma, which may help inform a more complicated regression, with multiple predictors/confounders.

We'll start with whether the patient was male or female:

### 9.7.1 Worked example

First we need to create a regression model using `glm()`. We will then summarise it using `summary()`

Note, we need to use the `family` option. Specifying '`binomial`' in `family` tells `glm()` to switch to logistic regression.

```
#Create a model

glm(died_melanoma.factor ~ sex.factor, data = mydata, family = "binomial")

## 
## Call: glm(formula = died_melanoma.factor ~ sex.factor, family = "binomial",
##          data = mydata)
##
## Coefficients:
## (Intercept)  sex.factorMale
##           -1.253          0.708
##
## Degrees of Freedom: 204 Total (i.e. Null);  203 Residual
## Null Deviance:      242.4
## Residual Deviance: 237.4      AIC: 241.4

model1 = glm(died_melanoma.factor ~ sex.factor, data = mydata, family = "binomial")

summary(model1)

## 
## Call:
## glm(formula = died_melanoma.factor ~ sex.factor, family = "binomial",
##      data = mydata)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -0.9565 -0.7090 -0.7090  1.4157  1.7344
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.2528    0.2143 -5.846 5.03e-09 ***
## sex.factorMale 0.7080    0.3169  2.235  0.0254 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 242.35  on 204  degrees of freedom
## Residual deviance: 237.35  on 203  degrees of freedom
## AIC: 241.35
##
## Number of Fisher Scoring iterations: 4
```

Now we have created the model - fantastic!

But this doesn't mean a lot to humans reading a paper - or us in fact.

The estimate output of `summary(model_1)` represents the logarithm of the odds ratio. The odds ratio would be a lot easier to understand.

Therefore, to sort that out we should exponentiate the output of the model! The `exp()` function will do this.

```
exp(model1$coefficients)
```

```
##      (Intercept) sex.factorMale
##      0.2857143     2.0300000
```

This gives us an odds ratio of 2.03 for males. That is to say, males are twice as likely to die from melanoma than females.

Now a confidence interval might be handy. As this will be the logarithm of the confidence interval, we should exponentiate it to make it understandable.

```
exp(confint(model1))
```

```
## Waiting for profiling to be done...
##                  2.5 %    97.5 %
## (Intercept)  0.1843592 0.4284939
## sex.factorMale 1.0914854 3.7938450
```

The 2.5% is the lower bound and the 97.5% is the upper bound of the 95% confidence interval.

So we can therefore say that being male doubles your chances of dying from melanoma with an Odds Ratio of 2.03 (95% confidence interval of 1.09 to 3.79)

### 9.7.2 Exercise

Repeat this for all the variables contained within the data, particularly:

`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!

---

## 9.8 Finalfit package

We have developed our `finalfit` package to help with advanced regression modelling. We will introduce it here, but not go into detail.

See [www.finalfit.org](http://www.finalfit.org) for more information and updates.

---

## 9.9 Summarise a list of variables by another variable

We can use the `finalfit` package to summarise a list of variables by another variable. This is very useful for “Table 1” in many studies.

```
library(finalfit)
dependent = "died_melanoma.factor"
explanatory = c("age", "sex.factor")

table_result = mydata %>%
  summary_factorlist(dependent, explanatory, p = TRUE)
```

label	levels	No	Yes	p
age	Mean (SD)	51.5 (16.1)	55.1 (17.9)	0.189
sex.factor	Female	98 (77.8)	28 (22.2)	0.024
	Male	50 (63.3)	29 (36.7)	

### 9.10 `finalfit` function for logistic regression

We can then use the `finalfit` function to run a logistic regression analysis with similar syntax.

```
dependent = "died_melanoma.factor"
explanatory = c("sex.factor")

model2 = mydata %>%
  finalfit(dependent, explanatory)
```

Dependent: died_melanoma.factor		No	Yes	OR (univariable)
sex.factor	Female	98 (66.2)	28 (49.1)	
	Male	50 (33.8)	29 (50.9)	2.03 (1.09-3.79, p=0.02)

## 9.11 Adjusting for multiple variables in R

Your first models only included one variable. It's time to scale them up.

Multivariable models take multiple variables and estimates how each variable predicts an event. It adjusts for the effects of each one, so you end up with a model that calculates the adjusted effect estimate (i.e. the odds ratio), upon an outcome.

When you see the term ‘adjusted’ in scientific papers, this is what it means.

### 9.11.1 Worked Example

Lets adjust for `age` (as a continuous variable), `sex.factor` and `stage.factor`. Then output them as odds ratios.

```
dependent = "died_melanoma.factor"
explanatory = c("age", "sex.factor", "stage.factor")

model3 = mydata %>%
  finalfit(dependent, explanatory)
```

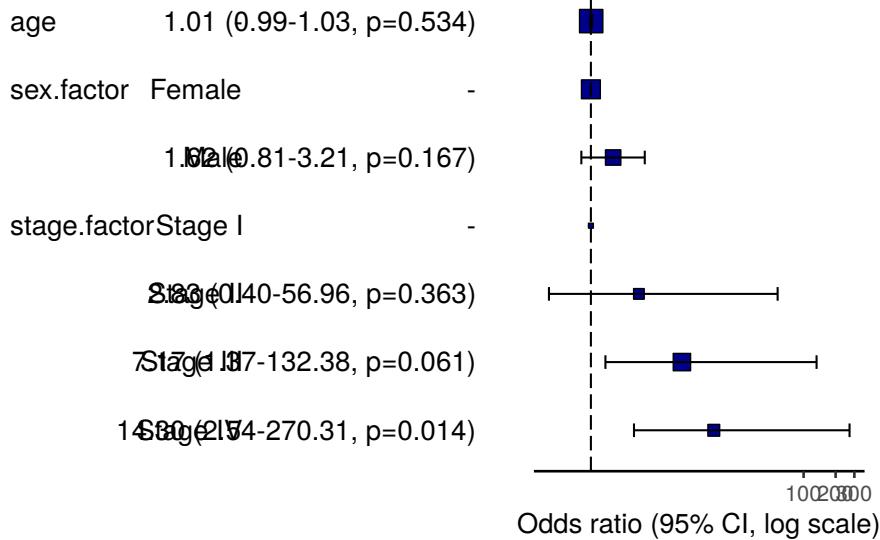
Dependent: died_melanoma.factor		No	Yes	OR
age	Mean (SD)	51.5 (16.1)	55.1 (17.9)	1.01 (0.99-1)
sex.factor	Female	98 (66.2)	28 (49.1)	
	Male	50 (33.8)	29 (50.9)	2.03 (1.09-3)
stage.factor	Stage I	18 (12.2)	1 (1.8)	
	Stage II	32 (21.6)	5 (8.8)	2.81 (0.41-56)
	Stage III	75 (50.7)	29 (50.9)	6.96 (1.34-128)
	Stage IV	23 (15.5)	22 (38.6)	17.22 (3.13-322)

```
or_plot(mydata, dependent, explanatory)
```

```
## Waiting for profiling to be done...
## Waiting for profiling to be done...
## Waiting for profiling to be done...
```

```
## Warning: Removed 2 rows containing missing values (geom_errorbarh).
```

### died\_melanoma.factor: OR (95% CI, p-value)



When we enter age into regression models, the effect estimate is provided in terms of per unit increase. So in this case it's expressed in terms of an odds ratio per year increase (i.e. for every year in age gained odds of death increases by 1.02).

#### 9.11.2 Exercise

Create a regression that includes ulcer.factor.

## 9.12 Advanced: Fitting the best model

Now we have our preliminary model. We could leave it there.

However, when you publish research, you are often asked to supply a measure of how well the model fitted the data.

There are different approaches to model fitting. Come to our course HealthyR-Advanced: Practical Logistic Regression. At this we describe use of the Akaike Information Criterion (AIC) and the C-statistic.

The C-statistic describes discrimination and anything over 0.60 is considered good. The closer to 1.00 the C-statistic is, the better the fit.

The AIC measure model fit with lower values indicating better fit.

These metrics are available here:

```
mydata %>%
  finalfit(dependent, explanatory, metrics=TRUE)

## Waiting for profiling to be done...

## Setting levels: control = 0, case = 1
## Setting direction: controls < cases

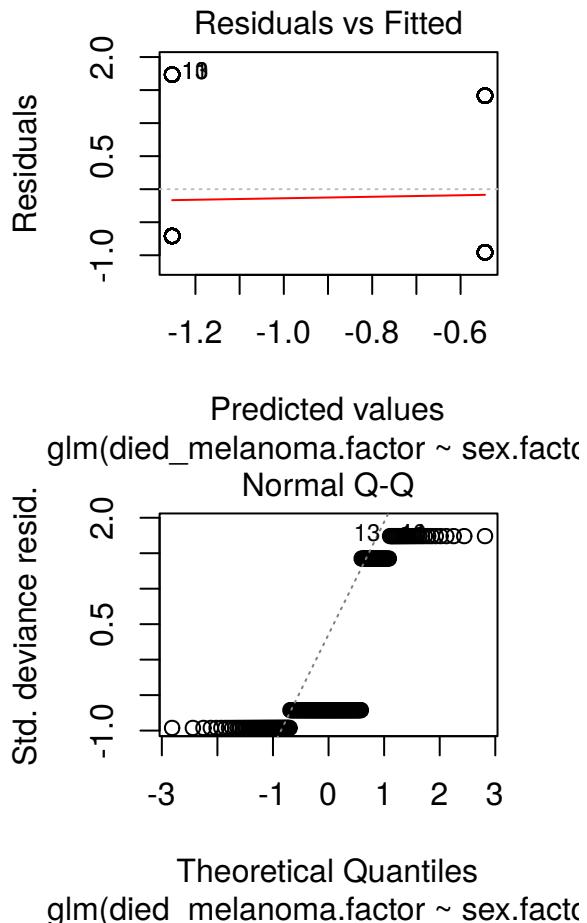
## [[1]]
##   Dependent: died_melanoma.factor           No      Yes
## 1                               age Mean (SD) 51.5 (16.1) 55.1 (17.9)
## 2       sex.factor    Female    98 (66.2)   28 (49.1)
## 3                           Male    50 (33.8)   29 (50.9)
## 4       stage.factor   Stage I    18 (12.2)    1 (1.8)
## 5                           Stage II   32 (21.6)    5 (8.8)
## 6                           Stage III  75 (50.7)   29 (50.9)
## 7                           Stage IV  23 (15.5)   22 (38.6)
##   OR (univariable)          OR (multivariable)
## 1    1.01 (0.99-1.03, p=0.163)    1.01 (0.99-1.03, p=0.534)
## 2                            -                      -
## 3    2.03 (1.09-3.79, p=0.025)    1.62 (0.81-3.21, p=0.167)
```

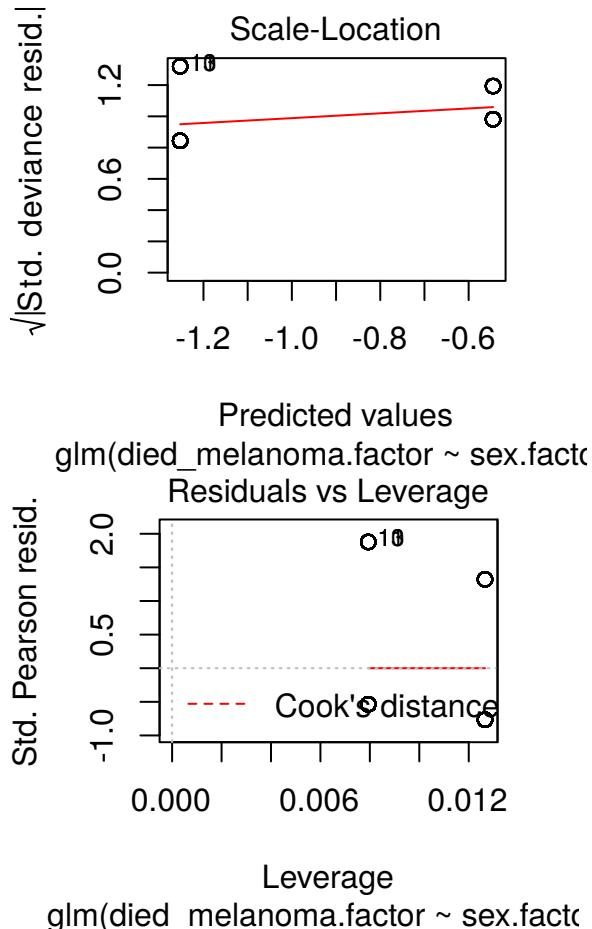
```
## 4
## 5 2.81 (0.41-56.12, p=0.362) 2.83 (0.40-56.96, p=0.363)
## 6 6.96 (1.34-128.04, p=0.065) 7.17 (1.37-132.38, p=0.061)
## 7 17.22 (3.13-322.85, p=0.008) 14.30 (2.54-270.31, p=0.014)
##
## [[2]]
## [1] "Number in dataframe = 205, Number in model = 205, Missing = 0, AIC = 232.3, C-statistic = 0.708, H&L"
```

### 9.12.1 Extra material: Diagnostics plots

While outwith the objectives of this course, diagnostic plots for `glm` models can be produced by:

```
plot(model1)
```





# **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 = ifelse(status == 2, 0, # "still alive"
                       1), # "died of melanoma" or "died of other causes"

    # Disease-specific survival
    status_dss = ifelse(status == 2, 0, # "still alive"
                        ifelse(status == 1, 1, # "died of melanoma"
                               0)), # "died of other causes is censored"

    # Competing risks regression
    status_crr = ifelse(status == 2, 0, # "still alive"
                        ifelse(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%
```

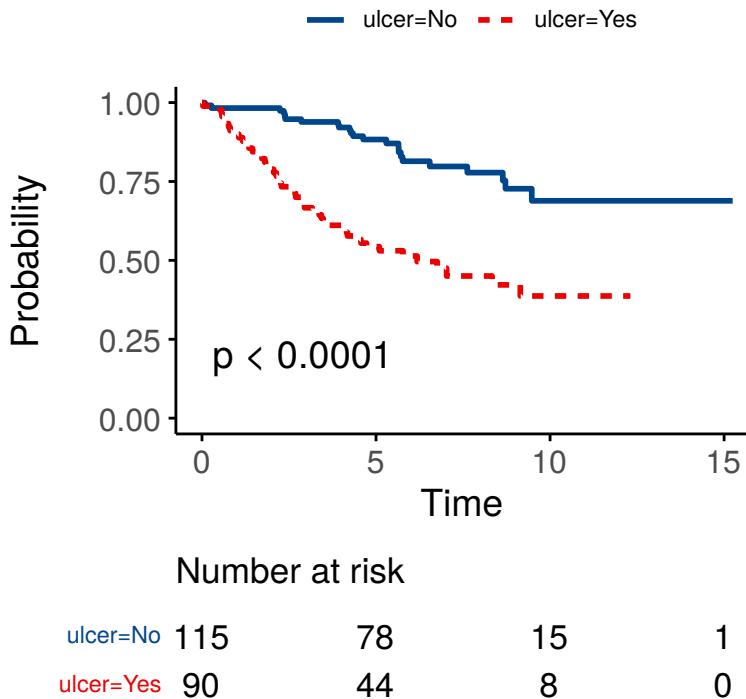
## 10.7 Kaplan Meier plot

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()
```

**TABLE 10.1:** CAPTION

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:** CAPTION

Overall survival		HR (univariable)	HR (multivariable)
Age (years)	Mean (SD)	52.5 (16.7)	1.03 (1.01-1.05, p<0.001)
Sex	Female	126 (61.5)	1.02 (1.01-1.04, p=0.005)
	Male	79 (38.5)	1.93 (1.21-3.07, p=0.006)
Tumour thickness (mm)	Mean (SD)	2.9 (3.0)	1.16 (1.10-1.23, p<0.001)
Ulcerated tumour	No	115 (56.1)	1.10 (1.03-1.18, p=0.004)
	Yes	90 (43.9)	3.52 (2.14-5.80, 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:** CAPTION

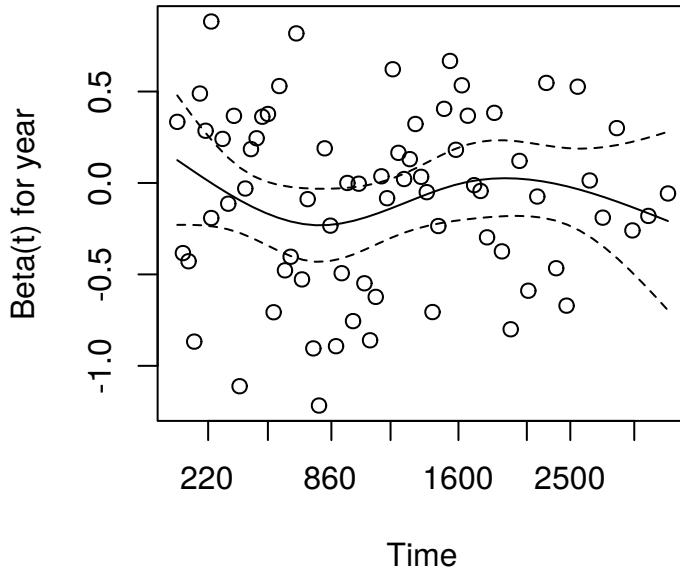
		all	HR (univariable)	HR (multivariable)	HR (multivariable reduced)
Age (years)	Mean (SD)	52.5 (16.7)	1.03 (1.01-1.05, p<0.001)	1.02 (1.01-1.04, p=0.005)	1.02 (1.01-1.04, p=0.003)
Sex	Female	126 (61.5)	-	-	-
	Male	79 (38.5)	1.93 (1.21-3.07, p=0.006)	1.51 (0.94-2.42, p=0.085)	-
Tumour thickness (mm)	Mean (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: CAPTION**

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()

##   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)

```

**TABLE 10.5:** CAPTION

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 `frailty()` method here is being superseded by the `coxme` package, and we look forward to incorporating this in the future.

### 10.8.6 Hazard ratio plot

A plot of any of the above models can be easily produced.

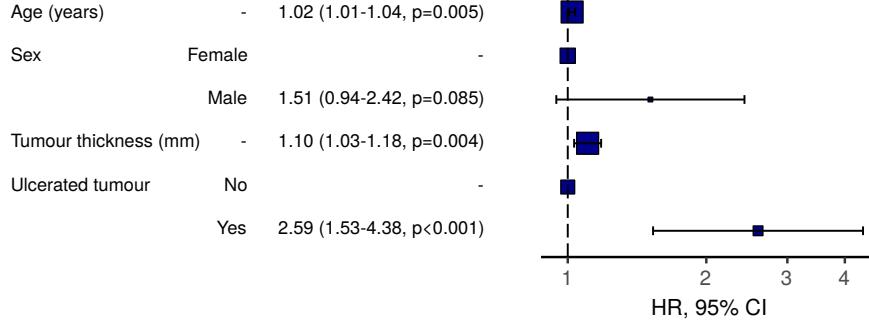
```

melanoma %>%
  hr_plot(dependent_os, explanatory)

## Dependent variable is a survival object
## Warning: Removed 2 rows containing missing values (geom_errorbarh).

```

### Survival: HR (95% CI, p-value)



## 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 `crr()`. 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:** CAPTION

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.

---

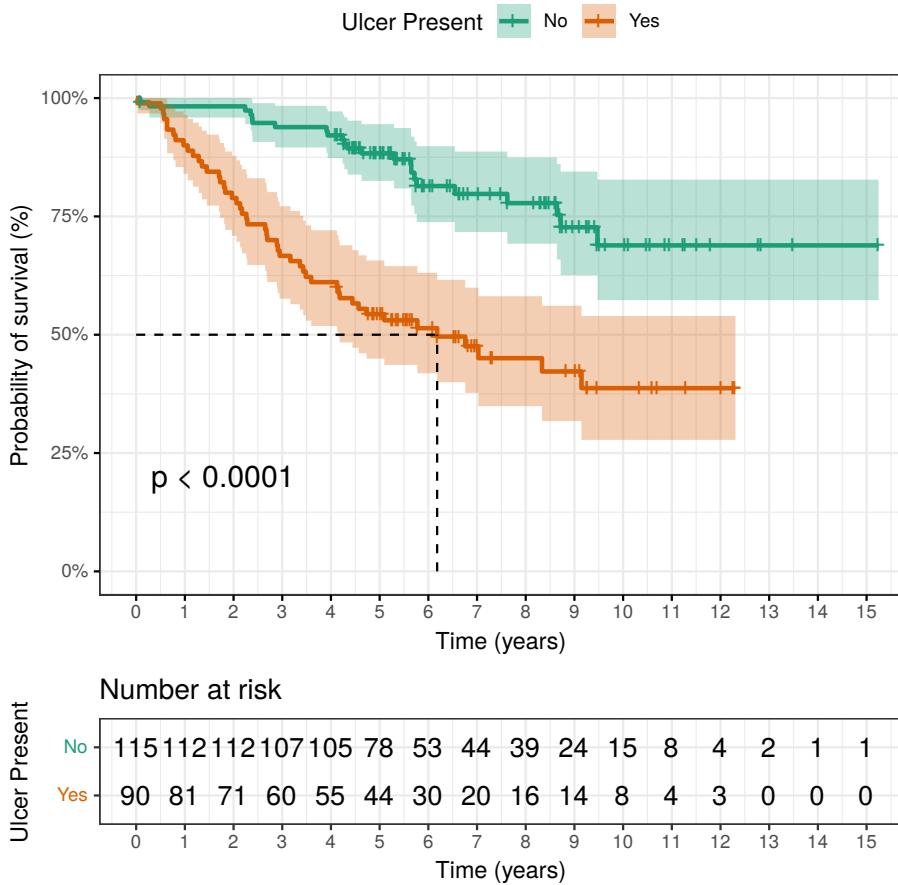
## 10.11 Exercise

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: ggpublisher
```

```
## 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 in-

dependent 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
```

Now we will 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 as
```

---

## 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**

---

---



# **11**

---

*Notebooks and markdown*



# **12**

---

## *Missing data*

---

### **12.1 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.

## 12.2 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.

### 12.2.1 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.

### 12.2.2 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!

### 12.2.3 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.

---

### 12.3 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.
- 

### 12.4 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_mcar = 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_mcar", "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         1.9  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.

## 12.5 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.

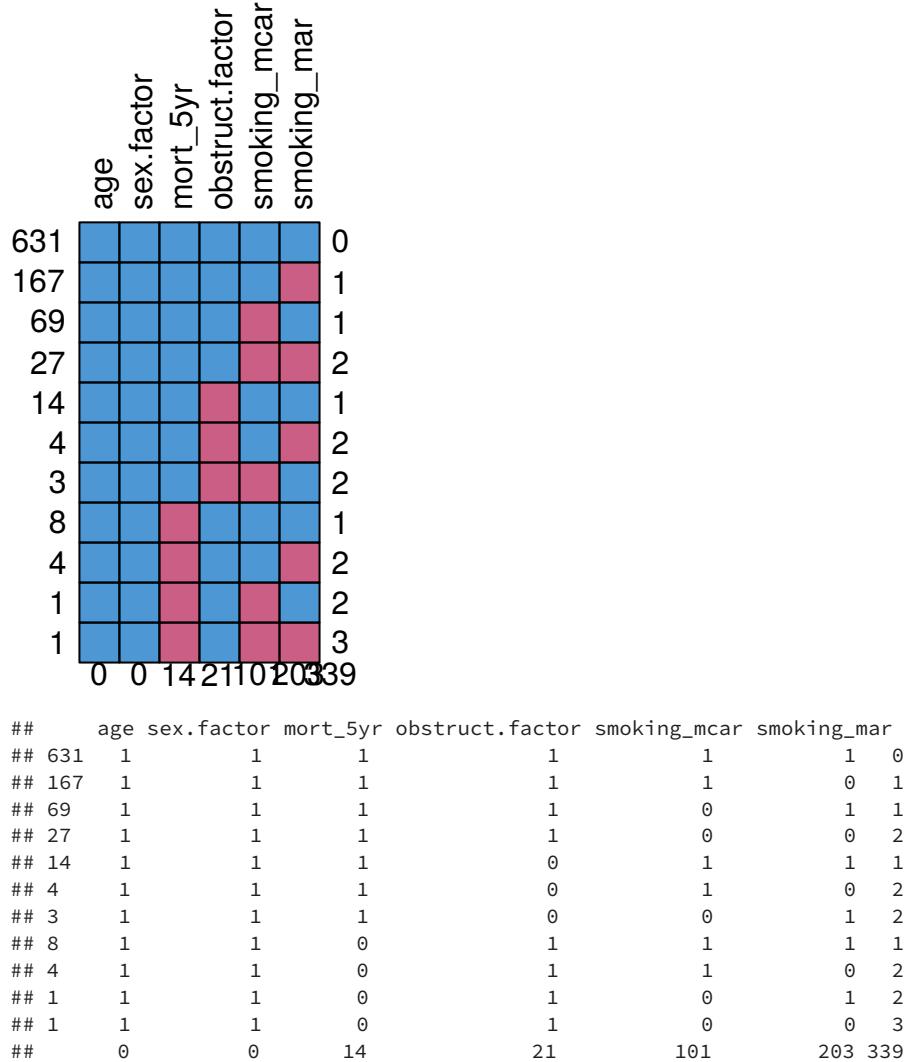
### 12.6 3. Look for patterns of missingness: `missing_pattern`

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.

### 12.6.1 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)
```

---

### 12.7 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 12.1:** CAPTION

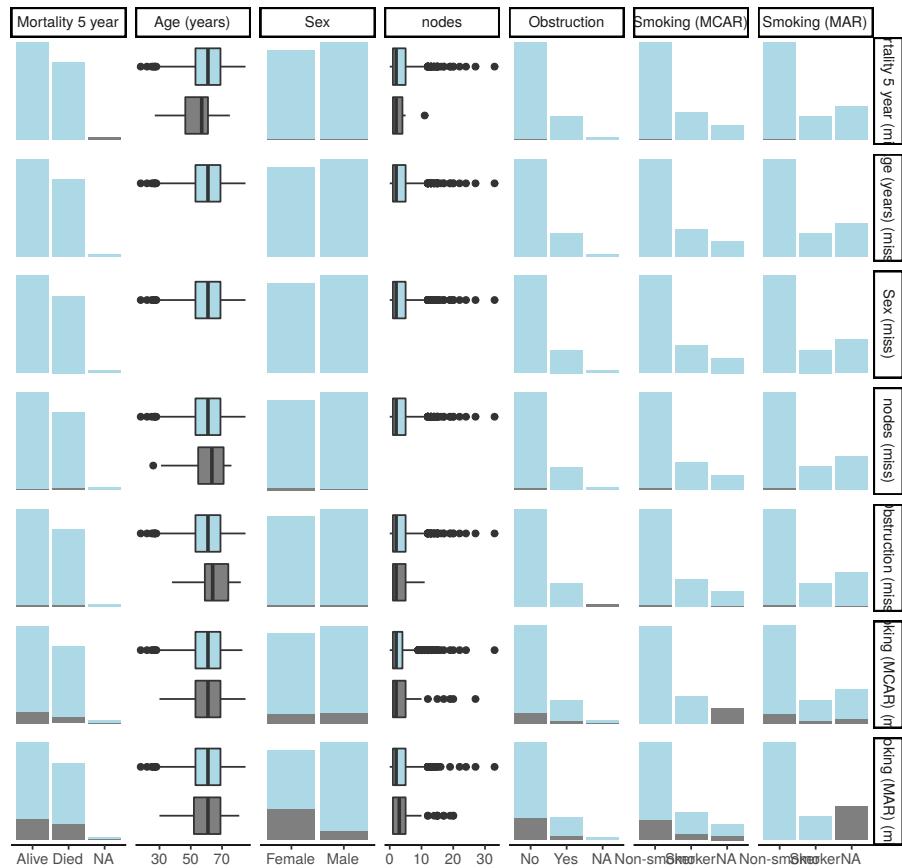
| label          | levels     | No          | Yes         | p     |
|----------------|------------|-------------|-------------|-------|
| Age (years)    | Mean (SD)  | 60.2 (11.5) | 57.3 (13.3) | 0.014 |
| Sex            | Female     | 346 (79.2)  | 91 (20.8)   | 0.290 |
|                | Male       | 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)   |       |
|                | Missing    | 78 (79.6)   | 20 (20.4)   |       |
| Smoking (MAR)  | 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)
```

Missing data matrix



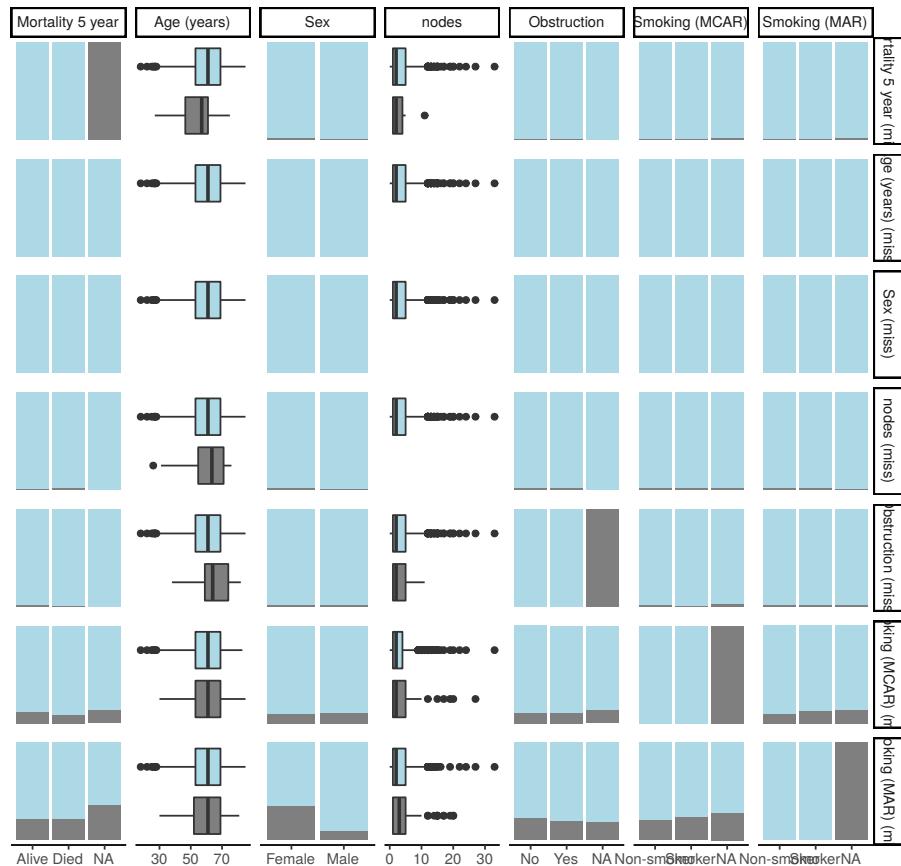
For continuous variables (age and nodes), the distributions of observed and missing data can be visually compared. Is there a difference between age and mortality above?

For discrete, data, counts are presented by default. It is often easier to compare proportions:

```
colon_s %>%
  missing_pairs(dependent, explanatory, position = "fill")
```

12.7 4. Check for associations between missing and observed data: `missing_pairs` / `missing_compare` 267

Missing data matrix



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()`.

```
explanatory = c("age", "sex.factor",
  "nodes", "obstruct.factor")
dependent = "smoking_mcar"

missing_mcar = colon_s %>%
  missing_compare(dependent, explanatory)
```

**TABLE 12.2:** CAPTION

| 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)  |       |

```
dependent = "smoking_mar"
missing_mar = colon_s %>%
  missing_compare(dependent, explanatory)
```

**TABLE 12.3:** CAPTION

| 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).

## 12.8 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.
```

---

## 12.9 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
- 

## 12.10 MCAR, MAR, or MNAR

### 12.10.1 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 not 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)
```

### Other considerations

- Sensitivity analysis
- Omit the variable
- Imputation

**TABLE 12.4:** CAPTION

| 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) |

- 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.

### 12.10.2 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\_comparishows 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 impu-

tation. 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 %>%
    glmmulti(dependent, explanatory) %>%
    fit2df(estimate_suffix = " (multivariable inc. smoking)") -> fit_multi

```

```

explanatory = c("age", "sex.factor",
  "nodes", "obstruct.factor")
colon_s %>%
  glmulti(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 12.5:** CAPTION

| 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=0.889) | 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         | 408 (56.7)  | 312 (43.3)  | -                         | -                         | -                               | -                         |
|               | 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)  | 296 (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 12.6:** CAPTION

| 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) |

### 12.10.3 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.



# **13**

---

## 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.

---

### **13.1 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?\*

---

### 13.2 `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.

---

### 13.3 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")
```

## 13.4 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
```

## 13.5 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'
```

### 13.6 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...
```

## 13.7 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
```

## 13.8 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
```

## 13.9 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'
```

---

### 13.10 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.

---

### 13.11 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")
```

---

### 13.12 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.

---

### 13.13 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.

# **14**

---

*Exporting tables and plots*



# **15**

---

## *RStudio settings, good practise*

---

### **15.1 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").

---

### **15.2 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**

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.7.15.



# **Index**

- analysis of variance (ANOVA), 137
- bookdown, xi
- continuous data**, 123
- functions
- aov, 138, 139
  - do, 136
  - ff\_glimpse, 126
  - filter, 126, 127, 129–133, 136, 137, 139, 143, 144, 146
  - gather, 143
  - glimpse, 126
  - group\_by, 136
  - head, 143
  - kruskal.test, 146
  - missing\_glimpse, 126
  - mutate, 134, 143, 146
  - pairwise.t.test, 139, 141
  - select, 134, 143
  - spread, 134
  - summarise, 134
  - summary, 138
  - summary\_factorlist, 146
  - t.test, 131, 135, 136
  - tidy, 132, 136, 139, 146
  - wilcox.test, 145
- knitr, xi
- non-parametric tests**, 142
- Mann-Whitney U, 144
  - Wilcoxon rank sum, 144
- pairwise testing, 139
- plotting
- facet\_grid, 126, 127, 129, 130, 143, 144
  - geom\_boxplot, 129, 130, 137, 144
  - geom\_histogram, 126, 143
  - geom\_jitter, 130, 144
  - geom\_line, 133
  - geom\_qq, 127, 144
  - geom\_qq\_line, 127, 144
  - ggtitle, 130
  - patchwork, 144
  - theme, 130, 144
  - xlab, 130
  - ylab, 130
- symbols**
- AND &, 27
  - assignment =, 18
  - comment #, 30
  - equal =, 27
  - greater or equal >=, 27
  - greater than >, 27
  - less or equal <=, 27
  - less than <, 27
  - not !=, 27
  - OR |, 27

pipe %>%, 20  
select column \$, 15

**t-test**, 130  
one-sample, 135  
paired, 133  
two-sample, 130  
tranformations, 142