Interactive Data Visualization in R

Group 5:

Michaela Kargl, Ajdin Mehic, Zoran Prodanovic, and David Seywald

706.057 Information Visualisation SS 2018
Graz University of Technology

13 May 2018

Abstract

R is a software environment specifically designed for statistical computing and graphics, as described in [TRF 2018]. This survey paper explores the capabilities of R with respect to data visualisation.

Several R packages provide functionality for *static data visualization*. In this survey paper the three packages graphics, lattice, and ggplot2 are described briefly. Examples of their usage are shown, and their specific strengths and weaknesses are discussed.

R can also be used for *interactive data visualisation*. This survey paper describes some of the most popular R packages for interactive data visualisation, such as shiny, htmlwidgets, ggiraph, ggvis, rbokeh, and plotly. For each of these packages its functionality is briefly described, and code examples are shown. Furthermore, these packages for interactive data visualisation are compared in an overview table, and recommendations are given regarding the usage of the packages for different visualisation tasks.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

R is a programming language and free software environment for statistical computing and graphics. it is widely used among statisticians and data miners who developer statistical software and data analysis. R is a GNU package. The source code is written primarily in C, Fortran and R.

Capabilities of R are extended through packages, which allow to used some specialized statistical techniques, import/export results etc. More than 12 000 packages are available.

R has a command line interface, but there is also a graphical front-end and integrated development environment such as R Studio. It includes a code editor, debugging and visualization tools. It is possible to view the result of written code inside R studio. However, this developing environment also makes it also possible to export the written code in form of .html file as well as publishing the written code online by deploying it on a server.

Chapter 2

Static Data Visualization in R

This chapter explains how R can be used for static data visualization. First, the two different graphics systems that exist in R are briefly introduced. Then, the three most popular R packages for static data visualization are described.

2.1 Graphics Systems in R

There are two fundamentally different, non compatible graphics systems existing within R: *graphics* and *grid.* The basic concepts of these two graphics systems, their similarities and main differences are briefly explained in the following paragraphs.

As described by Paul Murrell in Murrell [2006, pages 21, 164], the underlying concept for both R graphics systems is the "painters' model": Like a painter draws on a canvas, the output of *graphics* and *grid* functions occurs on the currently open "graphics device", which can be the screen or a file (.pdf, .png, .svg etc.). Similar to a painter's drawing, the new output of the *graphics* or *grid* functions is drawn on top of the previous output, and later output obscures any earlier output, when it is overlapping.

Regarding the drawing region, *graphics* and *grid* follow different concepts. In the *graphics* system the page is split into defined regions: the (current) plot region, where usually data points and lines are drawn, the (current) figure region, where usually the axis and labels are drawn, and the outer margins, where for example the title of the graphic is located. In the *grid* system, there are no pre-defined drawing regions. Instead arbitrary rectangular regions, so called viewports, can be defined, and the drawing is done in the currently active viewport. (A detailed description of the concept of viewports can be found in [Murrell 2018].)

Another aspect, where *graphics* and *grid* follow different concepts, is the output of the plotting. In the *graphics* system the output of the graphics functions is the graphical output on the screen or in a file. This output can only be modified by changing the R code and re-running the whole plotting process. In the *grid* system the graphics functions produce objects (graphic elements and viewports) representing the output. These objects can be saved and modified, and thus grid plots can be edited interactively. (A detailed description of the grid graphics model can be found in chapter 5 of [Murrell 2006].)

The *graphics* package includes high-level functions for producing complete plots, such as for example scatterplots, barcharts, histograms, linecharts, or piecharts. Therefore, it is widely used for quick data visualization. The *grid* package, however, contains only low-level functions for drawing graphical elements such as for example points or lines, and offers no functions for drawing complete plots. Thus, the *grid* package is usually not directly used for data visualisation, but there are two popular R packages for data visualization, the *lattice* package and the *ggplot2* package, which are built on *grid*.

The usage of these three R packages (*graphics*, *lattice*, and *ggplot2*) for static data visualization is described in more detail in the following chapter.

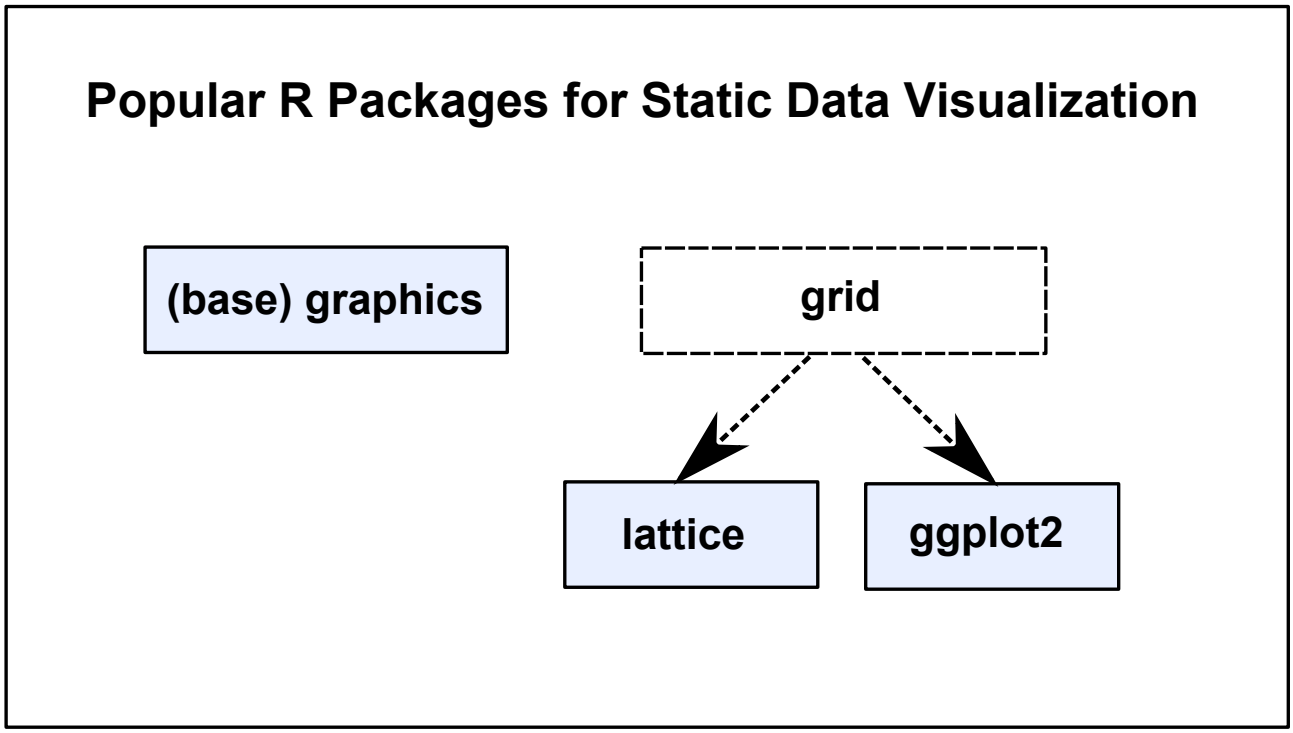

Figure 2.1: The three most popular R packages for static data visualization are *(base) graphics*, *lattice*, and *ggplot2*.

2.2 Popular R Packages for Static Data Visualization

Among the many packages that are available within R, the most popular packages for static data visualization are *graphics*, *lattice*, and *ggplot2*. The *graphics* package is often called "base graphics", as it is the traditional R graphics package, which is included in the standard R distribution. The *lattice* and *ggplot2* packages are built upon the *grid* graphics system, as shown in Figure 2.1. The *grid* package, as well as *lattice* and *ggplot2* are not included in the standard R distribution, they are add-on packages that need to be installed separately.

In the following subsections a closer look is taken into the usage of *graphics*, *lattice*, and *ggplot2* for static data visualisation. For each of these three R packages the underlying concept for creating graphs is explained. Code examples are shown to illustrate these underlying concepts, and the strengths and weaknesses of each of these three packages are discussed.

2.2.1 Usage of *graphics* for Static Data Visualization

Since the *graphics* package was one of the first packages for R extension, it is often called "base graphics" or "traditional graphics". The *graphics* package is a collection of high-level and low-level graphics functions.

A high-level graphics function creates a complete plot: it initialises the graphics window, sets the scale, and renders the graphic. For many standard graph-types, such as for example barcharts, linecharts, scatterplots, histograms, sunflowerplots, boxplots, mosaicplots..., there is a specific high-level function available in the *graphics* package. The most important high-level function in the *graphics* package, however, is the plot() function. The plot() function is a generic function, it produces different types of plots depending on the type of the data. For example, for numeric data the plot() function creates a scatterplot while it creates a barplot for factor data.

In addition to the high-level graphics functions, the *graphics* package provides also a set of low-level graphics functions, which add some elements (such as for example labels, points, lines, or arrows) to an existing

plot. All functions in the *graphics* package accept various parameters (such as colour, line-type, line-width, font-type, data-symbol...) to fine-tune the output and modify (almost) any aspect of the visual appearance of the plot.

In the usual work-flow for static data visualization with base *graphics*, first a high-level function is used to create a quick plot of the data, and then low-level functions are used to improve the visual appearance and comprehensibility of the graph by adding for example labels, annotations, or regression lines. Several books and tutorials, such as for example Kabacoff [2017b], Kabacoff [2011, chapter 3], Murrell [2012, chapters 2-3], and Teetor [2011, chapter 10], explain this workflow in detail for a lot of different plot-types.

The code example in Listing 2.1 shows how to utilise the base R *graphics* package for creating the simple scatterplot depicted in Figure 3.3: The first line of the code specifies that the mtcars data set is to be used for the graph. Then, the plot() function is used to create the scatterplot. Finally, in the last line of the code, the abline() function is used to add the red regression line to the plot.

```
1 data(mtcars)
2 plot(x=mtcars$cyl, y=mtcars$mpg,
3      xlab = "cyl", ylab = "mpg",
4      main = "mtcars: Cars' efficiency (mpg vs #cylinders)")
5 abline(lm(mtcars$mpg~mtcars$cyl), col = "red")
```

Listing 2.1: The code for building a simple scatterplot plus regression line with base R *graphics* basically consists of a high-level graphics function (- in this example the plot() function), which creates the graph, followed by a low-level graphics function (- in this example the abline() function), which adds further graphical elements to the graph.

For simple graphs, base R *graphics* is fairly straight forward to use. It produces quick graphs with just a few lines of code. Furthermore, data visualization with base R *graphics* is quite convenient, since *graphics* comes with the standard R distribution and no extra package needs to be installed. In fact also very sophisticated and complex graphs can be built with *graphics* functions, as the variety of available low-level graphics functions and the extensive set of parameters support building up graphs from scratch as well as customising graphs created with any of the high-level graphics functions available in the *graphics* package.

However, building complex graphs with base R *graphics* functions is quite tedious. As described in Teetor [2011, chapter 10] and shown in Rickert [2015], the code for complex graphs becomes confusing and bewildering when utilising base R *graphics* for creating complex graphs. For example the procedure for adding a legend to the graph is quite prone to errors, since not only the coordinates for positioning the legend must be picked manually but also the correspondence between the labels and the data-symbols, line-types, and colours is not automatic with base R *graphics*. Therefore, many experts, as for example Susane Johnston in Johnston [2013], recommend to use other graphics packages, namely lattice or ggplot2, for creating more complex graphs.

2.2.2 Usage of *lattice* for Static Data Visualization

The *lattice* package is built upon the *grid* graphics system, as already explained in a previous section and shown in Figure 2.1. Like almost all graphics-related R packages (with the exception of the base R *graphics* package) *lattice* is an add-on R package, which means that it is not included in the standard R distribution but must be installed separately.

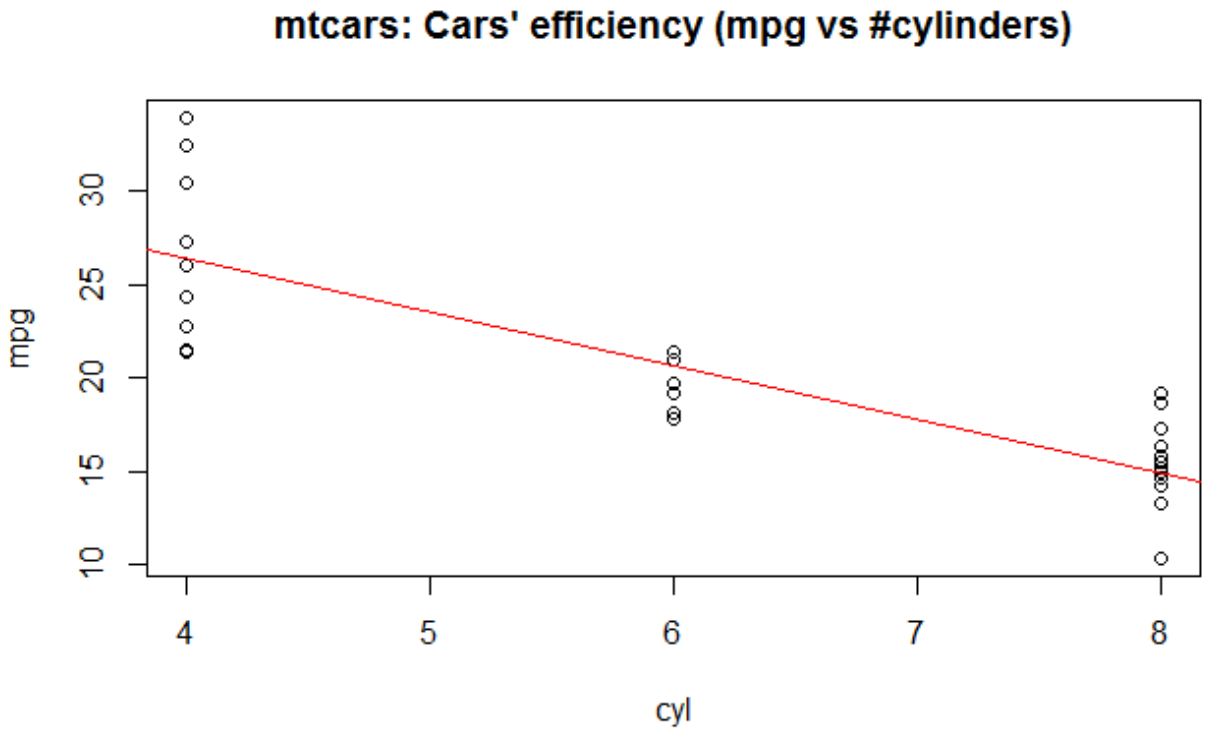

Figure 2.2: Scatterplot showing mpg versus cyl from the classic mtcars data set. This scatterplot was created utilising the base R *graphics* package.

The *lattice* package provides a bunch of high-level functions for creation of different types of plots. Most of these high-level functions have fairly descriptive names, such as for example barchart(), histogram(), xyplot(), contourplot(), and parallel(). In addition, *lattice* provides also a collection of so called panel.functions, which can be called within a high-level function to add additional elements such as for example lines, curves, or a grid. A detailed description of all available *lattice* functions can be found in the regularly updated lattice manual [Sarkar 2017].

In the usual work-flow for static data visualization with *lattice*, a suitable high-level function is selected to create the desired plot of the data, and then low-level panel.functions can be nested into the high-level function to add for example a regression line or a grid.

In the manual of the *lattice* package [Sarkar 2017], Deepayan Sarkar, the developer of the *lattice* package, describes *lattice* as:

> "a powerful and elegant high-level data visualization system inspired by Trellis graphics, with an emphasis on multivariate data"

This quote points to the underlying motivation for the development of the *lattice* package: *lattice* was developed with a focus on producing Trellis plots. Thus, as described in [Sarkar 2008], *lattice* was designed to effectively combine multiple plots in a page with properly coordinated scales, aspect ratios and labels. An example of a Trellis plot created with *lattice* is shown in Figure 2.3, and the code used for creating this Trellis plot example (adapted from [Kabacoff 2017a]) can be found in Listing 2.2.

Traditional graphs are implemented with *lattice* as a single-panel Trellis display. Figure 2.4 shows an example of such a single-panel Trellis display, which looks quite similar to the scatterplot created with base R

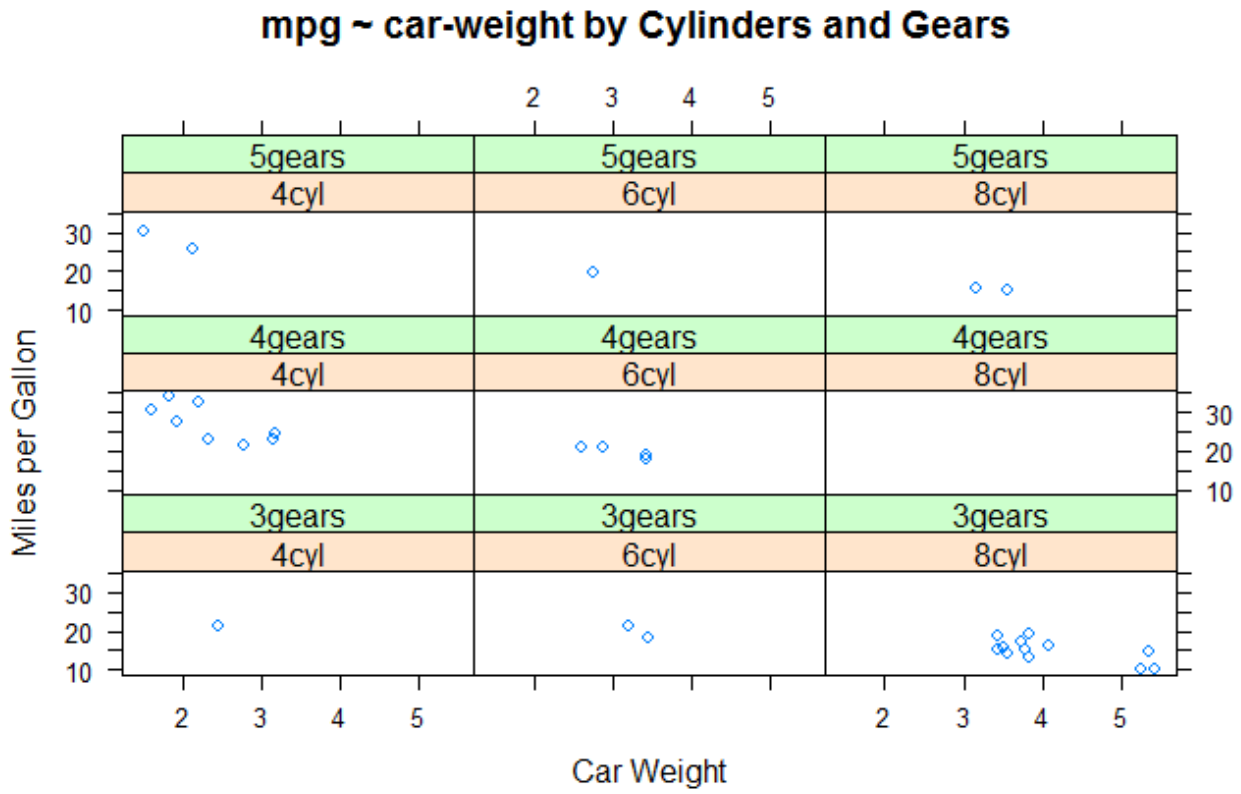

Figure 2.3: Example of a Trellis plot created with *lattice*. This plot shows the relationship of miles per gallon and car weight by number of cylinders and gears for the classic mtcars data set.

```
library(lattice) #use the lattice package
attach(mtcars) #use the mtcars data set

# create factors with value labels
gear.f<-factor(gear,levels=c(3,4,5),
    labels=c("3gears","4gears","5gears"))
cyl.f <-factor(cyl,levels=c(4,6,8),
   labels=c("4cyl","6cyl","8cyl"))

# draw scatterplots of mpg vs. wt depending on number of cylinders and gears
xyplot(mpg~wt|cyl.f*gear.f,
    main="mpg ~ car-weight by Cylinders and Gears",
   ylab="Miles per Gallon", xlab="Car Weight")
```

Listing 2.2: By using the *lattice* package, drawing a Trellis plot is straightforward. In this example code (adapted from [Kabacoff 2017a]) for the classic mtcars data set, the xyplot() function is used to draw a Trellis plot composed of scatterplots showing miles per gallon (mpg) versus car weight (wt) depending on the number of cylinders (cyl) and the number of gears (gears).

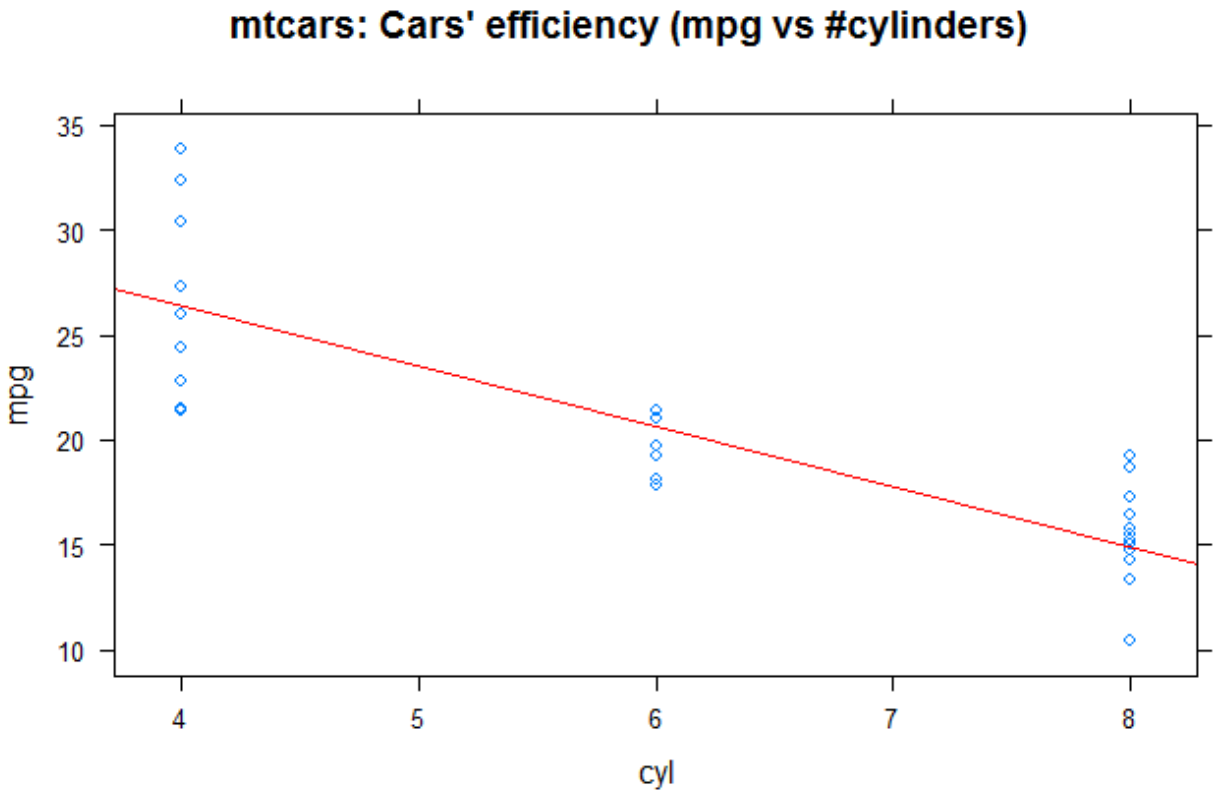

Figure 2.4: Scatterplot showing mpg versus cyl from the classic mtcars data set. This scatterplot was created utilising the *lattice* package.

graphics (see Figure 3.3). The *lattice* code used for creating the scatterplot shown in Figure 2.4 can be found in Listing 2.3.

The main advantage of *lattice* is, that many tasks that are rather complex and complicated in base R *graphics* are automated in *lattice*. For example, *lattice* takes automatically care for optimum positioning of the elements of the plot such as legends, and labels. Also the correspondence between the labels and the data-symbols, line-types, and colours in the legend is handled automatically by *lattice*. Since *lattice* is based on *grid*, it is easy to utilise grid functions to further fine-tune and enhance any plot created with *lattice*.

However, adding additional elements (for example a regression line or annotations and arrows) to the plot is more complicated in *lattice* than in base R *graphics*. Furthermore, since the complete *lattice* plot is put into one single function, for more complex graphics the code can easily become confusing and unclear, especially when there are many panel.functions nested within the high-level function.

2.2.3 Usage of *ggplot2* for Static Data Visualization

ggplot2 is, like *lattice*, also based on the *grid* graphics system. *ggplot2* is an add-on R package, which is not included in the standard R distribution but must be installed separately.

ggplot2 implements the concept of the "layered grammar of graphics". The concept of the "grammar of graphics", which helps to describe the components of a graphic, was developed by Leland Wilkinson in 2005. The "grammar of graphics" is explained thoroughly in [Wilkinson 2005]. Hadley Wickham, the developer of the *ggplot2* package, describes in [Wickham 2010], how the "grammar of graphics" is related to the way graphics are created with *ggplot2*. Rather than defining high-level functions for creating complete plots (as in base R *graphics* and *lattice*), in *ggplot2* a graphic is built from a set of independent components that can be

```
1 library(lattice)
2 xyplot(mpg ~ cyl, data = mtcars,
3        main = "mtcars: Cars' efficiency (mpg vs #cylinders)",
4        panel = function(x, y) {
5          panel.xyplot(x, y)
6          panel.abline(lm(y ~ x), col = "red")}
7        )
```

Listing 2.3: With *lattice*, the code for building a simple scatterplot plus regression line consists of a single high-level *lattice* function (- in this example the xyplot() function). The panel.abline() function for drawing the regression-line is nested into the high-level xyplot() function.

composed in various different ways. [Wickham 2016]

As described in [Wickham 2016], a *ggplot2* plot is composed of:

- *Data* that shall be visualised
- *Aesthetic mappings* describe, how variables (data) are mapped to visual properties such as for example colour, size, shape, distance from axis...
- *Geometric objects* such as points, lines, polygons...
- *Statistic transformations* such as for example binning counting observations (to get a histogram)...
- *Scales* draw legends and axes to map values in the data space to values in the aesthetics space
- *Coordinate System* such as for example Cartesian coordinates or polar coordinates; provides axis and grid lines
- *Faceting specification* describes how to split data into subsets and how to display these subsets as small multiples
- *Theme* specifies general aspects such as font size, background colour...

The code example in Listing 2.4 shows how the scatterplot depicted in Figure 2.5 was created with *ggplot2*. As can be seen in this code example, the *ggplot2* syntax, where the single functions describing the components of the plot (data, aesthetic mappings, geometric objects, labels, and title) are combined with "+" characters, makes the underlying "layered grammar of graphics" concept clearly visible. On the one hand, the layered structure of *ggplot2* helps to keep the code clear and graspable. On the other hand, the layered structure of the code also makes it easy to add further elements to the plot. For example, the red regression line here in this code example (Listing 2.4) is added to the plot by simply specifying an additional geometric object (geom) in code-line 5.

The underlying concept of the "layered grammar of graphics" makes *ggplot2* very versatile: composing graphics by combining independent components rather than defining high-level functions that build-up complete plots, enables also the creation and exploration of new, non-standard types of plots.

ggplot2 provides a large number of parameters to customise a plot. However, for many of these parameters useful defaults are specified, so that nice-looking plots can be created conveniently with the default settings.

ggplot2 is used by other developers as a basis for creation of new R add-on packages: Currently there are more than 40 extensions for *ggplot2* available on the CRAN repository.

```
library(ggplot2)
ggplot(data = mtcars) +
  aes(x = cyl, y = mpg) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE, col = "red") +
  xlab("cyl") +
  ylab("mpg") +
  ggtitle("mtcars: Cars' efficiency (mpg vs #cylinders)")
```

Listing 2.4: The *ggplot2* code clearly shows, how the single components (data, aesthetic mappings, geometric objects, labels, and title) are stacked to build a scatterplot plus regression line.

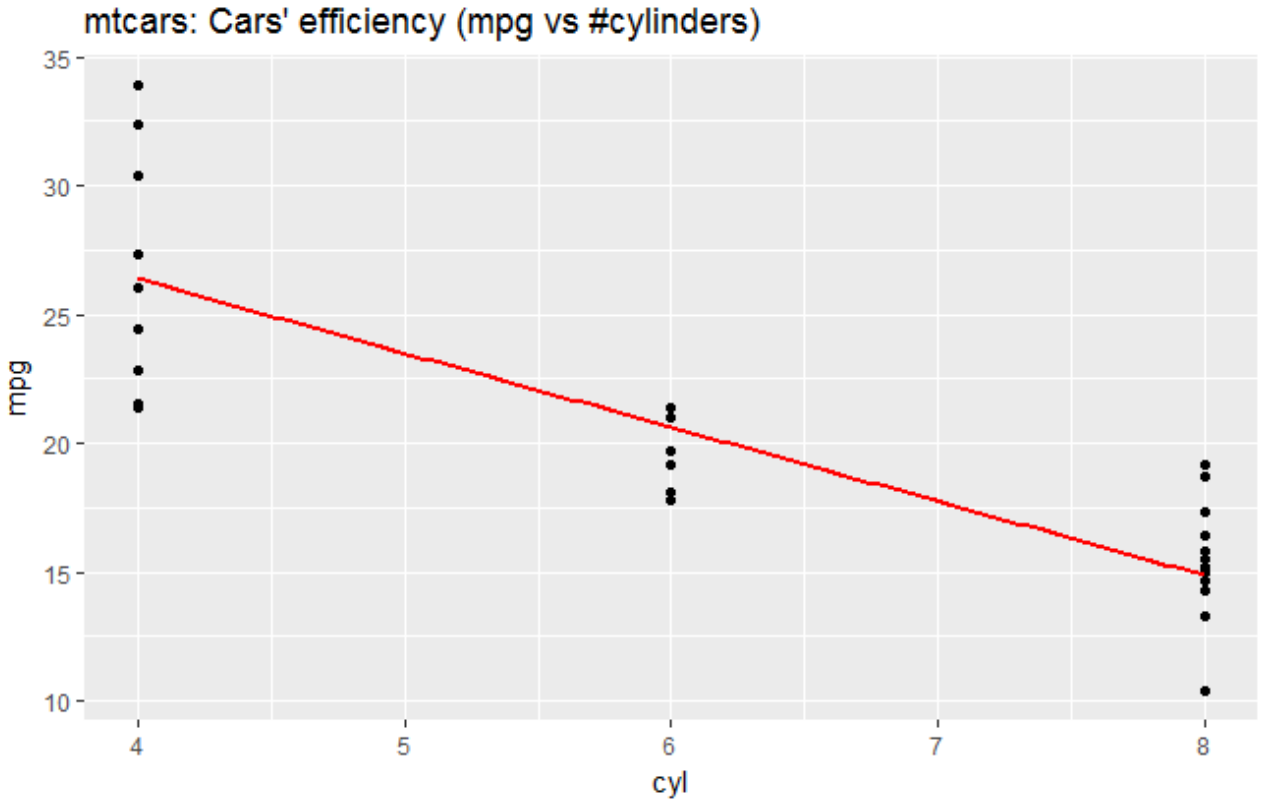

Figure 2.5: Scatterplot showing mpg versus cyl from the classic mtcars data set. This scatterplot was created utilising the *ggplot2* package.

However, *ggplot2* is not in all aspects superior. For example, it is substantially slower than base R *graphics* and also slower than *lattice*. Fine-tuning of plots by utilising *grid* functions is more complicated for *ggplot2* than for *lattice* plots.

2.2.4 Comparison of *graphics*, *lattice*, and *ggplot2*

In principle all three R packages (*graphics*, *lattice*, and *ggplot2* support creation, and customisation of static data visualizations. However, each of these packages has got its specific strengths and weaknesses.

- *graphics* is fast, convenient to use, and included in the standard R distribution. However, modification of an existing plot is not possible. *graphics* ist good to use for quick exploration of data and for simple standard plots.

- *lattice* deals automatically with spacing, margins, and positioning of multiple plots on one display. Graphical objects can be saved and modified (via *grid*). However, *lattice* cannot easily "add" elements (for example annotations...) to the plot. *lattice* is good to use for multivariate data visualisation and multiple plots on one display.

- *ggplot2* automatically deals with spacing and margins. *ggplot2* provides a lot of defaults (but customisable if desired), and there are many extensions available (for example ggiraph...). However, *ggplot2* is slower than *graphics* and *lattice*. Modification of saved graphical objects (via *grid* is more difficult than in *lattice*, and unfortunately *ggplot2* provides only a fixed aspect ratio for "faceting" plots. *ggplot2* is good to use for nice plots by default, and for animated plots (usage of *ggplot2* in combination with the extension *ggiraph*).

Chapter 3

Interactive Data Visualization in R

First, this chapter contains some of the most popular R packages which can be used for interactive data visualization. Later on we highlight a few other, interesting packages which we came across while researching and gathering data on the common ones.

3.1 Popular R Packages for Interactive Data Visualization

Among the many packages that are available within R, the most popular packages for interactive data visualization are shiny, ggvis, htmlwidgets, ggiraph, rbokeh and plotly. We had a closer look into each of those packages and the following chapters contain a short introduction to each package, highlighting advantages as well as providing a small code example with its corresponding output.

3.1.1 htmlwidgets

Htmlwidgets is a framework that connects JavaScript libraries to R programming language. It acts like a connector which simplifies usage of Javascript libraries in R programming language. The less amount of code lines is needed to produce same effect in R than it would have been needed when writing pure JavaScript code. This way, the user only has to be able to write code in R and he can make web page without any knowledge about HTML, CSS or JS.

There are various packages in R that take advantage of htmlwidgets framework (leaflet, dygraphs, networkD3, DataTables, rthreejs to name a few). An htmlwidget works like an R plot, but it also produces an interactive web visualization. While running the app inside R studio, one can export it as .html file and have it as a web page. The HTML code is minified and compressed. However, this makes it harder later if one wants to change details in the .html, because it is almost impossible to tell what the code is doing by just reading it: it's compressed and minified, and looks "ugly". From the positive side, writing code in R and using htmlwidgets takes much less time than writing the actual HTML/CSS/JS code.

Writing htmlwidgets code is possible in two ways:

- in the R console
- in R studio

The main difference between them is that when using R markdown, user has nicer interface and better overview of any error or warning messages. In both situations, user is able to see outcome of the code in the "Viewer" part and there is also a possibility of exporting current work in a form of .html file.

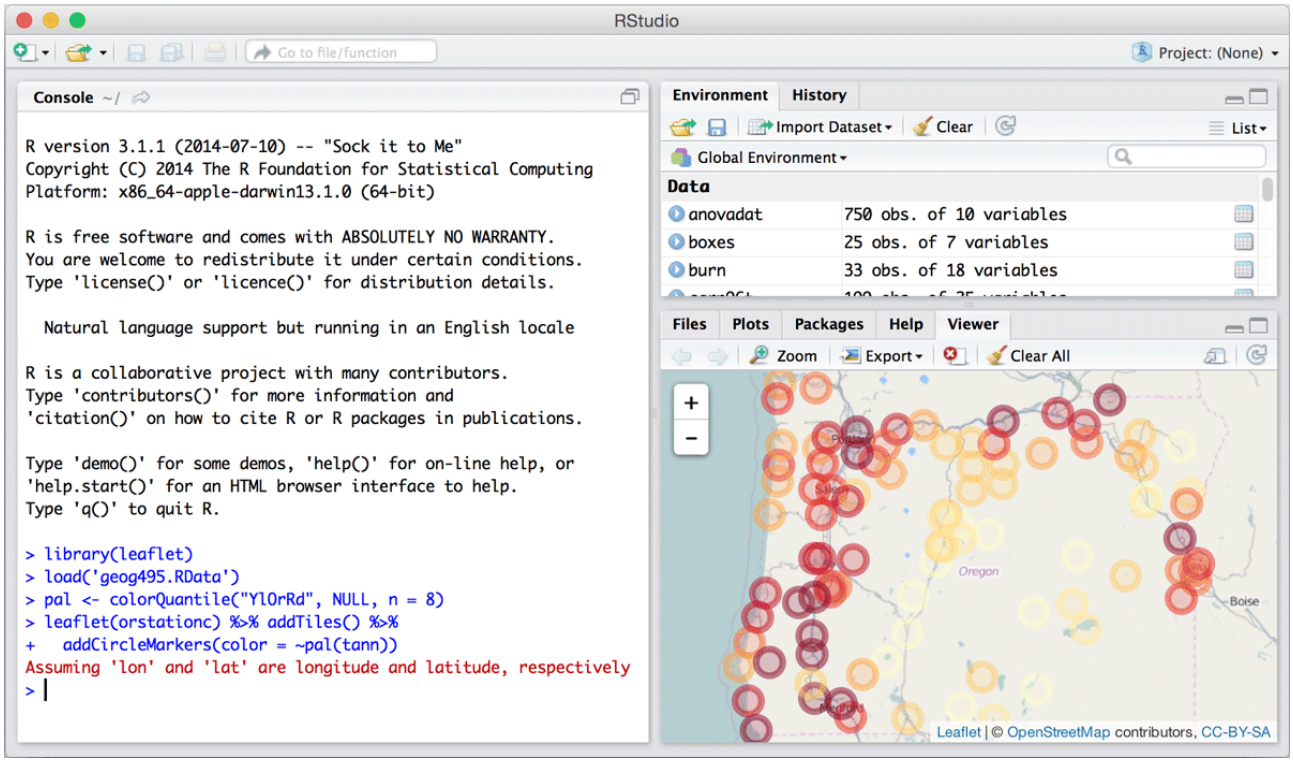

Figure 3.1: Example how to use Leaflet htmlwidget with the help of R console

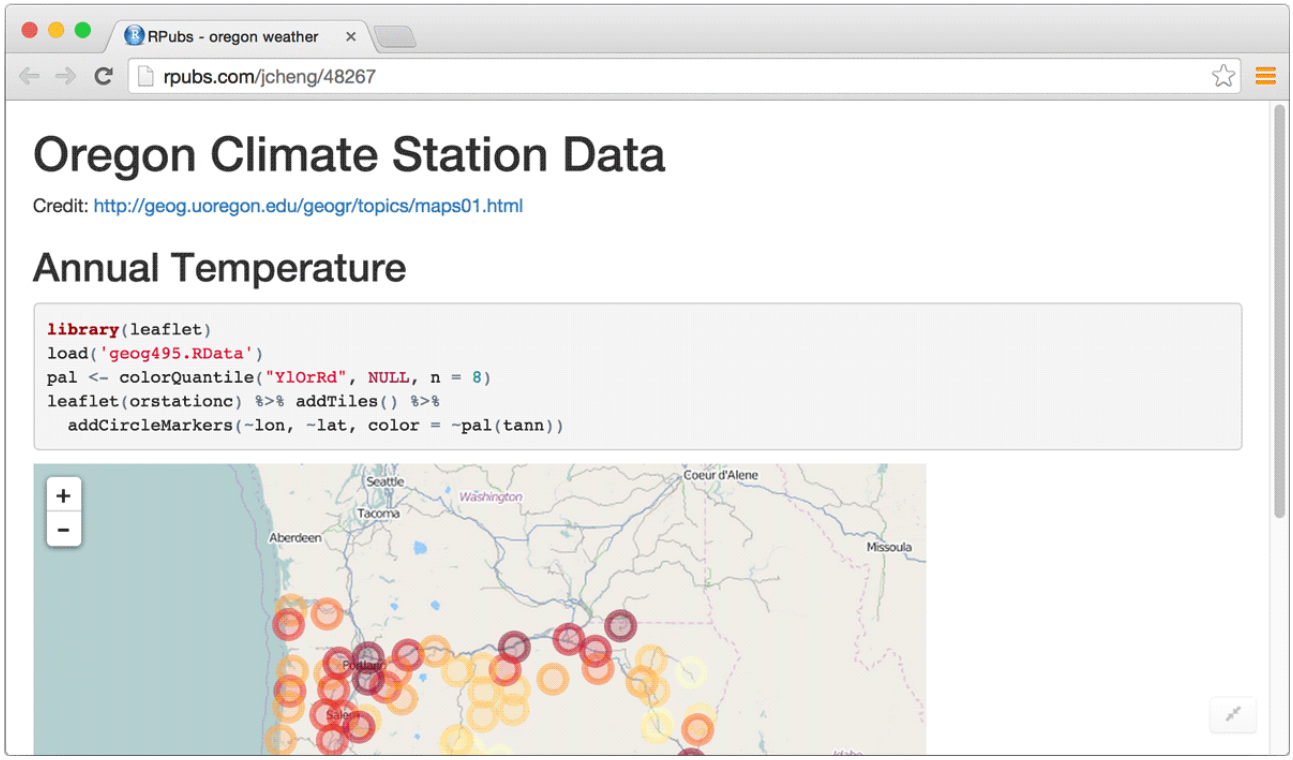

Figure 3.2: Example how to use htmlwidget with the help of R console

The main benefit of using htmlwidgets is that user does not have to be experienced web developer. He does not have to have any knowledge about HTML/CSS/JS coding. If he knows R programming language, he can use htmlwidgets and build websites by writing less amount of code lines with R, which would then be converted into HTML/CSS/JS. From the other side, a negative thing could be that after building .html file thanks for R's "export" function, it is really unclear what the code inside this file is doing. This is mainly because code is compressed and not readable from human perspective of view. So if user wants to do some changes in this .html file, it would not be that simple.

3.1.2 ggiraph

ggiraph is R package which allows users making ggplot interactive. This package is htmlwidget as well, since it allows building interactive graph which is converted into .html file.

The main difference between ggiraph and ggplot2 is that ggiraph is dynamic: it allows interaction with the user. From the other side, ggplot2 is static.

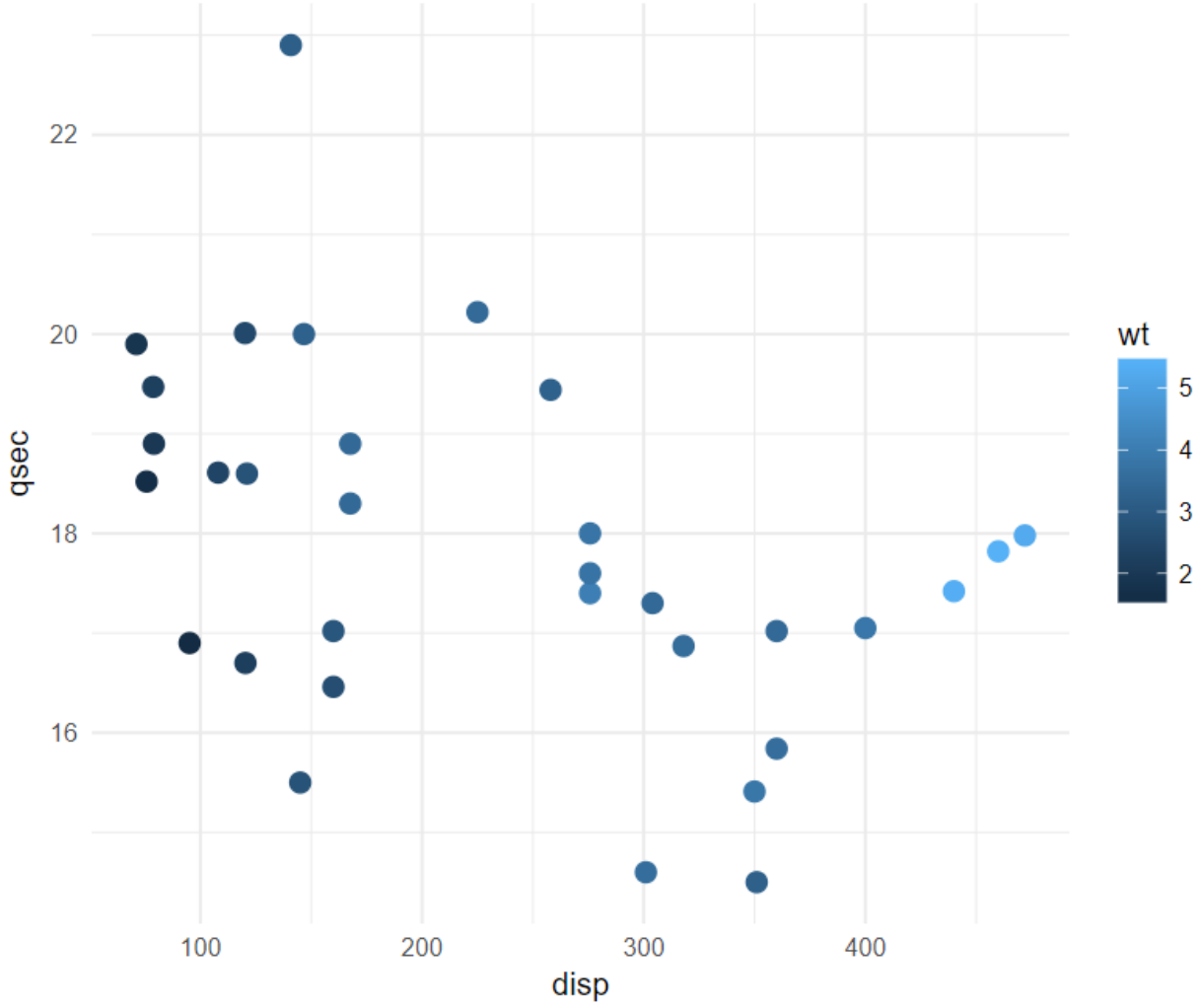

Figure 3.3: Example of graph drawn by ggiraph package

```
library(ggplot2)
library(ggiraph)

theme_set(theme_minimal())

dataset <- mtcars

dataset$carname <- row.names(dataset)

gg_point_1 <- ggplot(dataset, aes(x = disp, y = qsec, tooltip = carname, data_id =
    carname, color= wt) ) +

geom_point_interactive(size=3)

# htmlwidget call
ggiraph(code = {print(gg_point_1)}, tooltip_offx = 20, tooltip_offy = -10 )
```

Listing 3.1: ggiraph sample code for producing interactive graph

In the code sample it is shown that ggplot2 and ggiraph packages are being included. After that, dataset is being defined as well as labels to be shown on the graph. It's made in a way that when user hovers over a dot, he will see a description of the current data point he is checking.

3.1.3 shiny

The open source R package *shiny*, allows the user to create interactive webapplications without requiring knowledge of html, css or javascript. However, it is possible to enhance and style the visualizations with these techniques.
Other graphics can be used inside shiny apps, for example ggplot2, ggvis and htmlwidgets. Each shiny app consists of two different components, which are usually (used to be multiple files in the past) contained in a single file called *app.R*. These components are:

- UI component: the user interface object
- Server component: the server function with input and output collections
- ShinyApp: creates a shiny app with ui and server components as parameters

The syntax is easy to understand and uses a fluent style. One of the most important features of shiny is, that the inputs and outputs are connected "live" together. In other words the whole visualization is *reactive*. By changing any input variable (for example via a slider control), all connected outputs (for example chart values) are updated automatically. It is important to notice, that just the affected parts are updated and not the whole page being reloaded. The goal is to make it easy to capture inputs from a webpage, make them available in R and display the output again.

```
input values => R code => output values
```

Figure 3.4: shiny reactive structure

We can see in the following example how easy it is to create a simple reactive shiny webapplications without having to code any html, css or javascript. This example plots the built-in *faithful* dataset with a dynamic number of bins.

```
# Define UI for app that draws a histogram ----
ui <- fluidPage(
  # App title ----
  titlePanel("Hello Shiny!"),
  # Sidebar layout with input and output definitions ----
  sidebarLayout(
    # Sidebar panel for inputs ----
    sidebarPanel(
      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    # Main panel for displaying outputs ----
    mainPanel(
      # Output: Histogram ----
      plotOutput(outputId = "distPlot")
    )
  )
)
# Define server logic required to draw a histogram ----
server <- function(input, output) {
  # Histogram of the Old Faithful Geyser Data ----
  # with requested number of bins
  # This expression that generates a histogram is wrapped in a call
  # to renderPlot to indicate that:
  #
  # 1. It is "reactive" and therefore should be automatically
  #    re-executed when inputs (input$bins) change
  # 2. Its output type is a plot
  output$distPlot <- renderPlot({
    x    <- faithful$waiting
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = "#75AADB", border = "white",
         xlab = "Waiting time to next eruption (in mins)",
         main = "Histogram of waiting times")

    })
}
shinyApp(ui, server)
```

Listing 3.2: shiny sample code for producing an interactive barchart

To run this example (or any other shiny app in general):

```
1 library(shiny)
2 runExample("path to the folder containing the shiny app")
```

Listing 3.3: How to start a shiny app

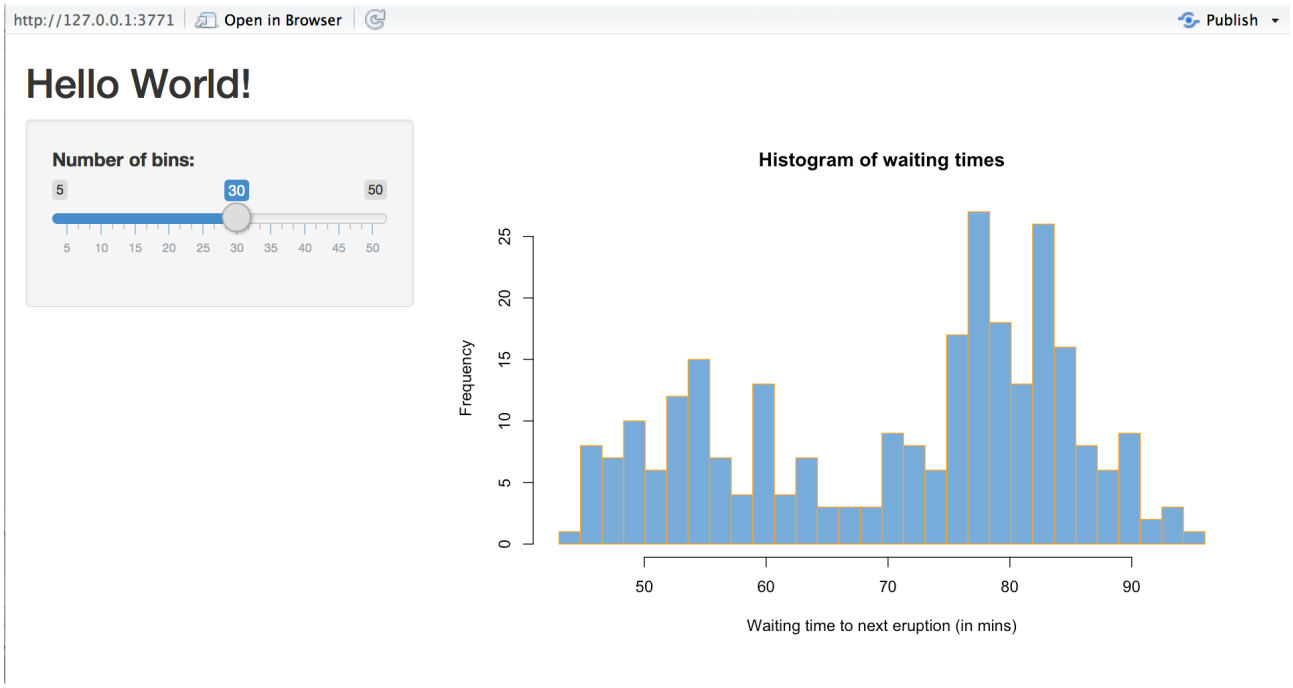

Figure 3.5: shiny interactive barchart example

3.1.3.1 Deploying shiny apps

There are different ways shiny apps can be deployed. For example:

- Open source shiny server (self hosted) - for Linux
- Shiny server pro (self hosted, commercial) - for Linux
- Cloud Service - shinyapps.io (free and paid plans available)

The open-source shiny server is relatively easy to install and configure, if you are familiar with the Linux operating system. Compared to the free version, the paid server version ($9,995 per year) offers additional features and comfort such as:

- Authentication (OAuth, SSL, Active Directory etc.)
- Better scalability
- Monitoring, metrics & management

If you do not can or want to install and maintain extra software, there is the possibility to deploy shiny apps to a cloud service, which will take care of hosting for you. One example for such a service is *shinyapps.io*. The procedure is as follows:

- Create an account at shinyapps.io (you will need a valid email address)
- Login at shinyapps.io and copy the access token code
- Install the R package *rsconnect* locally, for easier deployment
- Enter the access token code in RStudio preferences *(Tools -> Global Options -> Publishing)*
- Use rsconnect one-click publish button to deploy your app to shinyapps.io
- Manage your shiny apps on the shinyapps.io dashboard

The following table gives a short comparision of the different deployment options for shiny webapplications.

Type	Pros	Cons
Open source shiny server	Free, full control	Only Linux, more work to install/configure
Shiny server pro (paid)	Support, additional features (scaling, authentication etc.)	expensive, only Linux
shinyapps.io	Easy, no installation/hosting needed, limited free plan available	No full control, cloud service

Table 3.1: Shiny deployment options comparision

3.1.4 ggvis

In this section we describe the interactive data visualization package *ggvis*. It allows the user to declaratively create interactive graphics. The syntax is strongly inspired by the widely used static graphics package *ggplot2*. Users who are familiar with *ggplot2* should therefor have no problems understanding the principles of *ggvis*. One obvious difference is that ggvis uses a functional interface, so components are connected by using the pipe-like operator %>% instead of +.
Graphics created with ggvis can be run locally in RStudio or in a browser. If run in RStudio the result is displayed in the viewer panel, which is possible due to the fact that RStudio itself is also a web browser. It is also possible to use the *shiny* infrastructure to publish ggvis graphics.
The goal of ggvis, is to combine the best aspects of R and the Web. Graphics and transformations are programmed in R, and *Vega* is used to render the result in the browser.
gvvis and Vega have a similar relationship as gglpot2 and grid. The user does not need any knowdledge of Vega to use the basic functionality of ggvis.

```
1 library(ggvis)
2 library(dplyr)
3
4 # Barchart of the build-in pressure/temperature data set
5 pressure %>%
```

```
6   ggvis(~temperature, ~pressure) %>%
7   layer_bars()
```

Listing 3.4: ggvis sample code for producing a simple graph

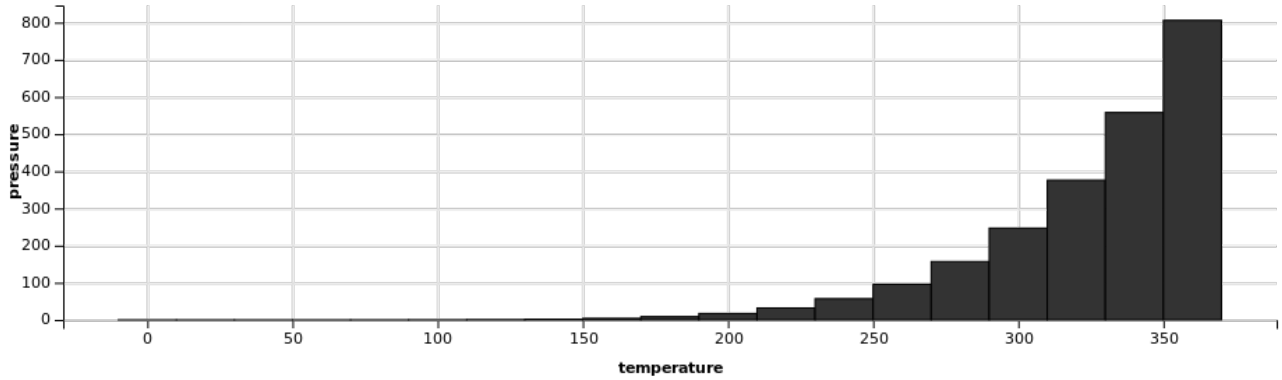

Figure 3.6: Example of a barchart drawn by ggvis

3.1.4.1 *ggvis* interaction with *shiny*

ggvis interactivity is built on top of shiny‘s reactive programming model. Most of the basic interactive controls in ggvis have a similar equivalent in shiny.
Compared to shiny however, ggvis functions have been implemented with the goal to simplify and minify the argument list. Some arguments are therefor optional in ggvis controls (for example *label*) and have a different order compared to the same shiny function.
It is also possible to include ggvis graphics within shiny, as we can see in the following examples.

```
1  library(shiny)
2  library(ggvis)
3
4  ui <- fluidPage(sidebarLayout(
5    sidebarPanel(
6      sliderInput("n", "Number of points", min = 1, max = nrow(mtcars),
7                  value = 10, step = 1),
8      uiOutput("plot_ui")
9    ),
10   mainPanel(
11     ggvisOutput("plot")
12   )
13 ))
14
15 server <- function(input, output, session){
16   # A reactive subset of mtcars
17   mtc <- reactive({ mtcars[1:input$n, ] })
18
19   # A simple visualisation. In shiny apps, need to register observers
20   # and tell shiny where to put the controls
21   mtc %>%
```

```
    ggvis(~wt, ~mpg) %>%
    layer_points() %>%
    bind_shiny("plot", "plot_ui")
}

shinyApp(ui = ui, server = server)
```

Listing 3.5: ggvis sample code for producing an interactive shiny graph

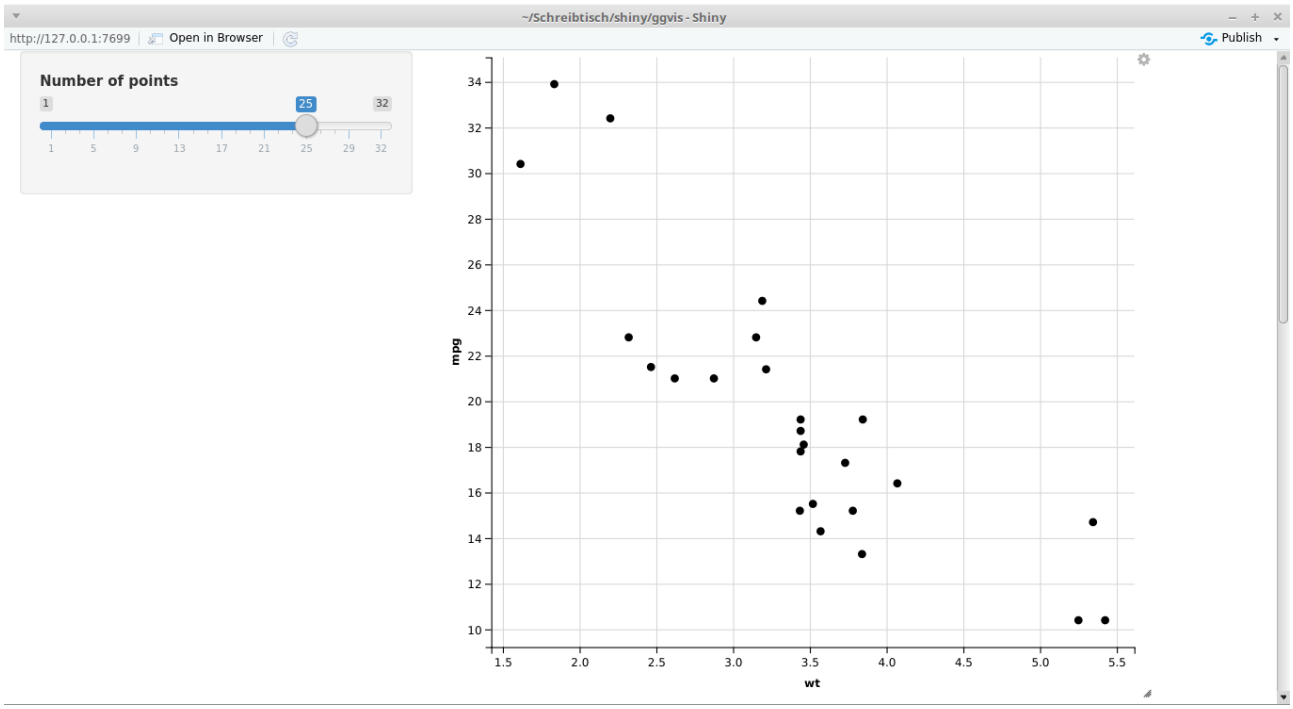

Figure 3.7: Example of an interactive graph in ggvis and shiny

3.1.5 rbokeh

Rbokeh is a native R plotting library with a flexible declarative interface for creating interactive web-based graphics, built upon the Bokeh visualization library. Plot rendering in rbokeh is done by using HTML canvas. Rbokeh provides a lot mechanisms for interactivity and furthermore even complex plots can be built quickly with a few simple commands.

The simplicity of rbokeh is shown in listing 3.6. In just a few lines of code it is possible to create a scatterplot where hovering a data point leads to further information. In this case the user has to provide which data shall be assigned to which axis, the data set itself, further attributes like the color scheme of the scatterplot as well as defining which information shall be provided on hover. The result can be seen in figure 3.8.

Unfortunately rbokeh is still under development, thus making it a good package to develop a simple prototype quickly but if real production code is needed it is recommended to use another package.

```
library(rbokeh)
```

```
figure() %>%
  ly_points(Sepal.Length, Petal.Length, data = iris,
            color = Species,hover = list(Sepal.Length, Petal.Length))
```

Listing 3.6: rbokeh sample code for producing a simple scatterplot using the Iris data set

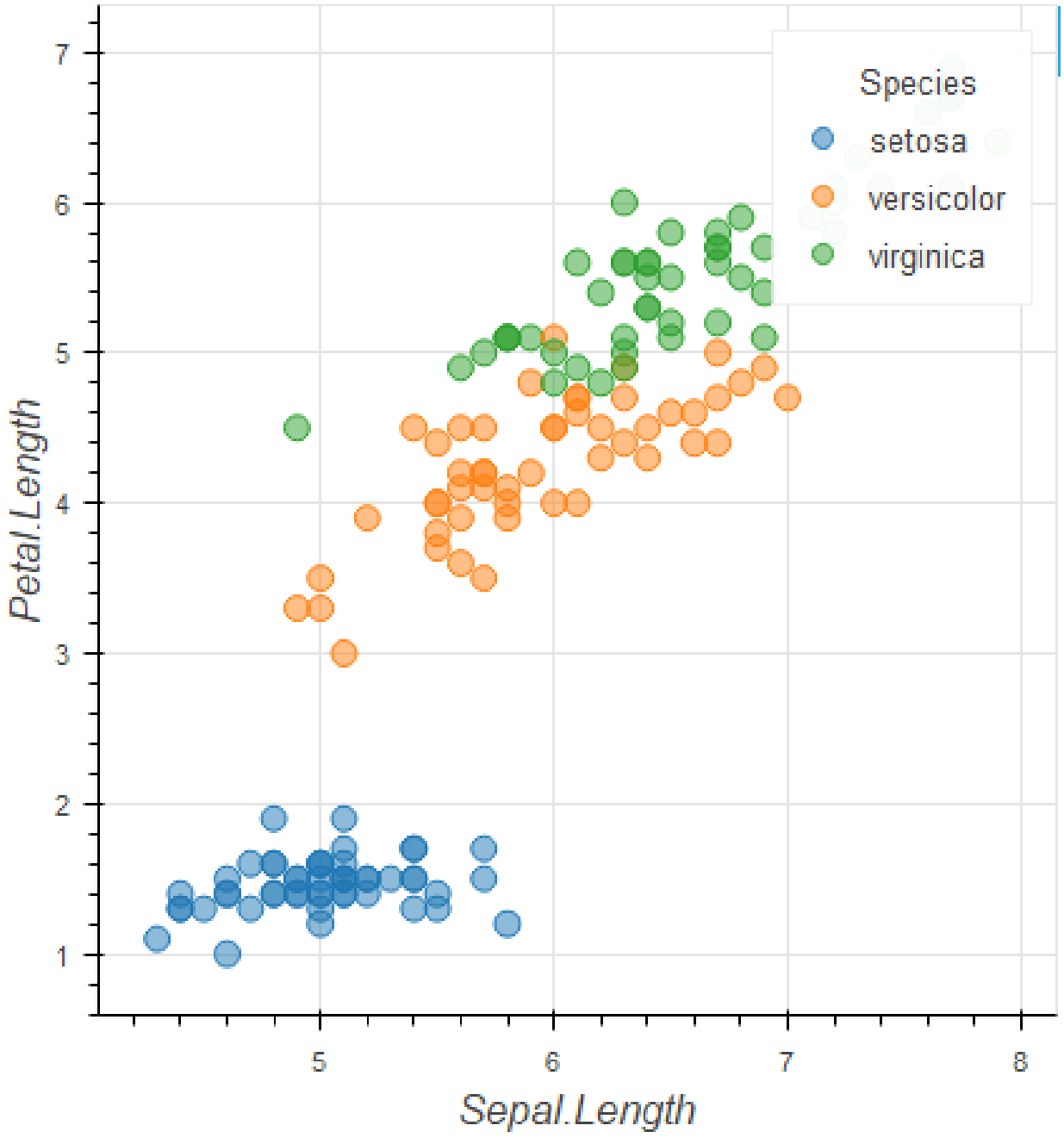

Figure 3.8: Example of scatterplot drawn by using rbokeh package

3.1.6 Plotly

Plotly is another package in R used to create interactive web-based plots with the help of the JavaScript library plotly.js. In general Plotly runs locally in the web browser or in the R studio viewer and the graphs are rendered through the htmlwidgets framework. Besides offline rendering it is also possible to publish plots to Plotly's web service and modify the data or share it with others to enable collaboration on graphs.

In listing 3.7 we recreated the same scatterplot as with rbokeh. The code is even shorter than in the previous example, one has just to provide the data, allocating the data to the axes as well as specifying the color attribute. Plotly also automatically provides information when the data points are hovered. The final result can be seen in figure 3.9.

```
1 library(plotly)
2 plot_ly(data = iris, x = ~Sepal.Length, y = ~Petal.Length, color = ~Species)
```

Listing 3.7: Plotly sample code for producing a simple scatterplot using the Iris data set

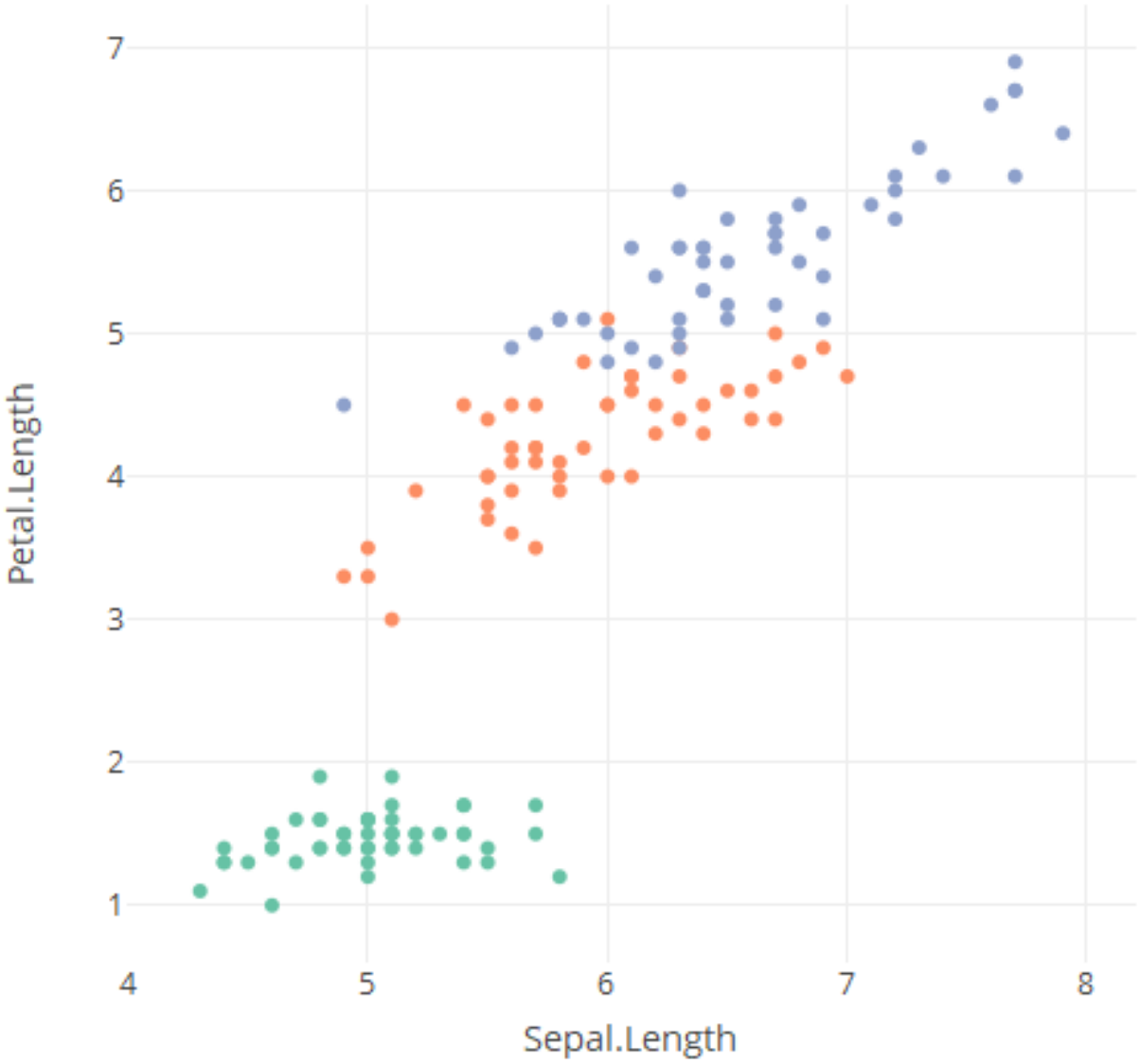

Figure 3.9: Example of scatterplot drawn by using Plotly package

3.2 Other R Packages for Interactive Data Visualization

While we researched on the most popular packages for interactive data visualization in R, discussed in the previous chapter, we also came across some other packages which attracted our attention. Following we want to give a brief introduction to *rCharts* and *highcharter*.

3.2.1 rCharts

The philosophy behind the rCharts package is to provide an easy way of creating, customizing and sharing interactive visualizations. rCharts uses a lattice styled plotting interface, which is a formula interface to specify plots. Furthermore the package supports multiple JavaScript charting libraries where it takes advantage of each ones strength. Since each library has multiple options to customize a plot, rCharts supports most of these

options. There are multiple ways to publish a visualization, either as a standalone page, embedded in a shiny app or in a tutorial/blog post.

3.2.2 highcharter

Highcharter is a wrapper for the Highcharts library which includes shortcut functions to plot R objects. The package consists of various chart types with the same style, e.g. scatter plots, line charts, bar charts etc. Only one function - *hchart(x)* - is needed to create different R objects. Highcharter also supports Highstock and Highmaps charts, where for example a candlestick chart can be created in two lines of code. Additionally it is possible to customize charts since highcharter comes with a huge variety of themes as well as plugins.

Chapter 4

Concluding Remarks

Bibliography

Johnston, Susane [2013]. *R Base Graphics: An Idiot's Guide*. RPubs by RStudio. 2013. https://rpubs.com/SusanEJohnston/7953 (cited on page 5).

Kabacoff, Robert I. [2011]. *R in Action - Data Analysis and Graphics with R*. 1st Edition. Manning Publications Co., 2011. ISBN 9781935182399 (cited on page 5).

Kabacoff, Robert I. [2017a]. *Lattice Graphs*. Quick-R, DataCamp. 2017. https://www.statmethods.net/advgraphs/trellis.html (cited on pages 6–7).

Kabacoff, Robert I. [2017b]. *Scatterplots*. Quick-R, DataCamp. 2017. https://www.statmethods.net/graphs/scatterplot.html (cited on page 5).

Murrell, Paul [2006]. *R Graphics*. 1st Edition. Chapman, Hall,CRC Taylor and Francis Group, 2006. ISBN 9781584884866. https://www.e-reading.club/bookreader.php/137370/C486x_APPb.pdf (cited on page 3).

Murrell, Paul [2012]. *R Graphics*. 2nd Edition. CRC Press Taylor and Francis Group, 2012. ISBN 9781439831779 (cited on page 5).

Murrell, Paul [2018]. *grid Graphics*. 2018. http://stat.ethz.ch/R-manual/R-patched/library/grid/doc/grid.pdf (cited on page 3).

Rickert, Joseph [2015]. *Some Basics for Base Graphics*. R-bloggers. 2015. https://www.r-bloggers.com/some-basics-for-base-graphics/ (cited on page 5).

Sarkar, Deepayan [2008]. *Lattice - Multivariate Data Visualization with R*. 1st Edition. Springer, 2008. ISBN 9780387759685 (cited on page 6).

Sarkar, Deepayan [2017]. *Package 'lattice'*. CRAN - The Comprehensive R Archive Network. 2017. https://cran.r-project.org/web/packages/lattice/lattice.pdf (cited on page 6).

Teetor, Paul [2011]. *R Cookbook - Practical Recipes for Visualizing Data*. 1st Edition. O'Reilly Media Inc., 2011. ISBN 9780596809157. http://www.bagualu.net/wordpress/wp-content/uploads/2015/10/R_Cookbook.pdf (cited on page 5).

TRF [2018]. *The R Project for Statistical Computing*. The R Foundation. 2018. https://www.r-project.org/ (cited on page I).

Wickham, Hadley [2010]. "A Layered Grammar of Graphics". *Journal of Computational and Graphical Statistics* 19.1 [2010], pages 3–28. doi:10.1198/jcgs.2009.07098 (cited on page 8).

Wickham, Hadley [2016]. *ggplot2 - Elegant Graphics for Data Analysis*. 2nd Edition. Springer, 2016. ISBN 9783319242750 (cited on page 9).

Wilkinson, Leland [2005]. *The Grammar of Graphics*. 2nd Edition. Springer, 2005. ISBN 9780387245447 (cited on page 8).