Advanced graphics in R

Antonio Gasparrini, Arturo de la Cruz Libardi

01 November 2023

Contents

This session introduces the GGPLOT2 package and graphical systems. Among various graphical methods in R, GGPLOT2 is the most elegant and advanced. The GGPLOT2 system and package implement the *grammar of graphics*, which provides a coherent theoretical framework for defining and building graphs. The actual process is based on sequential calls to specific functions that set the graphical structure, add layers of graphical elements, and define the graphical parameters. This allows fine control over every plotting aspect and the creation of fully-tailored graphs. The first two sections introduce GGPLOT2 and its framework. The third and fourth sections elaborate on the concepts of *geoms* and *aesthetic mapping*, which are key features of the GGPLOT2 system. The following two sections focus on visualisation, introducing first *scales* and *themes*, and then labelling, the coordinate system, and *faceting*. The last section briefly demonstrates how to export a GGPLOT2 graph.

Introduction to ggplot2

R is well-known for its exceptional capabilities for making graphs, provided through different systems. The *traditional* (or *base*) system is implemented through the standard package GRAPHICS, and offers a wide range of graphical tools. More sophisticated alternatives, based on the *grid* system, are implemented in packages such as LATTICE and, more recently GGPLOT2. The latter provides state-of-the-art graphical tools, and it is considered the most elegant and versatile option. The GGPLOT2 system and package implement the *grammar of graphics*, a coherent system for describing and building graphs that allows a consistent structure and syntax.

Compared to the base graphic system, in GGPLOT2 graphical features are controlled through the use of specific functions, not graphical parameters. This allows for comprehensive and fine control over every single

element of the plot, although it requires a more complex syntax and usage. In particular, in GGPLOT2 plots are created by sequential calls to various functions that set the plotting structures, define visual properties, add geometrical elements, and set graphical parameters. While this requires the knowledge of several different functions and technical details, it offers a unified framework for producing several types of high-level graphs using consistent processes and principles.

GGPLOT2 is part of *tidyverse*, a collection of R packages that are built using a coherent design and the same underlying data format. Tidyverse offers a wide range of tools to perform data management and other analytical tasks using a unified data structure that allows sequential operations. This makes it straightforward to combine data transformation, performed with other tidyverse packages, together with graphical outputs with GGPLOT2.

Before digging into the features of GGPLOT2, we first load the package, assuming it has been already installed, possibly together with the whole tidyverse set:

```
library(ggplot2)
```

In the next sections, we will provide a general introduction to GGPLOT2. The illustration is helped by real-data examples of creating graphs, but the focus is on discussing key aspects of the system, not on demonstrating the graphical capabilities of GGPLOT2. In addition, and more importantly, we can only offer a broad overview, without delving into the vast assortment of functionalities offered by the package.

Structure of ggplot2

In GGPLOT2, plots are created by a sequence of function calls. The first step is always a call to the function `ggplot()`, which creates the basic graphical structure. Let's see an example using the dataset MTCARS from the package DATASETS, specifically by representing the relationship between the weight of the car (variable `wt`) and its consumption (as miles per gallon, variable `mpg`). See `help(mtcars)` for details. This is the code:

```
ggplot(data=mtcars, mapping=aes(x=wt, y=mpg))
```

The first two arguments of `ggplot()` are `data` and `mapping`. The former identifies the dataset where the variables are stored, which, consistently with tidyverse, is usually expected in a tidy format. The argument `mapping` instead defines the *aesthetics*, namely visual properties of the graph that are passed through the function `aes()`. This function maps the variables to the aesthetics using non-standard evaluation with unquoted names, as usual in tidyverse. In this case, the two main aesthetics `x` and `y`, representing the *x* and *y*-axis, respectively, are mapped using the corresponding variables.

The result of the simple call to `ggplot()`, shown in the left panel of the figure below, is an empty plot where only the basic graphical structure has been set up. We can now add graphical elements to the plot by additional calls to specific functions called *geoms*. The sequence of calls is specified in GGPLOT2 by a special syntax involving the operator '+'. For instance, here we add points to the plot by calling the function `geom_point()`, which specifies a scatter plot. Here is the code:

```
ggplot(data=mtcars, mapping=aes(x=wt, y=mpg)) + geom_point()
```

The resulting plot is shown in the right panel of the figure below, illustrating how the geometrical elements corresponding to points have been added to the graphical structure created by the first call to `ggplot()`.

Similarly to the scatter plot created with the function `plot()` in the GRAPHICS system, this basic use of GGPLOT2 functions creates a simple graph, with no colour and uninformative labels. However, the default graphical aspects of GGPLOT2 offer better spacing (without the empty spaces around the plot region), a better magnification of the labels, and a grey background with an additional grid that improves readability.

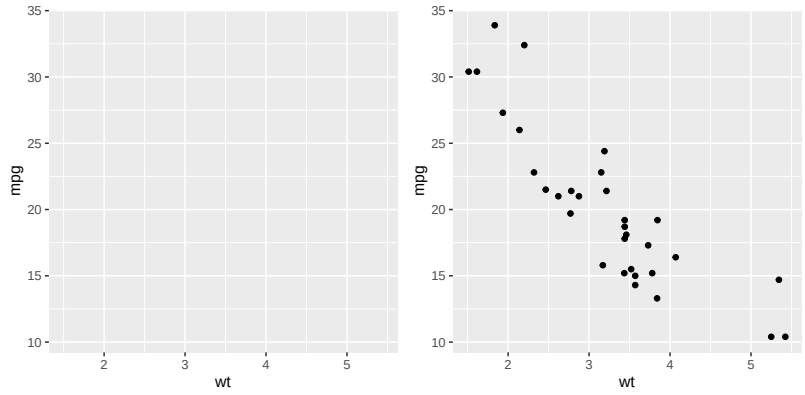

Figure 1: The basic plot structure (left) and the points added through a geom (right)

Using different geoms

Differently from the GRAPHICS system where different graphs can be created by calls to the same generic function `plot()`, GGPLOT2 does not embrace an object-oriented programming style. Various types of plots are obtained by using different geoms through separate functions `geom_*()`. These functions include arguments defining additional aesthetics, some general (such as `colour`, `fill`, `shape`, and `size`) and some specific to each geom.

Let's see some examples. We start with another scatter plot using the same function `geom_point()` seen above, but this time using the dataset BIRTHS in the package EPI (see `help(births)` for more information. We first load this contributed package (which might also be installed, if not):

```
library(Epi)
data(births)
```

We now visualise the relationship between birth weight (variable `bweight`) of the baby and gestational week (variable `gestwks`). The code:

```
ggplot(births, aes(x=gestwks, y=bweight)) +
  geom_point(colour="red", shape=20, size=4, alpha=0.2)
```

We can see how default choices for specific aesthetics can be replaced with different values for the geometrical objects (the points), specifically defining a red colour, a solid circle shape, and larger size, using corresponding arguments. An interesting aesthetic is `alpha`, which modifies the colour transparency using a range between 0 (fully transparent) to 1 (not transparent), thus improving the visualisation of overlapping points. The resulting plot is represented in the left panel of the figure below.

Multiple geoms can be used in the same plot to add various graphical elements and multiple *layers* of the plot, just by adding calls to other functions through the operator '+'. As an example, let's produce another scatter plot displaying the series of daily temperatures in New York during June 1973 using the observations in the dataset AIRQUALITY from the package DATASETS (see `help(airquality)` for details). Here is the code:

```
library(dplyr)
filter(airquality, Month==6) |>
  ggplot(aes(x=Day,y = Temp)) +
  geom_line(linetype="dotdash", linewidth=0.5) +
  geom_point(shape=22, size= 3, fill="green")
```

Here we first call the function `geom_line()` after `ggplot()` to represent the series with connected lines, before plotting the points with `geom_point()`. You can notice that specific line aesthetics are set through the arguments `linetype` and `linewidth` of `geom_line()`, while different point shapes and sizes are selected

for the points, in particular defining filled squares with green colour. The plot is shown in the right panel of the figure below.

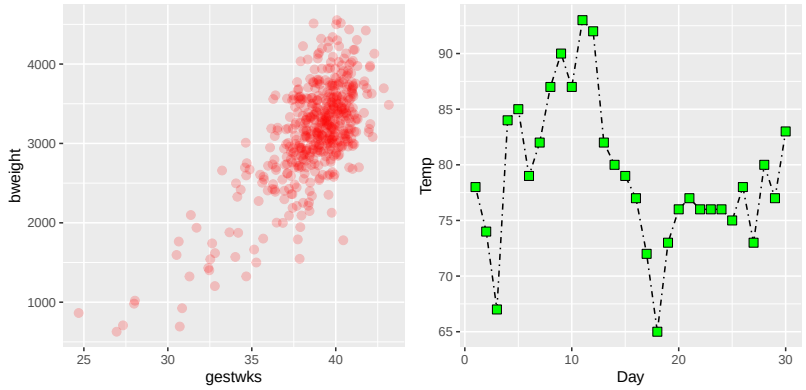

Figure 2: A simple scatter plot (left) and a scatter plot with lines (right) with non-default aesthetics

It is interesting to note that in the series of expressions above, we can first subset the data frame `airquality` using the function `filter()` from the package `dplyr` (which must be loaded into the session), and then call `ggplot()` and the following geom functions. This demonstrates how GGPLOT2 integrates with other tidyverse packages, and how a potentially long set of transformations and plotting can be coded using sequential calls through the pipe operator '`|>`' and the operator '+', respectively.

GGPLOT2 offers a wide set of geoms to produce different types of plots. See `help(ggplot2)` for the complete list. Below, we add some examples using the functions `geom_bar()`, `geom_histogram()` and `geom_boxplot()` to produce a bar plot, a histogram, and a box plot (also known as box-and-whiskers plot), respectively, using the same datasets introduced earlier in addition to ESOPH from the package DATASETS (see `help(esoph)` for details). The code is the following, with the three graphs represented in the figure below:

```
ggplot(mtcars, aes(x=carb)) + geom_bar(col=1, fill=3)
ggplot(births, aes(x=bweight)) + geom_histogram(col=1, fill="lightskyblue")
ggplot(esoph, aes(agegp, ncases)) + geom_boxplot(fill=4)
```

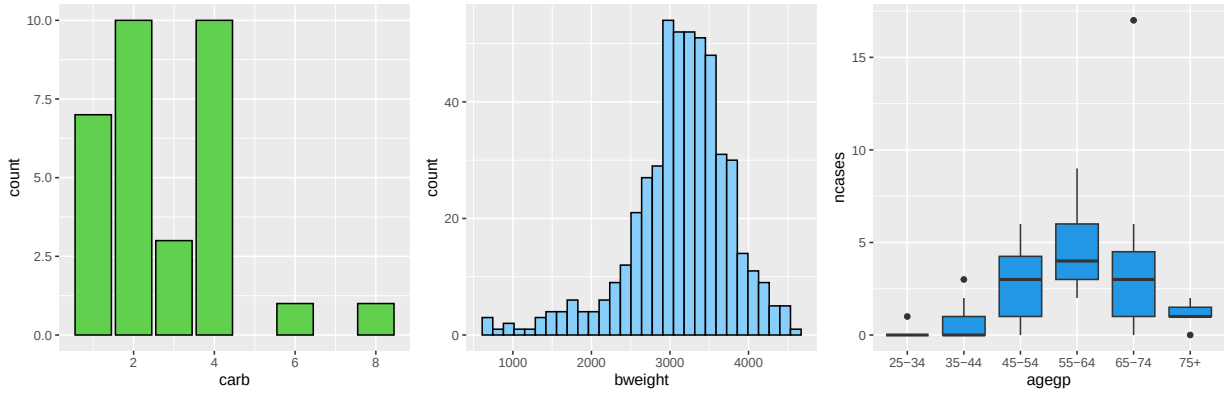

Figure 3: A bar plot (left), a histogram (mid), and a box plot (right) produced by different geom functions

Aesthetic setting and mapping

So far, we have seen two different aesthetic definitions in the plots above. In the first usage, the aesthetics `x` and `y` were *mapped* using the function `aes()` to specific variables that define the *x* and *y*-axis, respectively. In the second usage, other aesthetics such as `colour`, `shape`, or `linetype` were *set* to arguments of geom functions to choose the corresponding graphical features. Generally, we *map* through `aes()` if we want the visual attribute to vary based on the values of a variable, while we *set* if we want a fixed value for the aesthetic.

This allows a more sophisticated usage, whereby any aesthetic can be mapped to let the corresponding graphical features change depending on the value of continuous or categorical variables. Let's see an example:

```
ggplot(data=mtcars, mapping=aes(x=wt, y=mpg, col=factor(gear))) +
  geom_point(size=4)
```

Here, the size aesthetic is still set as an argument of `geom_point()`. However, colour (abbreviated as `col`) is moved into the call to `aes()` within `ggplot()`, specifically mapping the variable `gear` (transformed in a factor) to this aesthetic. This allows assigning different colours to points corresponding to different numbers of forward gears of the car, as illustrated in the resulting plot shown in the left panel of the figure below. You can notice that a legend is automatically included in the plot to display the mapping (it can be excluded by setting `show.legend=F` in the related geom function).

In addition to colour, we can also map any other aesthetics, using the same or different variables. For example, we can map `gear` also to shape by adding a similar expression within `aes()`:

```
ggplot(data=mtcars, mapping=aes(x=wt, y=mpg)) +
  geom_point(aes(col=factor(gear), shape=factor(gear)), size=4)
```

The plot, displayed in the right panel in the figure below, now assigns different colours and shapes to the points based on the number of forward gears. In this case, a single legend is created, while multiple legends would be shown if different variables are used for mapping.

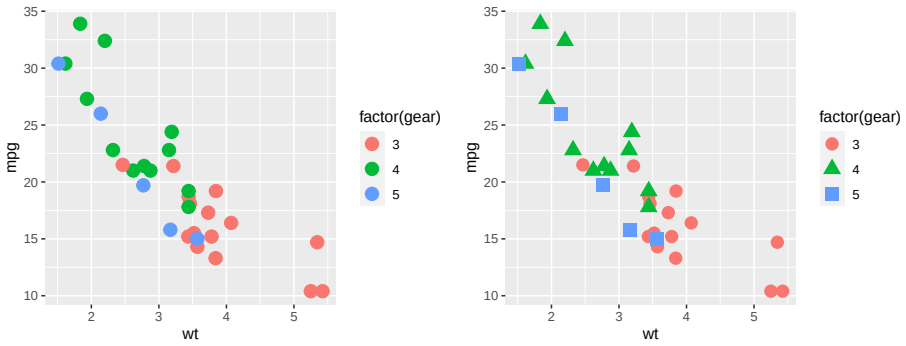

Figure 4: Examples of aesthetic mapping using one (left) or two (right) aesthetics

You can notice that, in the last example, the call to `aes()` has been moved to `geom_point()`. This is an example of *local* mapping, as an alternative to *global* mapping, the latter specified when it takes place within the first call to `ggplot()`, as in the previous examples. In the case of global mapping, the aesthetic definitions are by default *inherited* by all the subsequent geom layers of the plot, while with local mapping these are defined only for the specific geoms.

This system allows for sophisticated usage of the geom functions and aesthetic definitions. Let's see an example where we first add a call to `geom_smooth()` to the previous plot. This geom adds a fitting line to the scatter plot, estimated using different methods. Here, we use `method="lm"` to fit straight lines estimated through a linear model (see `help(geom_smooth)` for more information). The code:

```
ggplot(data=mtcars, mapping=aes(x=wt, y=mpg, col=factor(gear))) +
  geom_point(size=4) +
  geom_smooth(method="lm")
```

The resulting graph is shown in the left panel of the figure below. In this case, the global mapping of the colour aesthetic is inherited by all the geoms, and different fitting lines are created for the sets of points corresponding to different values of the variable `gear`.

Alternatively, we can make use of local mapping to define the colour aesthetic only within `geom_point()`:

```
ggplot(data=mtcars, mapping=aes(x=wt, y=mpg)) +
  geom_point(aes(col=factor(gear)), size=4) +
  geom_smooth(method="lm", col=1)
```

As shown in the right panel of the figure above, this alternative usage keeps the colour mapping for the points, but it allows a single fitting line to be added.

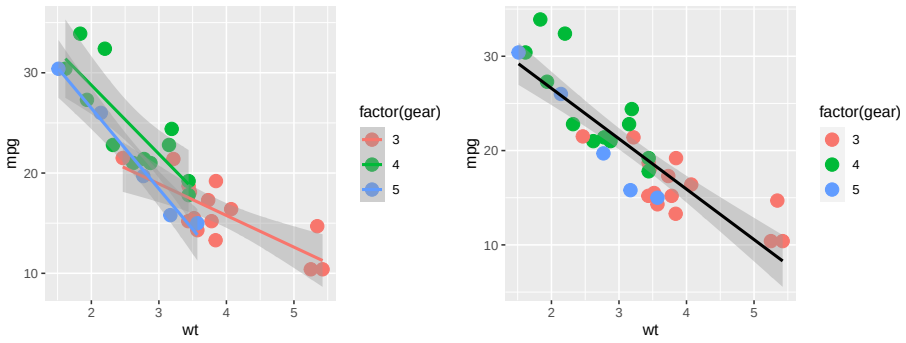

Figure 5: Group-specific (left) and overall (right) fitting lines added to a scatter plot

Inheritance can be overruled by an additional local call to `aes()` in the geoms, where aesthetics can be re-mapped using different variables, or by adding `inherit.aes=F` in the corresponding geom function. In addition, inheritance also applies to the `data` argument of `ggplot()`, which alternatively can be defined locally, allowing the plotting of data from different datasets within each geom.

Scaling and themes

Aesthetic mapping, either global or local, involves the actual translation of data values in the variable to specific visual properties of the plot, a process defined as *scaling*. This is exemplified by the selection of specific axis ranges and tick marks for the `x` and `y` aesthetics, or the assignment of colours and line types for `fill` and `linetype`, among others.

Normally, GGPLOT2 performs this scaling procedure internally, defining default *scales* that are suitable for most of the applications. However, in many cases it is important to control this process by defining scales with non-default features, allowing to fine-tune the plotting. This can be done through specific functions with form `scale_*_*()`, where the first asterisk is a placeholder for the selected aesthetic, and the second one for specific scaling processes. We provide some examples below to clarify the issue.

Before starting, we show another property of GGPLOT2, namely the possibility to assign (part of) a plot to an object through the usual assignment operator '<-' and then re-use it later. For instance, we can store the previous plot in an object `basic`:

```
basic <- ggplot(data=mtcars, mapping=aes(x=wt, y=mpg, col=factor(gear))) +
  geom_point(size=4)
```

This object can be called to display the plot, or used in further expressions to add layers or scales through the operator '+'. This property allows storing plots and simplifying the syntax.

We start by showing how to define a non-default scaling of the aesthetic `x` that specifies the *x*-axis. This is done through the function `scale_x_continuous()` (or similar) that includes a number of arguments to modify the representation of the axis (see `help(scale_x_continuous)` for details). For example:

```
basic + scale_x_continuous(n.breaks=10)
```

The direct definition of the scale overrides the default selection of the number of tick marks (and related labels and grid) using the argument `n.breaks`, as shown in the figure below (left panel). You can notice how the scale is added directly to the object `basic`, avoiding redefining (potentially long) expressions.

As another example, we change the default scaling of the colours using the function `scale_colour_brewer()`. Specifically, the `brewer` scales provide sequential, diverging, or qualitative colour schemes for categorical variables, in this case the factor transformation of `gear`, which was originally mapped to the `colour` aesthetic. Let's see:

```
basic + scale_colour_brewer(name="Gears (N)", palette="Greens")
```

The result is displayed in the mid panel of the figure below. Here, rather than using the default colour scheme, we define a specific green palette, in addition to changing the name to the scale that is also reported in the legend (see `help(scale_colour_brewer)` for more information).

In addition to scaling, and complementary to it, graphical features of plots in GGPLOT2 are defined by *themes*. Specifically, these refer to non-data components such as labels and fonts, background, and grid lines, as well as legends. Again, themes are pre-specified using default values, but all of them can be customised through the function `theme()`. For instance, we can change the previous plot and place the legend at the top using the following code, with the result displayed in the right panel of the figure below:

```
basic + scale_colour_brewer(name="Gears (N)", palette="Greens") +
  theme(legend.position="top")
```

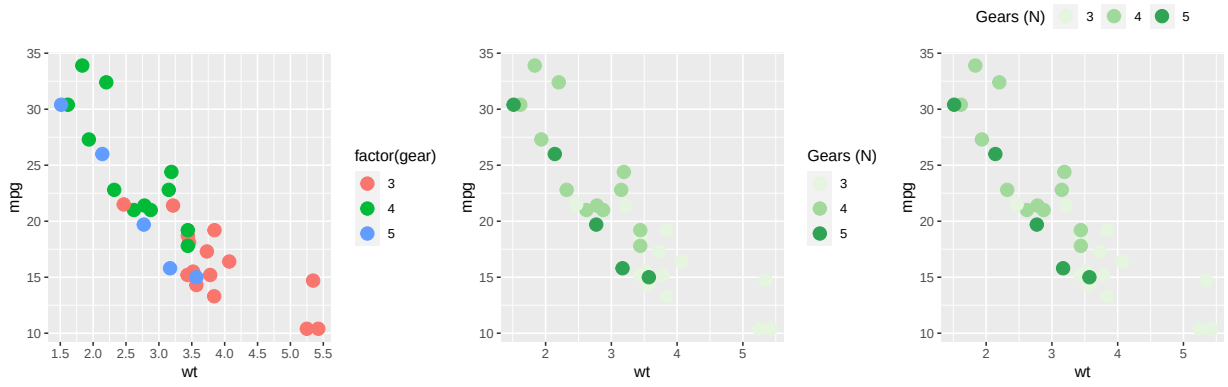

Figure 6: Changing the default scaling of the x-axis (left) and the colours (right)

As shown in the related help page (see `help(theme)`), the function `theme()` includes a long list of arguments to define various graphical features of the plot. However, there are pre-defined themes with consistent but alternative layouts that can be selected through dedicated functions with form `theme_*()`. For instance:

```
basic + scale_colour_brewer(name="Gears (N)", palette="Greens") + theme_bw()
basic + scale_colour_brewer(name="Gears (N)", palette="Greens") + theme_classic()
basic + scale_colour_brewer(name="Gears (N)", palette="Greens") + theme_dark()
```

The three options, shown in different panels of the figure below, offer alternative themes that can be suitable

in different situations. For example, by removing the light grey layout (or making it darker), it is now possible to visualise the light green points in the first group of `gear`.

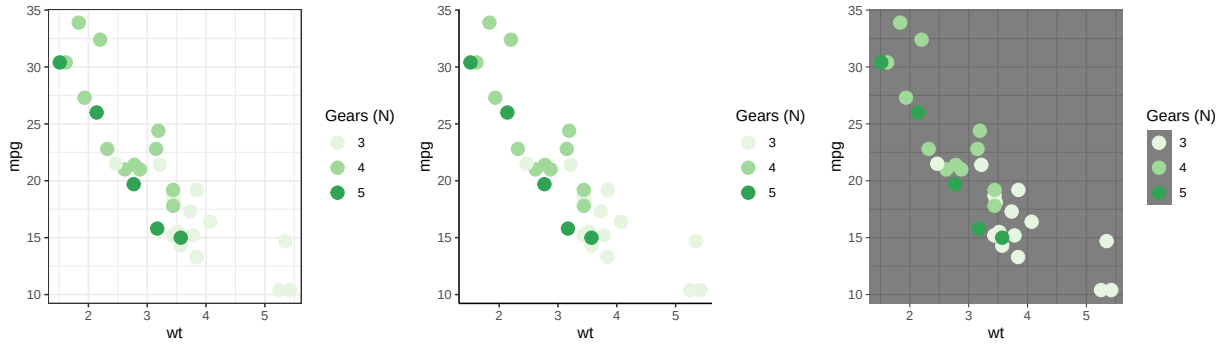

Figure 7: Different pre-defined themes

Similarly, the function `guides()` can be used to modify the scaling procedure, in particular regarding its mapping to the legend. See `help(guides)` for details.

The process of changing scales, themes, or guides involves the use of several internal functions and a complex syntax, and it is not possible to provide a comprehensive overview here. The suggestion is to look at the related help pages and examples, in addition to external documentation.

Labels, the coordinate system, and faceting

In this section, we complete the illustration of the main tools available in GGPLOT2 to customise plots, discussing labelling and layout features. Regarding the former, labels referring to different graphical components can be specified through the function `labs()`, for instance:

```
basic + labs(title="A title here", caption="Footnote here",
  x="Weight (x 1000 lb", y="Miles per gallon")
```

You can notice (left panel in the figure below) that a title, caption (footnote), and axis labels have been added to the graph. Alternatively, you can use dedicated functions `ggtitle()`, `xlab()`, and `ylab()` (see the related help pages).

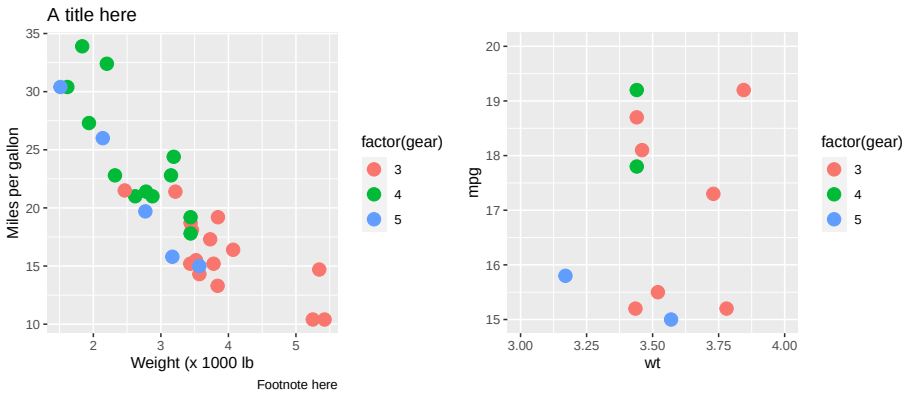

Figure 8: Added labels (left) and changes in the axis ranges (right)

The plotting of the graphical elements occurs within a pre-defined *coordinate system*, which by default is

the Cartesian system where independent *x* and *y* positions determine the location of the element. As usual, default settings for all the features of the system are automatically chosen, for instance the ranges for the *x* and *y*-axis. These can be modified using the functions `xlim()` and `ylim()`, respectively. For instance, we can focus the scatter plot created in the previous section on specific ranges using the following code:

```
basic + xlim(3,4) + ylim(15,20)
```

The result is displayed in the right panel of the figure above An alternative way of handling the coordinate system is offered by the function `coord_cartesian()` (see the related help page).

Another interesting layout feature is the possibility to split the plot into various panels, defined as *facets*. This is particularly useful when mapping aesthetics using categorical variables, especially if it makes the plot cluttered and difficult to make sense of. Faceting can be applied with the function `facet_wrap()` (or alternatively `facet_grid()`). Let's see an example:

```
ggplot(data=mtcars, mapping=aes(x=wt, y=mpg)) + geom_point(size=4) +
  facet_wrap(vars(gear), nrow=1)
```

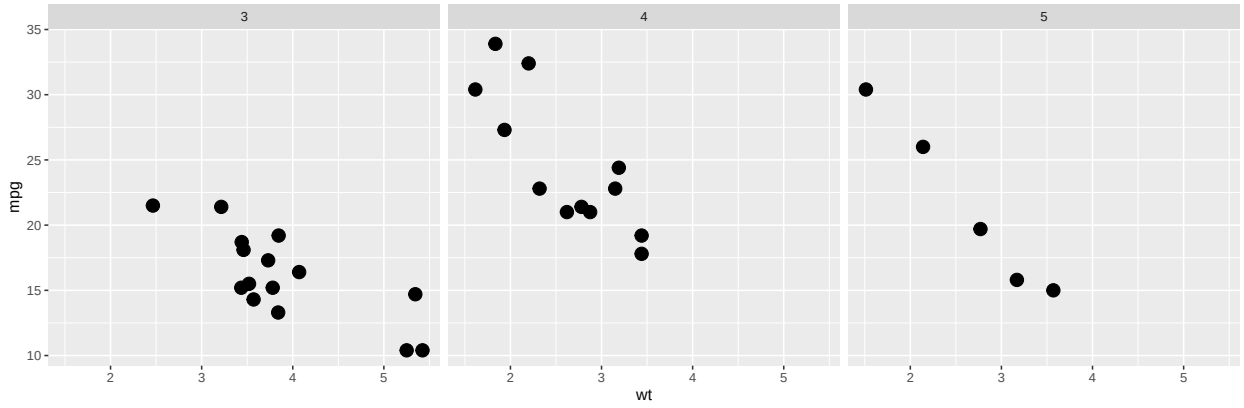

Figure 9: Multi-panel plot obtained through faceting

The resulting series of graphs, shown in the figure below, illustrate the relationship between `wt` and `mpg` in different groups of `gear`, as an alternative method to aesthetic mapping seen in the previous sections. By default, the panels use the same scaling of the axes, but this can be overruled by setting `scales="free"` (or `"free_x"` and `"free_y"` for specific axes). Note also that the package PATCHWORK offers more flexible ways to construct multi-panel graphs.

Exporting graphs

In this final section, we briefly illustrate the process of exporting graphs, that is saving it as an external file outside R. The traditional way is to use specific functions such as `pdf()`, which open an internal *graphic device*, then call plotting functions to produce the graph, and finally close the device through the function `dev.off()`.

This method perfectly works with GGPLOT2. However, this system provides a dedicated function, `ggsave()`, which simplifies the process and allows exporting graphs previously saved in objects. We illustrate an example by exporting the graph previously saved in the object `basic`, using the following code:

```
ggsave("files/myggplot.pdf", plot=basic, width=8, height=7)
```

The graph is now saved as the file MYGGPLOT.PDF in the folder FILES. By default, the last displayed graph is saved if the argument `plot` is left unspecified. In addition, the function `ggsave()` guesses the file format

based on the file extension (in this case, a pdf), although this can be directly set with the argument `device`. See `help(ggsave)` for more information.