

Cédric Scherer

Graphic Design with ggplot2

Create Beautiful Data Visualizations in R

To my son,
without whom I should have finished this book two years earlier

Contents

List of Figures	v
List of Tables	vii
Preface	ix
About the Author	xv
1 Introduction	1
1.1 Communicating Data	1
1.2 Coding Visualizations	2
1.3 Why R and ggplot2	3
2 Get Started	5
2.1 The Data	5
2.2 Working in R	7
2.2.1 Import data	7
2.2.2 Project-oriented workflows	8
2.2.3 Inspect data	9
2.2.4 Data types	9
2.2.5 Data wrangling	12
2.2.6 Quarto and Rmarkdown	15
3 The {ggplot2} Package	17
3.1 The Structure of a ggplot	17
3.1.1 Data	18
4 A walk-through example	19
4.1 Create a basic ggplot	19
4.2 Combine multiple layers	19
4.3 Create small multiples	22
4.4 Change the axis scaling	23
4.5 Use a custom color palette	24
4.6 Adjust labels	26
4.7 Apply a complete theme	27
4.8 Customize the theme	29
5 The Concepts of {ggplot2}	31
6 Quick Steps to Improve Your Graphic	41
7 Working with Axes	43

I How To Polish Your Graphics	39
8 Working with Text	57
8.1 Titles	57
Appendix	59
A More to Say	59
Bibliography	61
Index	63
.	63

List of Figures

2.1	The original and aggregated data sets in direct comparison: counts of bike shares registered by TfL over time with month encoded by colour. The left panel shows counts for every hour of the day, while in the right panel the hourly data was aggregated into two periods of the day (day and night).	5
2.2	Overview of the distribution of the boolean variables <code>is_workday</code> , <code>is_weekend</code> , and <code>is_holiday</code> (A), the categorical variable <code>weather_type</code> (B), and the continuous variables <code>count</code> , <code>temp</code> , <code>temp_feel</code> , <code>humidity</code> , and <code>wind_speed</code> (C) of the cleaned and aggregated bike sharing data set. In panel C, the correlation between the variables is shown as scatterplot encoded by <code>timeperiod</code> (upper triangle) and encoded by point density (lower triangle), highlighting the level of overlap of data points.	6
4.1	A basic scatter plot of feels-like temperature and reported TfL bike rents, created with the <code>{ggplot2}</code> package.	20
4.2	The same scatter plot, now with an additional GAM smoothing.	20
4.3	The points and smoothing lines colored by the time of the day.	21
4.4	We can map aesthetics and define properties for each layer individually.	22
4.5	With the <code>facet</code> functions, a visualization can quickly be split into small multiples.	23
4.6	<code>{ggplot2}</code> even allows to free the axis range—while ensuring equal axis spacing.	24
4.7	To adjust the formatting of axis labels, the respective axis needs to be addressed via the <code>scale_*</code> () functions.	25
4.8	We can overwrite the default labels with the <code>labs()</code> which also allows to add a title, subtitle, caption, and tag to the graphic.	27
4.9	By passing a vector of the same length as there are seasons to the <code>labels</code> argument of the <code>scale_color_*</code> () function, we can overwrite the labels used in the legend.	28
4.10	The graphic now comes with a new look and a non-default font used for all text labels.	28
4.11	After applying one of the complete themes, one can further style the theme to the personal needs.	30
5.1	A default scatter plot of temperature measured in Chicago created with the <code>ggplot2</code> package.	32
5.2	Again, the same data, now shown as a connected scatterplot as a combination of points and lines.	33
5.3	We can change the appearance of the geometry, here illustrated by turning the black dots in larger, red diamonds.	33
5.4	You can style each geometrical layer on its own. Each geometry also comes with a set of individual properties.	34

5.5	By applying aesthetic mapping to the <code>day_night</code> variable in the point layers, the two groups can be identified. Note that <code>ggplot2</code> automatically adds a legend to the plot.	35
5.6	Both, the points and the smoothing lines are encoded by daytime. Because of that, the smoothing is hard to see and best visible in the legend.	36
5.7	Overwriting the color encoding in the point (left) or the smoothing (right). Note that the grouping is removed as well which changes the behaviour of the smoothing geom.	36
5.8	We can change the appearance for all following plots generated with <code>ggplot2</code> by globally setting another theme via <code>theme_set()</code>	37

List of Tables

2.1 Overview of the 15 variables contained in the cleaned and aggregated bike sharing data set.	7
---	---

Preface

Back in 2016, I had to prepare my PhD introductory talk to inform about my plans for the next three years and to showcase my first preliminary results. I planned to create a visualization using small multiples to show various outcomes of the scenarios I ran with my simulation model. I was already using the R programming language for years and quickly came across the graphics library **ggplot2** which comes with the functionality to easily create small multiples. I never liked the syntax and style of base plots in R, so I immediately fell in love with the idea and implementation of **ggplot2**'s *Grammar of Graphics*. But because I was short on time, I plotted these figures by trial and error and with the help of lots of googling. The resource I came always back to was a blog entry called “Beautiful plotting in R: A **ggplot2** cheatsheet” by Zev Ross¹. After giving the talk which contained some decent plots thanks to the blog post, I decided to go through this tutorial step-by-step. I learned so much from it and directly started modifying the codes and adding additional code snippets, chart types, and resources.

Fast forward to 2019. I successfully finished my PhD and started participating in a weekly data visualization challenge called #TidyTuesday². Every week, a raw data set is shared with the aim to explore and visualize the data with **ggplot2**. Thanks to my experience with the **tidyverse** and especially **ggplot2** during my PhD and the open-source approach of the challenge that made it possible to learn from other participants, my visualizations quickly became more advanced and complex.

A few months later, I had built a portfolio of various charts and maps and decided to start working as an independent data visualization specialist. I am now using **ggplot2** every day: for my academic work, design requests, reproducible reports, educational purposes, and personal data visualization projects. What I especially love about my current job specification: It challenges and satisfies my creativity on different levels. Besides the creativity one can express in terms of chart choice and design, there is also creativity needed to come up with solutions and tricks to bring the most venturous ideas to life. At the same time, there is the gratification when your code works and *magically* translates code snippets to visuals.

The blog entry by Zev Ross was not updated since January 2016, so I decided to add more examples and tricks to my version, which was now hosted on my personal blog³. Step by step, my version became a unique tutorial that now contains for example also the fantastic **patchwork**, **ggttext** and **ggforce** packages, a section of custom fonts and colors, a collection of R packages tailored to create interactive charts, and several new chart types. The updated version now contains ~3.000 lines of code and 188 plots and received a lot of interest from **ggplot2** users from many different professional fields.

Today, on a sunny day in July 2021, this tutorial serves as the starting point for the book you hold in your hands. I hope you enjoy it as much as I enjoyed learning and sharing **ggplot2** wizardry!

¹<http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/>

²<https://github.com/rfordatascience/tidytuesday/blob/master/README.md>

³<https://www.cedricscherer.com/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/>

Why read this book

Often, people that use common graphic design and charting tools or have basic experience experience with **ggplot2** cannot believe what one can achieve with this graphics library—and I want to show you how one can create a publication-ready graphic that goes beyond the traditional scientific scatter or box plot.

ggplot2 is already used by a large and diverse group of graduates, researchers, and analysts and the current rise of R and the tidyverse will likely lead to an even increasing interest in this great plotting library. While there are many tutorials on **ggplot2** tips and tricks provided by the R community, to my knowledge there is no book that specifically addresses the complete design of specific details up to building an ambitious multipanel graphic with **ggplot2**. As a blend of strong grounding in academic foundations of data visualization and hands-on, practical codes, and implementation material, the book can be used as introductory material as well as a reference for more experienced **ggplot2** practitioners.

The book is intended for students and professionals that are interested in learning **ggplot2** and/or taking their default ggplots to the next level. Thus, the book is potentially interesting for **ggplot2** novices and beginners, but hopefully also helpful and educational for proficient users.

Among other things, the book covers the following:

- Look-up resource for every-day and more specific ggplot adjustments and design options
 - Practical hands-on introduction to **ggplot2** to quickly build appealing visualization
 - Discussion of best practices in data visualization (e.g. color choice, direct labeling, chart type selection) along the way
 - Coverage of useful **ggplot2** extension packages
 - Ready-to-start code examples
 - Reference implementations illustrating code solutions and design choices
-

How to read this book

This book can either serve as a textbook or as a reference. Depending on your skill level, some codes and tricks may already be known or not helpful at the moment. In case you want to directly jump to the chapters you find most promising or helpful, here are some suggestions:

- How do I get started with the code? → Chapter 2
 - I have no idea how **ggplot2** actually works and need a quick introduction → Chapter 3
-

Prerequisites

To run any of the materials locally on your own machine, you will need the following:

- A recent version of R (download from here⁴)
- Preferably an *Integrated Development Environment* (IDE) to store scripts and run code, e.g. RStudio (download from here⁵) or Visual Studio Code (download from here⁶)
- A set of R packages installed:
 - **tidyverse**⁷ that includes **ggplot2**⁸
 - **ggforce**⁹
 - **ggrepel**¹⁰
 - **ggtext**¹¹
 - **magick**¹²
 - **patchwork**¹³
 - **ragg**¹⁴
 - **rnatuarlearth**¹⁵
 - **scico**¹⁶
 - **sf**¹⁷

To install all packages in one go, run the following code in the R console:

```
install.packages(c(
  "tidyverse", "ggforce", "ggtext", "magick", "patchwork",
  "ragg", "rnaturallearth", "scico", "sf"
))
```

Software information and conventions

The book was written with the **knitr** package (Xie, 2015) and the **bookdown** package (Xie, 2022) with the following setup:

```
## R version 4.1.0 (2021-05-18)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 22621)
##
## Locale:
##   LC_COLLATE=German_Germany.1252
##   LC_CTYPE=German_Germany.1252
##   LC_MONETARY=German_Germany.1252
##   LC_NUMERIC=C
```

⁴<https://cloud.r-project.org/>

⁵<https://rstudio.com/products/rstudio/download/#download>

⁶<https://code.visualstudio.com/download>

⁷<https://www.tidyverse.org/>

⁸<https://ggplot2.tidyverse.org/>

⁹<https://ggforce.data-imaginist.com/>

¹⁰<https://ggrepel.slowkow.com/>

¹¹<https://wilkelab.org/ggtext/>

¹²<https://docs.ropensci.org/magick/>

¹³<https://patchwork.data-imaginist.com/>

¹⁴<https://ragg.r-lib.org/>

¹⁵<https://docs.ropensci.org/rnaturalearth/>

¹⁶<https://github.com/thomasp85/scico>

¹⁷<https://r-spatial.github.io/sf/>

```
##   LC_TIME=German_Germany.1252
## system code page: 65001
##
## Package version:
##   askpass_1.1           assertthat_0.2.1
##   backports_1.2.1        base64enc_0.1.3
##   bit_4.0.4              bit64_4.0.5
##   blob_1.2.2             bookdown_0.28
##   broom_1.0.0            bslib_0.4.0
##   cachem_1.0.6          callr_3.7.2
##   cellranger_1.1.0       cli_3.3.0
##   clipr_0.7.1            colorspace_2.0-3
##   compiler_4.1.0          cpp11_0.4.2
##   crayon_1.5.1           curl_4.3.2
##   data.table_1.14.2       DBI_1.1.2
##   dbplyr_2.1.1           digest_0.6.29
##   dplyr_1.0.9             dtplyr_1.2.0
##   ellipsis_0.3.2         evaluate_0.16
##   fansi_0.5.0             farver_2.1.0
##   fastmap_1.1.0          forcats_0.5.2
##   fs_1.5.2                gargle_1.2.0
##   generics_0.1.3          ggplot2_3.3.6
##   glue_1.6.2               googledrive_2.0.0
##   googlesheets4_1.0.0      graphics_4.1.0
##   grDevices_4.1.0          grid_4.1.0
##   gtable_0.3.0             haven_2.4.3
##   highr_0.9                hms_1.1.2
##   htmltools_0.5.2          httr_1.4.2
##   ids_1.0.1                isoband_0.2.5
##   jquerylib_0.1.4          jsonlite_1.7.2
##   knitr_1.39                labeling_0.4.2
##   lattice_0.20.44          lifecycle_1.0.1
##   lubridate_1.8.0           magrittr_2.0.3
##   MASS_7.3.54                Matrix_1.3.3
##   memoise_2.0.1             methods_4.1.0
##   mgcv_1.8.35               mime_0.12
##   modelr_0.1.8              munsell_0.5.0
##   nlme_3.1.152              openssl_2.0.2
##   patchwork_1.1.2            pillar_1.8.1
##   pkgconfig_2.0.3            prettyunits_1.1.1
##   processx_3.7.0            progress_1.2.2
##   ps_1.6.0                  purrr_0.3.4
##   R6_2.5.1                  ragg_1.2.2
##   rappdirs_0.3.3            RColorBrewer_1.1.3
##   readr_2.1.2                readxl_1.4.1
##   rematch_1.0.1              rematch2_2.1.2
##   reprex_2.0.2               rlang_1.0.4
##   rmarkdown_2.16              rstudioapi_0.14
##   rvest_1.0.2                 sass_0.4.2
##   scales_1.2.1                selectr_0.4.2
##   splines_4.1.0              stats_4.1.0
```

```
##   stringi_1.7.5      stringr_1.4.1
##   sys_3.4            systemfonts_1.0.3
##   textshaping_0.3.6  tibble_3.1.8
##   tidyrr_1.2.0        tidyselect_1.1.2
##   tidyverse_1.3.2     tinytex_0.41
##   tools_4.1.0         tzdb_0.1.2
##   utf8_1.2.2          utils_4.1.0
##   uuid_1.0.3          vctrs_0.4.1
##   viridisLite_0.4.1   vroom_1.5.7
##   withr_2.5.0          xfun_0.31
##   xml2_1.3.3          yaml_2.2.1
```

Package names are in bold text and wrapped into curly brackets (e.g. **{ggplot2}**), and inline code and file names are formatted in a monospaced typewriter font (e.g. `read_csv("data.csv")`). Function names are followed by parentheses (e.g. `ggplot2::ggplot()`). Notes are formatted as coloured, italic text.

Acknowledgments

Thanks to David Grubbs, Alberto Cairo, Emily Riederer, Oscar Baruffa, and Malcolm Barrett for all your constructive feedback.

Cédric Scherer
Berlin, Germany

About the Author

Dr Cédric Scherer is a graduated computational ecologist with a passion for good design. In 2020, he combined his expertise in analyzing and visualizing large data sets in R with his interest in design and his perfectionism to become an independent data designer and data visualization consultant.

Cédric has created visualizations across all disciplines, purposes, and styles. Due to regular participation to social data challenges, he is now well known for complex and visually appealing figures, entirely made with **ggplot2**, that look as they have been created with a vector design tool.

As a data visualization specialist, he does rarely create dashboards but acts as a consultant and designer improving chart and design choices and workshop coach teaching data visualization principles and courses on R and **ggplot2**. He also uses R and the **tidyverse** packages to automate data analyses and plot generation, following the philosophy of a reproducible workflow.

1

Introduction

1.1 Communicating Data

Communicating data is critical for many of us, no matter if scientists, journalists, or analysts. How we present data affects the engagement of and interpretation by the audience. Showing data in an honest, meaningful—and maybe sometimes even playful or artistic—way is the art of ***data visualization*** or ***information visualization***. Data visualization can be described as the transformation of numbers into visual quantities, encoded by forms, positions, and colors. In the best case, a well-designed data visualization helps to amplify cognition, facilitate insights, discover, spark curiosity, explain, and make decisions.

Data visualizations, or broadly speaking ***information graphics***, are often classified as being either exploratory or explanatory. ***Exploratory graphics*** are generated to understand the data and search for the relevant information. ***Explanatory graphics*** aim to communicate the derived information between people (Koponen and Hildén, 2019). In contrast to exploratory graphics, the creation of engaging explanatory graphics involves not only the display of data but also requires many choices with regard to the storytelling and design.

When designing visualizations myself or looking at the work of others, the most important question to me is the ***purpose*** of the graphic. Without a clear understanding of the purpose, it is impossible to design an effective and engaging visualization. The same applies when evaluating a visualization: without the consideration of the purpose—the audience, the message, the mood—the designer had in mind when creating the visualization, the critique of design choices often becomes obsolete. A common assumption is that the single aim of data visualizations is to guide decisions. This might be true for business or scientific applications that aim for precision and accuracy by creating ***pragmatic visualizations*** (Kosara, 2007).

At the same time, it is ignorant to assume that efficiency and functionality are the main purpose of every visualization. Many of the great visualizations we have seen and that stick to our mind go beyond the precise, informative display of data¹. They experiment with new approaches, use clever, unusual ways to tell stories or were designed simply to transport joy, curiosity or concern. In some cases, the design and visual novelty may even be the main focus with the aim to create a novel, artistic experience for the viewer. Such artworks are not necessarily created to maximize discovery or communication but to elicit emotions and can be termed ***affective graphics***².

As a ***creator***, clearly defining the purpose of a visualization helps to make decisions about the data, the chart type, and the design. As a ***reader***, identifying the purpose helps rating the quality of the presentation. Some people like to think that there is a single best approach

¹However, I am not saying that these are the only ones that are great—there are definitely several magnificent pragmatic visualizations that come to my mind!

²Credit to the term ***affective graphics*** goes to Alberto Cairo, thank you for sharing your thoughts with me.

to visualize data: the one that has survived the test of time and is the most efficient to quantify information. Some believe that a chart has to be designed in a *neutral* way. I strongly disagree with both opinions, for multiple reasons. The most important: Every time we present the data, we make decisions; and it is not about *if* we make decisions but *which*. Chart types are not inherently ‘right’ or ‘wrong’ but might be more or less suitable for the purpose. Colors are associated with some emotional value—how could we pick one that has a neutral meaning, association or emotion for every person that might look at your visualization?

Even if we agree on the ‘right’ decisions—the best chart type and a neutral color encoding, likely some shades of grey—we still can’t ensure that all people interpret it in the same way. People will always find their own message in graphs and the interpretation will likely differ based on individual differences through culture, attitude and mood.

A quote from Alberto Cairo that is close to my heart sums it up brilliantly:

Visualizations can be designed and experienced in various ways, by people of various backgrounds, and in various circumstances. That’s why reflecting on the purpose of a visualization is paramount before we design it—or before we critique it. ([Cairo, 2021](#))

In the optimal case, the decisions made by the creator are based on some thoughtful consideration of the following:

- *data* — which information is meaningful and robust?
- *audience* — what do readers already know?
- *context* — how will the reader encounter the visualization?
- *story* — what is the main message of the visualization?
- *goal* — which chart type is suitable to transport the story?
- *design* — how can I facilitate engagement and understanding?

While some decisions might (and should) be made before crafting the visualization, the creation of purposeful, well-designed graphics is an iterative process. Rarely³ the first draft is what ends up being printed on physical material or being displayed on your computer or smartphone screen. Nowadays, computational approaches ease the cyclic process of prototyping, exploring, testing and designing the best visual encoding of information for a given purpose.

1.2 Coding Visualizations

As data visualizations involve the quantitative representation of variables, an environment that allows to handle, wrangle and quantify data is preferential. Classical design software

³I was very tempted to write “never” but I don’t have data to support this claim...

is great to create vector-based graphics of all kinds but must often be paired with a ‘visualization tool’ if the data and/or the chart type becomes more complex. While there are many tools that allow to quickly create specific chart types (e.g. DataWrapper, Flourish, RAWgraphs), often also with beautiful and very sensible defaults, such chart builder do not provide full flexibility. Furthermore, the combination of a suite of tools might be ...

By using a computational, code-driven approach we can combine all steps related to data visualization in the same environment: from the data import and cleaning to the precise and flexible encoding of quantitative information with custom designs. Programming languages such as JavaScript, Python, or R have a much steeper learning curve but at the same time allow users to create almost any visualization one can think of. Furthermore, they come with several *extension libraries* (e.g. D3.js, echarts, Vega, Matplotlib, ggplot2) that provide additional approaches or add more opportunities to existing code.

Data visualizations that are generated with code have several other benefits. The *reproducibility* of code makes the process more efficient by being able to update the data or to use the code as a template for future projects. The *transparency* of coded (and well-documented) data workflows increases trust. The *scalability* of code allows to produce the graphics for multiple data sets and use cases.

Of course, the visualization does not need to be created by code alone. Switching from a code-based approach to a vector-graphics tool makes a lot of sense in use cases where reproducibility does not matter or graphics are stand-alone artworks. Honestly, in terms of efficiency and freedom, a combined approach is likely the best approach in such a case.

With that in mind, knowing how to code visualizations is likely beneficial in any data-related field.

1.3 Why R and ggplot2

As a computational ecologist, I’ve learned and used a range of different tools and programming languages for various purposes such as data wrangling, statistical analyses, and model building. The open-source language R was and is the programming language most widely used by ecologists to handle and analyze ecological data (Sciaiani et al., 2018). Consequently, I was *of course* using R in my daily life as an scientific researcher.

Nowadays, R plays a crucial part in many data-related workflows, no matter if for scientific, educational, or industrial use cases. Thanks to the ever growing R community and the rich collection of libraries that add additional functionality and simplify workflows, R is an attractive programming language that has outgrown of its original purpose: statistical analyses. Today, R can serve as tool to generate automated reports, develop stand-alone web apps, and draft presentation slides, books, and web pages. And to design high-level, publication-ready visualizations.

Even though R—similar to most programming languages—has a steep learning curve, the level of functionality, flexibility, automation, and reproducibility offered can be a major benefit also in a design context:

- The layered approach of **ggplot2** opens the possibility to build any type of visualization.
- Various extension packages add missing functionality.
- Script-based workflows instead of *point-and-click* approaches allow for reproducibility—

which means you can simply (in theory) run the code again after receiving new data or create thousands of visualizations for various data sets in no time.

- Sharing code is becoming the golden standard in many fields and thus facilitates transparency and credibility as well as modification and creative advancement.
- A helpful community and many free resources simplify learning experiences and the search for solutions.
- The visualizations created in R can be exported as vector files and thus allow for post-processing with vector graphics software like Adobe Illustrator, Inkscape or Figma.

2

Get Started

2.1 The Data

We are using historical data for bike sharing in London in 2015 and 2016, provided by *TfL* (*Transport for London*)¹. The data was collected from the TfL data base and is ‘Powered by TfL Open Data’. The processed data set contains hourly information on the number of rented bikes and was combined with weather data acquired from freemeteo.com. The data was contributed to the Kaggle online community² by Hristo Mavrodiev.

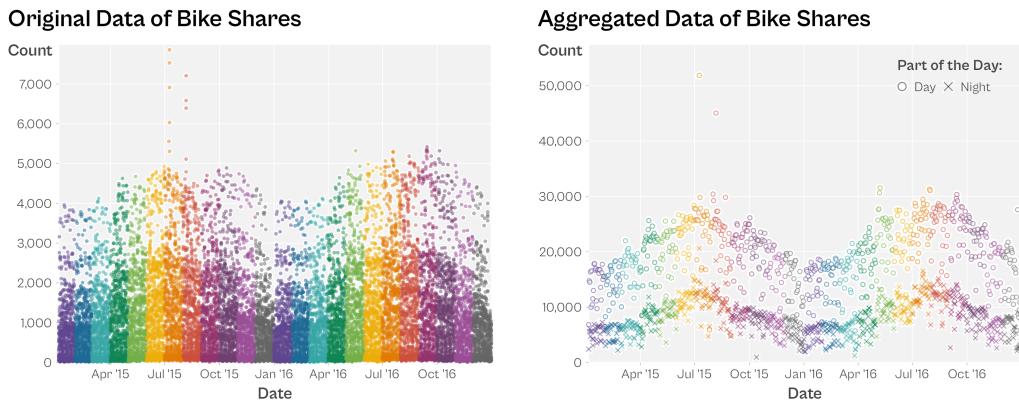


FIGURE 2.1: The original and aggregated data sets in direct comparison: counts of bike shares registered by TfL over time with month encoded by colour. The left panel shows counts for every hour of the day, while in the right panel the hourly data was aggregated into two periods of the day (day and night).

To make the visualizations manageable and patterns more insightful, we are using a modified data set with all variables aggregated for day (6:00am–5:59pm) and night (6:00pm–5:59am) (2.1). The bike counts were summarized while all weather-related variables were averaged. Finally, for the weather type, the most common was used and, in case of a tie, one of the most common types was randomly chosen. The modified data set contains 14 variables (columns) with 1,454 observations (rows). To give you a better idea what the data set contains, a visual overview of the variables is provided in table 2.1 and figure 2.2.

COMMENT: Decide on a version to provide and overview of the variables as table or list.

¹<https://tfl.gov.uk/modes/cycling/santander-cycles>

²<https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset>

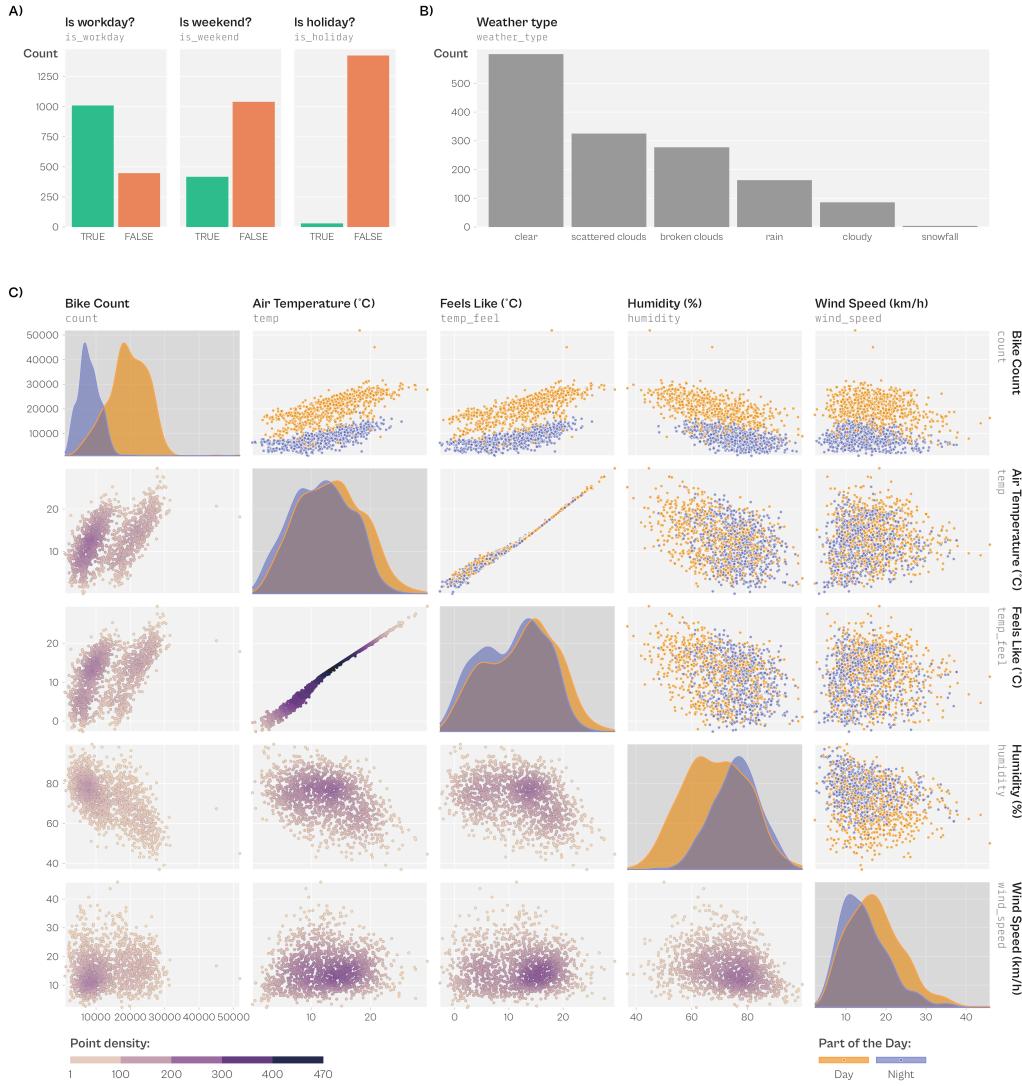


FIGURE 2.2: Overview of the distribution of the boolean variables `is_workday`, `is_weekend`, and `is_holiday` (A), the categorical variable `weather_type` (B), and the continuous variables `count`, `temp`, `temp_feel`, `humidity`, and `wind_speed` (C) of the cleaned and aggregated bike sharing data set. In panel C, the correlation between the variables is shown as scatterplot encoded by `timeperiod` (upper triangle) and encoded by point density (lower triangle), highlighting the level of overlap of data points.

TABLE 2.1: Overview of the 15 variables contained in the cleaned and aggregated bike sharing data set.

Variable	Description
date	Date encoded as ‘YYYY-MM-DD’
day_night	‘day’ (6:00am–5:59pm) or ‘night’ (6:00pm–5:59am)
year	‘2015’ or ‘2016’
month	‘1’ (January) to ‘12’ (December)
season	‘0’ (spring), ‘1’ (summer), ‘2’ (autumn) or ‘3’ (winter)
count	Sum of bikes rented
is_workday	‘TRUE’ being Monday to Friday and no official holiday
is_weekend	‘TRUE’ being Saturday or Sunday
is_holiday	‘TRUE’ being an official holiday in the UK
temp	Average air temperature (°C)
temp_feel	Average feels like temperature (°C)
humidity	Average air humidity (%)
wind_speed	Average wind speed (km/h)
weather_type	Most common weather typed

2.2 Working in R

`{ggplot2}` can be used even if you know little about the R programming language. However, the knowledge of certain basic principles is at least helpful and probably indispensable for advanced plots. This section will give you a short overview of workflows and the very basics needed. The overview makes use of the `{tidyverse}`, a package collection designed for data science in R. However, multiple other options exist to import, inspect, and wrangle your data if you prefer not to work with the `{tidyverse}` for these steps³.

2.2.1 Import data

You need to import data to be able to work with it in the current session. The data can be imported from a local directory or directly from a web source. Nowadays, all common and some less common data formats can easily be imported. For traditional tabular data as .txt or .csv one can use the `{readr}` package.

We use the `read_csv()` function to load the TfL data as .csv file directly from a web URL. To access the URL and data later, we are storing the link as `url_data` and the data set as `bikes` by using the *assignment arrow* `<-`. The `col_types` argument within the ‘`read_csv()`’ function allows to specify the column types. For example, `i` represents integer values, `f` encodes factors, and `l` turns a column into a logical, boolean variable that only can have two states, `TRUE` or `FALSE`. You’ll find more on the different *data types* later in this chapter.

³Note that the `{ggplot2}` package itself belongs to the `{tidyverse}` as well.

```
url_data <- "https://cedricscherer.com/data/london-bikes.csv"
bikes <- readr::read_csv(file = url_data, col_types = "Dcffffilllldddfc")
```

The `::` is called “namespace” and can be used to access a function without loading the package. Here, you could also run `library(readr)` first and `bikes <- read_csv(url_data)` afterwards.

If you want to load data that is stored locally, you specify the path to the file instead:

```
path_data <- "C://path/to/my/data/london-bikes.csv" ## mocked-up name for Win users
bikes <- readr::read_csv(file = path_data, col_types = "Dcffffilllldddfc")
```

Note that the syntax of the path differs between operating systems. While in Windows subdirectories are separated with backslashes, Mac and Linux uses slashes. Also, absolute paths differ in their syntax: Windows machines specify the drive letter, here `C://`, or the server name; on Mac/Linux machines absolute paths start with `/users/`.

Instead of relying on the absolute path, which likely differs between users and operating systems, you can also use relative paths. Those are specified with a leading dot, e.g. as `./data/london-bikes.csv`, and look for the file relative from the working directory. You can retrieve and change your working directory with `getwd()` and `setwd()`. However, relying on the default working directory or setting a custom one will likely cause problems the moment someone else wants to run your code⁴.

2.2.2 Project-oriented workflows

Preferably, we do not want to rely on absolute paths and the default or manually set working directories. So-called project-oriented workflows aim to organize each piece of work in a self-contained, bundled directory. This directory includes all relevant files needed for the project, such as scripts and images. By ensuring that paths are set relative and in a way that they are understood by any operating system, we guarantee that the project will run on any machine and can easily moved around.

If you are working in Rstudio, *Rstudio projects* provide such a project-oriented workflow. When opening the project via the associated `.Rproj` file, it ensures that the working directory is correctly set and points to the project’s top-level directory (i.e. the folder that contains the `.Rproj` file).

When using Rstudio projects, a helpful package to navigate to files of interest is the `{here}` package⁶. The function `here()` will create paths relative to the top-level directory:

```
## point to the csv inside the "data" subdirectory of the project directory
path_data <- here::here("data", "london-bikes.csv")
```

⁴A detailed reasoning why you should not use `setwd()` is provided in Chapter 2 of “What they forgot you to teach about R”⁵.

⁶<https://here.r-lib.org/>

2.2.3 Inspect data

After importing the data, it is advisable to have a look at the data. Does the object stored in R match the dimensions of your original data file? Are the variables displayed correctly? You can print the data by simply running the name of the object, here `bikes`.

```
bikes
```

```
## # A tibble: 1,454 x 14
##   date      day_ni~1 year month season count is_wo~2
##   <date>    <chr>   <fct> <fct> <fct>  <int> <lgl>
## 1 2015-01-04 day     2015  1     3      6830 FALSE
## 2 2015-01-04 night   2015  1     3      2404 FALSE
## 3 2015-01-05 day     2015  1     3      14763 TRUE
## 4 2015-01-05 night   2015  1     3      5609 TRUE
## 5 2015-01-06 day     2015  1     3      14501 TRUE
## 6 2015-01-06 night   2015  1     3      6112 TRUE
## 7 2015-01-07 day     2015  1     3      16358 TRUE
## 8 2015-01-07 night   2015  1     3      4706 TRUE
## 9 2015-01-08 day     2015  1     3      9971 TRUE
## 10 2015-01-08 night  2015  1     3      5630 TRUE
## # ... with 1,444 more rows, 7 more variables:
## #   is_weekend <lgl>, is_holiday <lgl>, temp <dbl>,
## #   temp_feel <dbl>, humidity <dbl>, wind_speed <dbl>,
## #   weather_type <fct>, and abbreviated variable names
## #   1: day_night, 2: is_workday
```

As we have used the `{readr}` package, our data is stored as a *tibble* (class `tbl_df` and related) which is the `{tidyverse}` subclass of a traditional data frame (class `data.frame`). There are other data structures in R such as lists and matrices; however, in this book we are going to use only data frames, more precisely tibbles (besides spatial data formats in Chapter XYZ).

On the top of the output, you can directly see that our data set consists of `format(length(bikes), big.mark = ",")` variables (columns) frame with 1,454 observations (rows). Also, the tibble output shows you the first ten rows. Alternatively you can inspect the data with the help of `str()` or `tibble::glimpse()` to print a transposed version.

2.2.4 Data types

If you have looked carefully, you may have noticed that a tibble prints also the data type of each column, e.g. `<chr>`. In our case, we have specified the types of the columns manually when importing the data; if not specified, `readr::read_csv()` as most other import functions will guess the data type for each column based on the first x observations.

The data encoding is especially important when exploring chart options and writing ggplot code. You should be familiar if the data is encoded as quantitative or qualitative, if it contains missing or unusual values. Thus, it is always worth to check the classes and the values of the columns.

In R there are multiple low-level data types, and some of the `{ggplot2}` behavior will depend on the type of the variable(s) used for plotting. The six basic data types are:

- character
- factor
- numeric
- integer
- logical
- complex

To examine the data type of an object, use the `class()` function. The `$` symbol allows you to access single columns of a data frame, e.g. `bikes$day_night`:

```
class(bikes$day_night)
## [1] "character"

class(bikes$weather_type)
## [1] "factor"

class(bikes$temp)
## [1] "numeric"

class(bikes$count)
## [1] "integer"

class(bikes$is_weekend)
## [1] "logical"
```

Variables of type `character`, `factor`, and `logical` will be handled as **categorical, qualitative data** while `numeric`, `integer`, and `complex`⁷ are treated as **numerical, quantitative data**.

The most important difference between the categorical types is their sorting: while `character` and also `logical` values are sorted alphabetically, `factor` variables have a specified order, defined by the so-called *levels*. Note that numbers can be treated as well and thus we can change their intrinsic order, too. We can inspect the order with the `levels()`:

```
levels(bikes$weather_type) ## an example of a factor with strings
## [1] "broken clouds"      "clear"
## [3] "cloudy"              "scattered clouds"
## [5] "rain"                "snowfall"
levels(bikes$season) ## an example of a factor with numbers
## [1] "3" "0" "1" "2"
```

Also, we can manually change the order of a factor by supplying a vector of level names to the `factor()` function:

```
bikes$season_type_mix <- factor(bikes$season, levels = c("2", "0", "3", "1"))
levels(bikes$season_type_mix)
## [1] "2" "0" "3" "1"
```

⁷As variables of type `complex` are rarely used in a data visualization context, we are going to ignore this data type.

The `{forcats}` package⁸ provides a set of useful functions to reorder factor levels. Important functions to quickly change to order of variables include `fct_rev()`, `fct_reorder()`, `fct_infreq()`, `fct_inorder()`, and `fct_lump()`:

```
bikes$weather_type_rev <- forcats::fct_rev(bikes$weather_type)
levels(bikes$weather_type_rev)
## [1] "snowfall"          "rain"
## [3] "scattered clouds" "cloudy"
## [5] "clear"             "broken clouds"

bikes$weather_type_rank <- forcats::fct_infreq(bikes$weather_type)
levels(bikes$weather_type_rank)
## [1] "clear"           "scattered clouds"
## [3] "broken clouds"   "rain"
## [5] "cloudy"          "snowfall"
```

The single difference between the two numerical types `numeric` and `integer` is simply-speaking the existence of floating point numbers: while `numeric` variables store decimals, `integer` variables are stored as whole numbers. If you want to force integer values, you can specify them as `1L`.

```
head(bikes$temp) ## numeric
## [1] 2.167 2.792 8.958 7.125 9.000 6.708

as.integer(head(bikes$temp)) ## integer
## [1] 2 2 8 7 9 6
```

As shown in the last command, we can also convert data types. Be careful though, as R has internal rules how to *coerce* one data type into another. The same also happens if you specify a vector with different data types. In the following example, the `integer`, `numeric`, and `logical` values are coerced to `character` values. Afterwards, we explore the coercion behavior by converting the vector to other data types:

```
my_vector <- c(1L, 0.2, "ggplot", FALSE)
my_vector
## [1] "1"      "0.2"    "ggplot" "FALSE"

class(my_vector)
## [1] "character"

as.numeric(my_vector)
## [1] 1.0 0.2 NA NA

as.integer(my_vector)
## [1] 1 0 NA NA

as.factor(my_vector)
```

⁸<https://forcats.tidyverse.org/>

```
## [1] 1      0.2    ggplot FALSE
## Levels: 0.2 1 FALSE ggplot

as.logical(my_vector)
## [1] NA    NA    NA FALSE
```

When changing the data type of our `character` vector, some values are successfully converted (e.g. `1L` and `0.2` as `numeric` or `FALSE` as `logical`) while some are “wrongly” interpreted (e.g. `0.2` is converted into `0L` as `integer` or `FALSE` into “`FALSE`” as `character`). Some others are stored as `NA` representing *unknown, not available values* (e.g. “`ggplot`” as `numeric` or `1` as `logical`). Conversion between different data types is very useful but be careful and always inspect the output of the conversion for potential coercion mistakes.

Another important data type is the `Date` class. Dates are either represented as the number of days since a specified origin or converted `character` type with a structure of YYYY-MM-DD:

```
class(bikes$date)
## [1] "Date"

as.Date(1, origin = "1970-01-01")
## [1] "1970-01-02"

as.Date("2007-06-10")
## [1] "2007-06-10"

as.Date("2022-09-16") - as.Date("2007-06-10")
## Time difference of 5577 days
```

Furthermore, `POSIXct` and `POSIXlt` are able to represent full time stamps including a date and time. The two POSIX date/time classes only differ in the way that the values are stored internally.

```
as.POSIXct("2007-06-10 12:34:56")
## [1] "2007-06-10 12:34:56 CEST"

as.POSIXlt("2007-06-10 12:34:56")
## [1] "2007-06-10 12:34:56 CEST"
```

2.2.5 Data wrangling

Often, the data imported might not be in the right format to plot it with `{ggplot2}`. The preparation of the data—e.g. converting data types and setting factor levels, aggregating values, estimating data summaries, reshaping the data set or combining it with another source—is called *data wrangling, data munging, or data manipulation*.

To explore and retrieve summary estimates of individual variables, the following functions are useful:

- `min()`, `max()`, `range()` to extract extremes of numerical data

- `quantile()` to get an idea of the distribution numerical data
- `unique()` to get all unique entries, helpful for categorical data
- `length(unique())` to count all unique entries

```
min(bikes$temp) ## add na.rm = TRUE in case it returns `NA`
## [1] 0.125

range(bikes$date)
## [1] "2015-01-04" "2016-12-31"

quantile(bikes$count)
##    0%   25%   50%   75%  100%
## 953  7508 11965 19412 51870

unique(bikes$day_night)
## [1] "day"   "night"

length(unique(bikes$weather_type))
## [1] 6
```

The `{dplyr}`⁹ and other `{tidyverse}` packages provide a simple and intuitive syntax to wrangle data. There are a ton of unique functions, but knowing a handful is enough to empower you to bring your data in the right format.

There are five main functions of `{dplyr}`, called *verbs* in this context:

- `filter()`: Pick rows with matching criteria
- `select()`: Pick columns with matching criteria
- `arrange()`: Reorder rows
- `mutate()`: Create new variables
- `summarize()` or `summarise()`: Sum up variables

All of these functions follow a consistent syntax: `verb(data, condition)`. Note that we specify the data and columns individually and that within the tidyverse, column names are referred to without quotation marks:

```
## only keep more than 1000 shares during the night
filter(bikes, day_night == "night", count > 1000)

## only keep 4 columns and reorder those
select(bikes, date, count, weather_type, temp)

## order by day_night and decreasing bike shares
arrange(bikes, day_night, -count)

## add a column with temperature encoded as °F
mutate(bikes, temp_fahrenheit = temp * 1.8 + 32)
```

⁹<https://dplyr.tidyverse.org/>

```
## calculate the mean count across all observations
summarize(bikes, count_avg = mean(count))
```

Another important and very powerful function from the `{dplyr}` package is `group_by()`: when a data set is grouped into subsets, we can apply any operation for each group *within a single data frame*. While you could nest those functions, the common approach within the tidyverse is the use of *pipes*. Pipes take the output of one function and send it directly to the next which avoids nested operations and allows for a more logical order of functions and their arguments. The `{tidyverse}` pipe is encoded as `%>%`; a base R pipe is available as `|>` as well. Have a look at the following silly example:

```
## nested functions, read inside out + disconnected function inputs
go_to_work(
  breakfast(
    wake_up(
      Cedric, alarm = "06:30")
    ),
    c("coffee", "croissant")
  ),
  mode = "bus", delay = 10
)

## piped version with "Cedric" passed to the next function call
Cedric %>%
  wake_up(alarm = "06:30") %>%
  breakfast(c("coffee", "croissant")) %>%
  go_to_work(mode = "bus", delay = 10)
```

Let's create the workflow for some serious data preparation: In a first step, we are going to estimate the average temperature for night rents per year and season:

```
bikes_summarized <-
  bikes %>%
  ## only keep night observations
  filter(day_night == "night") %>%
  ## create 8 subsets (4 seasons x 2 years)
  group_by(season, year) %>%
  ## calculate total counts and mean temperature per subgroup
  summarize(
    count = sum(count),
    temp_avg = mean(temp)
  )

bikes_summarized

## # A tibble: 8 x 4
## # Groups:   season [4]
##   season year     count temp_avg
##   <fct>  <fct>   <int>    <dbl>
```

```

## 1 3      2015  459469   7.65
## 2 3      2016  489136   6.91
## 3 0      2015  693703   9.93
## 4 0      2016  676427   9.43
## 5 1      2015  1020524  17.2
## 6 1      2016  1044592  17.7
## 7 2      2015  685534   12.4
## 8 2      2016  744827   12.3

```

After filtering out “day” cases, the grouping by `season` and `year` allows us to summarize counts and average temperatures for each of the eight groups. Note that the resulting tibble is still grouped by `season` (as shown in the output with the [4] explicitly stating the number of current subsets).

In a next step, we regroup our data to add a column with shares of nightly bike rents per season on a yearly basis. Afterwards, we remove all subsets by calling `ungroup()` and clean our data set by reordering and removing columns and sorting rows by year and season:

```

bikes_summarized %>%
  ## add column with relative shares per year
  group_by(year) %>%
    mutate(share_year = count / sum(count)) %>%
  ## remove grouping
  ungroup() %>%
  ## reorder columns and remove days column
  select(year, season, count, share_year, temp_avg) %>%
  ## sort by year and season
  arrange(year, season)

## # A tibble: 8 x 5
##   year  season  count share_year temp_avg
##   <fct> <fct>   <int>     <dbl>     <dbl>
## 1 2015  3        459469    0.161     7.65
## 2 2015  0        693703    0.243     9.93
## 3 2015  1       1020524   0.357    17.2
## 4 2015  2       685534    0.240    12.4
## 5 2016  3        489136    0.166     6.91
## 6 2016  0        676427    0.229     9.43
## 7 2016  1       1044592   0.354    17.7
## 8 2016  2       744827    0.252    12.3

```

2.2.6 Quarto and Rmarkdown

3

The {ggplot2} Package

ADD SHORT HISTORY OF GGPLOT2

When looking into the package description of the **{ggplot2}** package, it states the following:

{ggplot2} is a system for declaratively creating graphics, based on The Grammar of Graphics¹. You provide the data, tell **{ggplot2}** how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

The most important insight from that technical description is that (1) it relies on a framework called the grammar of graphics (hence the gg in **{ggplot2}**) and (2) we map variables to aesthetics. This is an important concept when writing **{ggplot2}** code.

3.1 The Structure of a ggplot

In general, a ggplot is built up from the following components:

1. **Data:**
The raw data that you want to plot.
2. **Aesthetics aes():**
The mapping of variables to visual properties, such as position, color, size, shape, and transparency.
3. **Layers:**
The representation of the data on the plot panel which is a combination of the geometric shapes (**geom_**) representing the data and the statistical transformation ('stat_') of the data, such as fitted curves, counts, and data summaries.
4. **Scales scale_:**
The control of the mapping between the data and the aesthetic dimensions, such as data range to positional aesthetics or qualitative or quantitative values to colors.
5. **Coordinate system coord_:**
The transformation used for mapping data coordinates into the plane of the graphic.

¹https://link.springer.com/chapter/10.1007/978-3-642-21551-3_13

6. **Facets facet_:**

The arrangement of the data into a grid of plots (also known as *trellis* or *lattice plot*, or simply *small multiples*).

7. **Visual themes theme():**

The overall visual (non-data) details of a plot, such as background, grid lines, axes, typefaces, sizes, and colors.

The number of elements may vary depending on how you group them and whom you ask. This list is based on the list provided in the “ggplot2” book by Hadley Wickham².

A basic ggplot needs three things that you have to specify: the *data*, *aesthetics*, and a *layer*. All other components can be further combined and modified to customize your chart.

You can think of a ggplot as a receipt for a dish: it can be based on a few or a diversity of ingredients. Also, you are free to add additional ingredients to spice-up your creation (literally and visually).

Similarly, you can build rather basic charts such as scatter plots or histograms with only a few lines code. But {ggplot2} also allows to create rather complex charts that combine multiple geometries, statistical transformations and maybe even data sets. On top, it is up to you how much effort you take to polish the plot. You can rely on the defaults used for data-related aesthetics (e.g. default axis breaks and color palettes) and non-data aspects (e.g. complete themes). Or you decide to modify the data-related aesthetics such as axes and color palettes and/or customize the theme elements of your graphic to your needs.

3.1.1 Data

No data visualization without data.

²<https://ggplot2-book.org/introduction.html>

4

A walk-through example

To illustrate the utility and flexibility of `{ggplot2}` to create complex, well-designed visualizations with a handful of components, we are going to create a set of four plots. For each combination of year and time of the day, we draw a scatter plot of bike rents including a linear fitting to show overall trends with feels-like temperature.

The creation of the plot itself will require only four lines of code! One to specify the data and global aesthetics, two for the scatter and fitting, and another one to create small multiples.

With a few more lines of code, we will adjust the labels, add annotations, and apply a custom color palette. Finally, we apply a personalized theme using custom typefaces.

4.1 Create a basic ggplot

Once the `{ggplot2}` package is loaded, we can create a basic ggplot by specifying the *data*, *aesthetics*—the positional mapping of the variables—, and a *geometry* (Fig. 4.1).

```
library(ggplot2)

ggplot(data = bikes) + ## data
  aes(x = temp_feel, y = count) + ## aesthetics
  geom_point() ## geometric layer
```

In most cases, you'll find ggplot code in which the aesthetics are supplied inside the `ggplot()` call; however, both versions are valid.

```
ggplot(data = bikes, mapping = aes(x = temp_feel, y = count)) +
  geom_point()
```

Due to so-called *implicit matching*, we can omit the argument names `data` and `mapping` in the `ggplot()` call as long as we first add the `data` object and afterwards the `aes()` function. You can read more about how to specify data and aesthetics in Chapter XYZ.

4.2 Combine multiple layers

One can also add several layers, no matter if geometric shapes or statistical transformations (Fig. 4.2)—and this is where the magic and fun starts!

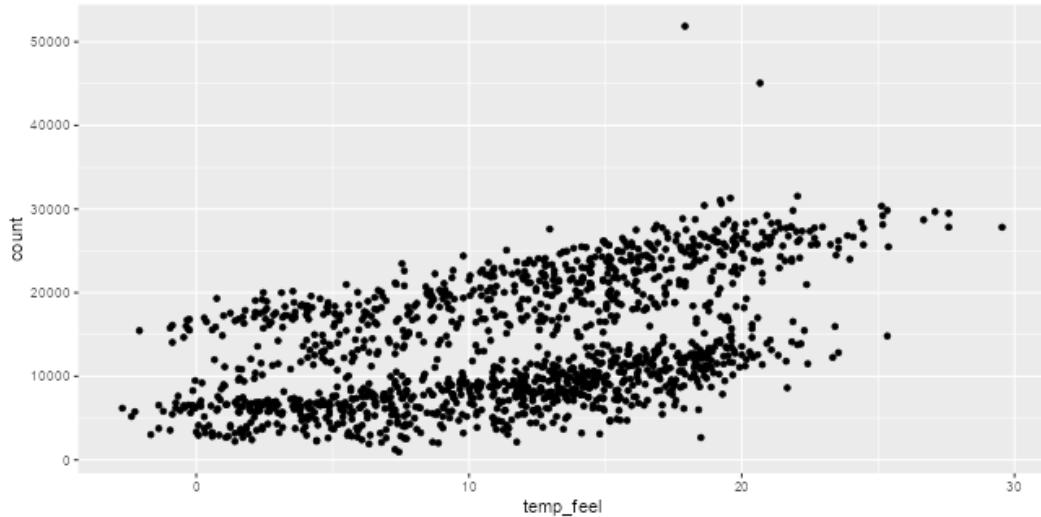


FIGURE 4.1: A basic scatter plot of feels-like temperature and reported TfL bike rents, created with the `{ggplot2}` package.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point() +
  ## add a GAM smoothing
  stat_smooth()
```

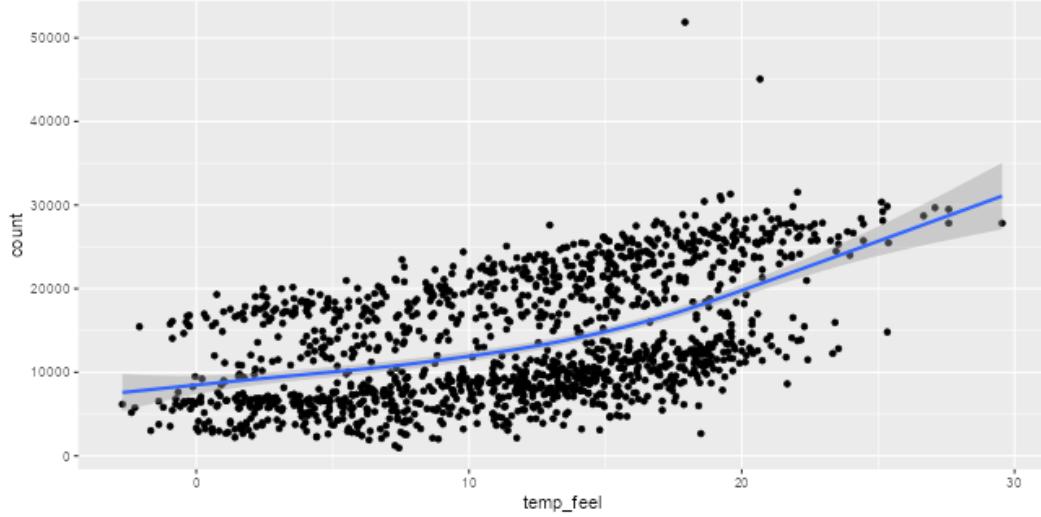


FIGURE 4.2: The same scatter plot, now with an additional GAM smoothing.

By looking at these scatter plots, we can actually identify two different trends for day and night. We can highlight both groups by mapping the `day_night` variable to `color` (Fig. 4.3). Note that the mapping is applied to both, `geom_point()` and `geom_smooth()` and consequently the latter creates two separate smoothings.

```
ggplot(bikes, aes(x = temp_feel, y = count, color = day_night)) +
  geom_point() +
  stat_smooth()
```

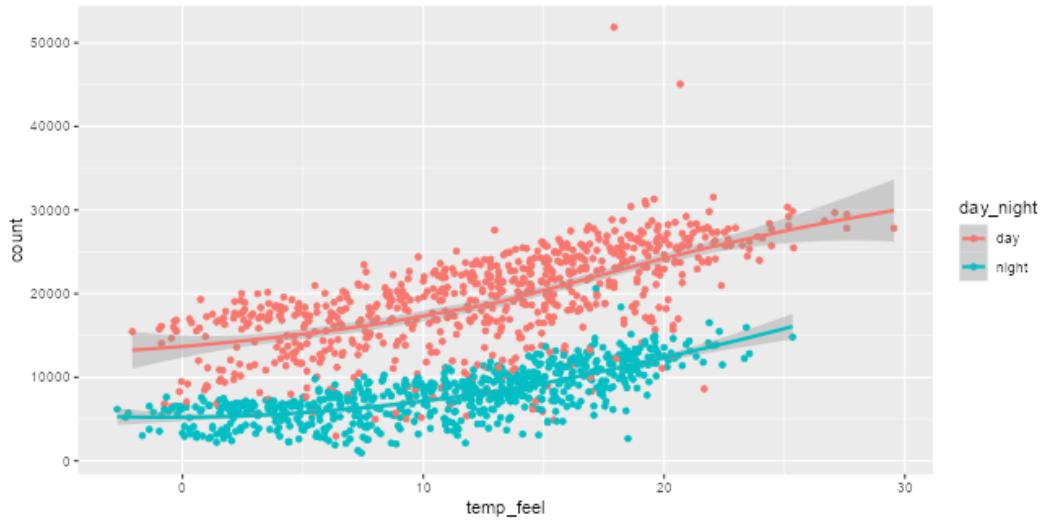


FIGURE 4.3: The points and smoothing lines colored by the time of the day.

Aesthetics can also be defined for each layer and are then applied locally to the respective geometry or statistical transformation only. The `group` aesthetics allows to create subsets without changing the visual appearance (in contrast to aesthetics such as `color` or `shape`).

Furthermore, each layer has its own arguments to change their behavior and appearance. Let's also add some transparency to the points and turn the smoothing into a linear fitting (Fig. 4.4). As we are not mapping aesthetics but setting properties, we have to place those adjustments outside the `aes()` call.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    ## color mapping only applied to points
    aes(color = day_night),
    ## larger points with 50% opacity
    alpha = .5, size = 2
  ) +
  stat_smooth(
    ## invisible grouping to create two trend lines
    aes(group = day_night),
    ## use linear fitting + black smoothing lines
    method = "lm", color = "black"
  )
```

More information on the different geometries is provided in Chapter XYZ and a deep-dive into the power of statistical transformation is Chapter XYZ. The different aesthetics and the differences of global versus local mapping of aesthetics is explained in Chapter XYZ.

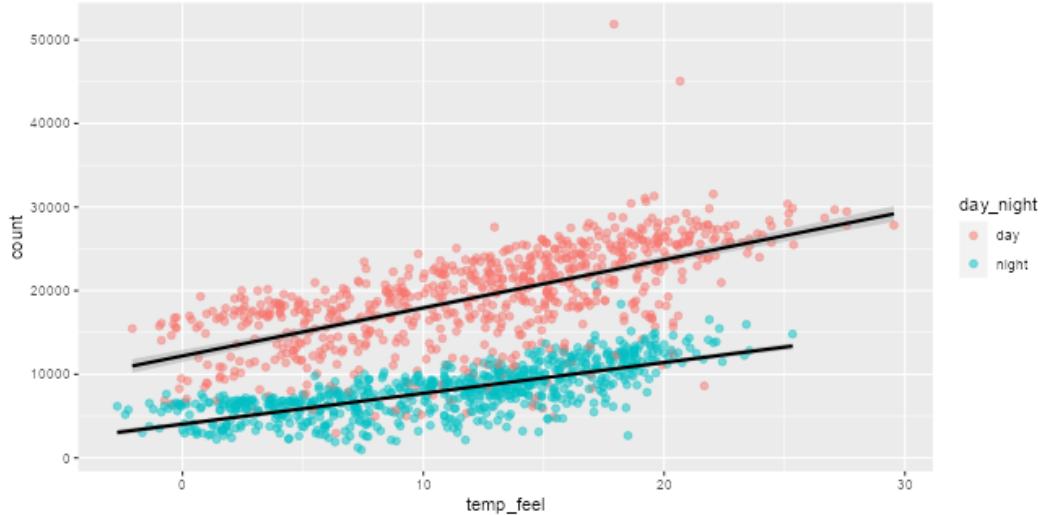


FIGURE 4.4: We can map aesthetics and define properties for each layer individually.

4.3 Create small multiples

A fantastic feature of `{ggplot2}` is its ability to quickly split a single visualization in a set of so-called *small multiples*: the same visualization, no matter how complex it is, is applied to subsets contained in the same data set. In `{ggplot2}`, such small multiples are called *facets* which are conditional on the variables defined in the `facet_*`() function (Fig. 4.5).

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(aes(
    color = day_night), alpha = .5, size = 2
  ) +
  stat_smooth(
    aes(group = day_night), method = "lm", color = "black"
  ) +
  ## small multiples based on time of the day (rows) and year (columns)
  facet_grid(day_night ~ year)
```

Consistency across axes is considered good practice as varying axis ranges are harder to compare and the potential of misleading viewers increases in case the differences keep unnoticed—or even worse are not noticeable. The implementation of `facet` in `{ggplot2}` ensures consistency across all small multiples by default, as illustrated by the empty space in the lower row as there are no values above ~21,000 reported rents during night.

The `facet` functionality also comes with the option to overwrite the default behavior by “freeing” the positional scales. Furthermore, one can ensure equal axis spacing by freeing the space as well (Fig. 4.6). This setting allows to efficiently use the available space while ensuring comparability across plots and decreasing the potential of misleading the viewer. The differences between the two ways to create small multiples, namely `facet_grid()` and `facet_wrap()`, as well as ways to adjust and annotate facets are explained in Chapter XYZ.

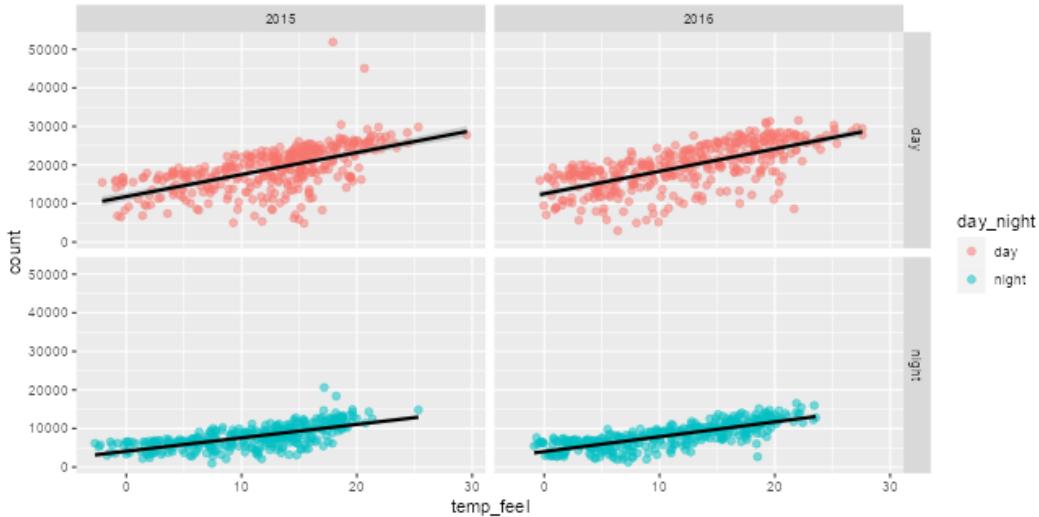


FIGURE 4.5: With the facet functions, a visualization can quickly be split into small multiples.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    aes(color = day_night), alpha = .5, size = 2
  ) +
  stat_smooth(
    aes(group = day_night), method = "lm", color = "black"
  ) +
  facet_grid(
    ## free y axis range + scale heights respectively
    day_night ~ year, scales = "free_y", space = "free_y"
  )
```

4.4 Change the axis scaling

For every aesthetic, `{ggplot2}` applies so-called *scales* that translate between variable ranges (data) and property ranges (aesthetic). For the positional axes, a set of `scale_x_*`() and `scale_y_*`() controls their behavior. The `breaks` argument defines the placement of the axes ticks while the `labels` argument controls the labels next to those ticks. The labels can be either overwritten by a vector of the same length as the breaks (as for `x` in our example) or a function that returns a vector based on the breaks (as for `y` in our example). There are many more arguments such as `expand` to control the padding towards the ends of the respective axis or `trans` to transform the scale. You can read more about how to control scales in Chapter XYZ and about label adjustment and styling in Chapter XYZ.

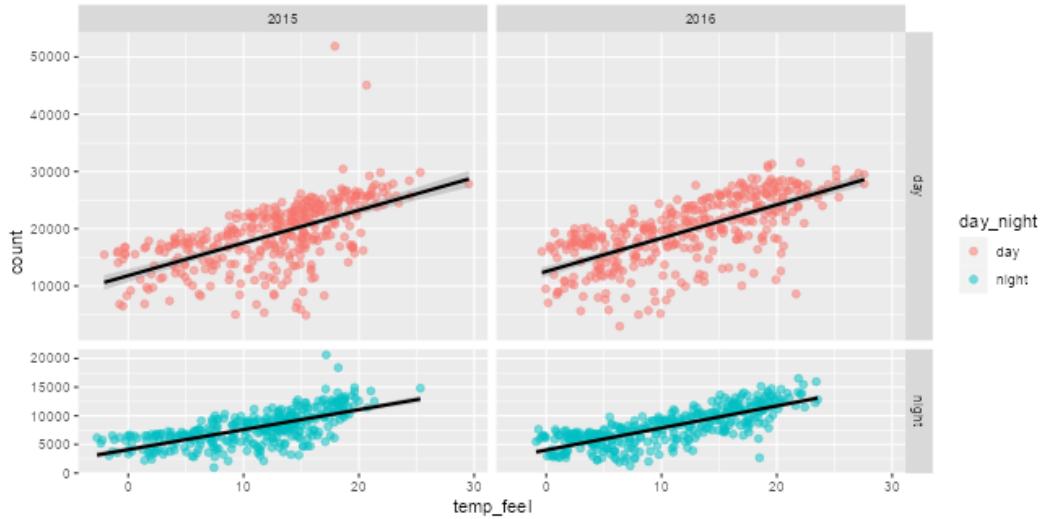


FIGURE 4.6: {ggplot2} even allows to free the axis range—while ensuring equal axis spacing.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    aes(color = season), alpha = .5, size = 2
  ) +
  stat_smooth(
    aes(group = day_night), method = "lm", color = "black"
  ) +
  facet_grid(
    day_night ~ year, scales = "free_y", space = "free_y"
  ) +
  ## x axis: add °C symbol + 5°C spacing
  scale_x_continuous(
    breaks = -1:6*5, labels = function(x) paste0(x, "°C"), expand = c(mult = 0, add = 1)
  ) +
  ## y axis: add a thousand separator + consistent spacing across rows
  scale_y_continuous(
    breaks = 0:5*10000, labels = scales::label_comma(), expand = c(mult = .1, add = 0)
  )
```

4.5 Use a custom color palette

The `scales` do not only control the behavior of the axes but also all other aesthetics such as `color`, `shape` or `alpha`. In case of categorical colors, `scale_color_manual()` enables us to overwrite the default color set by passing a vector of colors.

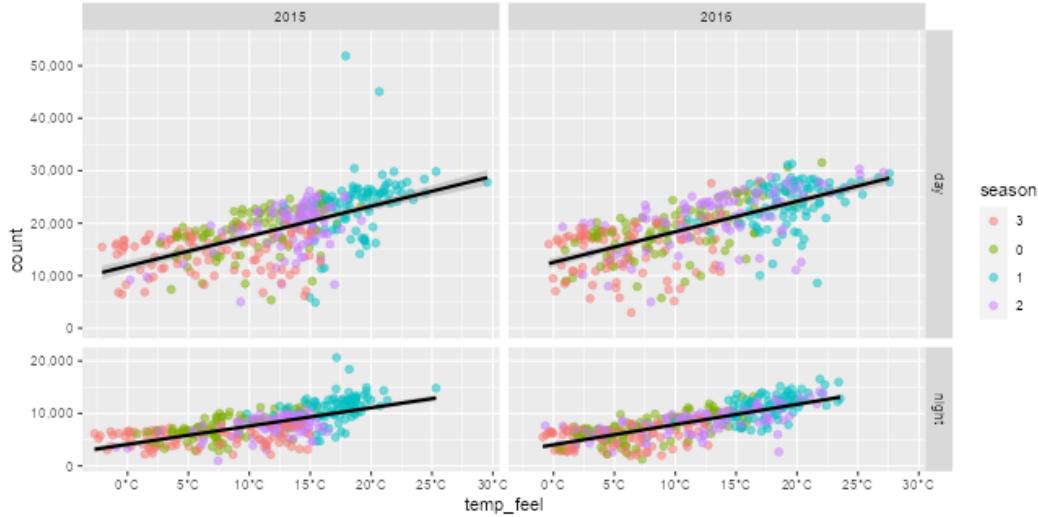


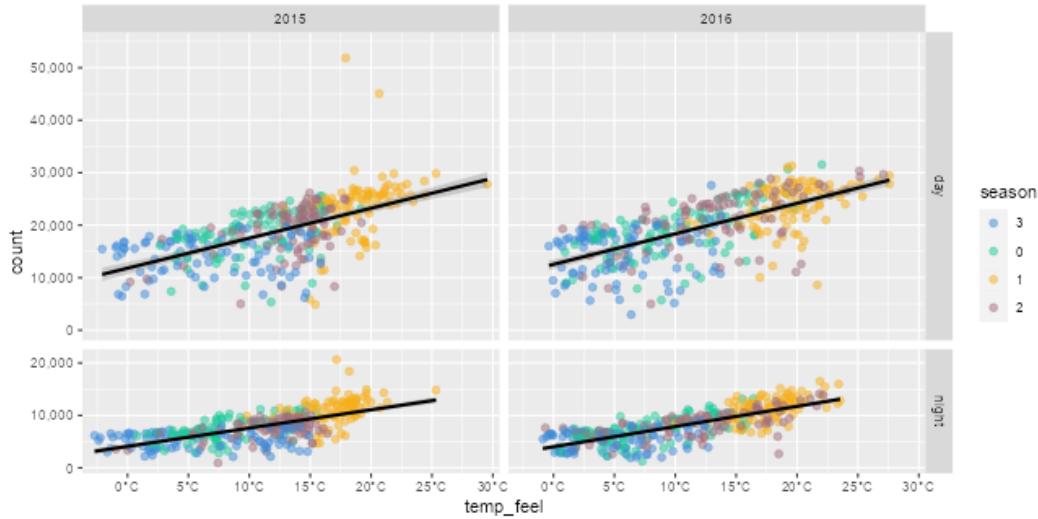
FIGURE 4.7: To adjust the formatting of axis labels, the respective axis needs to be addressed via the `scale_*`() functions.

```

g1 <- ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    aes(color = season), alpha = .5, size = 2
  ) +
  stat_smooth(
    aes(group = day_night), method = "lm", color = "black"
  ) +
  facet_grid(
    day_night ~ year, scales = "free_y", space = "free_y"
  ) +
  scale_x_continuous(
    breaks = -1:6*5, labels = function(x) paste0(x, "°C"), expand = c(mult = 0, add = 1)
  ) +
  scale_y_continuous(
    breaks = 0:5*10000, labels = scales::label_comma(), expand = c(mult = .1, add = 0)
  ) +
  ## use a custom color palette for season colors
  scale_color_manual(
    values = c("#3c89d9", "#1ec99b", "#F7B01B", "#a26e7c")
  )

g1

```



The colors will be applied in order, i.e. the levels of the factor or alphabetical in case of character variables. To ensure correct mapping, one can also use a named vector.

You will learn more about the different color and fill scales and how to manipulate colors in Chapter XYZ. The chapter also features an example how to create a corporate color or fill scale to ensure consistent color use.

Furthermore, we are storing the current plot in an object named `g1`. This `ggplot` object can be extended afterwards. Storing intermediate `ggplot` outputs is especially helpful if the code becomes rather long and in case you have multiple variants of the same chart, for example when building up a chart stepwise. Another common use case is the creation of graphics in multiple languages.

4.6 Adjust labels

For now, we have ignored that `{ggplot2}` uses the variable names as specified in our `bikes` object for data-related labels. Resist the temptation to change the variable names in your original data set! We can easily overwrite those by referring to the respective aesthetic (`x`, `y`, and `color`) in the `labs()` function which we add to our plot. Here, we can also specify a title, subtitle, caption, and tag (subtitle and tag are not shown in the following example; Fig. 4.8).

```
g2 <- g1 +
  labs(
    ## overwrite axis and legend titles
    x = "Average Feels-Like Temperature", y = NULL, color = NULL,
    ## add plot title and caption
    title = "Trends of reported bike rents versus feels-like temperature in London.",
    caption = "Data: TfL (Transport for London), Jan 2015-Dec 2016"
  )
```

g2

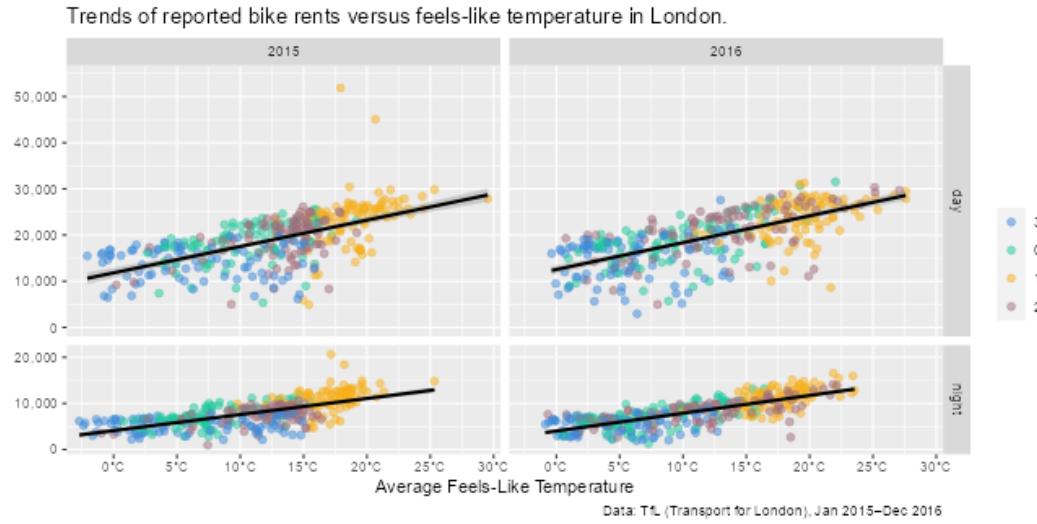


FIGURE 4.8: We can overwrite the default labels with the `labs()` which also allows to add a title, subtitle, caption, and tag to the graphic.

Let's also overwrite the cryptic numeric encoding of the seasons. Similar to the positional scales, we can pass a vector or function to the `labels` argument in the `scale_color_*`() component:

```
g3 <- g2 +
  scale_color_manual(
    values = c("#3c89d9", "#1ec99b", "#F7B01B", "#a26e7c"),
    labels = c("Winter", "Spring", "Summer", "Autumn")
  )

g3
```

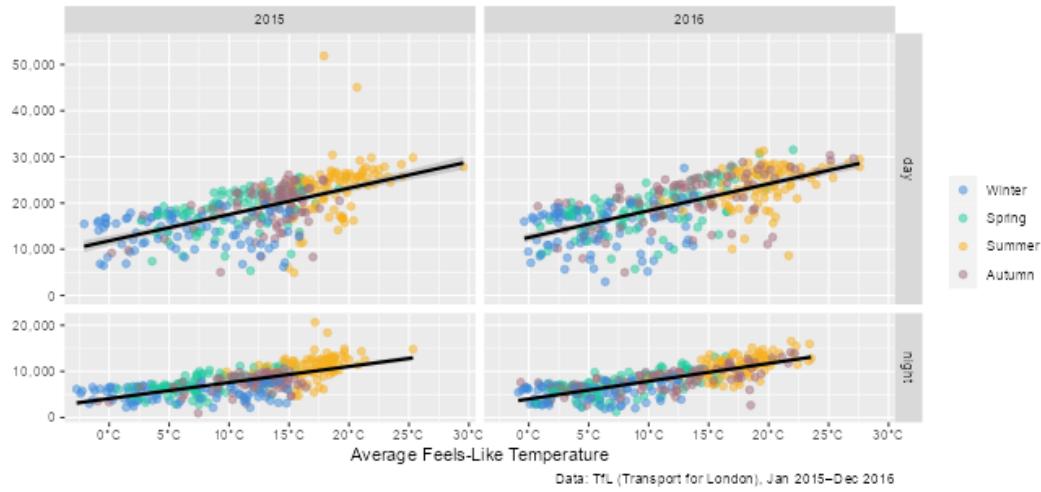
Note that the code is returning a message informing you that the present color scale has been replaced. As you can only apply one scale per aesthetic, we also have to pass the color vector again to use the custom colors from before.

4.7 Apply a complete theme

`{ggplot2}` comes with a set of so-called *complete themes*. We can add one of these to overwrite the `theme_grey()` by one of the others (run `?theme_grey` for a full list of available complete themes).

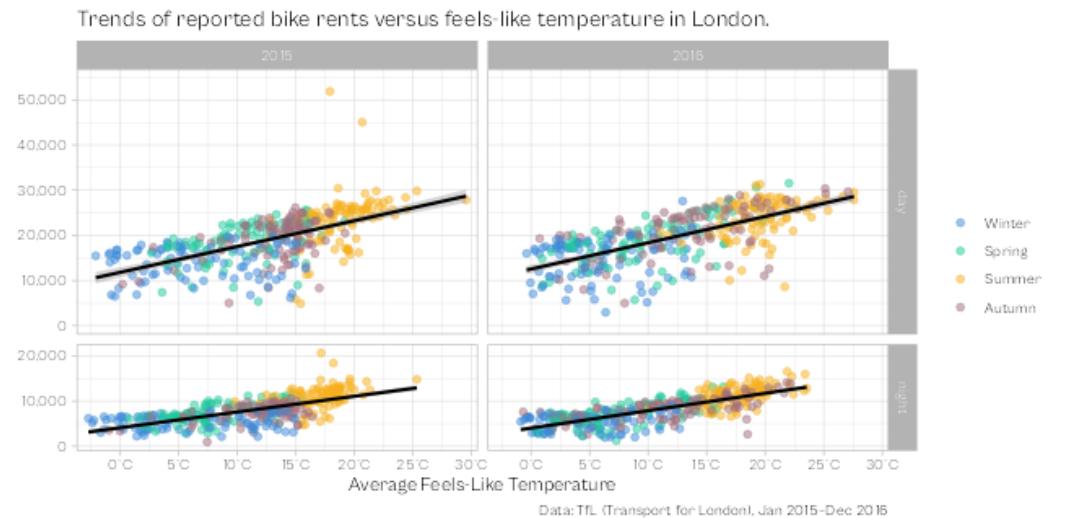
When adding a theme to a ggplot, we can already overwrite the `base_size` of the theme elements, which translates to the size of text labels and line widths. Furthermore, we can

Trends of reported bike rents versus feels-like temperature in London.

**FIGURE 4.9:** By passing a vector of the same length as there are seasons to the `labels` argument of the `scale_color_*`() function, we can overwrite the labels used in the legend.

set a non-default typeface by supplying a locally installed typeface in the `base_family` argument.

```
g3 +
  ## add theme with a custom font + larger element sizes
  theme_light(
    base_size = 12, base_family = "Cabinet Grotesk"
  )
```

**FIGURE 4.10:** The graphic now comes with a new look and a non-default font used for all text labels.

4.8 Customize the theme

The complete themes follow a set of rules that overwrite the choices made for the default `theme_grey()`. By using a `theme()` call after (!) adding a complete theme, one can adjust the appearance of single elements (e.g. `plot.title` or `panel.grid.major.x`) as well as set some pre-set options such as the positions of titles, captions and legends (`plot.title.position`, `plot.caption.position` and `legend.position`).

In general, theme elements can be of class `text` (e.g. `plot.title`), `line` (e.g. `panel.grid.major.x`), or `rectangle` (e.g. `panel.background`). In addition, we can use `element_blank()` to remove an element entirely as set for `panel.grid.minor` in our example.

```
g3 +
  theme_light(
    base_size = 12, base_family = "Cabinet Grotesk"
  ) +
  ## theme adjustments
  theme(
    plot.title.position = "plot", ## left-align title
    plot.caption.position = "plot", ## right-align title
    plot.title = element_text(face = "bold", size = rel(1.3)), ## larger bold title
    axis.text = element_text(family = "Tabular"), ## monospaced font for axes
    axis.title.x = element_text(## left-aligned, grey x axis label
      hjust = 0, color = "grey30", margin = margin(t = 12)
    ),
    strip.text = element_text(## larger bold facet labels
      face = "bold", size = rel(1.15)
    ),
    panel.grid.major.x = element_blank(), ## no vertical grid lines
    panel.grid.minor = element_blank(), ## no minor grid lines
    legend.position = "top", ## place legend above plot
    legend.text = element_text(size = rel(1)) ## larger legend labels
  )
)
```

The book covers themes in full detail in Chapter XYZ. Here you learn how to leverage complete themes, which other complete themes are provided by extension packages, and how to create your own custom, corporate theme to apply a cohesive and easily reproducible style to all of your graphics.

RECAP

LEAD OVER TO NEXT CHAPTER

Trends of reported bike rents versus feels-like temperature in London.

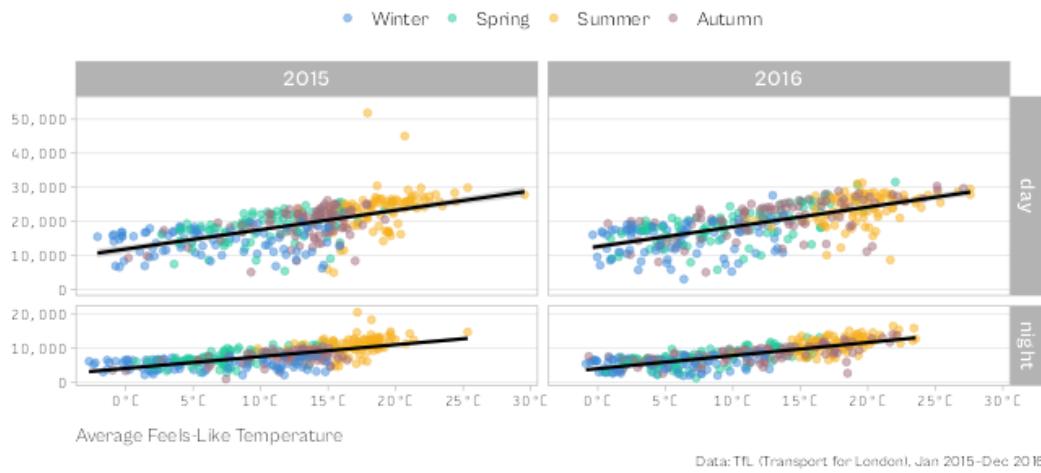


FIGURE 4.11: After applying one of the complete themes, one can further style the theme to the personal needs.

5

The Concepts of {ggplot2}

Creating a basic ggplot

First, to be able to use the functionality of ggplot2 we have to load the package (which we can also load via the tidyverse package collection):

```
#library(tidyverse)
library(ggplot2)
```

The syntax of ggplot2 is very different from plotting functionality of provided by base R. We always start to define a plotting object by calling `ggplot(data = df)` which just tells ggplot2 that we are going to work with that data. In most cases, you might want to plot two variables—one on the x and one on the y axis. These are positional aesthetics and thus we add `aes(x = var1, y = var2)` to the `ggplot()` call (yes, the `aes()` stands for aesthetics). However, there are also cases where one has to specify only one or even three or more variables.

We specify the data outside `aes()` and add the variables that ggplot maps the aesthetics to inside `aes()`.

Here, we map the variable date to the x position and the variable temp to the y position. Later, we will also map variables to all kind of other aesthetics such as color, size, and shape.

```
ggplot(data = bikes, mapping = aes(x = temp_feel, y = count))
```

Only a panel is created when running this. Why? This is because ggplot2 does not know how we want to plot that data—we still need to provide a geometry!

ggplot2 allows you to store the current ggobject in a variable of your choice by assigning it to a variable, in our case called `g`. You can extend this ggobject later by adding other layers, either all at once or by assigning it to the same or another variable.

Thanks to implicit matching we can rewrite the code as follows: `ggplot(bikes, aes(date, count))`). But be aware that the order matters! I suggest to use a mixture if you feel confident to do so.

```
ggplot(bikes, aes(x = temp_feel, y = count))
```

Omitting the arguments `data` and `mapping` saves you a ton of typing when creating dozens to hundreds ggplot's per day but being specific about x and y is a good idea. This is the syntax I am using throughout the book.

There are many, many different geometries (called geoms because each function usually

starts with `geom_`) one can add to a `ggplot` by default (see here for a full list) and even more provided by extension packages (see here for a collection of extension packages). Let's tell `ggplot2` which style we want to use, for example by adding `geom_point()` to create a scatter plot:

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point()
```

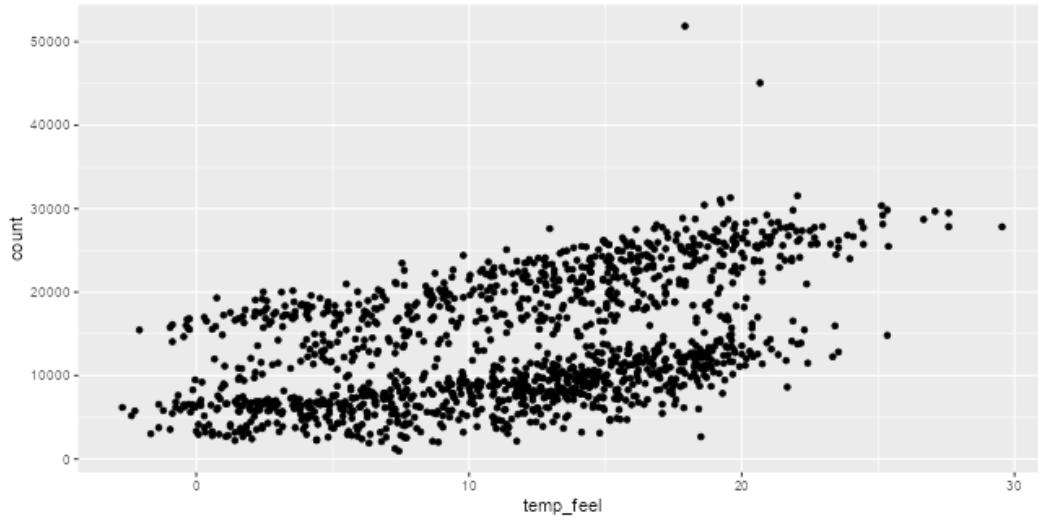


FIGURE 5.1: A default scatter plot of temperature measured in Chicago created with the `ggplot2` package.

One can also combine several geometric layers—and this is where the magic and fun starts!

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point() +
  geom_smooth()
```

That's it for now about geometries. No worries, we are going to learn several plot types in Chapter charts.

Change Properties of Geometries

Within the `geom_*` command, you already can manipulate visual aesthetics such as the color, shape, and size of your points. Let's turn all points to large fire-red diamonds!

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(color = "firebrick", shape = "diamond", size = 2)
```

`ggplot2` understands both color and colour as well as the short version col.

You can use preset colors (here is a full list) or hex color codes, both in quotes, and even RGB/RGBA colors by using the `rgb()` function.

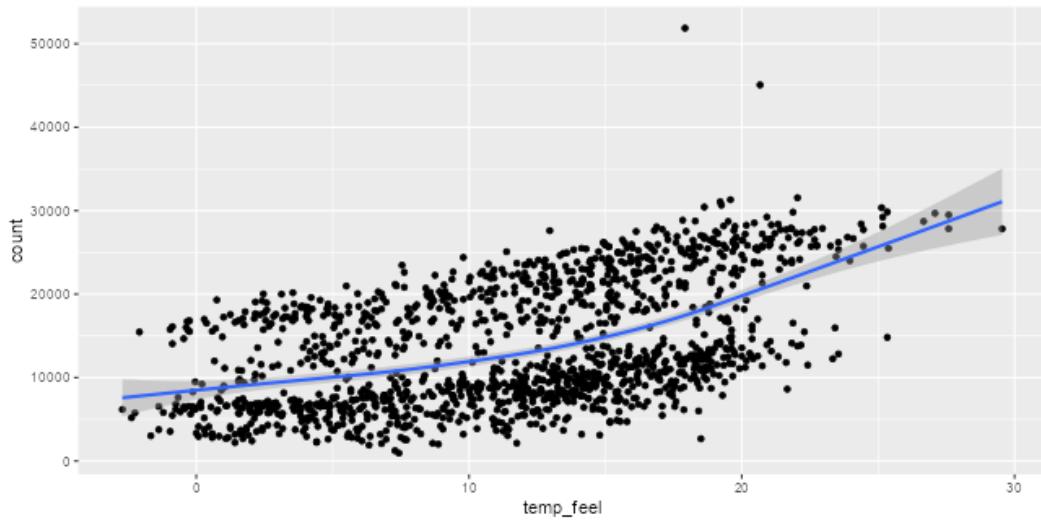


FIGURE 5.2: Again, the same data, now shown as a connected scatterplot as a combination of points and lines.

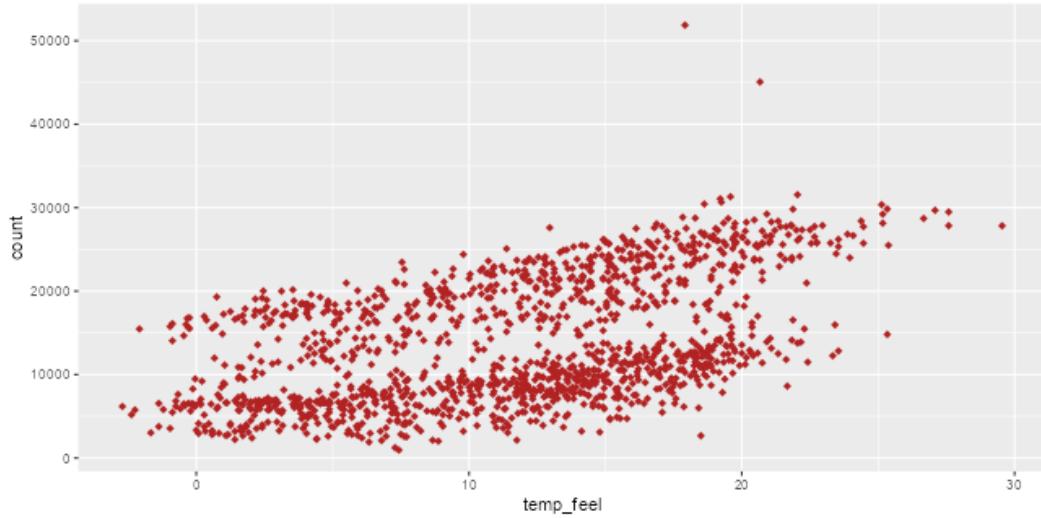


FIGURE 5.3: We can change the appearance of the geometry, here illustrated by turning the black dots in larger, red diamonds.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(color = "#b22222", shape = "diamond", size = 2)

ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(color = rgb(178, 34, 34, maxColorValue = 255), shape = "diamond", size = 2)
```

Each geom comes with its own properties (called arguments) and the same argument may result in a different change depending on the geom you are using.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(color = "firebrick", shape = "diamond", size = 2) +
  geom_smooth(formula = y ~ poly(x, 4), se = FALSE,
             color = "gray40", size = 2)
```

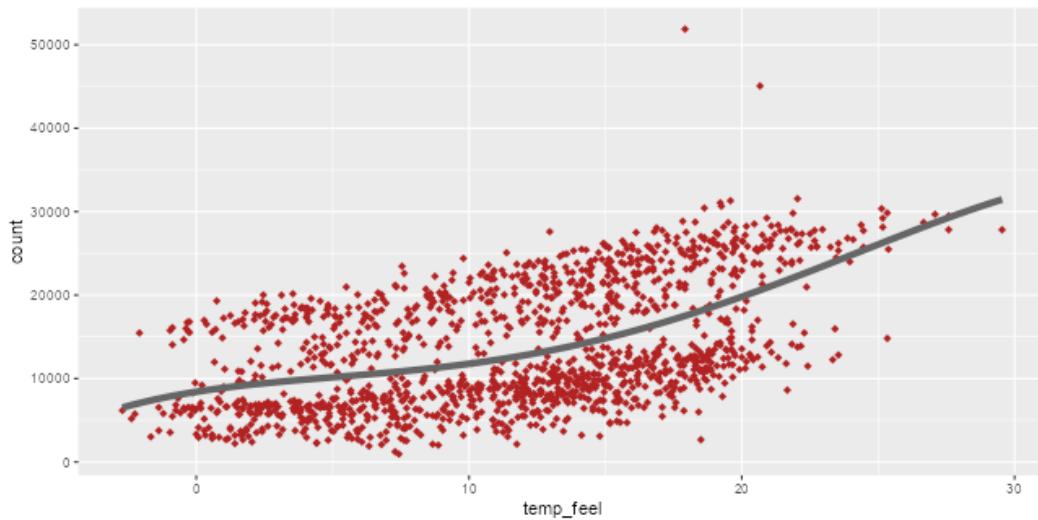


FIGURE 5.4: You can style each geometrical layer on its own. Each geometry also comes with a set of individual properties.

Mapping Data to Aesthetics

You already have seen two positional aesthetics, x and y that can be used in combination with the aes() function to map variables to the x- and y-axis, respectively. There are many more aesthetic attributes we are going to use throughout the book. Some are related to positions such as ymin and ymax while others change the appearance of the layer based on the variables they are mapped to such as color and shape.

As with the x and y, the mapping needs to be wrapped into aes() so that ggplot knows that you are referring to columns of your data set. There are two different levels on which you can apply aesthetic mappings: either for all layers or for individual layers only.

In case we want to color our points based on the period of the day to reveal the two patterns, we add aes(color = day_night) to our point layer geom_point():

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(aes(color = day_night)) +
  geom_smooth()
```

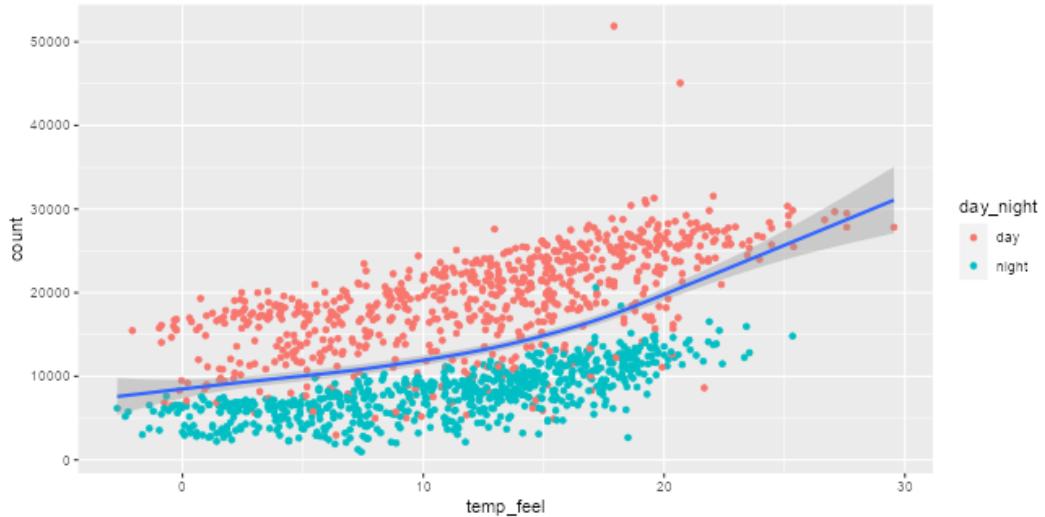


FIGURE 5.5: By applying aesthetic mapping to the `day_night` variable in the point layers, the two groups can be identified. Note that `ggplot2` automatically adds a legend to the plot.

By supplying the aesthetic mapping inside `ggplot()` all layers are applying the same mapping. Consequently, the grouping and coloring is also used for the smoothing:

```
ggplot(bikes, aes(x = temp_feel, y = count, color = day_night)) +
  geom_point() +
  geom_smooth()
```

The aesthetic mapping to both, `geom_point()` and `geom_smooth()`, is also reflected in the legend which now shows points, lines, and ribbons.

Note that you can overwrite the mapping specified in `ggplot()` in individual layers by either supplying a fixed value to that property:

```
ggplot(bikes, aes(x = temp_feel, y = count, color = day_night)) +
  geom_point(color = "black") +
  geom_smooth()

ggplot(bikes, aes(x = temp_feel, y = count, color = day_night)) +
  geom_point() +
  geom_smooth(color = "black")
```

Replace the default ggplot2 theme

And to illustrate some more of ggplot's versatility, let's get rid of the grayish default ggplot2

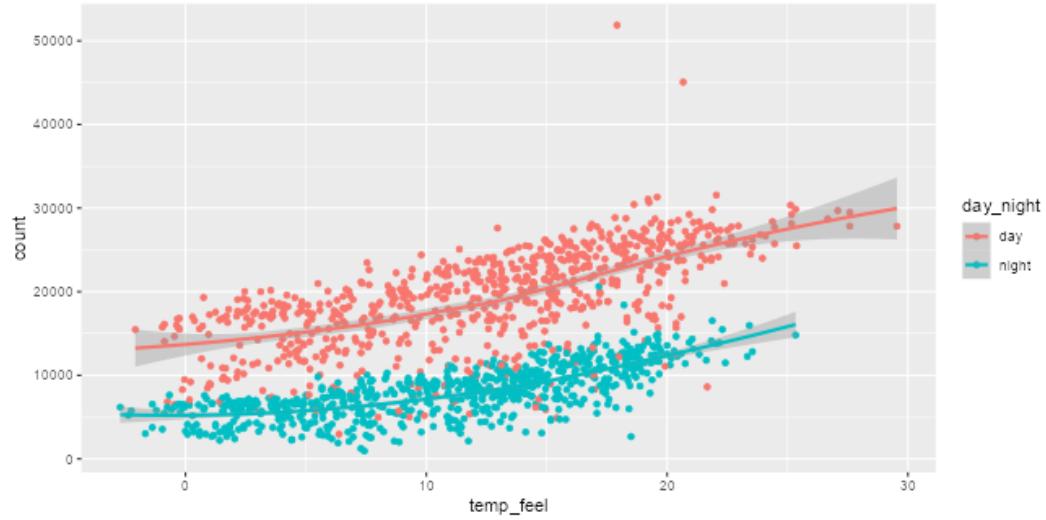


FIGURE 5.6: Both, the points and the smoothing lines are encoded by daytime. Because of that, the smoothing is hard to see and best visible in the legend.

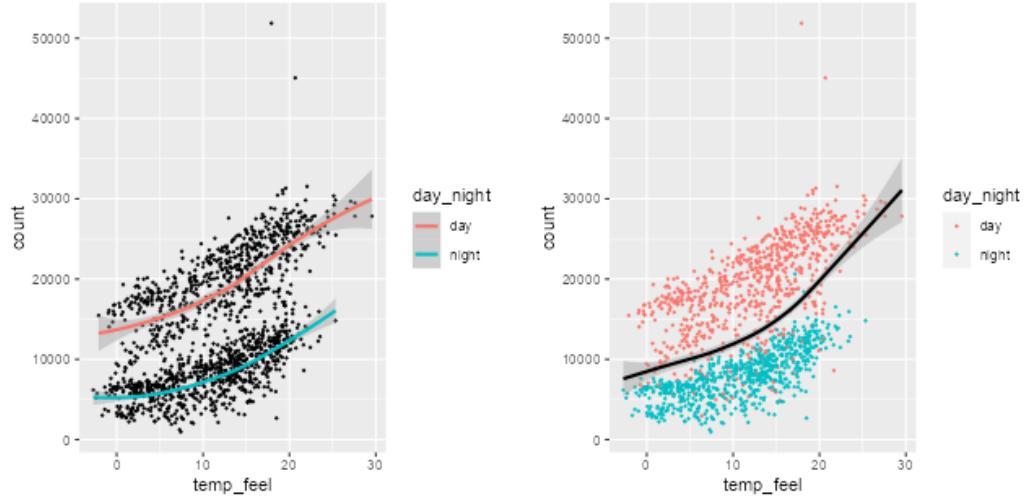


FIGURE 5.7: Overwriting the color encoding in the point (left) or the smoothing (right). Note that the grouping is removed as well which changes the behaviour of the smoothing geom.

look by setting a different built-in theme, e.g. `theme_bw()`. One can add a theme directly to a ggplot composition or setting a theme globally—by calling `theme_set()` all following plots will have the same black'n'white theme. In the same step, we can also increase the `base_size` of the plot elements. The default base size is 11 and tends to be too small, at least for my workflow.

```
theme_set(theme_bw(base_size = 16))

ggplot(bikes, aes(x = temp_feel, y = count, color = day_night)) +
  geom_point()
```

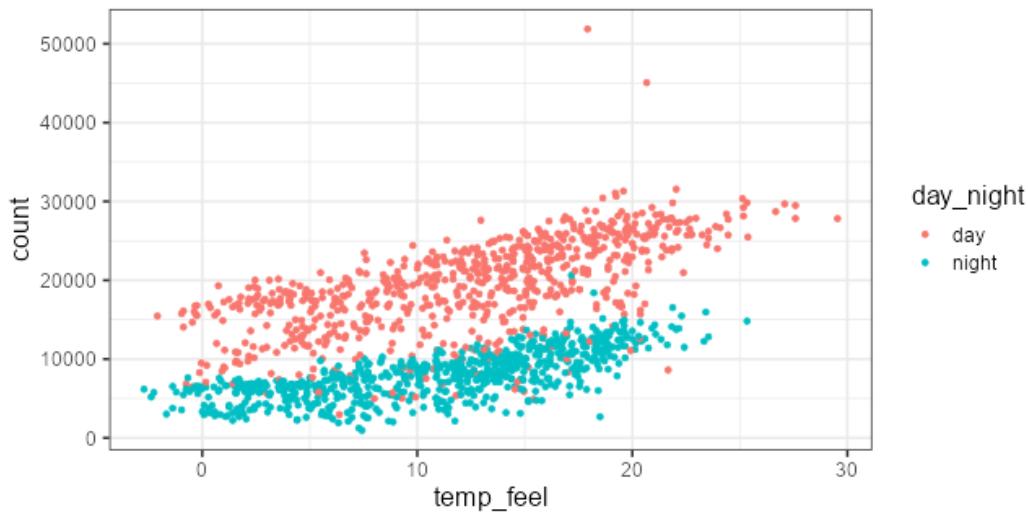


FIGURE 5.8: We can change the appearance for all following plots generated with `ggplot2` by globally setting another theme via `theme_set()`.

You can find more on how to use built-in themes and how to customize themes in the section `themes`. From the next chapter on, we will also use the `theme()` function to customize particular elements of the theme.

`theme()` is an essential command to manually modify all kinds of theme elements (texts, rectangles, and lines). To see which details of a ggplot theme can be modified have a look here—and take some time, this is a looong list.

Export ggplot2 as graphic files

Part I

How To Polish Your Graphics

6

Quick Steps to Improve Your Graphic

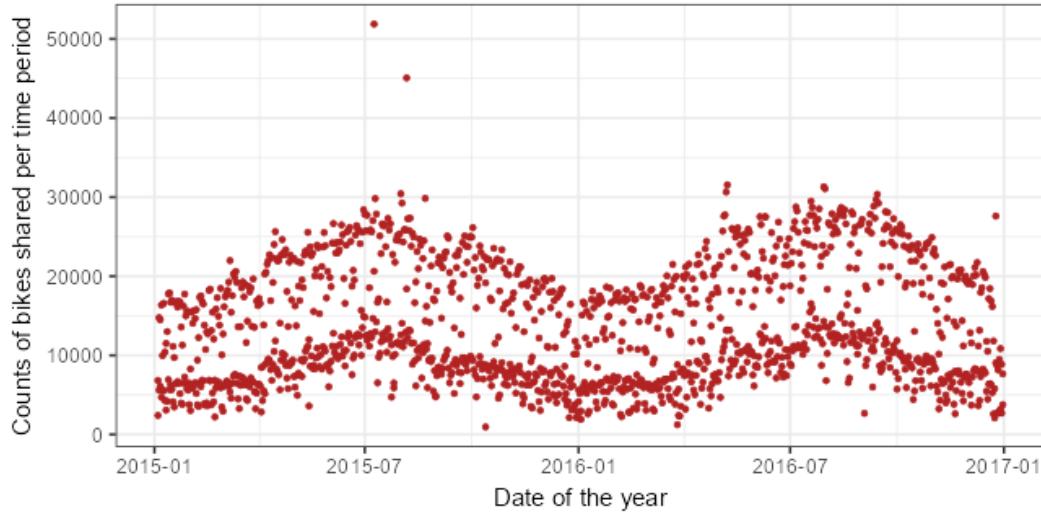
7

Working with Axes

7.0.0.0.1 Increase Space between Axis and Axis Titles

`theme()` is an essential command to modify particular theme elements (texts and titles, boxes, symbols, backgrounds, ...). We are going to use them a lot! For now, we are going to modify text elements. We can change the properties of all or particular text elements (here axis titles) by overwriting the default `element_text()` within the `theme()` call:

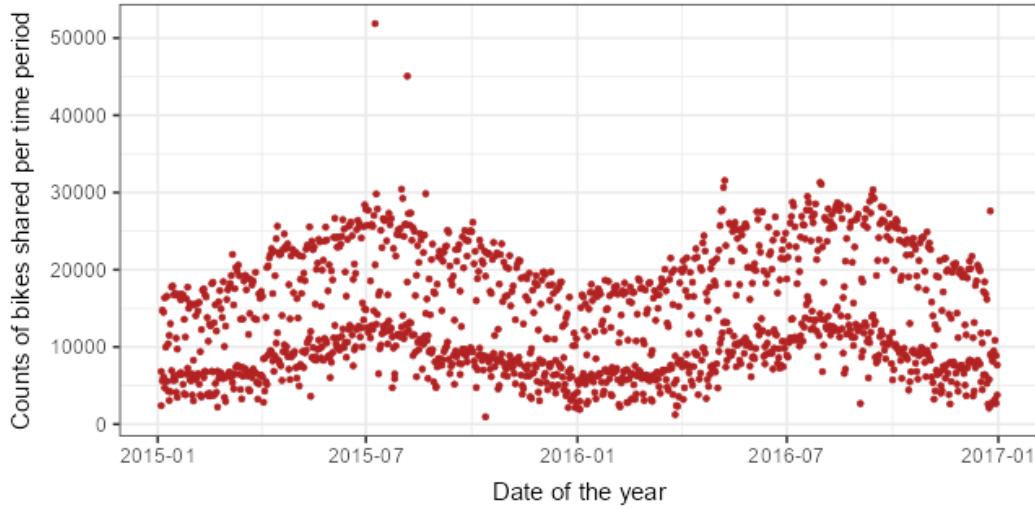
```
ggplot(bikes, aes(x = date, y = count)) +  
  geom_point(color = "firebrick") +  
  labs(x = "Date of the year",  
       y = "Counts of bikes shared per time period") +  
  theme(axis.title.x = element_text(vjust = 0, size = 15),  
        axis.title.y = element_text(vjust = 2, size = 15))
```



`vjust` refers to the vertical alignment, which usually ranges between 0 and 1 but you can also specify values outside that range. Note that even though we move the axis title on the y axis horizontally, we need to specify `vjust` (which is correct from the label's perspective). You can also change the distance by specifying the margin of both text elements:

```
ggplot(bikes, aes(x = date, y = count)) +  
  geom_point(color = "firebrick") +  
  labs(x = "Date of the year",  
       y = "Counts of bikes shared per time period") +
```

```
theme(axis.title.x = element_text(margin = margin(t = 10), size = 15),
      axis.title.y = element_text(margin = margin(r = 10), size = 15))
```



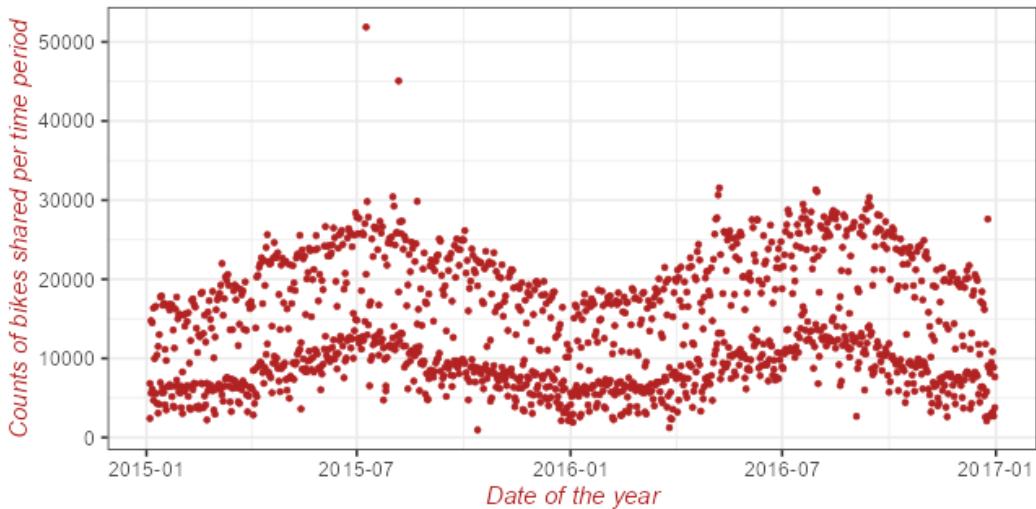
The labels `t` and `r` within the `margin()` object refer to *top* and *right*, respectively. You can also specify the four margins as `margin(t, r, b, l)`. Note that we now have to change the right margin to modify the space on the y axis, not the bottom margin.

A good way to remember the order of the margin sides is “*t-r-ou-b-l-e*”.

7.0.0.0.2 Change Aesthetics of Axis Titles

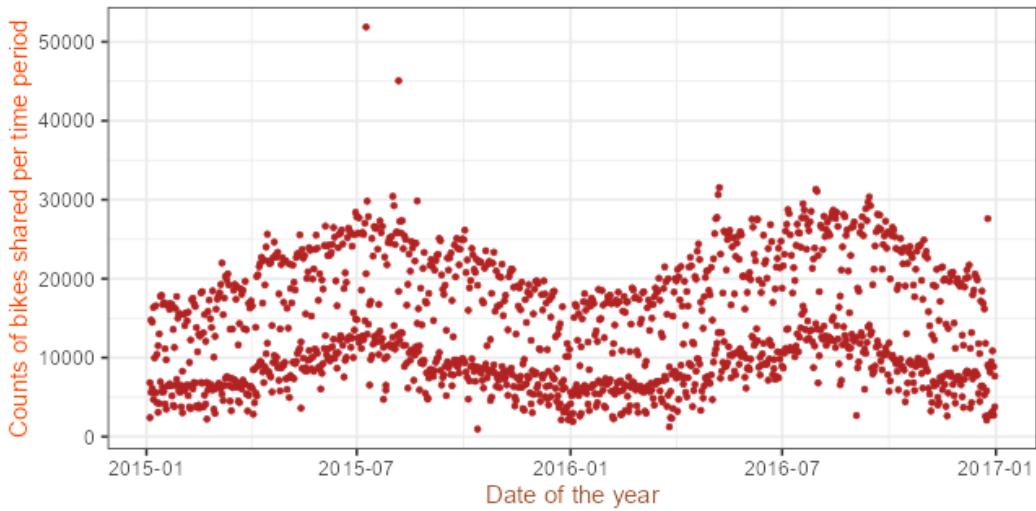
Again, we use the `theme()` function and modify the element `axis.title` and/or the subordinated elements `axis.title.x` and `axis.title.y`. Within the `element_text()` we can for example overwrite the defaults for `size`, `color`, and `face`:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.title = element_text(size = 15, color = "firebrick",
                                  face = "italic"))
```



The `face` argument can be used to make the font bold or italic or even `bold.italic`.

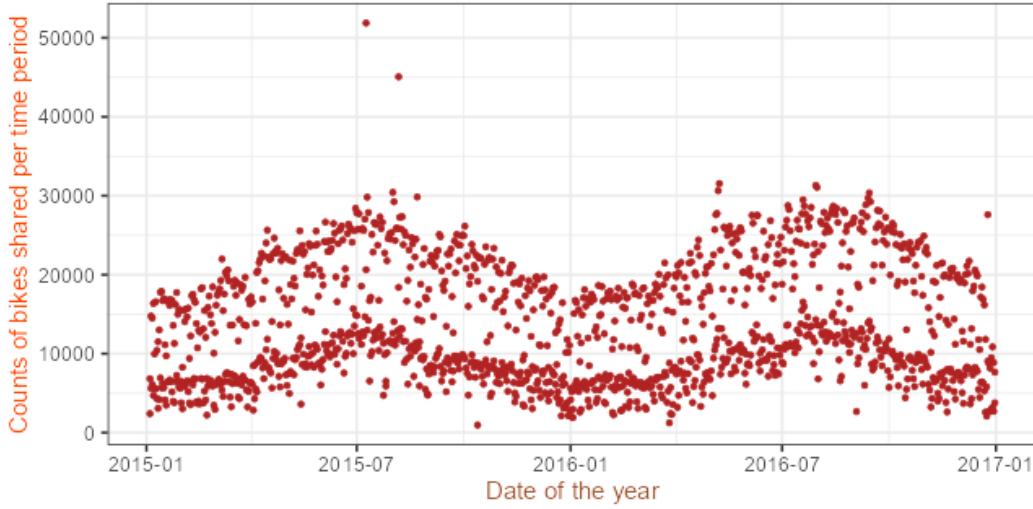
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.title.x = element_text(color = "sienna", size = 15),
        axis.title.y = element_text(color = "orangered", size = 15))
```



You could also use a combination of `axis.title` and `axis.title.y`, since `axis.title.x` inherits the values from `axis.title`. Expand to see example.

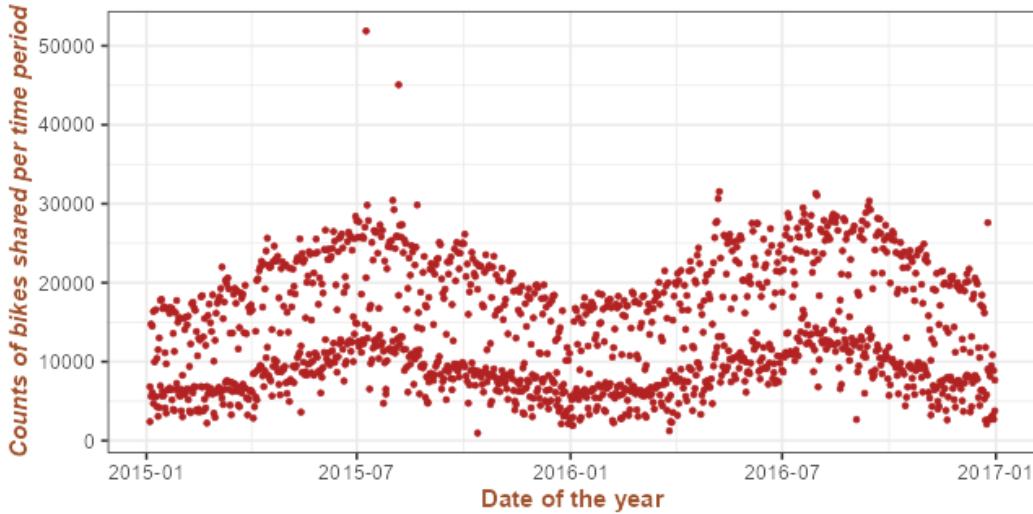
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
```

```
y = "Counts of bikes shared per time period") +
theme(axis.title = element_text(color = "sienna", size = 15),
axis.title.y = element_text(color = "orangered", size = 15))
```



One can modify some properties for both axis titles and other only for one or properties for each on its own:

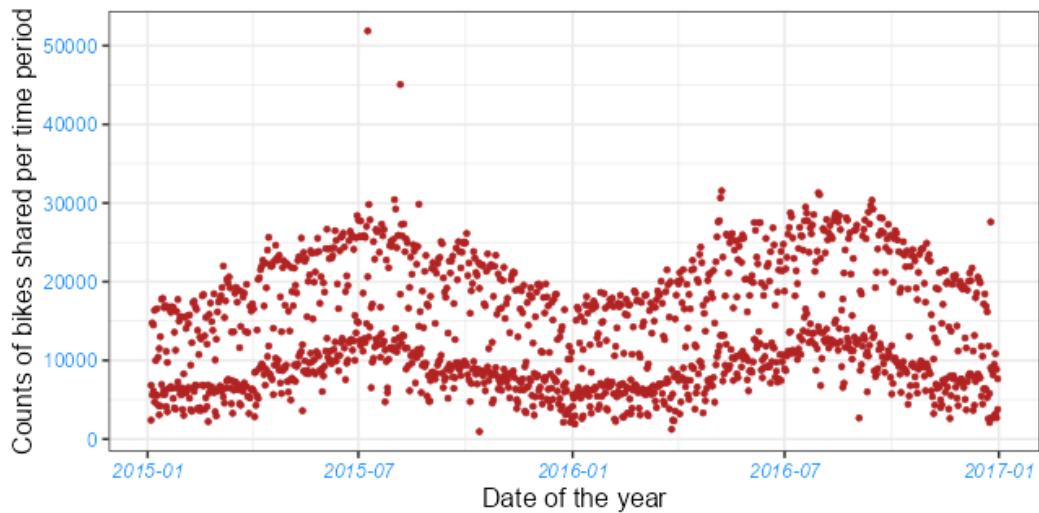
```
ggplot(bikes, aes(x = date, y = count)) +
geom_point(color = "firebrick") +
labs(x = "Date of the year",
y = "Counts of bikes shared per time period") +
theme(axis.title = element_text(color = "sienna", size = 15, face = "bold"),
axis.title.y = element_text(face = "bold.italic"))
```



7.0.0.0.3 Change Aesthetics of Axis Text

Similarly, you can also change the appearance of the axis text (here *the numbers*) by using `axis.text` and/or the subordinated elements `axis.text.x` and `axis.text.y`:

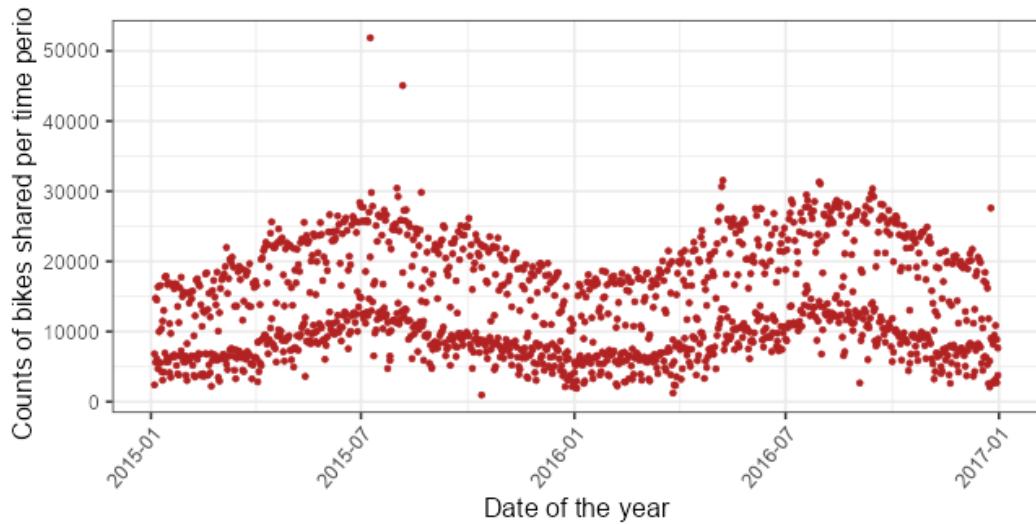
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.text = element_text(color = "dodgerblue", size = 12),
        axis.text.x = element_text(face = "italic"))
```



7.0.0.0.4 Rotate Axis Text

Specifying an `angle` allows you to rotate any text elements. With `hjust` and `vjust` you can adjust the position of the text afterwards horizontally (0 = left, 1 = right) and vertically (0 = top, 1 = bottom):

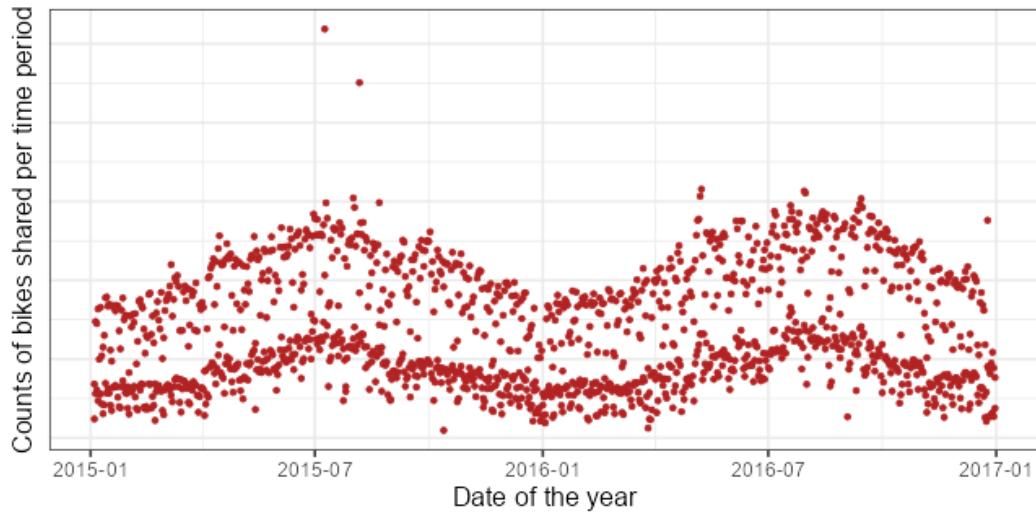
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.text.x = element_text(angle = 50, vjust = 1, hjust = 1, size = 12))
```



7.0.0.0.5 Remove Axis Text & Ticks

There may be rarely a reason to do so—but this is how it works:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.ticks.y = element_blank(),
        axis.text.y = element_blank())
```



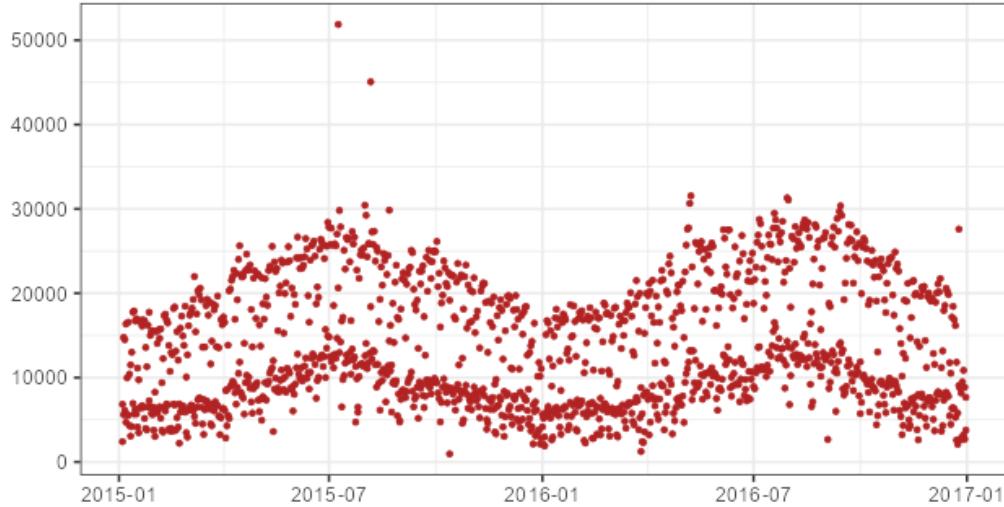
I introduced three theme elements—text, lines, and rectangles—but actually there is one more: `element_blank()` which removes the element (and thus is not considered an official element).

If you want to get rid of a theme element, the element is always `element_blank()`.

7.0.0.6 Remove Axis Titles

We could again use `theme_blank()` but it is way simpler to just remove the label in the `labs()` (or `xlab()`) call:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = NULL, y = "")
```

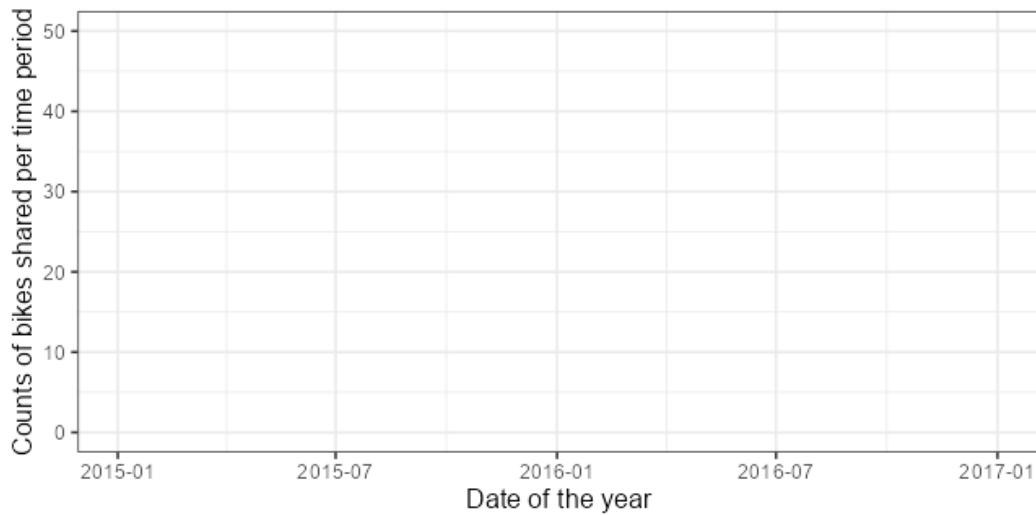


Note that `NULL` removes the element (similarly to `element_blank()`) while empty quotes `" "` will keep the spacing for the axis title and simply print nothing.

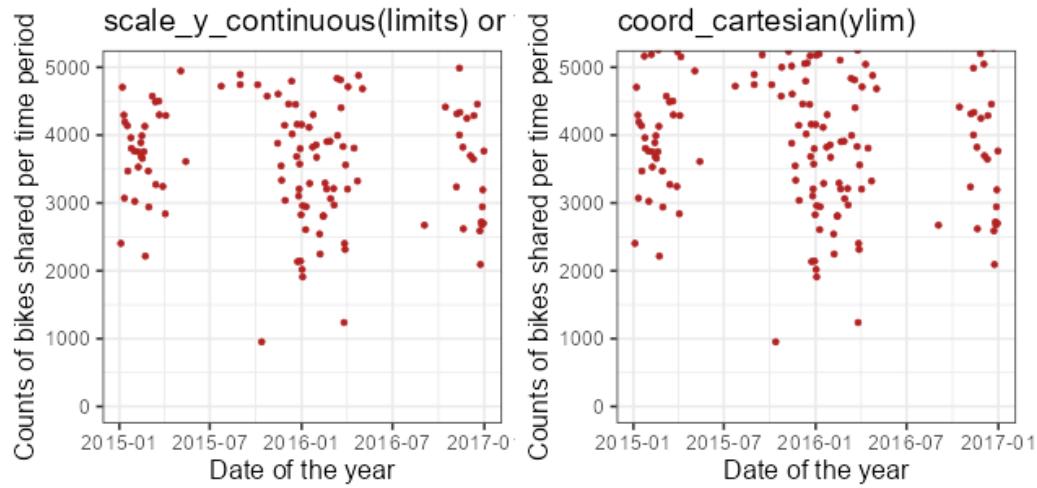
7.0.0.7 Limit Axis Range

Sometimes you want to zoom into take a closer look at some range of your data. You can do this without subsetting your data:

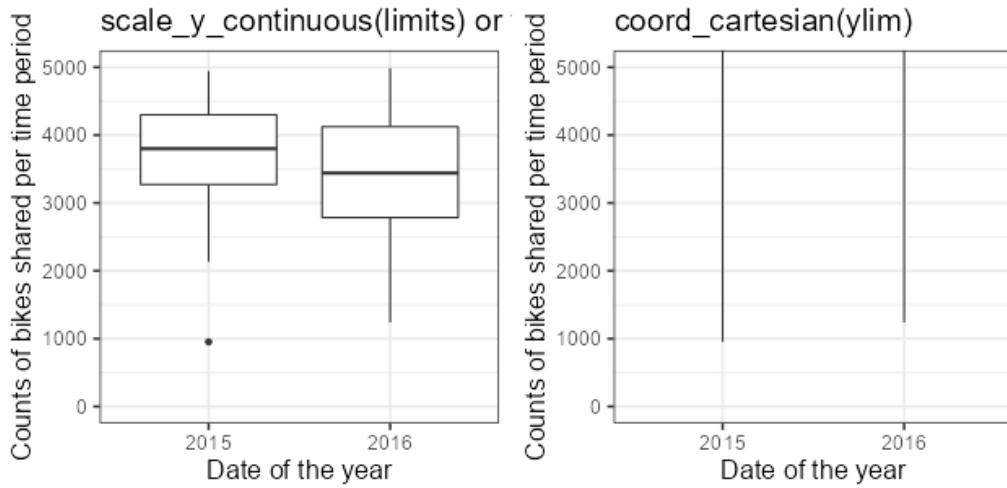
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  ylim(c(0, 50))
```



Alternatively you can use `scale_y_continuous(limits = c(0, 50))` or `coord_cartesian(ylim = c(0, 50))`. The former removes all data points outside the range while the second adjusts the visible area and is similar to `ylim(c(0, 50))`. You may wonder: *So in the end both result in the same*. But not really, there is an important difference—compare the two following plots:



You might have spotted that on the left there is some empty buffer around your y limits while on the right points are plotted right up to the border and even beyond. This perfectly illustrates the subsetting (left) versus the zooming (right). To show why this is important let's have a look at a different chart type, a box plot:



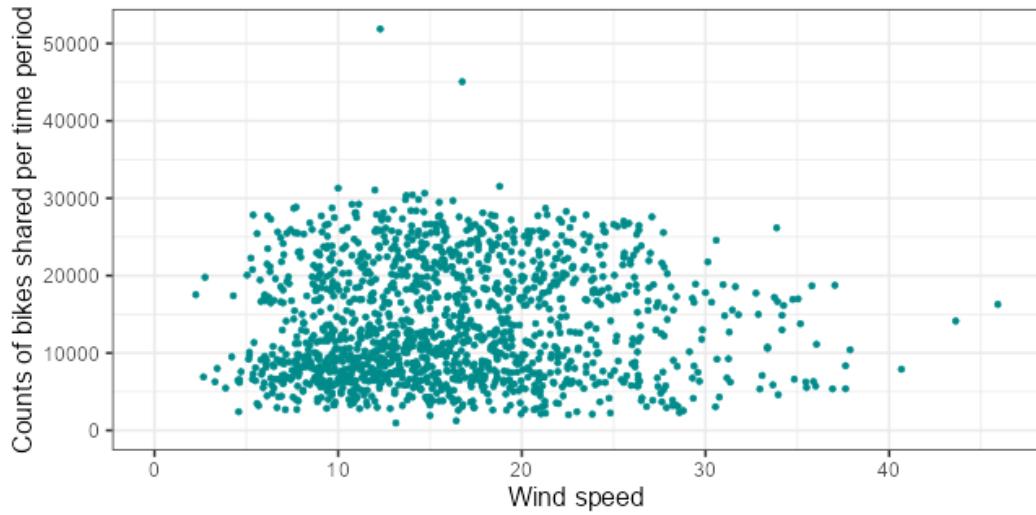
Um. Because `scale_x|y_continuous()` subsets the data first, we get completely different (and wrong, at least if in the case this was not your aim) estimates for the box plots! I hope you don't have to go back to your old scripts now and check if you *maybe* have manipulated your data while plotting and did report wrong summary stats in your report, paper or thesis...

7.0.0.0.8 Force Plot to Start at Origin

Related to that, you can force R to plot the graph starting at the origin:

```
library(tidyverse)

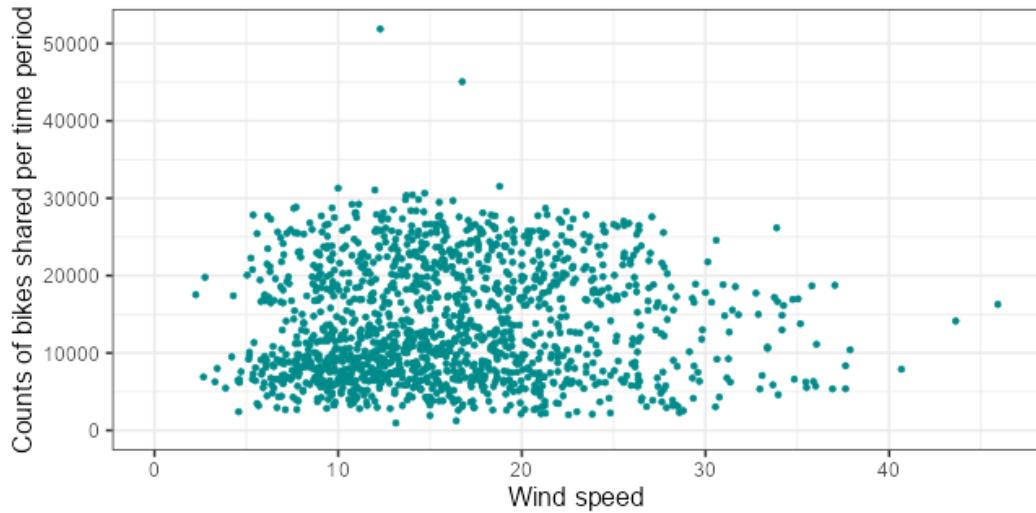
ggplot(bikes, aes(x = wind_speed, y = count)) +
  geom_point(color = "darkcyan") +
  labs(x = "Wind speed",
       y = "Counts of bikes shared per time period") +
  expand_limits(x = 0, y = 0)
```



Using `coord_cartesian(xlim = c(0, NA), ylim = c(0, NA))` will lead to the same result. Expand to see example.

```
library(tidyverse)

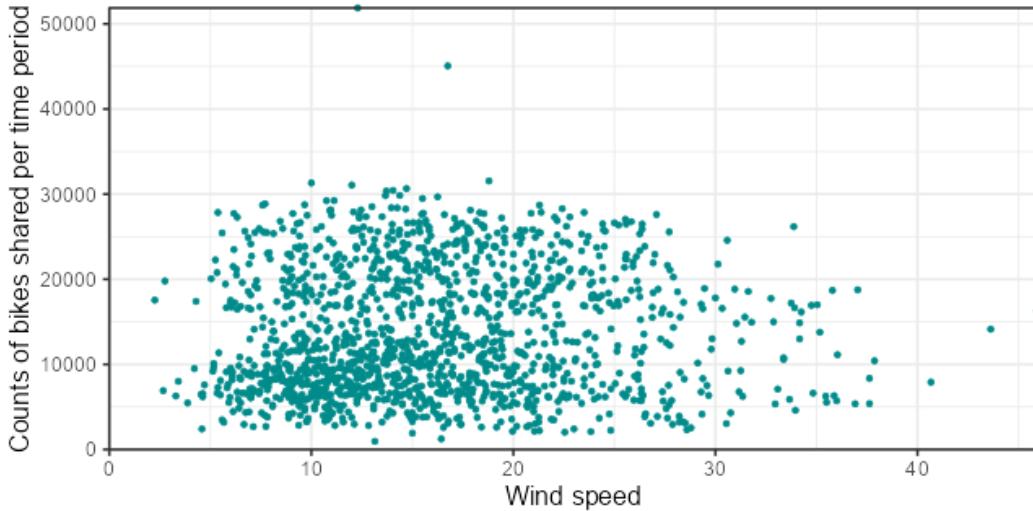
ggplot(bikes, aes(x = wind_speed, y = count)) +
  geom_point(color = "darkcyan") +
  labs(x = "Wind speed",
       y = "Counts of bikes shared per time period") +
  coord_cartesian(xlim = c(0, NA), ylim = c(0, NA))
```



But we can also force it to *literally* start at the origin!

```
ggplot(bikes, aes(x = wind_speed, y = count)) +
  geom_point(color = "darkcyan") +
```

```
labs(x = "Wind speed",
     y = "Counts of bikes shared per time period") +
  expand_limits(x = 0, y = 0) +
  coord_cartesian(expand = FALSE, clip = "off")
```



The argument `clip = "off"` in any coordinate system, always starting with `coord_*`, allows to draw outside of the panel area.

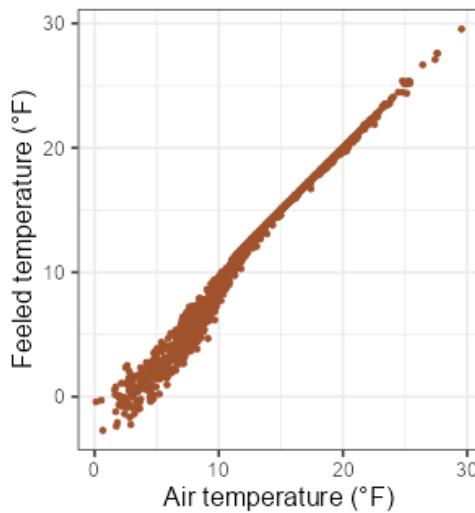
Here, I call it to make sure that the tick marks at `c(0, 0)` are not cut. See the Twitter thread by Claus Wilke¹ for more details.

7.0.0.9 Axes with Same Scaling

For demonstrating purposes, let's plot temperature against temperature with some random noise. The `coord_equal()` is a coordinate system with a specified ratio representing the number of units on the y-axis equivalent to one unit on the x-axis. The default, `ratio = 1`, ensures that one unit on the x-axis is the same length as one unit on the y-axis:

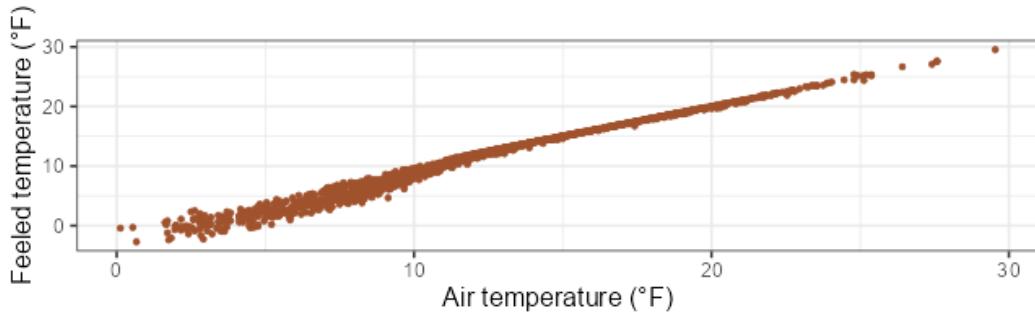
```
ggplot(bikes, aes(x = temp, y = temp_feel)) +
  geom_point(color = "sienna") +
  labs(x = "Air temperature (°F)", y = "Feeled temperature (°F)") +
  coord_fixed()
```

¹<https://twitter.com/clauswilke/status/991542952802619392?lang=en>



Ratios higher than one make units on the y axis longer than units on the x-axis, and vice versa:

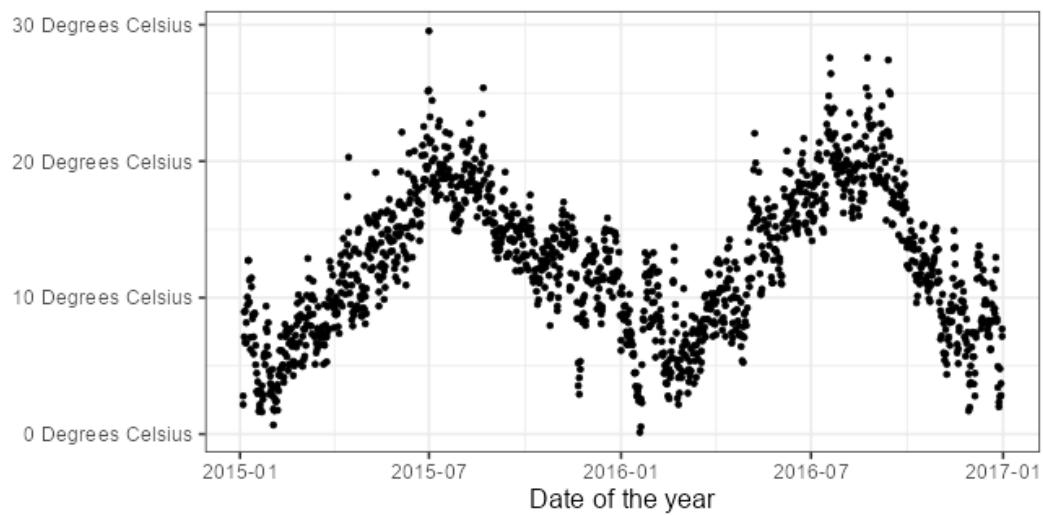
```
ggplot(bikes, aes(x = temp, y = temp_feel)) +
  geom_point(color = "sienna") +
  labs(x = "Air temperature (°F)", y = "Felt temperature (°F)") +
  coord_fixed(ratio = 1/5)
```



7.0.0.0.10 Use a Function to Alter Labels

Sometimes it is handy to alter your labels a little, perhaps adding units or percent signs without adding them to your data. You can use a function in this case:

```
ggplot(bikes, aes(x = date, y = temp)) +
  geom_point() +
  labs(x = "Date of the year", y = NULL) +
  scale_y_continuous(label = function(x) {return(paste(x, "Degrees Celsius"))})
```



- work with `scale_*_date()`

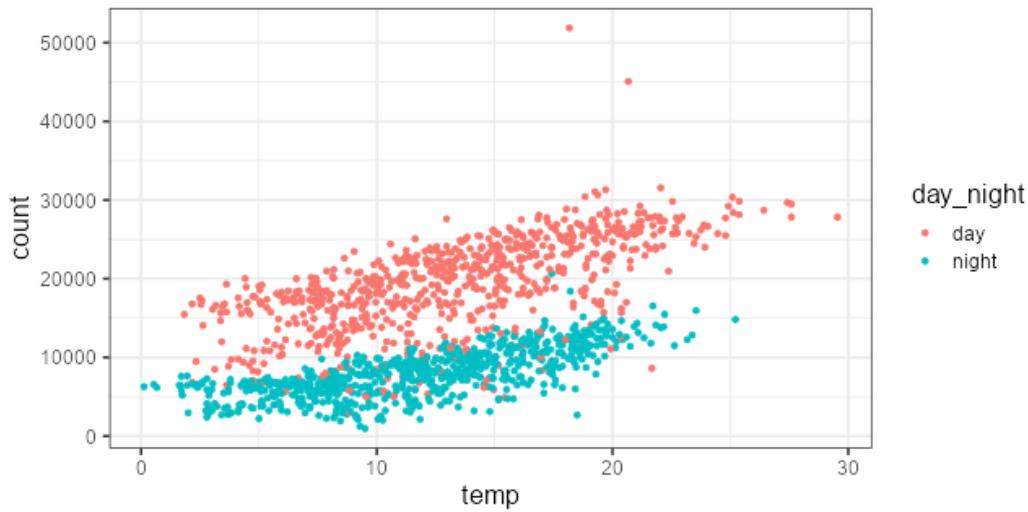
8

Working with Text

8.1 Titles

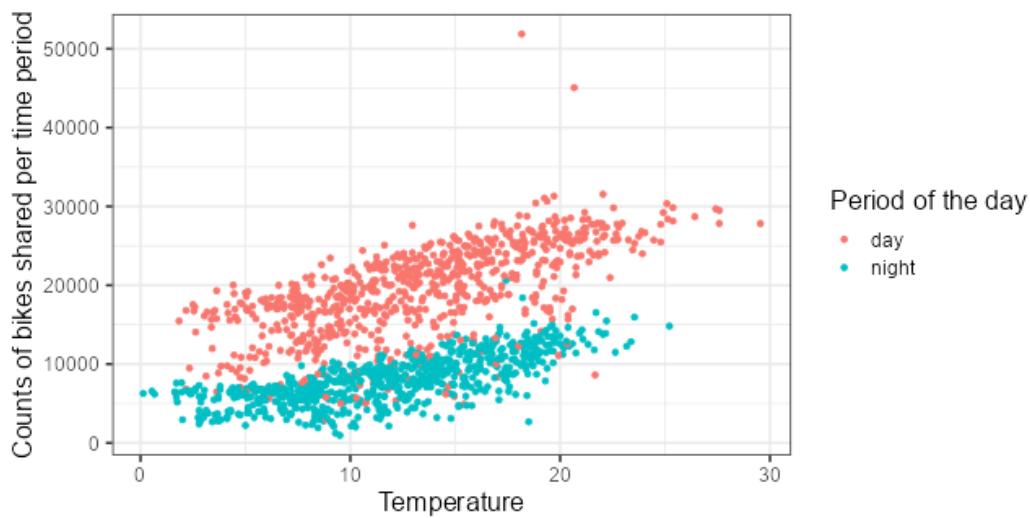
By default, **ggplot2** uses the column names specified for the aesthetic mappings as titles. That means, one could overwrite the column names in the source data. Usually the names are optimized for developing code (i.e. they contain no white spaces or unusual symbols) and it is better to leave them as they are. Also, the columns might be addressed somewhere else in the script and overwriting their names sounds like a bad idea in such a case.

```
ggplot(bikes, aes(y = count, x = temp, color = day_night)) +  
  geom_point()
```



Let's add some well-written labels to the axes and legend without changing the input object. For this, we add the `labs()` function and provide a character string for each label we want to change (here `x`, `y`, and `color`):

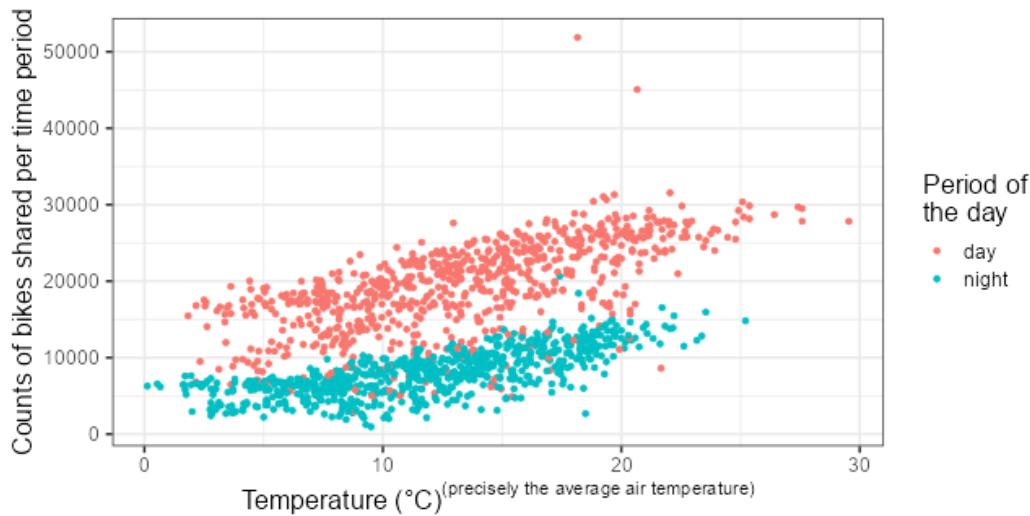
```
ggplot(bikes, aes(y = count, x = temp, color = day_night)) +  
  geom_point() +  
  labs(x = "Temperature",  
       y = "Counts of bikes shared per time period",  
       color = "Period of the day")
```



You can also add each axis title via `xlab()` and `ylab()` to specify each label individually. However, I recommend to use the `labs()` as (a) it allows you to add more labels to your plot and (b) collects all labels in one place.

By adding `\n` to a string you force a line break. Usually you can also specify symbols by simply adding the symbol itself (here `"°"`). In case you want to add more complex expressions you can use the `expression()` function. Here we use it in combination with `paste()` to concatenate multiple strings. With this approach we can for example add superscripts:

```
ggplot(bikes, aes(y = count, x = temp, color = day_night)) +
  geom_point() +
  labs(x = expression(paste("Temperature (°C)"^"(precisely the average air temperature)")),
       y = "Counts of bikes shared per time period",
       color = "Period of\nthe day")
```



A

More to Say

Yeah! I have finished my book, but I have more to say about some topics. Let me explain them in this appendix.

To know more about **bookdown**, see <https://bookdown.org>.

Bibliography

- Cairo, A. (2021). Orthodoxy and eccentricity. In *Data Sketches by Nadieh Bremer and Shirley Wu*, pages 12–13. Chapman and Hall/CRC, Boca Raton, Florida, United States. ISBN 978-036-70-0012-7.
- Koponen, J. and Hildén, J. (2019). *Data visualization handbook*. Aalto ARTS Books, Espoo, Finland, 1st edition. ISBN 978-952-60-7449-8.
- Kosara, R. (2007). Visualization criticism - the missing link between information visualization and art. In *2007 11th International Conference Information Visualization (IV '07)*, pages 631–636.
- Sciaiani, M., Fritsch, M., Scherer, C., and Simpkins, C. E. (2018). Nlmr and landscapetools: An integrated environment for simulating and modifying neutral landscape models in r. *Methods in Ecology and Evolution*, 9(11):2240–2248.
- Xie, Y. (2015). *Dynamic documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, United States, 2nd edition. ISBN 978-1498716963.
- Xie, Y. (2022). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.28.

Index

bookdown, xi

knitr, xi