© Thomas Mailund 2022

T. Mailund, *Beginning Data Science in R 4*

https://doi.org/10.1007/978-1-4842-8155-0_4

4. Visualizing Data

Thomas Mailund[1]

(1) Aarhus, Denmark

Nothing tells a story about your data as powerfully as good plots. Graphics captures your data much better than summary statistics and often shows you features that you would not be able to glean from summaries alone.

R has excellent tools for visualizing data. Unfortunately, it also has more tools than you really know what to do with. There are several different frameworks for visualizing data, and they are usually not compatible, so you cannot easily combine the various approaches.

In this chapter, we look at graphics in R, and we cannot possibly cover all the plotting functionality, so I will focus on two frameworks. The first is the basic graphics framework. It is not something I frequently use or recommend that you use, but it is the default for many packages, so you need to know about it. The second is the `ggplot2` framework, which is my preferred approach to visualizing data. It defines a small domain-specific language for constructing data and is perfect for exploring data as long as you have it in a data frame (and with a little bit more work, for creating publication-ready plots).

Basic Graphics

The basic plotting system is implemented in the `graphics` package. You usually do not have to include the package

```
library(graphics)
```

since it is already loaded when you start up R, but you can use

```
library(help = "graphics")
```

to get a list of the functions implemented in the package. This list isn't exhaustive, though, since the primary plotting function, `plot()`, is generic and many packages write extensions to it to specialize plots.

In any case, you create basic plots using the `plot()` function. This function is a so-called generic function, which means that what it does depends on the input it gets. So you can give it different first arguments to get plots of various objects.

The simplest plot you can make is a scatter plot, plotting points for *x* and *y* values (see Figure 4-1):

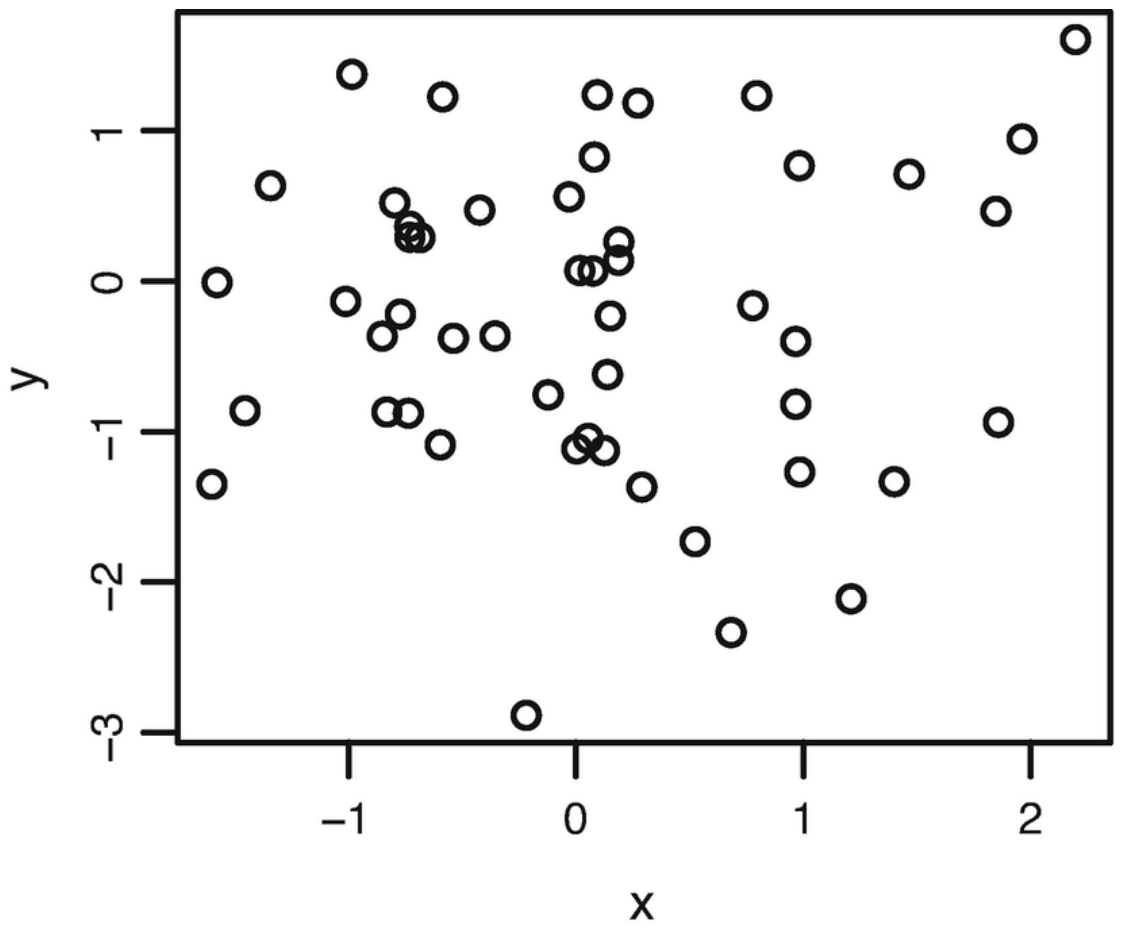

Figure 4-1 Scatter plot

```
x <- rnorm(50)
y <- rnorm(50)
plot(x, y)
```

The `plot()` function takes a data argument you can use to plot data from a data frame, but you cannot write code like this to plot the `cars` data from the `datasets` package:

```
data(cars)
cars %>% plot(speed, dist, data = .)
```

Despite giving `plot()` the data frame, it will not recognize the variables for the x and y parameters, and so adding plots to pipelines requires that you use the `%$%` operator to give `plot()` access to the variables in a data frame. So, for instance, we can plot the `cars` data like this (see Figure 4-2):

```
cars %$% plot(speed, dist, main="Cars data",
                   xlab="Speed", ylab="Stopping distance")
```

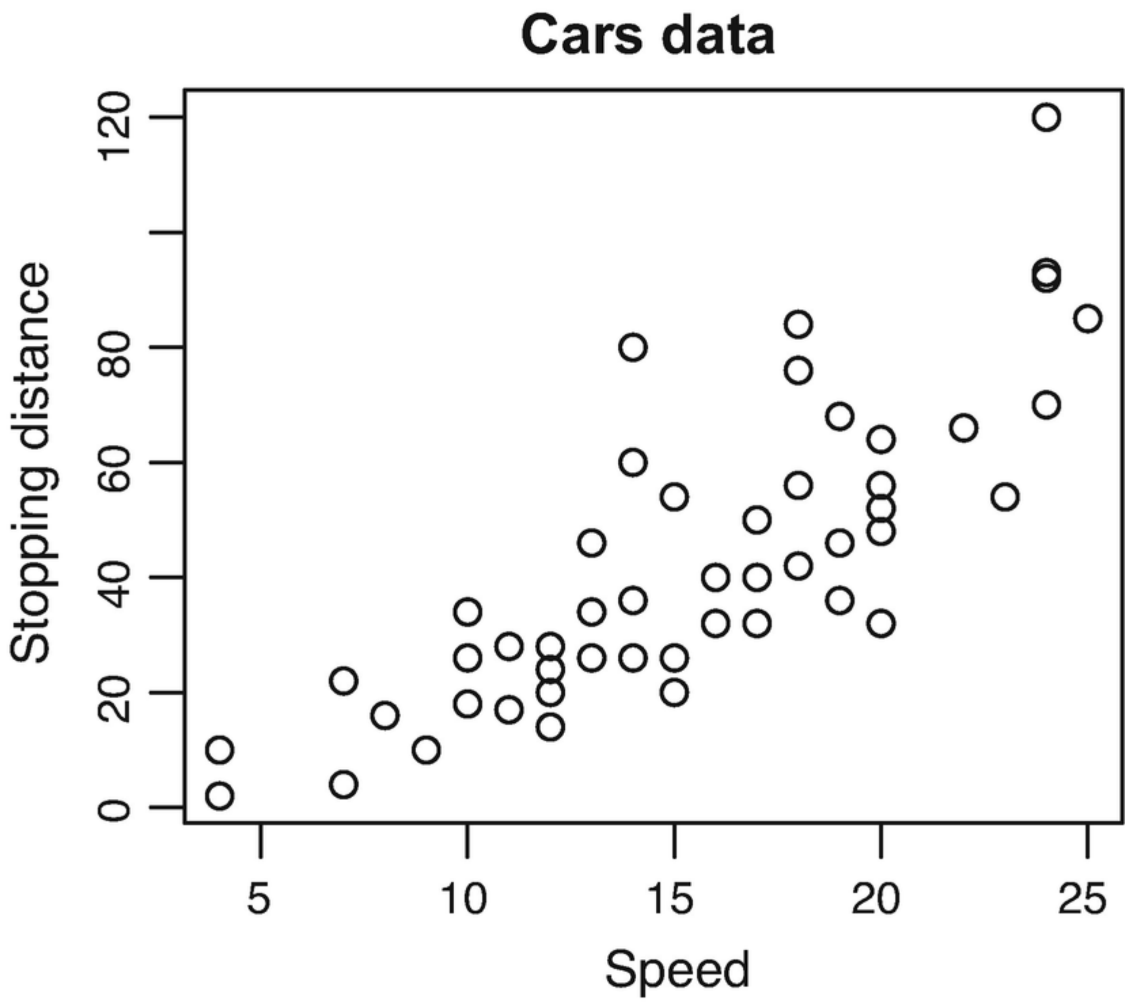

Figure 4-2 Scatter plot of speed and distance for cars

Here, we use `main` to give the figure a title and `xlab` and `ylab` to specify the axis labels.

The data argument of `plot()` is used when the variables to the plot are specified as a formula. The `plot()` function then interprets the formula as specifying how the data should be plotted. If the x and y values are specified in a formula, you can give the function a data frame that holds the variables and plot from that:

```
cars %>% plot(dist ~ speed, data = .)
```

Here, you must use the `%>%` operator and not `|>` since you need the left-hand side to go into the `data` argument and not the first argument.

By default, the plot shows the data as points, but you can specify a `type` parameter to display the data in other ways such as lines or histograms (see Figure 4-3):

```
cars %$% plot(speed, dist, main="Cars data", type="h",
              xlab="Speed", ylab="Stopping distance")
```

To get a histogram for a single variable, you should use the function `hist()` instead of `plot()` (see Figure 4-4):

```
cars %$% hist(speed)
```

What is meant by `plot()` being a generic function (something we will cover in much greater detail in Chapter 12) is that it will have different functionality depending on what parameters you give it.

Different kinds of objects can have their own plotting functionality, though, and often do. This is why you probably will use basic graphics from time to time even if you follow my advice and use `ggplot2` for your own plotting.

Linear regression, for example, created with the `lm()` function, has its specialized plotting routine. Try evaluating the following expression:

```
cars %>% lm(dist ~ speed, data = .) %>% plot()
```

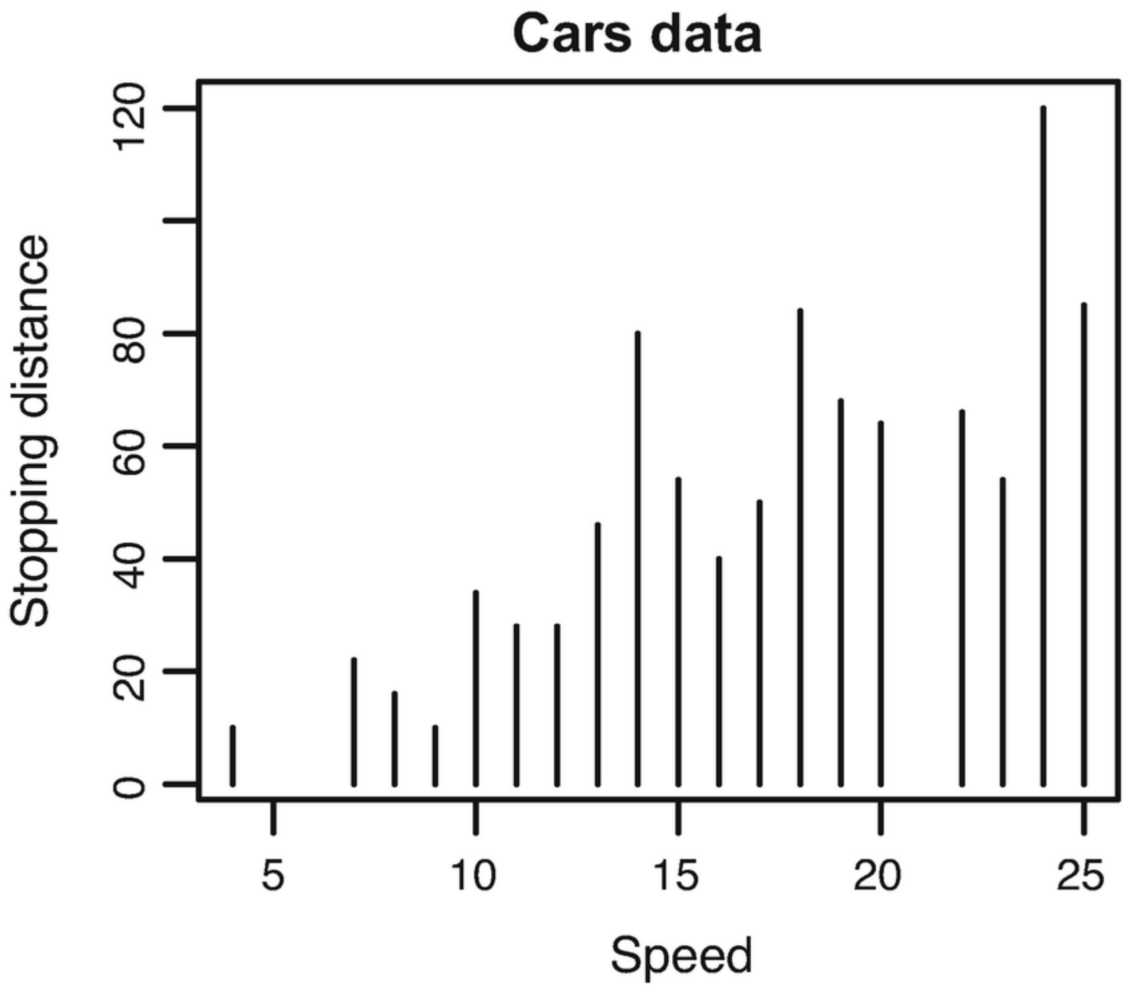

Figure 4-3 Histogram plot of speed and distance for cars

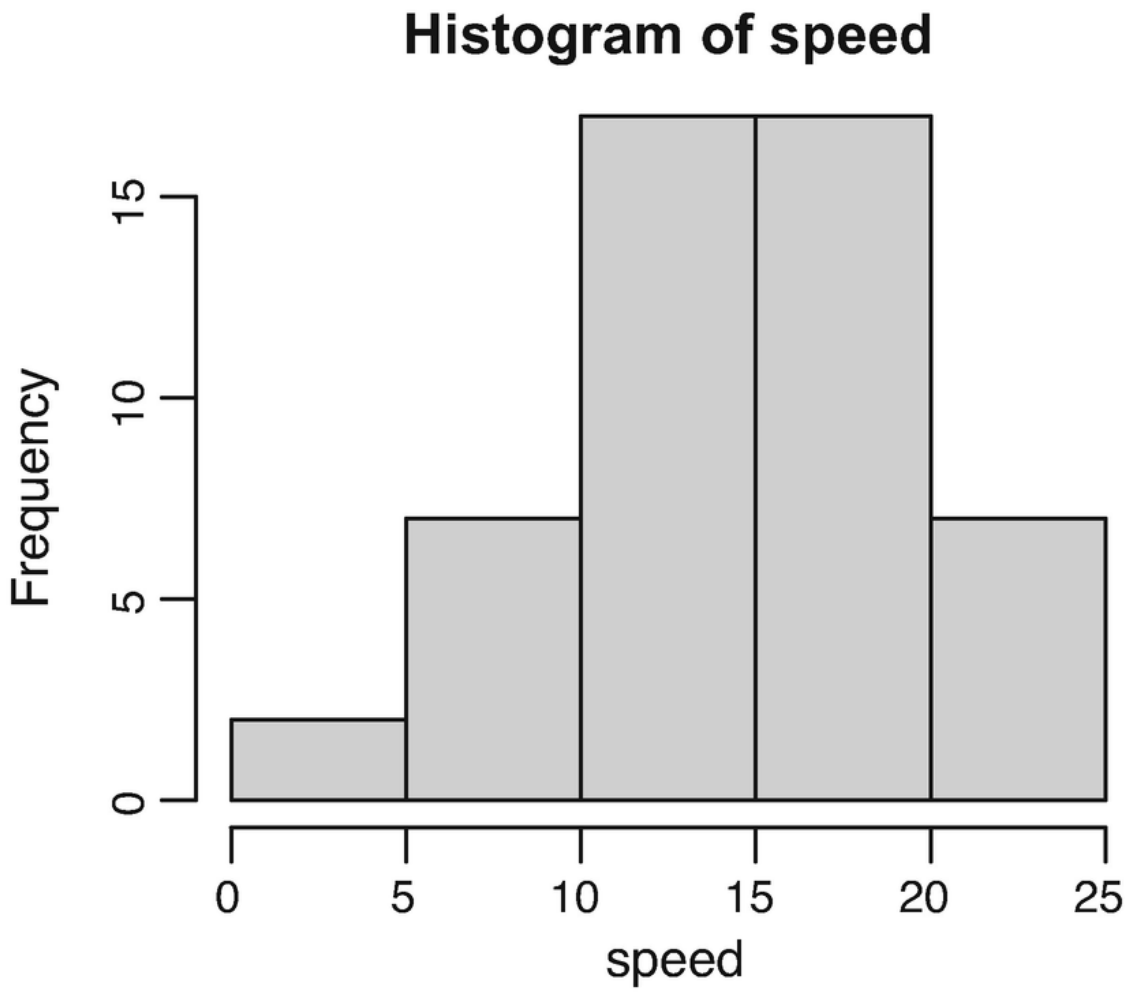

Figure 4-4 Histogram for cars speed

It will give you several summary plots for visualizing the quality of the linear fit.

Many model fitting algorithms return a fitted object that has specialized plotting functionality like this, so when you have fitted a model, you can always try to call `plot()` on it and see if you get something useful out of that.

Functions like `plot()` and `hist()` and a few more create new plots, but there is also a large number of functions for annotating a plot. Functions such as `lines()` and `points()` add lines and points, respectively, to the current plot rather than making a new plot.

We can see them in action if we want to plot the `longley` data set and want to see both the unemployment rate and people in the armed forces over the years:

```
data(longley)
```

Check the documentation for `longley` (`?longley`) for a description of the data. The data has various statistics for each year from 1947 to 1962 including the number of people unemployed (variable `Unemployed`) and the number of people in the armed forces (variable `Armed.Forces`). To plot both of these on the same plot, we can first plot `Unemployed` against years (variable `Year`) and then add lines for `Armed.Forces`. See Figure 4-5.

```
longley %>% plot(Unemployed ~ Year, data = ., type = 'l')
longley %>% lines(Armed.Forces ~ Year, data = ., col =
"blue")
```

This almost gets us what we want, but the y-axis is chosen by the `plot()` function to match the range of y values in the call to `plot()`, and the `Armed.Forces` doesn't quite fit into this range. To fit both, we have to set the limits of the y-axis which we do with parameter `ylim` (see Figure 4-6):

```
longley %$% plot(Unemployed ~ Year, type = 'l',
                 ylim = range(c(Unemployed, Armed.Forces)))
longley %>% lines(Armed.Forces ~ Year, data = ., col =
"blue")
```

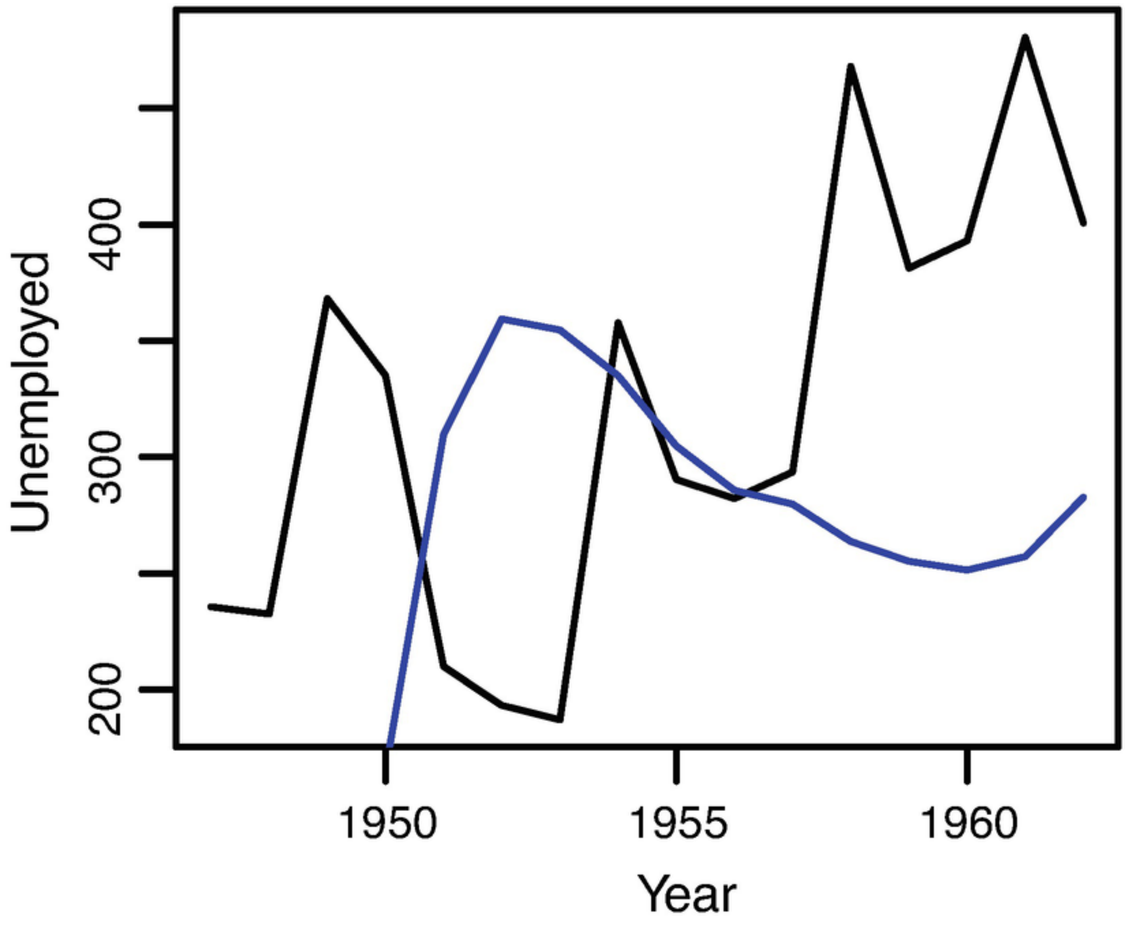

Figure 4-5 Longley data showing Unemployed and Armed.Forces. The y-axis doesn't cover all of the Armed.Forces variable

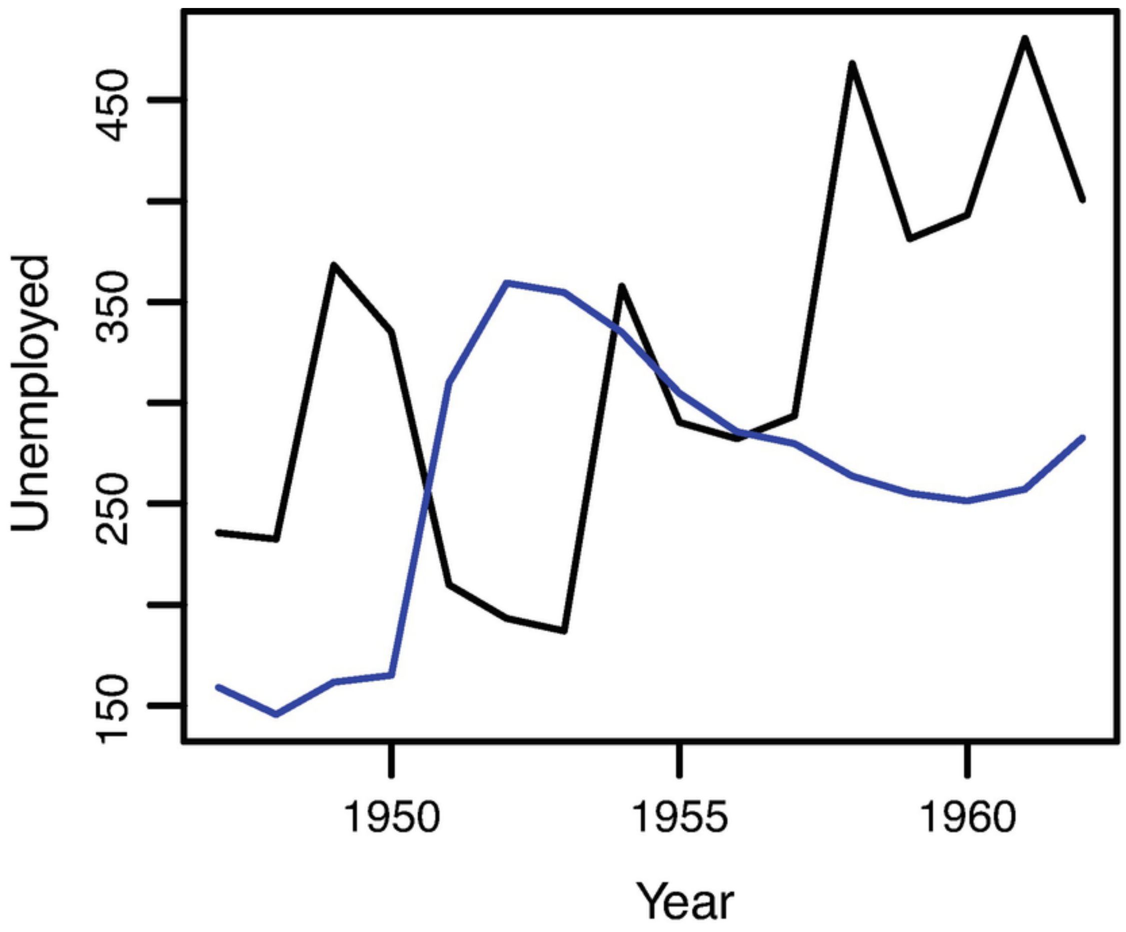

Figure 4-6 Longley data showing Unemployed and Armed.Forces. The y-axis is wide enough to hold all the data

Like `plot()`, the other plotting functions are usually generic. This means we can sometimes give them objects such as fitted models. The `abline()` function is one such case. It plots lines of the form $y = a + bx$, but there is a variant of it that takes a linear model as input and plots the best fitting line defined by the model. So we can plot the `cars` data together with the best-fitted line using the combination of the `lm()` and `abline()` functions (see Figure 4-7):

```
cars %>% plot(dist ~ speed, data = .)
cars %>% lm(dist ~ speed, data = .) %>% abline(col = "red")
```

Plotting using the basic graphics usually follows this pattern. First, there is a call to `plot()` that sets up the canvas to plot on—possibly adjusting the axes to make sure that later points will fit in on it. Then any additional data points are plotted—like the second time series we saw in the `longley` data. Finally, there might be some annotation like adding text labels or margin notes (see functions `text()` and `mtext()` for this).

If you want to select the shape of points or their color according to other data features, for example, plotting the `iris` data with data points in different colors according to the `Species` variable, then you need to map features to columns (see Figure **4-8**):

```
color_map <- c("setosa" = "black",
               "versicolor" = "grey40",
               "virginica" = "grey75")
iris %$% plot(Petal.Length ~ Petal.Width,
              col = color_map[Species])
```

The basic graphics system has many functions for making publication-quality plots, but most of them work at a relatively low level. You have to map variables to colors or shapes explicitly if you want a variable to determine how points should be displayed. You have to set the `xlim` and `ylim` parameters to have the right *x*- and *y*-axes if the first points you plot do not cover the entire range of the data you want to plot. If you change an axis—say log-transform—or if you flip the *x*- and *y*-axes, then you will usually need to update several function calls. If you want to have different subplots—so-called facets—for different subsets of your data, then you have to subset and plot this explicitly.

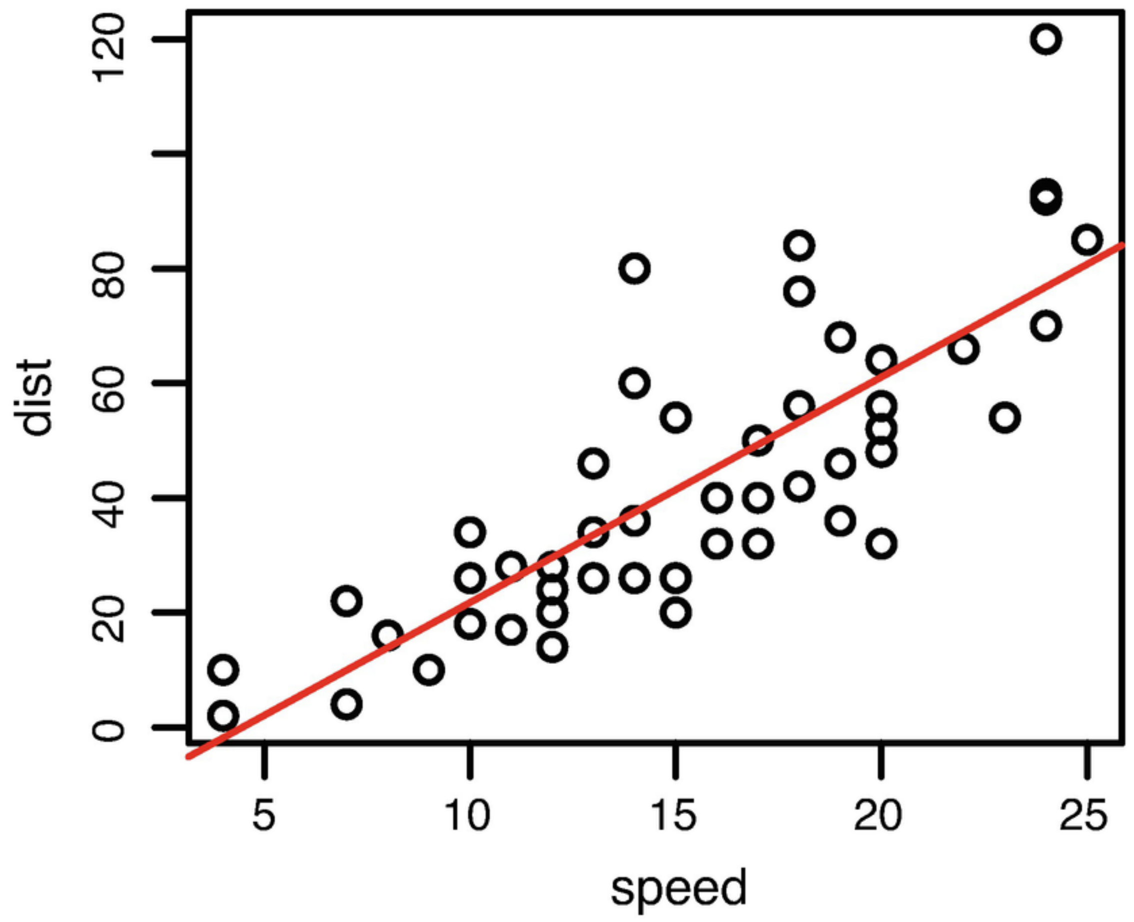

Figure 4-7 The cars data points annotated with the best fitting line

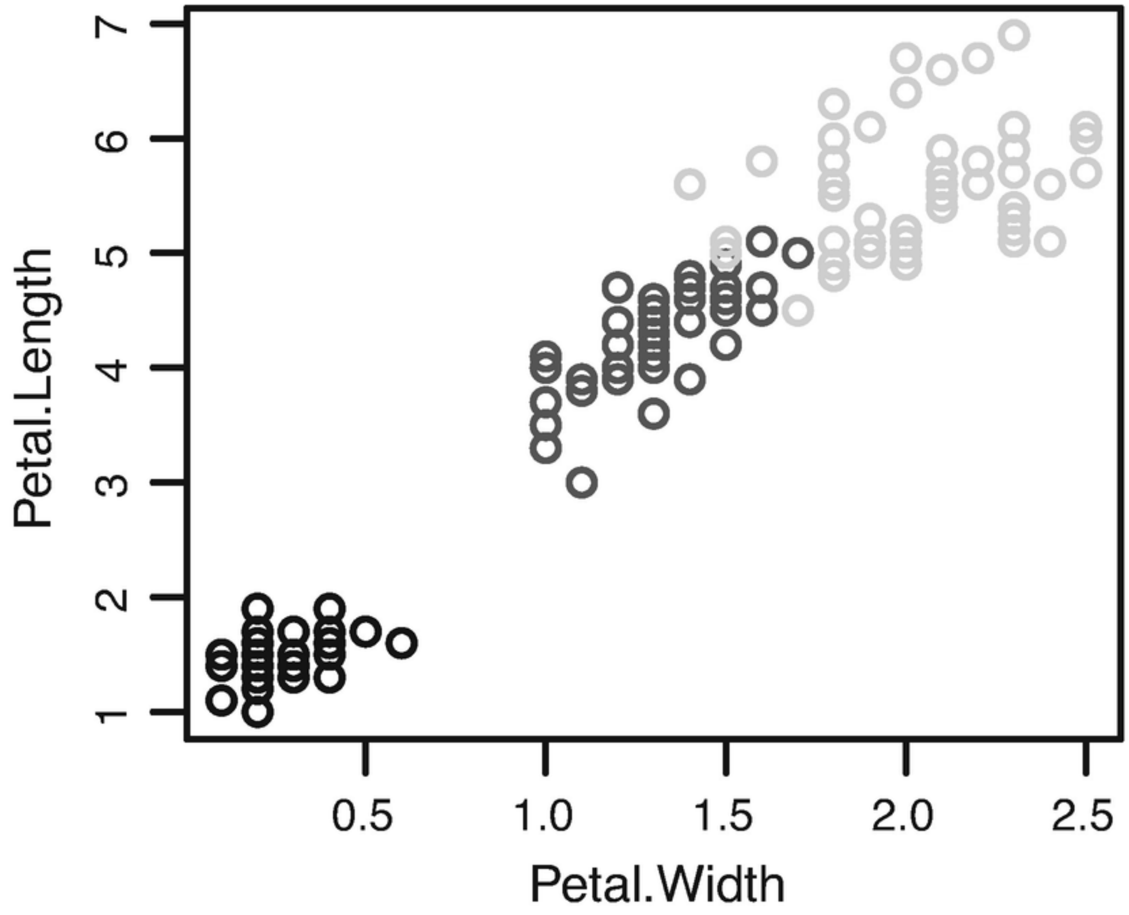

Figure 4-8 Iris data plotted with different colors for different species

So while the basic graphics system is powerful for making good-looking final plots, it is not necessarily optimal for exploring data where you often want to try different ways of visualizing it.

The Grammar of Graphics and the `ggplot2` Package

The `ggplot2` package provides an alternative to the basic graphic that is based on what is called the “grammar of graphics.” The idea here is that the system gives you a small domain-specific language for creating plots (similar to how `dplyr` provides a domain-specific language for manipulating data frames). You construct plots through a list of function calls—similar to how you would work with basic graphics—but these function calls do not directly write on a canvas independently of each other. Rather, they all manipulate a plot by either modifying it—scaling axes or splitting data into subsets that are plotted on different facets—or adding layers of visualization to the plot.

To use it, you, of course, need to import the library:

```
library(ggplot2)
```

and you can get a list of functions it defines using

```
library(help = "ggplot2")
```

I can only give a very brief tutorial-like introduction to the package here. There are full books written about `ggplot2` if you want to learn more details. After reading this chapter, you should be able to construct basic plots, and you should be able to find information about how to make more intricate plots by searching online.

We ease into `ggplot2` by first introducing the `qplot()` function (it stands for quick plot). This function works similar to `plot()`—although it handles things a little differently—but creates the same kind of objects that the other `ggplot2` functions operate on, and so it can be combined with those.

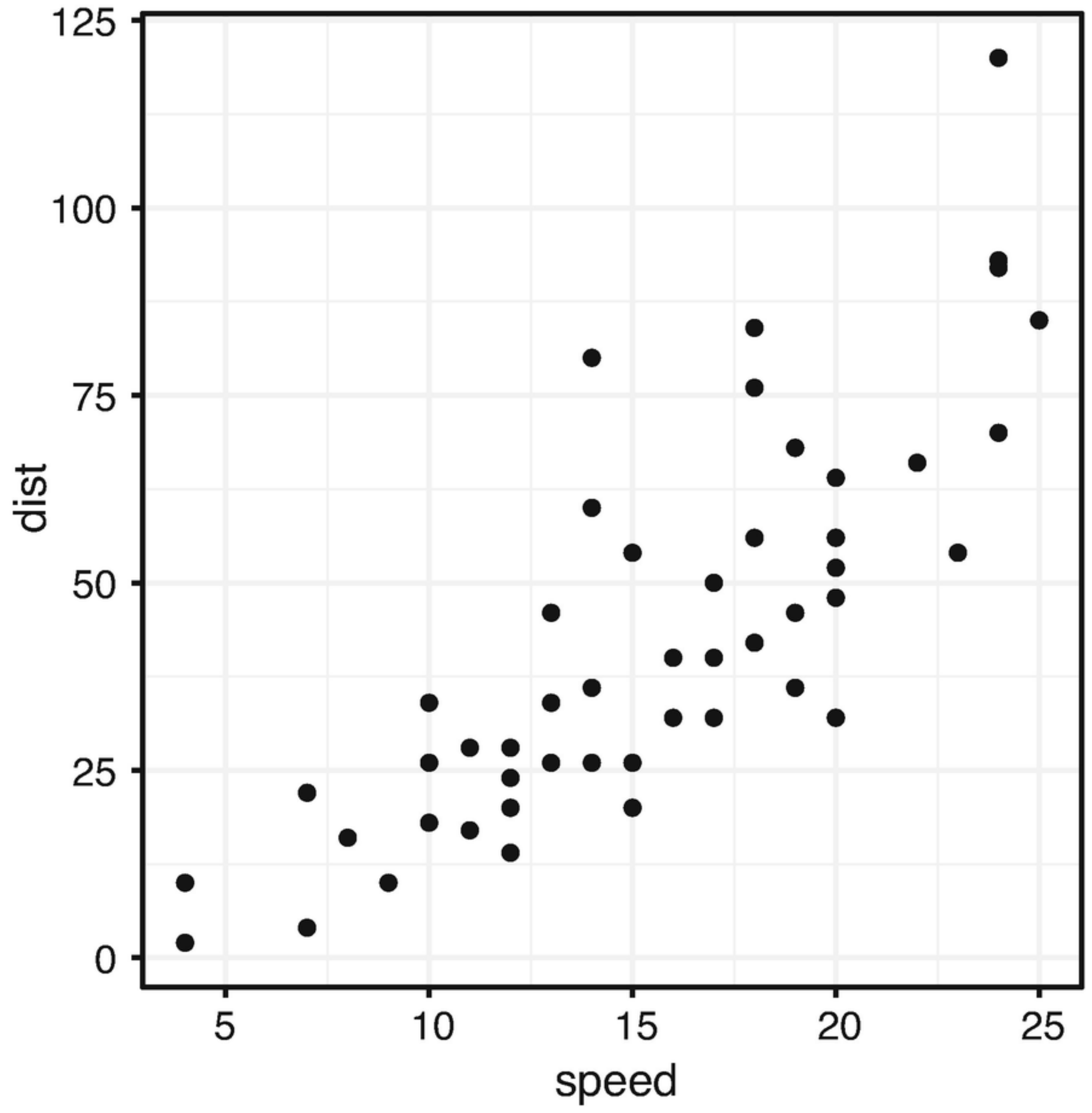

Figure 4-9 Plot of the cars data using qplot (ggplot2)

Using `qplot()`

The `qplot()` function can be used to plot simple scatter plots the same way as the `plot()` function. To plot the `cars` data (see Figure 4-9), we can use

```
cars %>% qplot(speed, dist, data = .)
```

What happens is slightly different, though. The `qplot()` function creates a `ggplot` object rather than directly plotting. It is just that when such objects are printed, which happened at the end of the statement, the effect of printing is that they are plotted. That sounds a bit confusing, but it is what happens. The function used for printing R objects is a generic function, so the effect of printing an object depends on what the object implements for the `print()` function. For `ggplot` objects, this function plots the object. It works well with the kind of code we write, though, because in the preceding code the result of the entire expression is the return value of `qplot()`. When this is evaluated at the outermost level in the R prompt, the result is printed. So the `ggplot` object is plotted.

The preceding code is equivalent to

```
p <- cars %>% qplot(speed, dist, data = .)
p
```

which is equivalent to

```
p <- cars %>% qplot(speed, dist, data = .)
print(p)
```

The reason that it is the `print()` function rather than the `plot()` function—which would otherwise be more natural—is that the `print()` function is the function that is automatically called when we evaluate an expression at the R prompt. By using `print()`, we don't need to print objects explicitly, we just need the plotting code to be at the outermost level of the program. If you create a plot inside a function, however, it isn't automatically printed, and you do need to do this explicitly.

I mention all these details about objects being created and printed because the typical pattern for using `ggplot2` is to build such a `ggplot` object, do various operations on it to modify it, and then finally plot it by printing it.

When using `qplot()`, some transformations of the plotting object are done before `qplot()` returns the object. The quick in quick plot consists of `qplot()` guessing at what kind of plot you are likely to want and then doing transformations on a plot to get there. To get the full control of the final plot, we skip `qplot()` and do all the transformations explicitly—I personally never use `qplot()` anymore myself—but to get started and getting familiar with `ggplot2`, it is not a bad function to use.

With `qplot()`, we can make the visualization of data points depend on data variables in a more straightforward way than we can with `plot()`. To color the `iris` data according to `Species` in `plot()`, we needed to code up a mapping and then transform the `Species` column to get the colors. With `qplot()`, we just specify that we want the colors to depend on the `Species` variable (see Figure **4-10**):

```
iris %>% qplot(Petal.Width, Petal.Length ,
               color = Species, data = .)
```

We get the legend for free when we are mapping the color like this, but we can modify it by doing operations on the `ggplot` object that `qplot()` returns, should we want to.

You can also use `qplot()` for other types of plots than scatter plots. If you give it a single variable to plot, it will assume that you want a histogram instead of a scatter plot and give you that (see Figure **4-11**):

```
cars %>% qplot(speed, data = ., bins = 10)
```

If you want a density plot instead, you simply ask for it (see Figure **4-12**):

```
cars %>% qplot(speed, data = ., geom = "density")
```

Similarly, you can get lines, box plots, violin plots, etc. by specifying a geometry. Geometries determine how the underlying data should be visualized. They might involve calculating some summary statistics, which they do when we create a histogram or a density plot, or they might just visualize the raw data, as we do with scatter plots. Still, they all describe how data should be visualized. Building a plot with `ggplot2` involves adding geometries to your data, typically more than one geometry. To see how this is done, though, we leave `qplot()` and look at how we can create the plots we made earlier with `qplot()` using geometries instead.

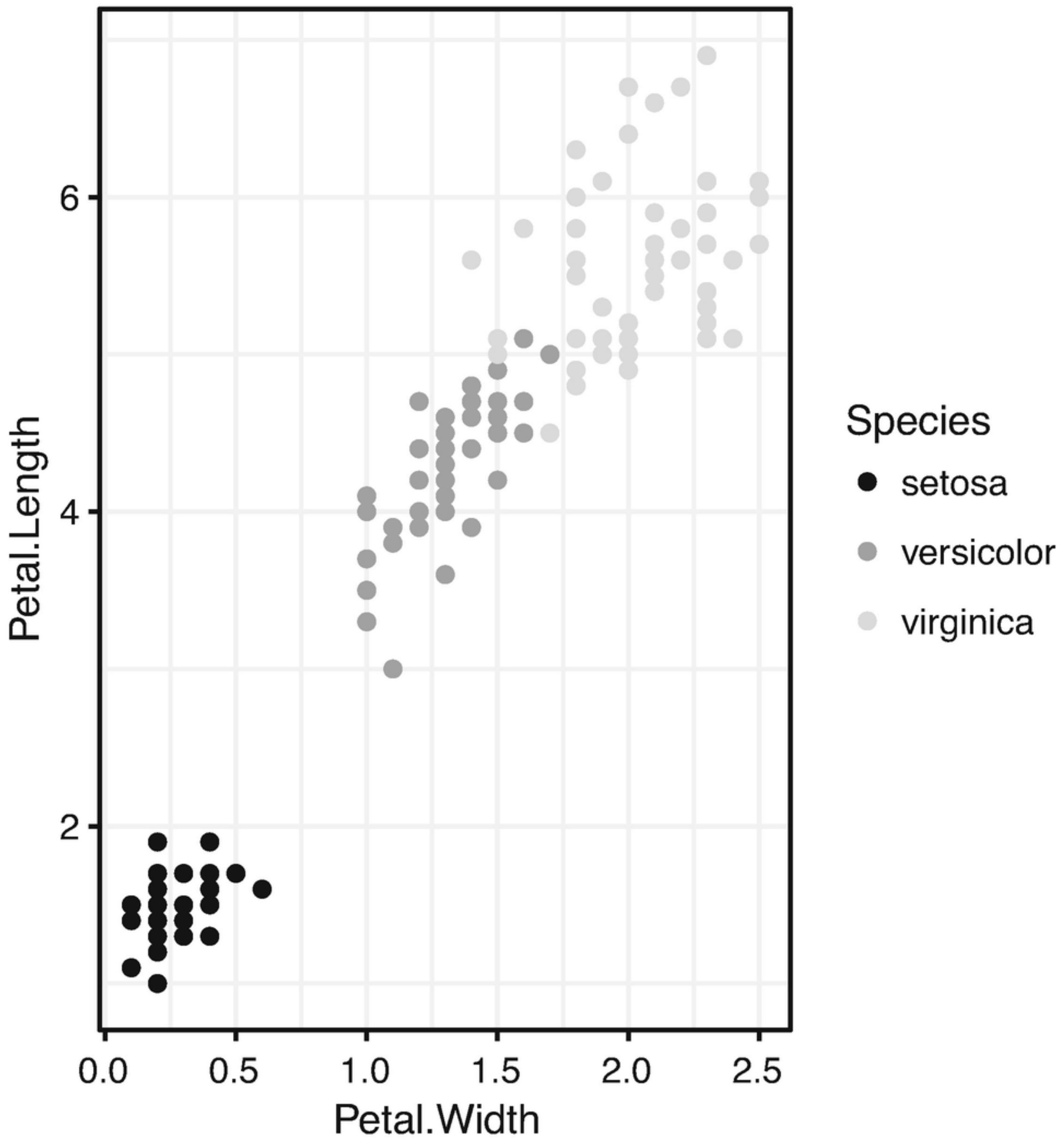

Figure 4-10 Plot of iris data with colors determined by the species. Plotted with qplot (ggplot2)

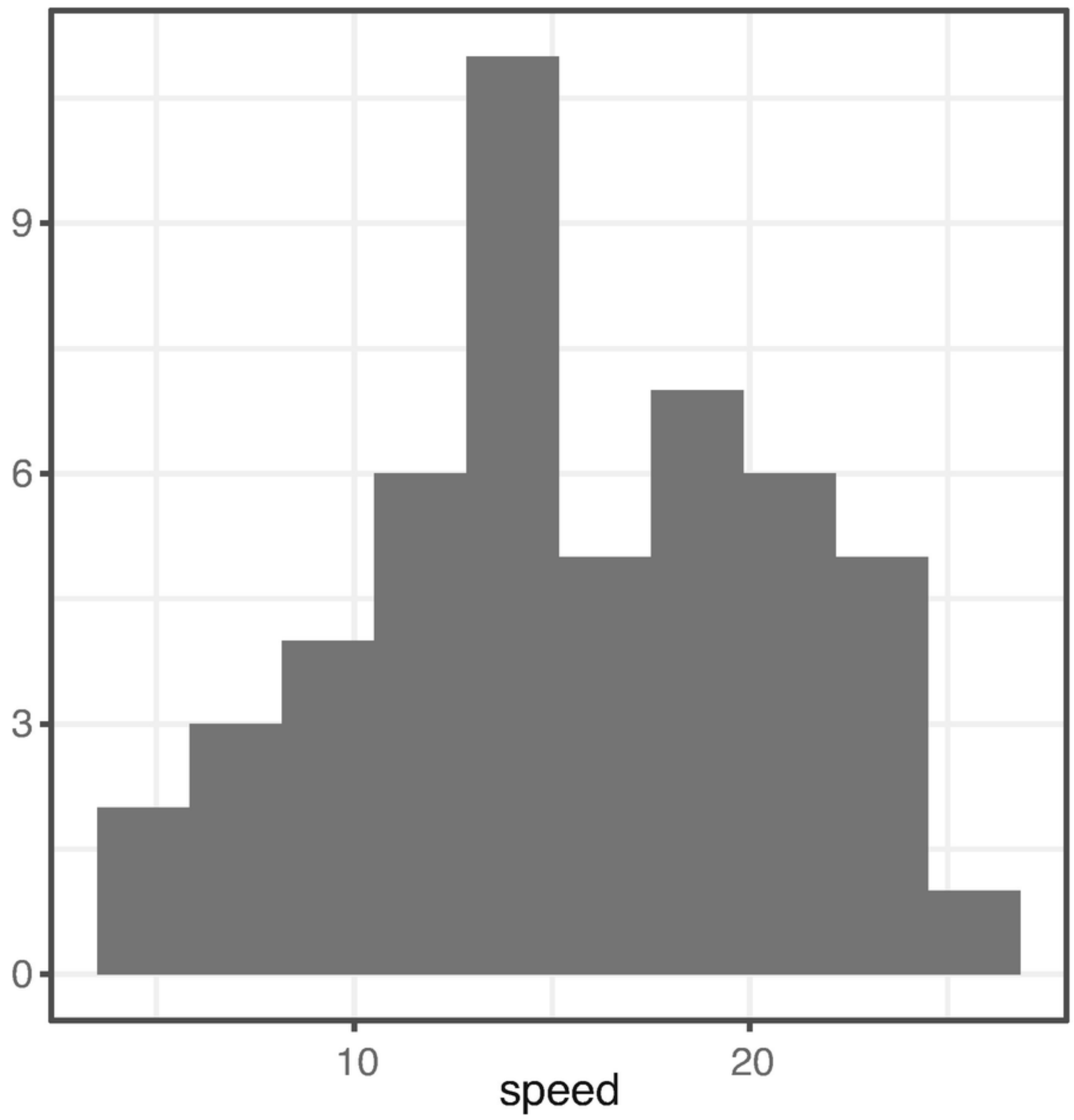

Figure 4-11 Histogram of car speed created using qplot (ggplot2)

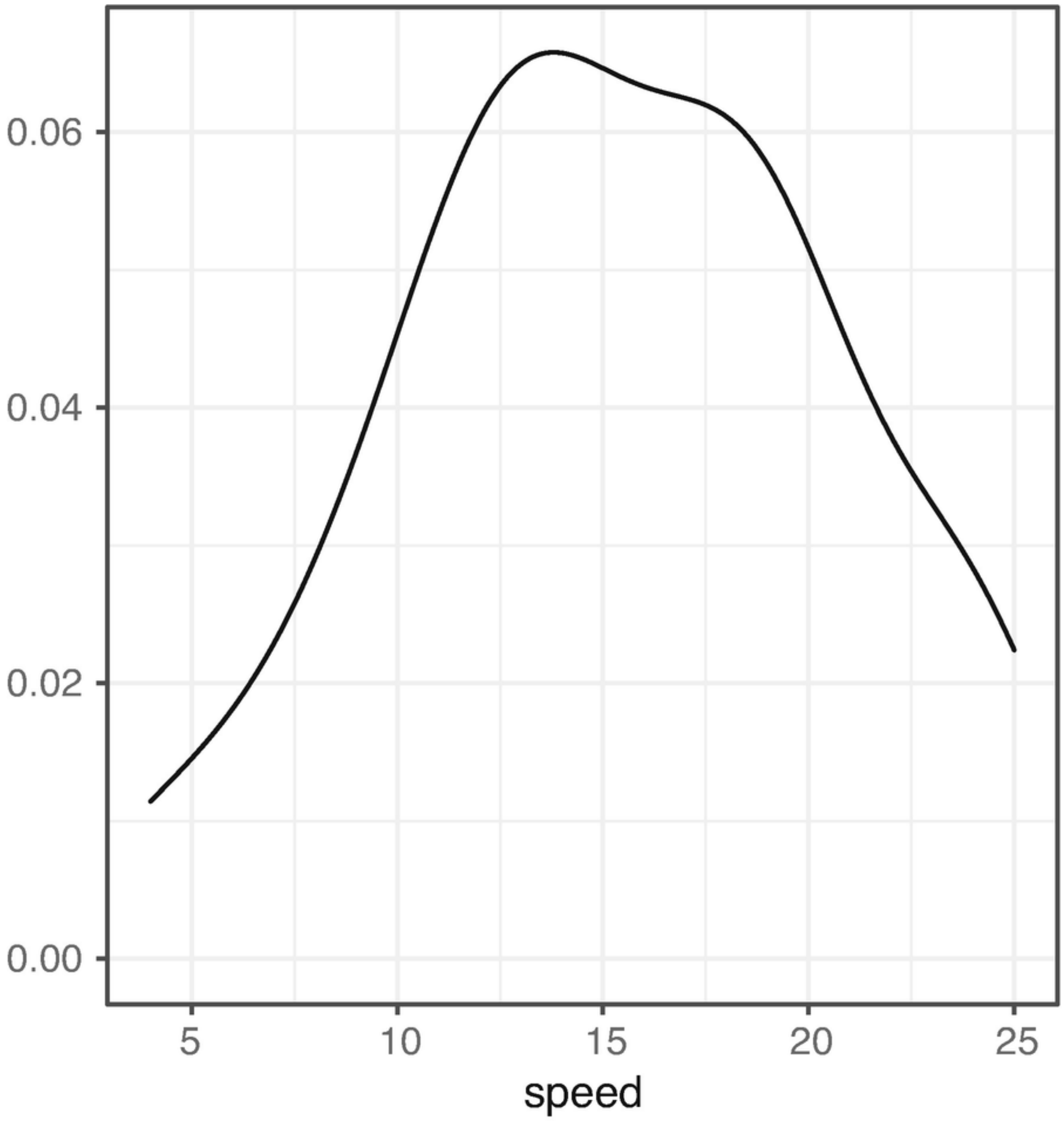

Figure 4-12 Density of car speed created using qplot (ggplot2)

Using Geometries

By stringing together several geometry commands, we can either display the same data in different ways—for example, scatter plots combined with smoothed lines—or put several data sources on the same plot. Before we see more complex constructions, though, we can see how the preceding `qplot()` plots could be made by explicitly calling geometry functions.

We start with the scatter plot for `cars` where we used

```
cars %>% qplot(speed, dist, data = .)
```

To create this plot using explicit geometries, we want a `ggplot` object, we need to map the `speed` parameter from the data frame to the x-axis and the `dist` parameter to the y-axis, and we need to plot the data as points:

```
ggplot(cars) + geom_point(aes(x = speed, y = dist))
```

We create an object using the `ggplot()` function. We give it the `cars` data as input. When we give this object the data frame, the following operations can access the data. It is possible to override which data frame the data we plot comes from, but unless otherwise specified, we have access to the data we gave `ggplot()` when we created the initial object. Next, we do two things in the same function call. We specify that we want x and y values to be plotted as points by calling `geom_point()`, and we map `speed` to the x values and `dist` to the y values using the "aesthetics" function `aes()`. Aesthetics are responsible for mapping from data to graphics. With the `geom_point()` geometry, the plot needs to have x and y values. The aesthetics tell the function which variables in the data should be used for these.

The `aes()` function defines the mapping from data to graphics just for the `geom_point()` function. Sometimes, we want to have different mappings for different geometries, and sometimes we do not. If we want to share aesthetics between functions, we can set it in the `ggplot()` function call instead. Then, like the data, the following functions can access it, and we don't have to specify it for each subsequent function call:

```
ggplot(cars, aes(x = speed, y = dist)) + geom_point()
```

The `ggplot()` and `geom_point()` functions are combined using `+`. You use `+` to string together a series of commands to modify a `ggplot` object in a way very similar to how we use `%>%` to string together a sequence of data manipulations. The only reason that these are two different operators here is historical; if the `%>%` operator had been in common use when `ggplot2` was developed, it would most likely have used that. As it is, you use `+`. Because `+` works slightly different in `ggplot2` than `%>%` does in `magrittr`, you cannot just use a function name when the function doesn't take any arguments, so you need to include the parentheses in `geom_point()`.

Since `ggplot()` takes a data frame as its first argument, a typical pattern is first to modify data in a string of `%>%` or `|>` operations and then give it to `ggplot()` and follow that with a series of + operations. Doing that with `cars` would provide us with this simple pipeline—in larger applications, more steps are included in both the `%>%` pipeline and the + plot composition:

```
cars %>% ggplot(aes(x = speed, y = dist)) + geom_point()
```

For the `iris` data, we used the following `qplot()` call to create a scatter plot with colors determined by the `Species` variable:

```
iris %>% qplot(Petal.Width, Petal.Length ,
               color = Species, data = .)
```

The corresponding code using `ggplot()` and `geom_point()` looks like this:

```
iris %>% ggplot() +
  geom_point(aes(x = Petal.Width, y = Petal.Length,
                 color = Species))
```

Here, we could also have put the aesthetics in the `ggplot()` call instead of the `geom_point()` call.

When you specify the color as an aesthetic, you let it depend on another variable in the data. If you instead want to hardwire a color—or any graphics parameter in general—you simply have to move the parameter assignment outside the `aes()` call. If `geom_point()` gets assigned a color parameter, it will use that color for the points; if it doesn't, it will get the color from the aesthetics (see Figure 4-13):

```
iris |> ggplot() +
  geom_point(aes(x = Petal.Width, y = Petal.Length),
                 color = "grey50")
```

The `qplot()` code for plotting a histogram and a density plot

```
cars %>% qplot(speed, data = ., bins = 10)
cars %>% qplot(speed, data = ., geom = "density")
```

can be constructed using `geom_histogram()` and `geom_density()`, respectively:

```
cars |> ggplot() + geom_histogram(aes(x = speed), bins = 10)
cars |> ggplot() + geom_density(aes(x = speed))
```

You can combine more geometries to display the data in more than one way. Doing this isn't always meaningful depending on how data is summarized—combining scatter plots and histograms might not be so useful. However, we can, for example, make a plot showing the car speed both as a histogram and a density (see Figure 4-14):

```
cars |> ggplot(aes(x = speed, y = ..count..)) +
  geom_histogram(bins = 10) +
  geom_density()
```

It just requires us to call both `geom_histogram()` and `geom_density()`. We do also need to add an extra aesthetics option for the y value. The reason is that histograms by default will show the counts of how many observations fall within a bin on the y-axis, while densities integrate to one. By setting `y = ..count..`, you tell both geometries to use counts as the y-axis. To get densities instead, you can use `y = ..density..`.

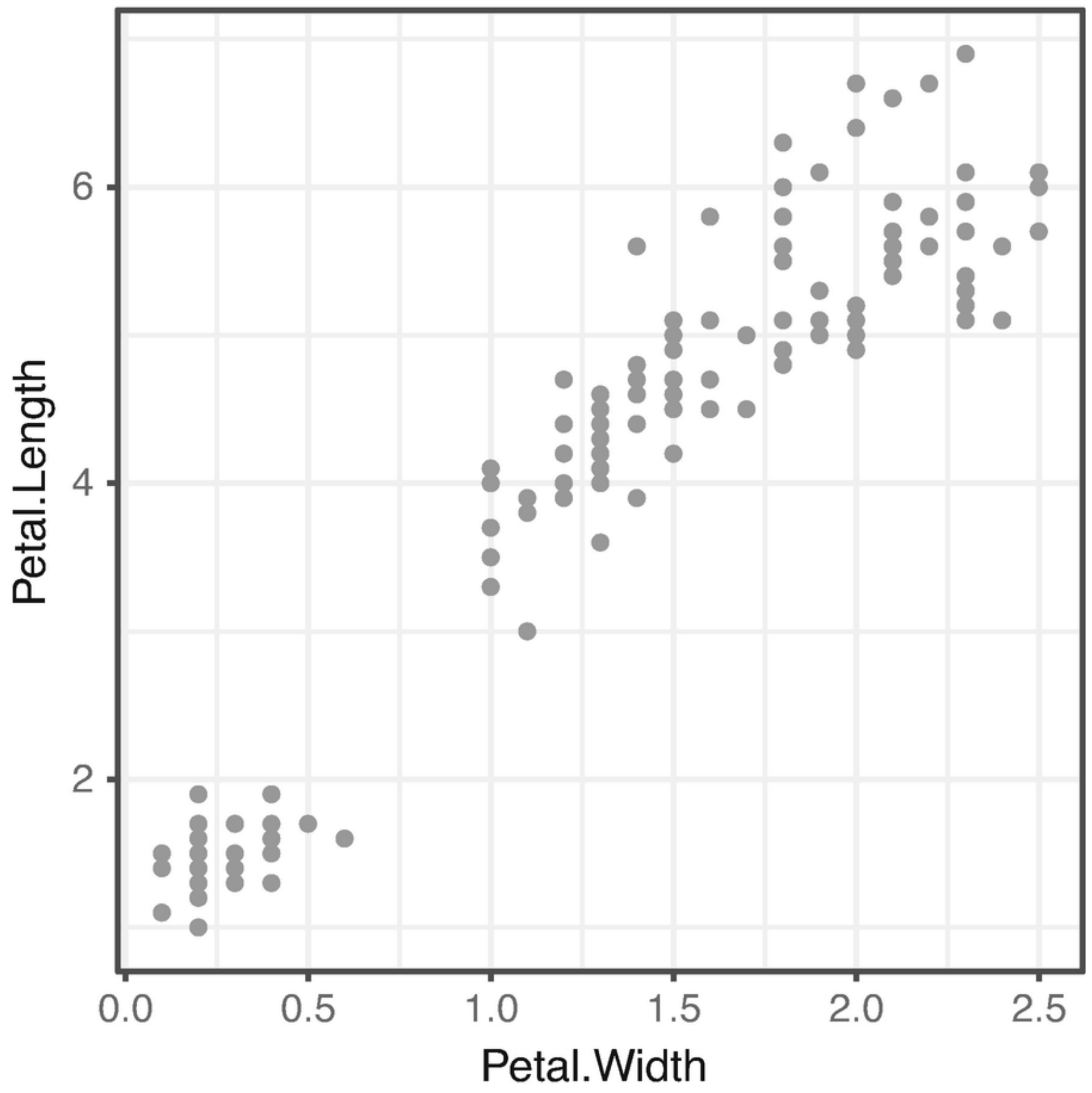

Figure 4-13 Iris data where the color of the points is hardwired

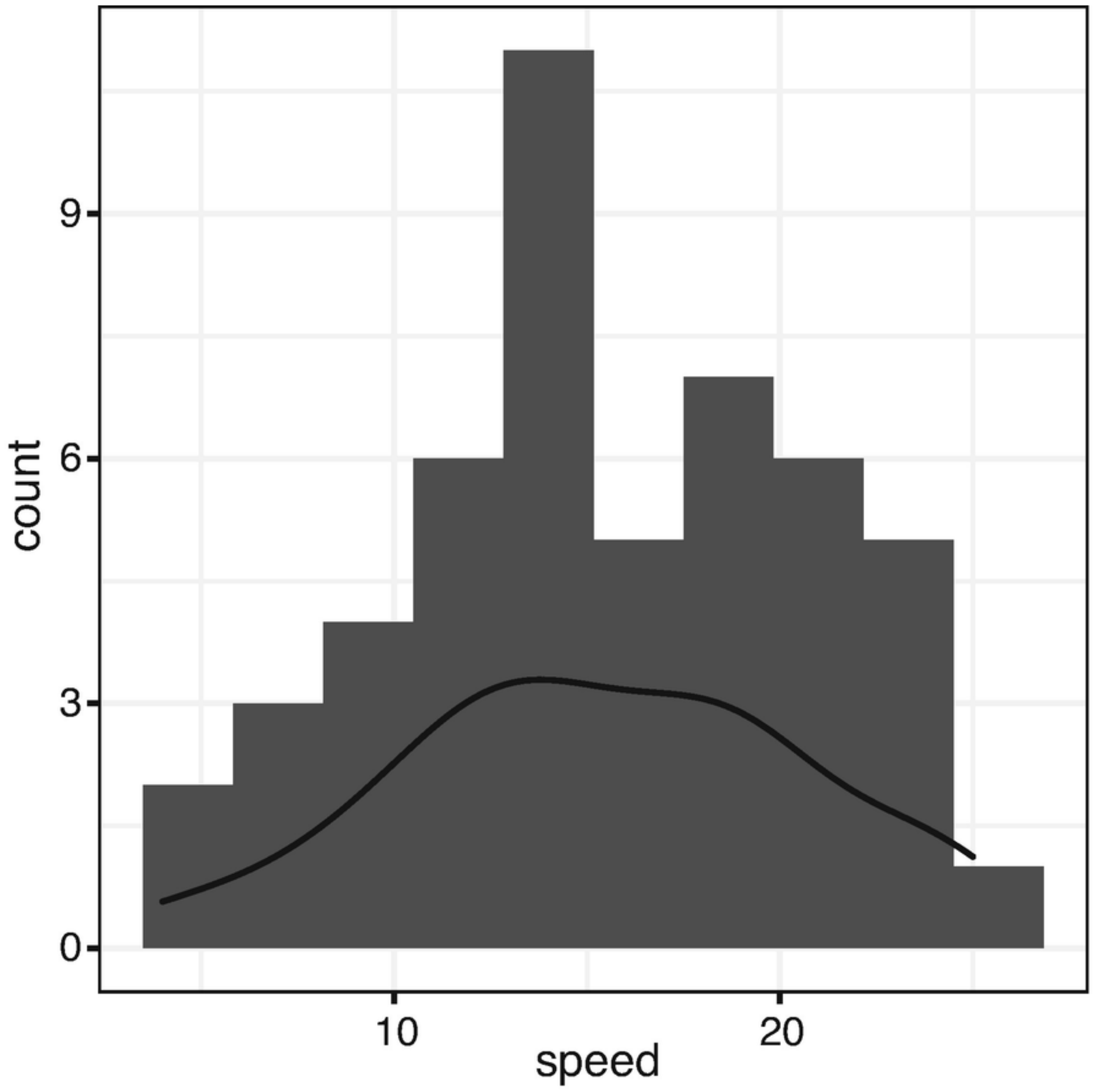

Figure 4-14 Combined histogram and density plot for speed from the cars data

We can also use combinations of geometries to show summary statistics of data together with a scatter plot. We added the result of a linear fit of the data to the scatter plot we did for the `cars` data with `plot()`. To do the same with `ggplot2`, we add a `geom_smooth()` call (see Figure 4-15):

```
cars %>% ggplot(aes(x = speed, y = dist)) +
  geom_point() + geom_smooth(method = "lm")
## `geom_smooth()` using formula 'y ~ x'
```

The message we get from `geom_smooth` is that it used the formula `y ~ x` in the linear model to smooth the data. It will let us know when we use a default instead of explicitly providing a formula for what we want smoothed. Here, it just means that it is finding the best line between the x and y values, which is exactly what we want. You could make the formula explicit by writing `geom_smooth(formula = y ~ x, method = "lm")`, or you could use a different formula, for example, `geom_smooth(formula =`

`y ~ 1, method = "lm")`, to fit the y values to a constant, getting a horizontal line to fit the mean y value (you can try it out). The default is usually what we want.

Earlier, we told the `geom_smooth()` call to use the linear model method. If we didn't do this, it would instead plot a loess smoothing (see Figure 4-16):

```
cars %>% ggplot(aes(x = speed, y = dist)) +
  geom_point() + geom_smooth()
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

We can also use more than one geometry to plot more than one variable. For the `longley` data, we could use two different `geom_line()` to plot the `Unemployed` and the `Armed.Forces` data (see Figure 4-17):

```
longley %>% ggplot(aes(x = Year)) +
  geom_line(aes(y = Unemployed)) +
  geom_line(aes(y = Armed.Forces), color = "blue")
```

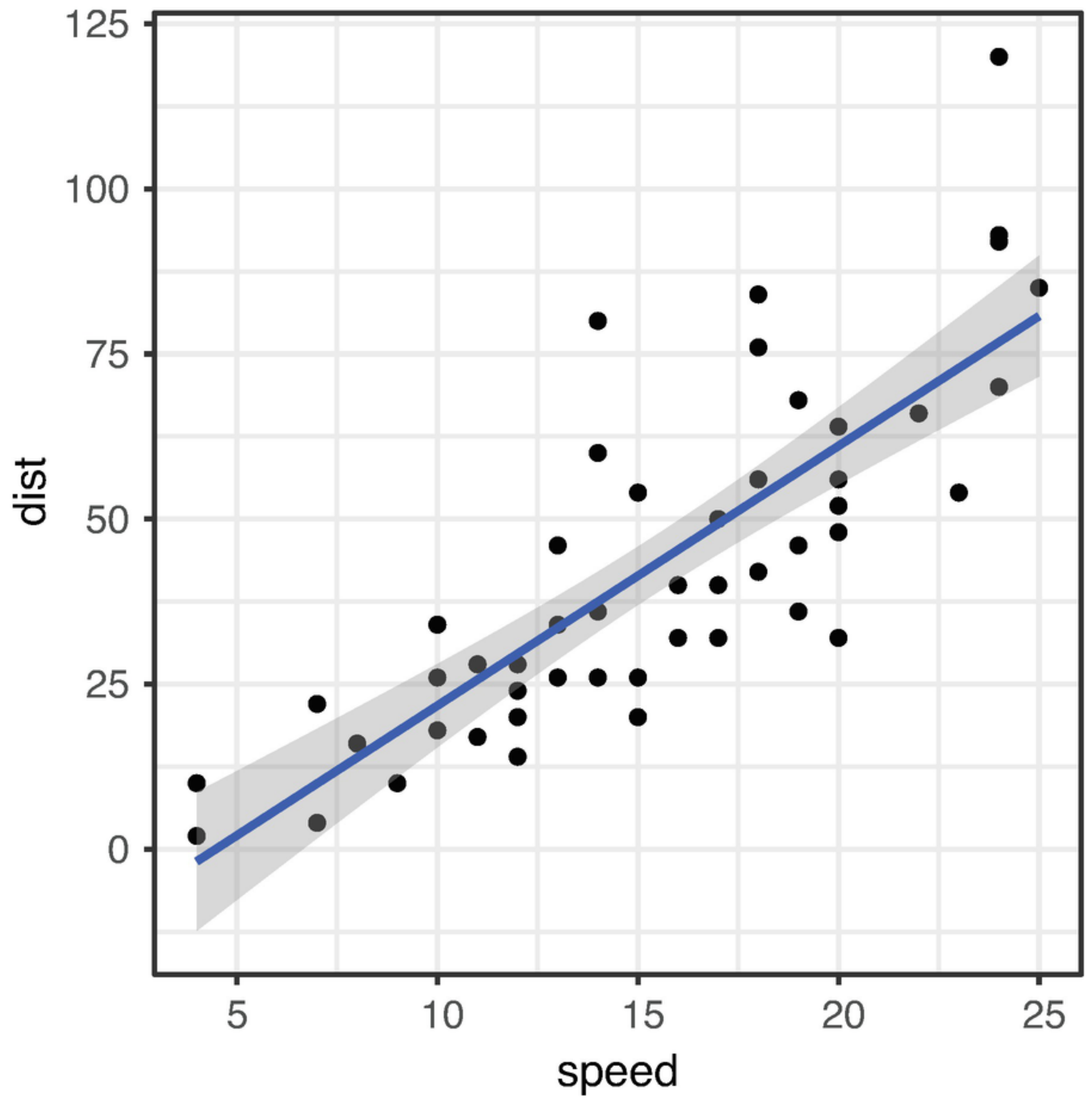

Figure 4-15 Cars data plotted with a linear model smoothing

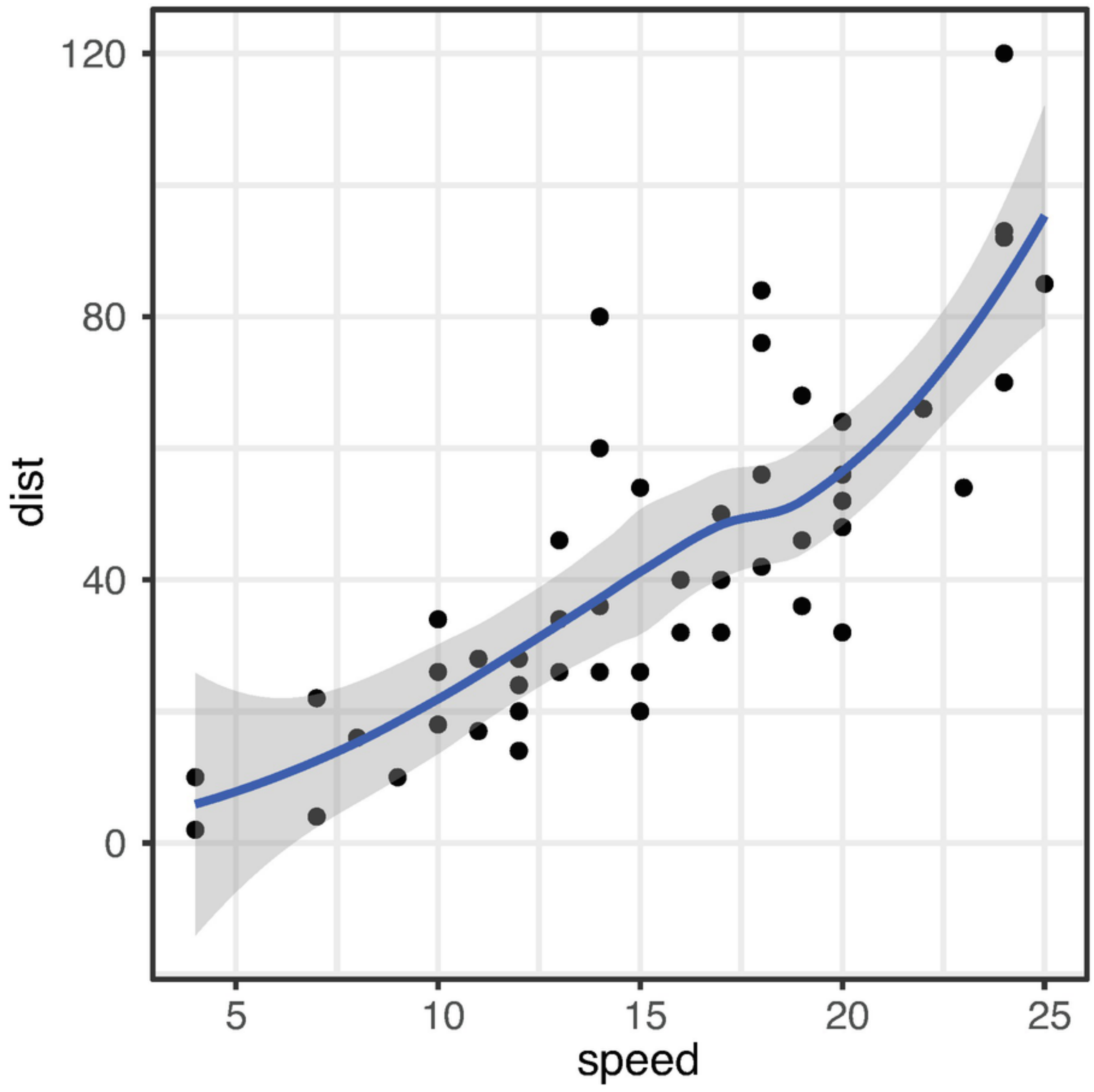

Figure 4-16 Cars data plotted with a loess smoothing

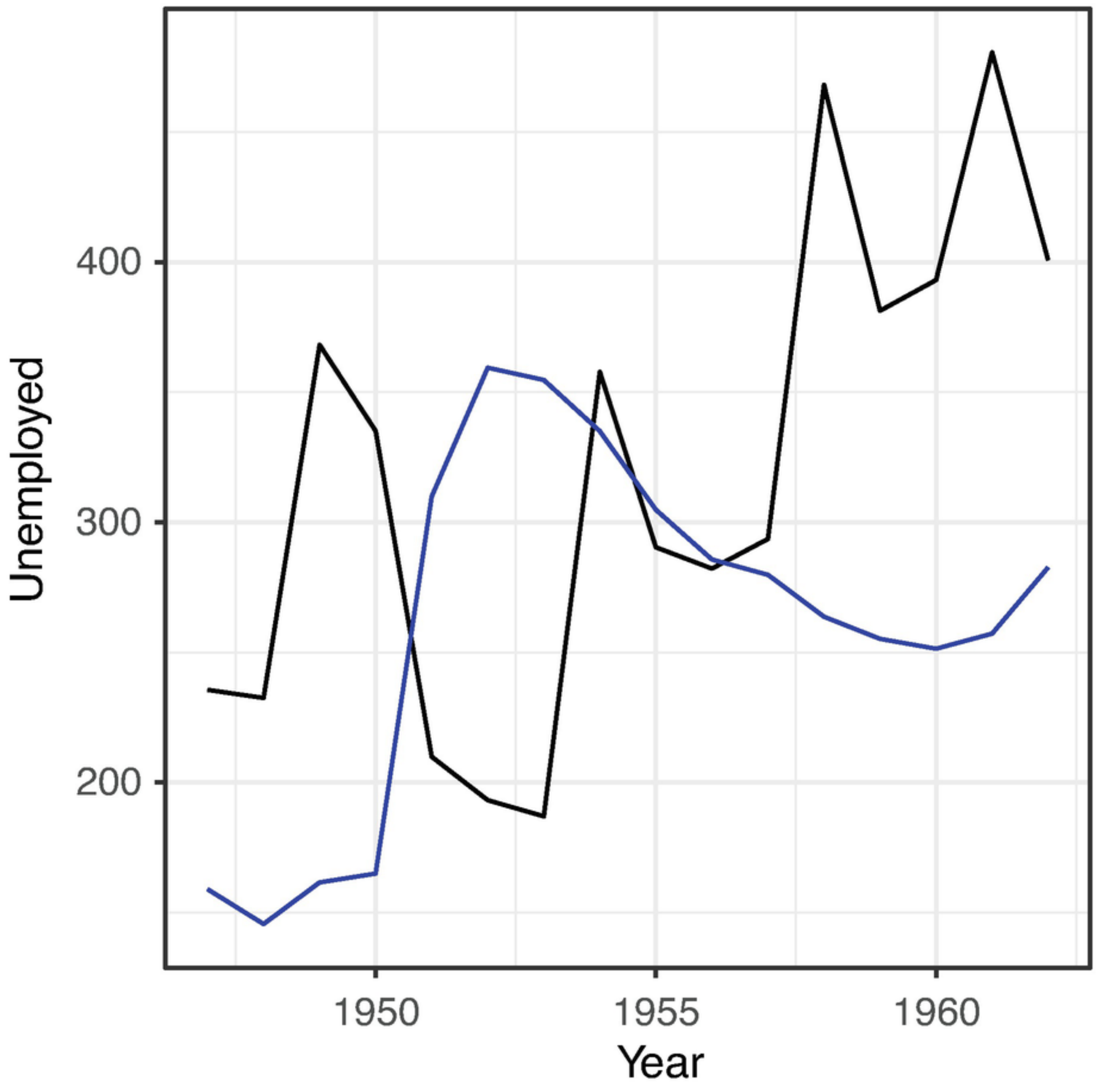

Figure 4-17 Longley data plotted with ggplot2

Here, we set the x value aesthetics in the `ggplot()` function since it is shared by the two `geom_line()` geometries, but we set the y value in the two calls, and we set the color for the `Armed.Forces` data, hardwiring it instead of setting it as an aesthetic. Because we are modifying a plot rather than just drawing on a canvas with the second `geom_line()` call, the y-axis is adjusted to fit both lines. We, therefore, do not need to set the y-axis limit anywhere.

We can also combine `geom_line()` and `geom_point()` to get both lines and points for our data (see Figure 4-18):

```
longley %>% ggplot(aes(x = Year)) +
  geom_point(aes(y = Unemployed)) +
  geom_line(aes(y = Unemployed)) +
  geom_point(aes(y = Armed.Forces), color = "blue") +
  geom_line(aes(y = Armed.Forces), color = "blue")
```

Plotting two variables using different aesthetics like this is fine for most applications, but it is not always the optimal way to do it. The problem is that we are representing that the two measures, `Unemployed` and `Armed.Forces`, are two different measures we have per year and that we can plot together in the plotting code. The data is not reflecting this as something we can compute on. Should we want to split the two measures into subplots instead of plotting them in the same frame, we would need to write new plotting code. A better way is to reformat the data frame such that we have one column telling us whether an observation is `Unemployment` or `Armed.Forces` and another giving us the values and then set the color according to the first column and the y-axis according to the other. We can do this with the `pivot_longer` function from the `tidyr` package (see Figure 4-19):

```
longley %>%
  pivot_longer(
    c(Unemployed, Armed.Forces),
    names_to = "Class",
    values_to = "Number of People"
  ) %>%
  ggplot(aes(x = Year, y = `Number of People`, color =
Class)) +
  geom_line()
```

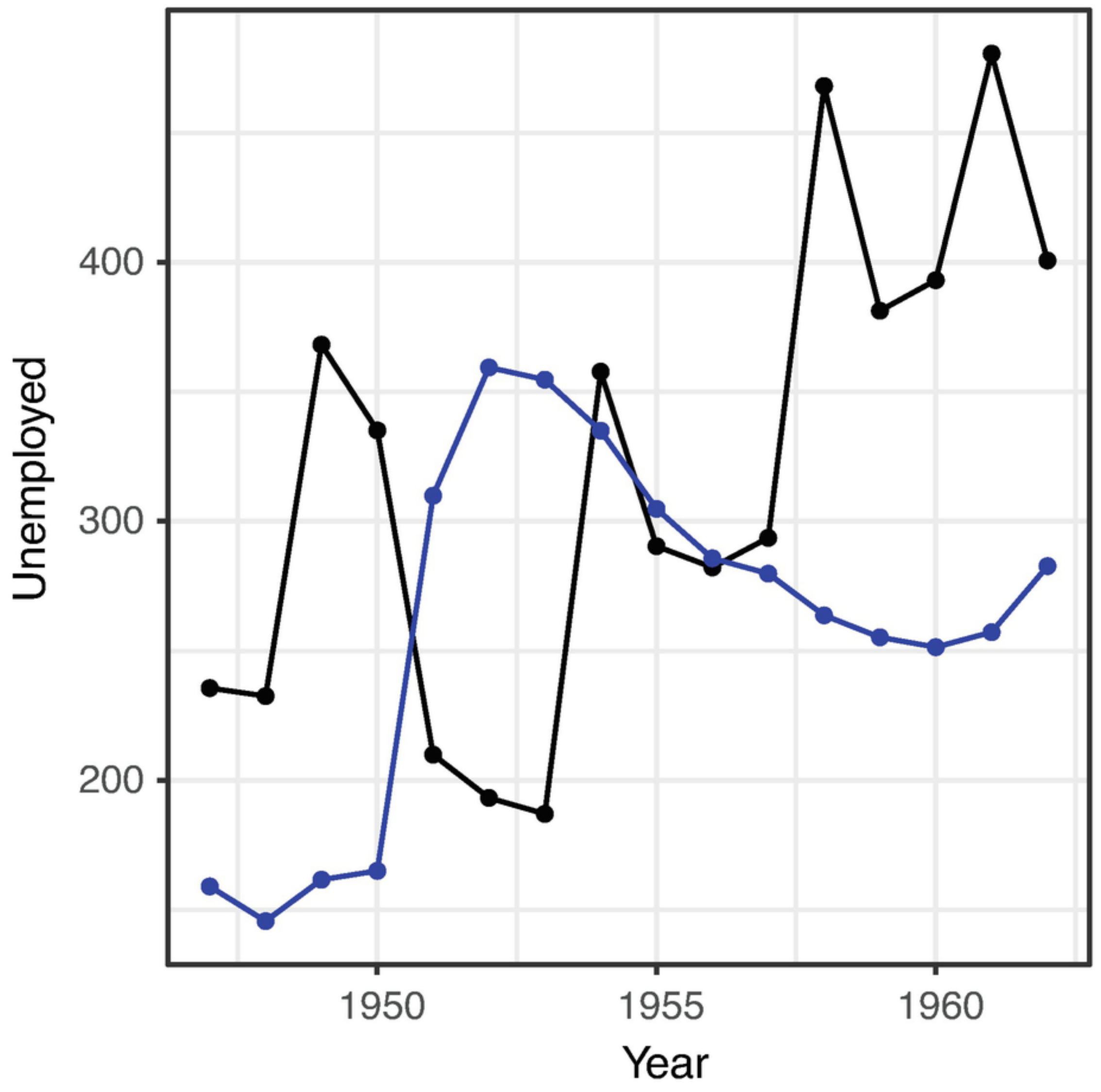

Figure 4-18 Longley data plotted with ggplot2 using both points and lines

In the `pivot_longer` expression, we are saying that we want to transform the `Unemployed` and the `Armed.Forces` columns. These are two different classes from the statistics, so we put the original column names into a new column called `Class`. The two columns count the number of people in the two classes, so the values from the two original classes will go into a new column that we name `Number of People`. The names in the `pivot_longer` expression are strings, and we can put anything there, but the `y` value in the `aes()` expression has to be a valid variable name, and those cannot have spaces, and we need to escape the string. We do that using backticks.

Once we have transformed the data, we can change the plot with little extra code. If, for instance, we want the two values on different facets, we can simply specify this (instead of setting the colors) (see Figure 4-20):

```
longley %>%
  pivot_longer(
    c(Unemployed, Armed.Forces),
    names_to = "Class",
    values_to = "Number of People"
  ) %>%
  ggplot(aes(x = Year, y = `Number of People`)) +
  geom_line() +
  facet_grid(Class ~ .)
```

Facets

Facets are subplots showing different subsets of the data. In the preceding example, we show the `Armed.Forces` variable in one subplot and the `Unemployed` variable in another. You can specify facets using one of two functions: `facet_grid()` creates facets from a formula `rows ~ columns`, and `facet_wrap()` creates facets from a formula `~ variables`. The former creates a row for the variables on the left-hand side of the formula and a column for the variables on the right-hand side and builds facets based on this. In the preceding example, we used “`key ~ .`”, so we get a row per key. Had we used “`. ~ key`” instead, we would get a column per key. The `facet_wrap()` doesn’t explicitly set up rows and columns, it simply makes a facet per combination of variables on the right-hand side of the formula and wraps the facets in a grid to display them.

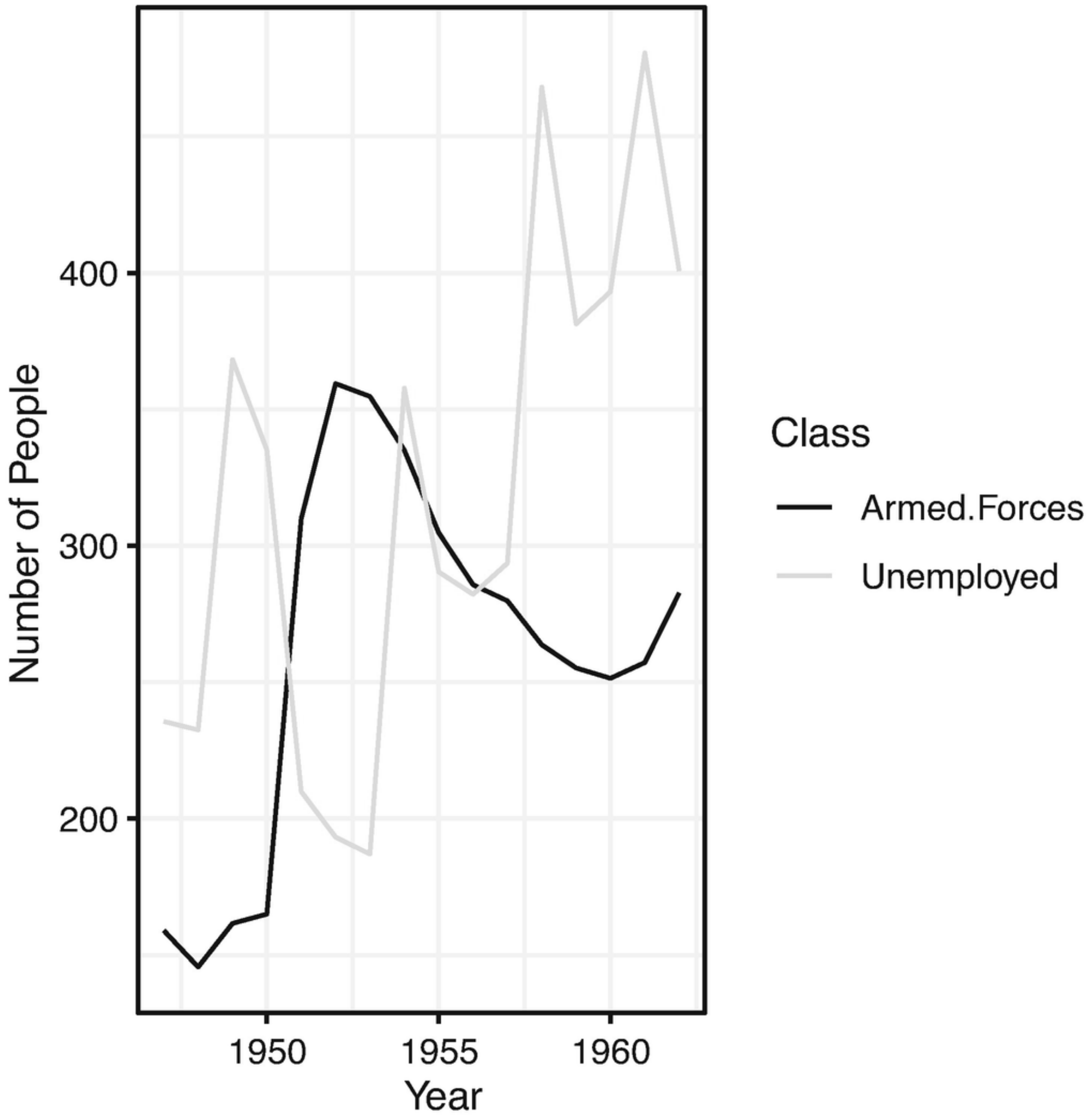

Figure 4-19 Longley data plotted using tidy data

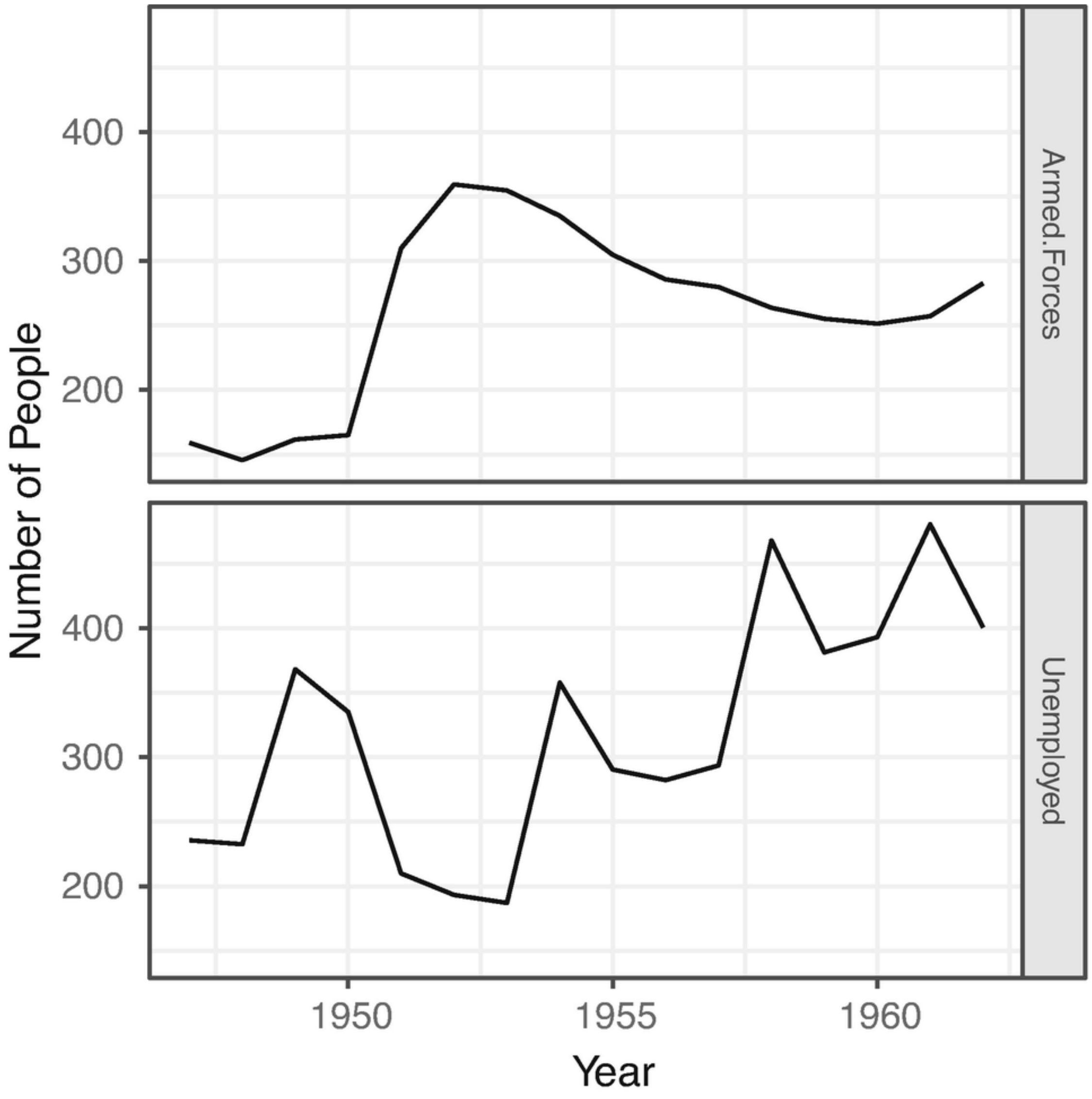

Figure 4-20 Longley data plotted using facets

By default, ggplot2 will try to put values on the same axes when you create facets using facet_grid(). So in the preceding example, the Armed.Forces are shown on the same x- and y-axes as Unemployment even though the y values, as we have seen, are not covering the same range. We can use the scales parameter to change this. Facets within a column will always have the same x-axis, however, and facets within a row will have the same y-axis.

We can see this in action with the iris data. We can transform the iris data, so every column except Species gets squashed into two key-value columns using pivot_longer. We can select everything except selected columns by putting a - in front of their name when we select them. Then we can plot the measurements for each separate species like this:

```
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value)) +
  geom_boxplot() +
  facet_grid(Measurement ~ .)
```

We plot the four measurements for each species in different facets, but they are on slightly different scales, so we will only get a good look at the range of values for the largest range. We can fix this by setting the y-axis free; contrast Figures 4-21 and 4-22.

```
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value)) +
  geom_boxplot() +
  facet_grid(Measurement ~ ., scale = "free_y")
```

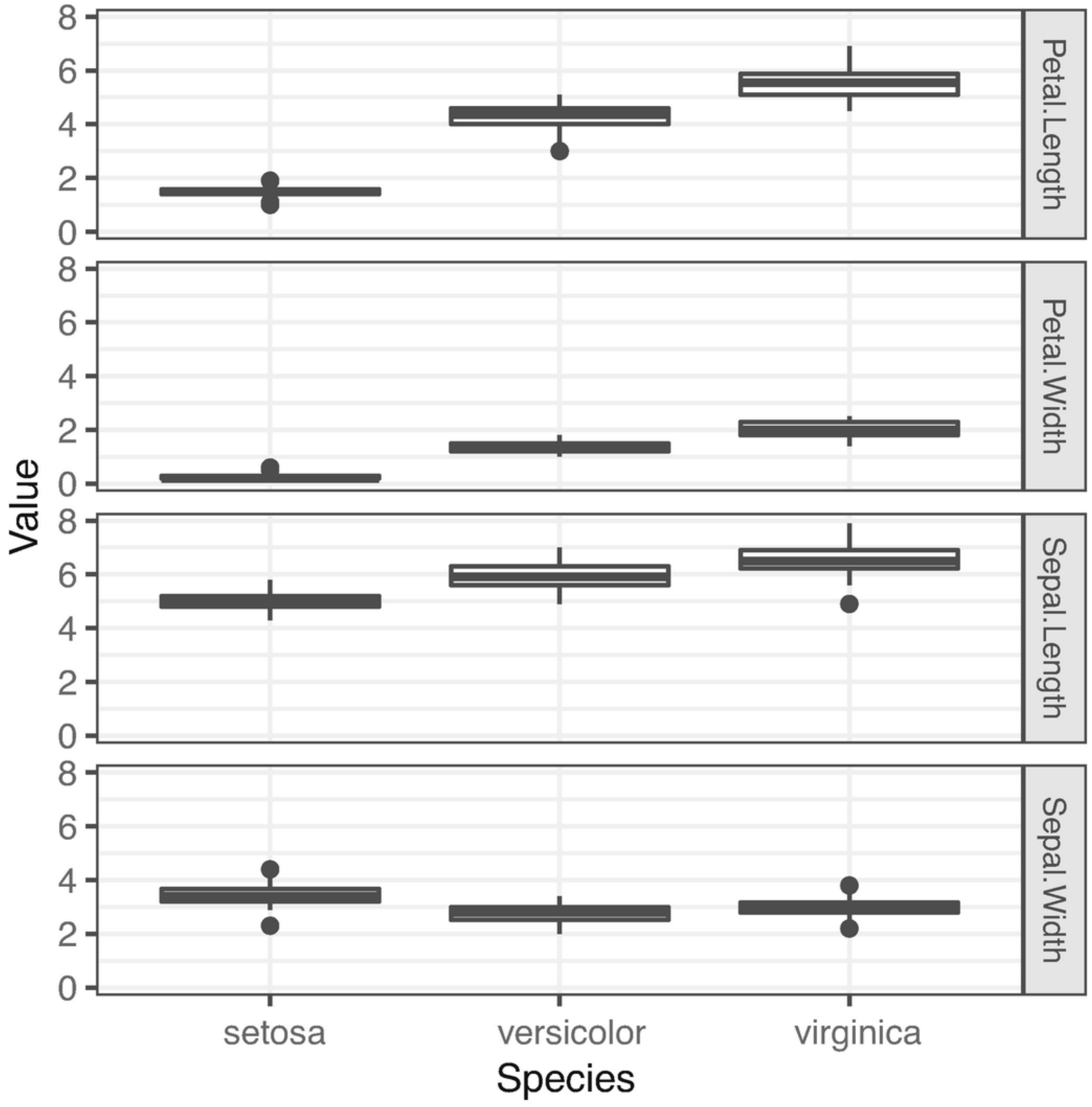

Figure 4-21 Iris measures plotted on the same y-axis

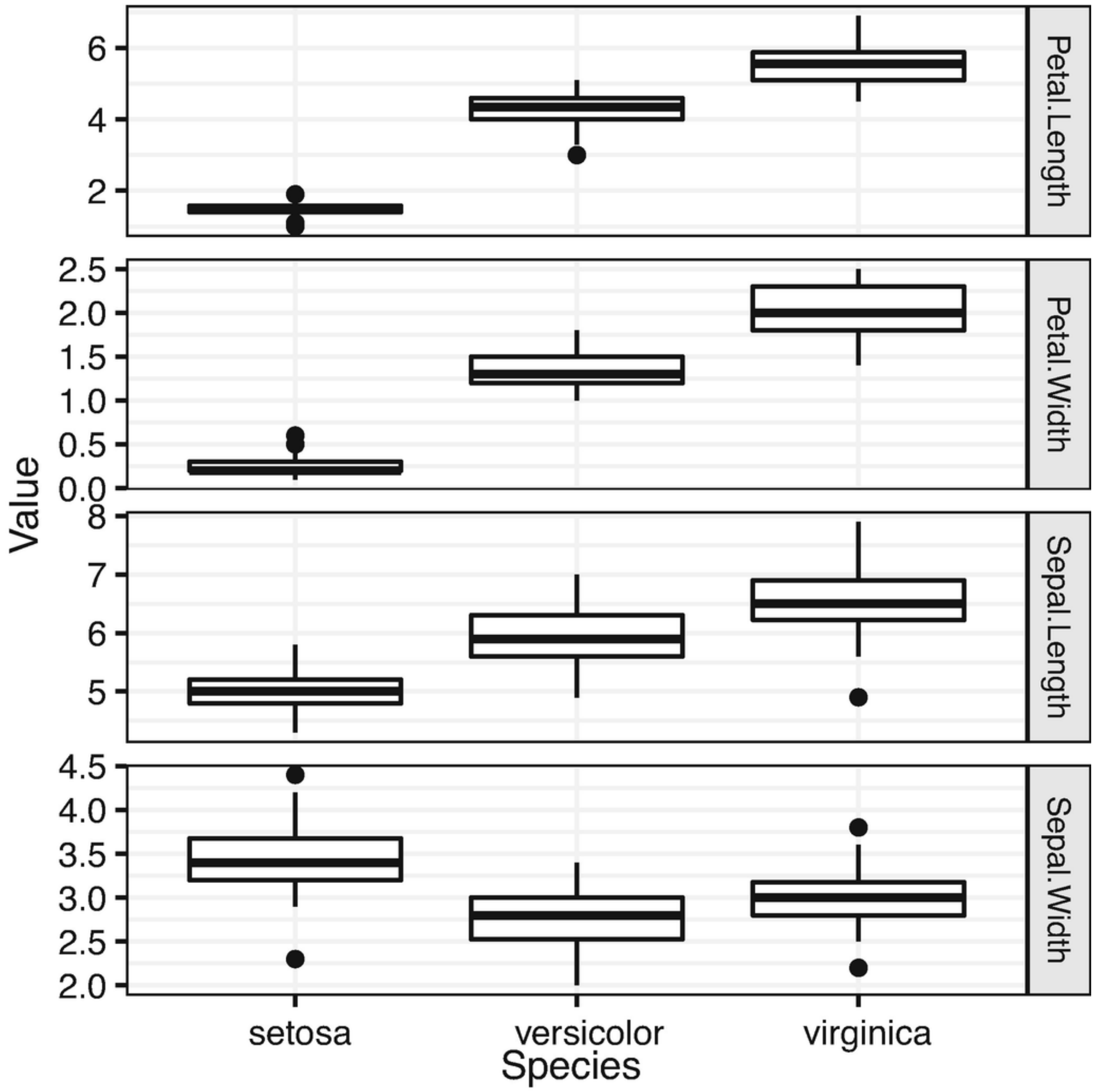

Figure 4-22 Iris measures plotted on different y-axes

By default, all the facets will have the same size. You can modify this using the `space` variable. This is mainly useful for categorical values if one facet has many more of the levels than another.

The labels used for facets are taken from the factors in the variables used to construct the facet. This is a good default, but for print quality plots, you often want to modify the labels a little. You can do this using the `labeller` parameter to `facet_grid()`. This parameter takes a function as an argument that is responsible for constructing labels. The easiest way to construct this function is by using another function, `labeller()`. You can give `labeller()` a named argument specifying a factor to make labels for with lookup tables for mapping levels to labels. For the `iris` data, we can use this to remove the dots in the measurement names (see Figure 4-23):

```
label_map <- c(Petal.Width = "Petal Width",
               Petal.Length = "Petal Length",
               Sepal.Width = "Sepal Width",
               Sepal.Length = "Sepal Length")
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value)) +
  geom_boxplot() +
  facet_grid(Measurement ~ ., scale = "free_y",
             labeller = labeller(Measurement = label_map))
```

Scaling

Geometries specify part of how data should be visualized and scales another. The geometries tell `ggplot2` how you want your data mapped to visual components, like points or densities, and scales tell `ggplot2` how dimensions should be visualized. The simplest scales to think about are the x- and y-axes, where values are mapped to positions on the plot as you are familiar with, but scales also apply to visual properties such as colors.

The simplest use we can make of scales is just to put labels on the axes. We can also do this using the `xlab()` and `ylab()` functions, and if setting labels were all we were interested in, we would, but as an example, we can see this use of scales. To set the labels in the `cars` scatter plot, we can write

```
cars %>%
  ggplot(aes(x = speed, y = dist)) +
  geom_point() + geom_smooth(method = "lm") +
  scale_x_continuous("Speed") +
  scale_y_continuous("Stopping Distance")
```

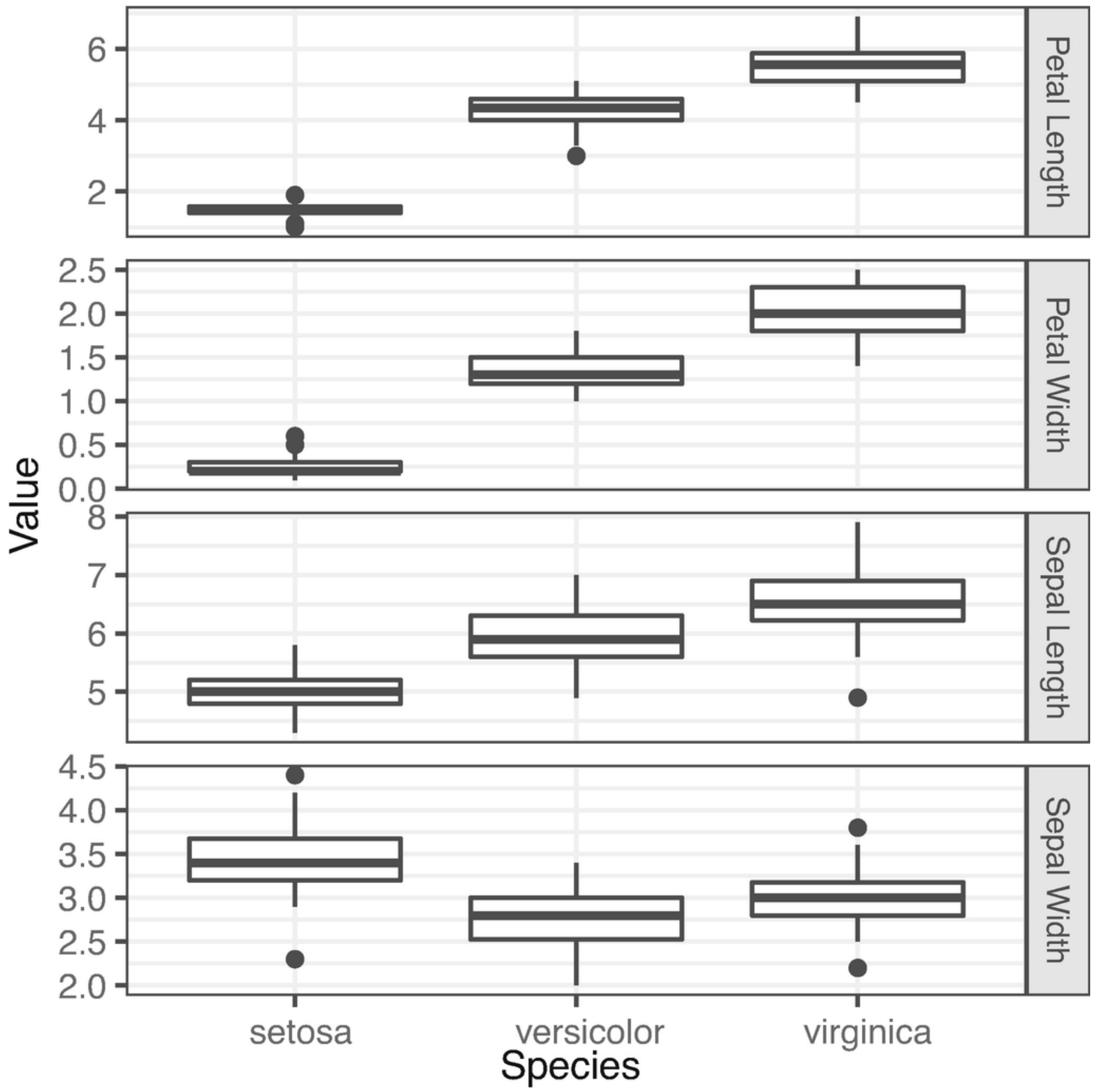

Figure 4-23 Iris measures with measure labels adjusted

Both the x- and y-axes are showing a continuous value, so we scale like that and give the scale a name as the parameter. This will then be the names put on the axis labels. In general, we can use the `scale_x/y_continuous()` functions to control the axis graphics, for instance, to set the breakpoints shown. If we want to plot the `longley` data with a tick mark for every year instead of every five years, we can set the breakpoints to every year:

```
longley %>%
  pivot_longer(
    c(Unemployed, Armed.Forces),
    names_to = "Class",
    values_to = "Number of People"
  ) %>%
  ggplot(aes(x = Year, y = `Number of People`)) +
  geom_line() +
  scale_x_continuous(breaks = 1947:1962) +
  facet_grid(Class ~ .)
```

You can also use the scale to modify the labels shown at tick marks or set limits on the values displayed.

Scales are also the way to transform data shown on an axis. If you want to log-transform the x- or y-axis, you can use the `scale_x/y_log10()` functions, for instance. This usually leads to a nicer plot compared to plotting data you log-transform yourself since the plotting code then knows that you want to show data on a log scale rather than showing transformed data on a linear scale.

To reverse an axis, you use `scale_x/y_reverse()`. This is better than reversing the data mapped in the aesthetic since all the plotting code will just be updated to the reversed axis; you don't need to update x or y values in all the function geometry calls. For instance, to show the speed in the `cars` data in decreasing instead of increasing order, we could write

```
cars %>%
  ggplot(aes(x = speed, y = dist)) +
  geom_point() +
  geom_smooth(method = "lm") +
  scale_x_reverse("Speed") +
  scale_y_continuous("Stopping Distance")
```

Neither axis has to be continuous. If you map a factor to `x` or `y` in the aesthetics, you get a discrete axis; see Figure 4-24 for the `iris` data plotted with the factor `Species` on the x-axis.

```
iris %>%
  ggplot(aes(x = Species, y = Petal.Length)) +
  geom_boxplot() +
  geom_jitter(width = 0.1, height = 0.1)
```

Since `Species` is a factor, the x-axis will be discrete, and we can show the data as a box plot and the individual data points using the jitter geometry. If we want to modify the x-axis, we need to use `scale_x_discrete()` instead of `scale_x_continuous()`.

We can, for instance, use this to modify the labels on the axis to put the species in capital letters:

```
iris %>%
  ggplot(aes(x = Species, y = Petal.Length)) +
  geom_boxplot() +
  geom_jitter(width = 0.1, height = 0.1) +
  scale_x_discrete(labels = c("setosa" = "Setosa",
                              "versicolor" = "Versicolor",
                              "virginica" = "Virginica"))
```

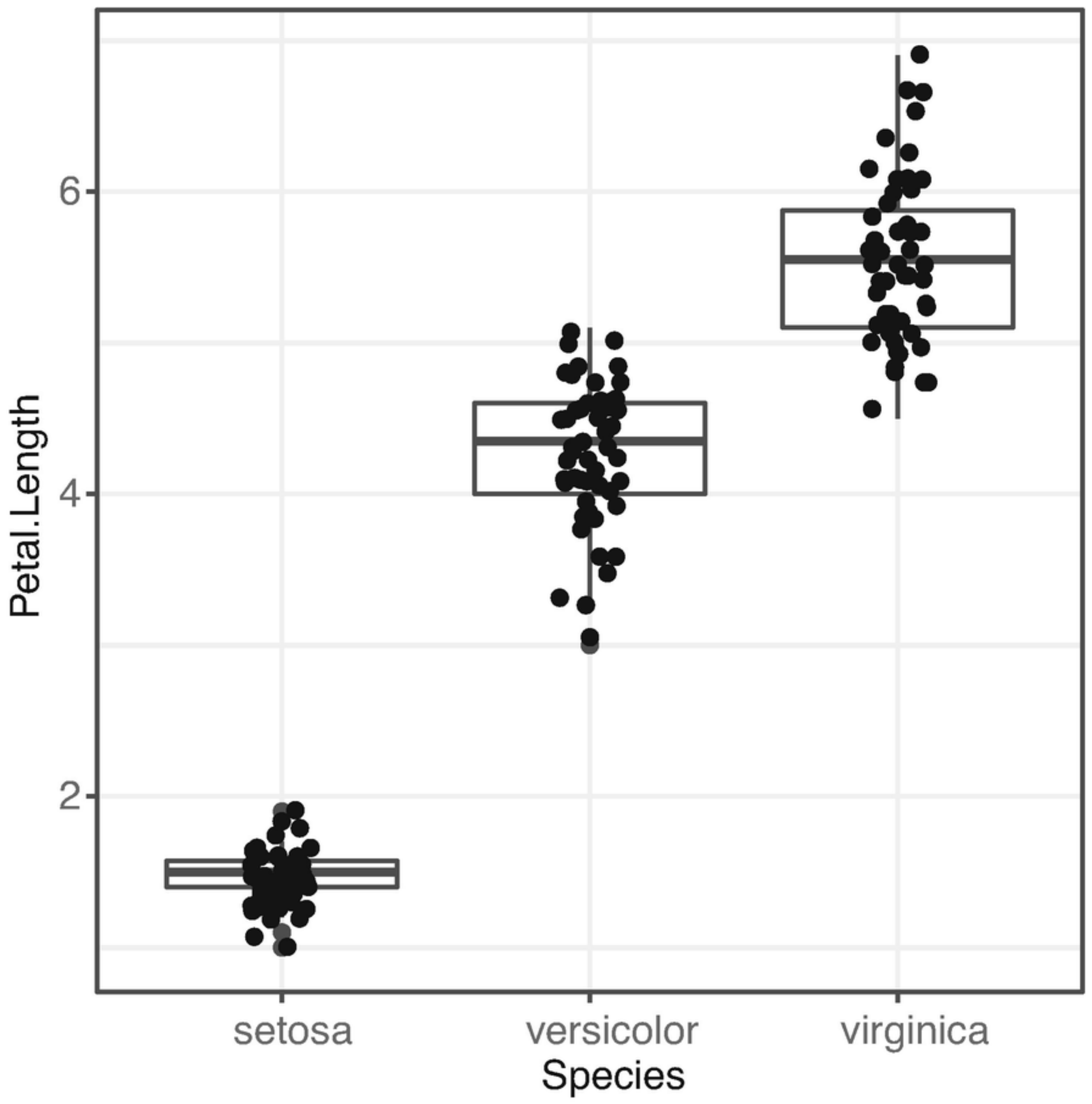

Figure 4-24 Iris data plotted with a factor on the x-axis

We provide a map from the data levels to labels. There is more than one way to set the labels, but this is by far the easiest.

Scales are also used to control colors. You use the various `scale_color_` functions to control the color of lines and points, and you use the `scale_fill_` functions to control the color of filled areas.

We can plot the `iris` measurements per species and give them a different color for each species. Since it is the boxes we want to color, we need to use the `fill` aesthetics. Otherwise, we would color the lines around the boxes. See Figure 4-25.

```
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value, fill = Species)) +
  geom_boxplot() +
  facet_grid(Measurement ~ ., scale = "free_y",
             labeller = labeller(Measurement = label_map))
```

There are different ways to modify color scales. There are two classes, as there are for axes, discrete and continuous. The `Species` variable in `iris` is discrete, so to modify the fill color, we need one of the functions for that. The simplest is just to give a color per species explicitly. We can do that with the `scale_fill_manual()` function (see Figure 4-26):

```
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value, fill = Species)) +
  geom_boxplot() +
  scale_fill_manual(values = c("black", "grey40", "grey60"))
+
  facet_grid(Measurement ~ ., scale = "free_y",
             labeller = labeller(Measurement = label_map))
```

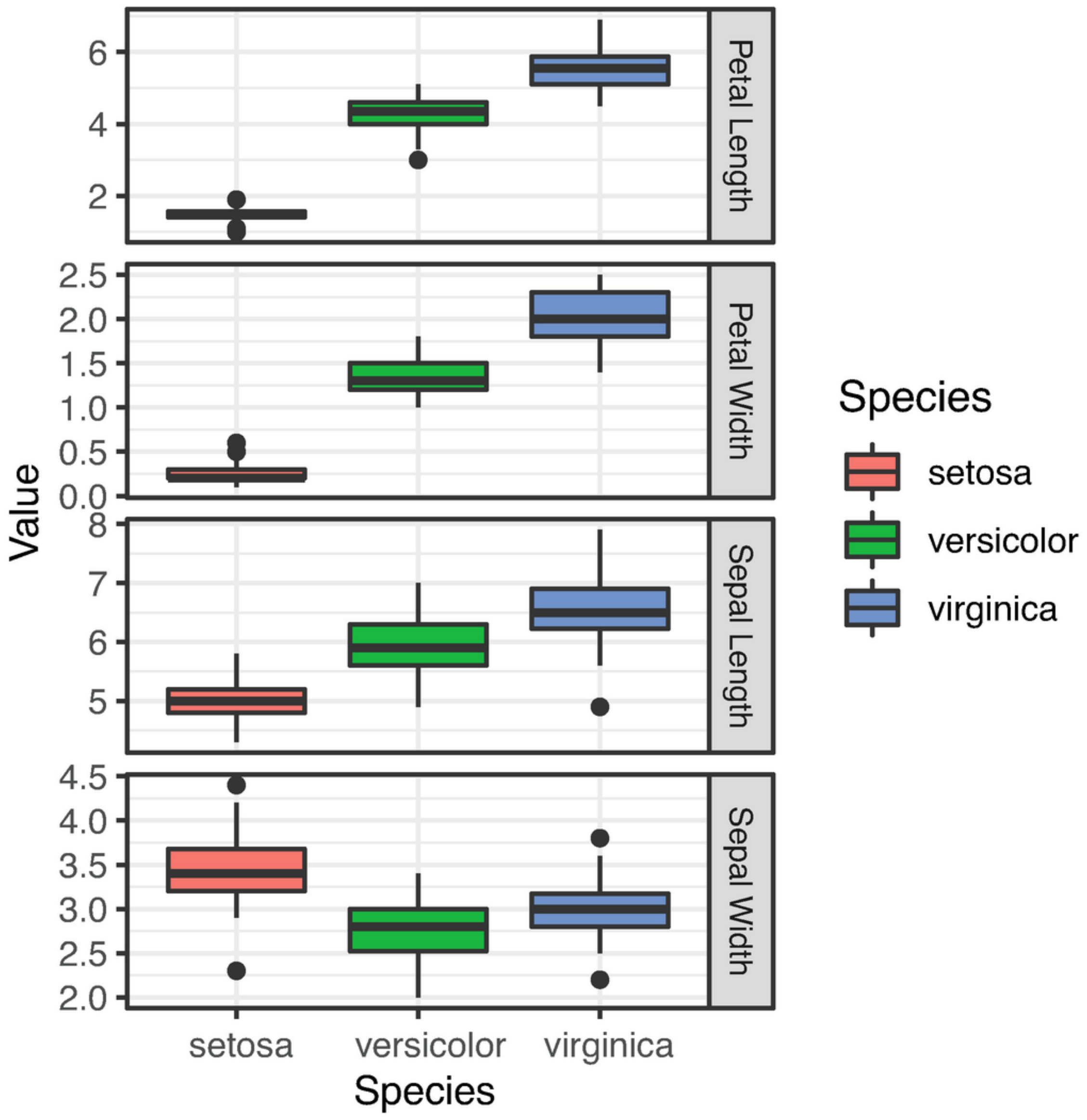

Figure 4-25 Iris data plotted with default fill colors

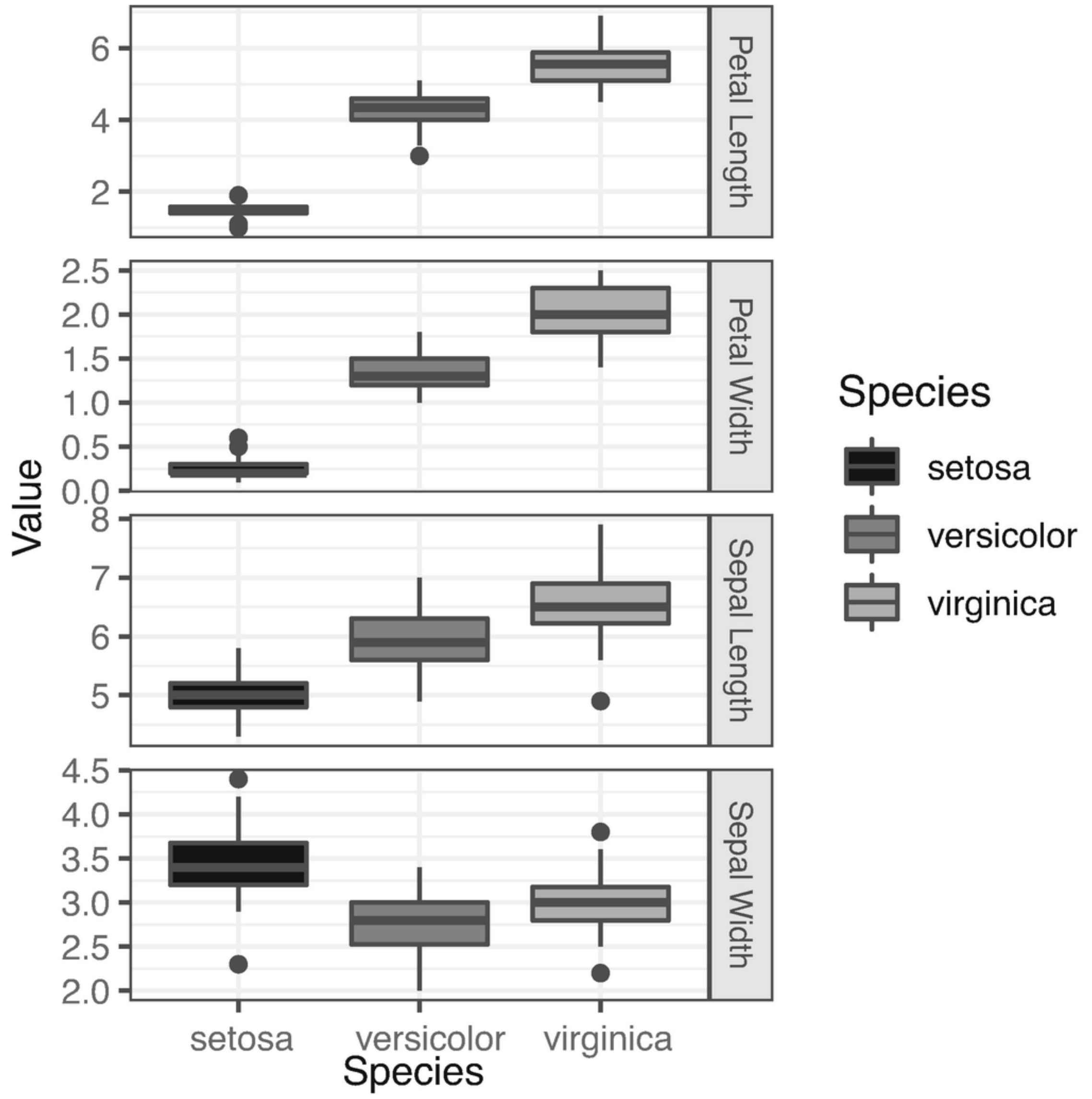

Figure 4-26 Iris data plotted with custom fill colors

Explicitly setting colors is a risky business, though, unless you have a good feeling for how colors work together and which combinations can be problematic for color blind people. It is better to use one of the "brewer" choices. These are methods for constructing good combinations of colors (see `http://colorbrewer2.org`), and you can use them with the `scale_fill_brewer()` function (see Figure 4-27):

```
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value, fill = Species)) +
  geom_boxplot() +
  scale_fill_brewer(palette = "Greens") +
  facet_grid(Measurement ~ ., scale = "free_y",
             labeller = labeller(Measurement = label_map))
```

Themes and Other Graphics Transformations

Most of using `ggplot2` consist of specifying geometries and scales to control how data is mapped to visual components, but you also have much control over how the final plot will look through functions that only concern the final result.

Most of this is done by modifying the so-called theme. If you have tried the examples I have given in this chapter yourself, the results might look different from the figures in this book. This is because I have set up a default theme for the book using the command

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

The `theme_bw()` sets up the final visual appearance of the figures you see here. You can add a theme to a plot using `+` as you would any other `ggplot2` modification or set it as default as I have done here. There are several themes you can use; you can look for functions that start with `theme_`, but all of them can be modified to get more control over a plot.

Besides themes, various other functions also affect the way a plot looks. There is far too much to cover here on all the things you can do with themes and graphics transformations, but I can show you an example that should give you an idea of what can be achieved.

Figure 4-27 Iris data plotted with brewer fill colors

You can, for instance, change coordinate systems using various `coord_` functions—the simplest is just flipping x and y with `coord_flip()`. This can, of course, also be achieved by changing the aesthetics, but flipping the coordinates of a complex plot can be easier than updating aesthetics several places. For the `iris` plot we have looked at before, I might want to change the axes.

I also want to put the measurement labels on the left instead of on the right. You can control the placement of facet labels using the `switch` option to `facet_grid()`, and giving the `switch` parameter the value `y` will switch the location of that label:

```
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value, fill = Species)) +
  geom_boxplot() +
  scale_x_discrete(labels = c("setosa" = "Setosa",
                              "versicolor" = "Versicolor",
                              "virginica" = "Virginica")) +
  scale_fill_brewer(palette = "Greens") +
  facet_grid(Measurement ~ ., switch = "y",
             labeller = labeller(Measurement = label_map)) +
  coord_flip()
```

If I just flip the coordinates, the axis labels on the new x-axis will be wrong if I tell the `facet_grid()` function to have a free y-axis. With a free y-axis, it would have different ranges for the y values, which is what we want, but after flipping the coordinates, we will only see the values for one of the y-axes. The other values will be plotted as if they were on the same axis, but they won't be. So I have removed the `scale` parameter to `facet_grid()`. Try to put it back and see what happens.

Figure 4-28 Iris with flipped coordinates and switched facet labels

The result so far is shown in Figure 4-28. We have flipped coordinates and moved labels, but the labels look ugly with the color background. We can remove it by modifying the theme using `theme(strip.background = element_blank())`. It just sets the strip.background, which is the graphical property of facet labels, to a blank element, so in effect it removes the background color. We can also move the legend label using a theme modification: `theme(legend.position="top")`.

```
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value, fill = Species)) +
  geom_boxplot() +
  scale_x_discrete(labels = c("setosa" = "Setosa",
                              "versicolor" = "Versicolor",
                              "virginica" = "Virginica")) +
  scale_fill_brewer(palette = "Greens") +
  facet_grid(Measurement ~ ., switch = "y",
             labeller = labeller(Measurement = label_map)) +
  coord_flip() +
  theme(strip.background = element_blank()) +
  theme(legend.position="top")
```

The result is now as seen in Figure 4-29. It is pretty close to something we could print. We just want the labelled species to be in capital letters just like the axis labels.

Figure 4-29 Iris data with theme modifications

Well, we know how to do that using the `labels` parameter to a scale so the final plotting code could look like this:

```
label_map <- c(Petal.Width = "Petal Width",
               Petal.Length = "Petal Length",
               Sepal.Width = "Sepal Width",
               Sepal.Length = "Sepal Length")
species_map <- c(setosa = "Setosa",
                 versicolor = "Versicolor",
                 virginica = "Virginica")
iris %>%
  pivot_longer(
    -Species,
    names_to = "Measurement",
    values_to = "Value"
  ) %>%
  ggplot(aes(x = Species, y = Value, fill = Species)) +
  geom_boxplot() +
  scale_x_discrete(labels = species_map) +
  scale_fill_brewer(palette = "Greens", labels = species_map)
+
  facet_grid(Measurement ~ ., switch = "y",
             labeller = labeller(Measurement = label_map)) +
  coord_flip() +
  theme(strip.background = element_blank()) +
  theme(legend.position="top")
```

and the result is seen in Figure 4-30.

Figures with Multiple Plots

Using facets covers many of the situation where you want to have multiple panels in the same plot, but not all. You use facets when you want to display different subsets of the data in separate panels but essentially have the same plot for the subsets. Sometimes, you want to combine different types of plots, or plots of different data sets, as subplots in different panels. For that, you need to combine otherwise independent plots.

The `ggplot2` package doesn't directly support combining multiple plots, but it can be achieved using the underlying graphics system, `grid`. Working with basic `grid`, you have many low-level tools for modifying graphics, but for just combining plots, you want more high-level functions, and you can get that from the `gridExtra` package.

Figure 4-30 Final version of the iris plot

To combine plots, you first create them as you normally would. So, for example, we could make two plots of the `iris` data like this:

```
petal <- iris %>% ggplot() +
  geom_point(aes(x = Petal.Width, y = Petal.Length,
                     color = Species)) +
   scale_color_grey() +
  theme(legend.position="none")
sepal <- iris %>% ggplot() +
  geom_point(aes(x = Sepal.Width, y = Sepal.Length,
                     color = Species)) +
  scale_color_grey() +
  theme(legend.position="none")
```

We then import the `gridExtra` package:

```
library(gridExtra)
```

and can then use the `grid.arrange()` function to create a grid of plots, putting in the two plots we just created (see Figure 4-31):

```
grid.arrange(petal, sepal, ncol = 2)
```

Another approach I like to use is the `plot_grid()` function from the `cowplot` package. This package contains several functions developed by Claus O. Wilke (where the `cow` comes from) for his plotting needs, and loading it will redefine the default `ggplot2` theme. You can use the `theme_set()` function to change it back if you don't like the theme that `cowplot` provides.

Anyway, creating a plot with subplots using `cowplot`, we have to import the package:

```
library(cowplot)
```

Figure 4-31 Combining two plots of the iris data using grid.arrange

Figure 4-32 Combining two plots of the iris data using cowplot

If we don't want the theme it sets here, we need to change it again using `theme_set()`, but otherwise we can combine the plots we have defined before using `plot_grid()` (see Figure 4-32):

```
plot_grid(petal, sepal, labels = c("A", "B"))
```

With the `patchwork` package, combining plots is even easier. You can just add them together to get them next to each other:

```
library(patchwork)
petal + sepal
```

The pipe operator does the same thing as the plus, so this composition does the same thing:

```
petal | sepal
```

If you want to stack one plot over another, you use `/`:

```
petal / sepal
```

You can combine these using parentheses, so if you want the `petal` and `sepal` plots next to each other and over another `sepal` plot, you can use

```
(petal + sepal) / sepal
```

Exercises

In the previous chapter, you should have imported a data set and used `dplyr` and `tidyr` to explore it using summary statistics. Now do the same thing using plotting. If you looked at summary statistics, try representing these as box plots or smoothed scatter plots. If you have different variables that you used `tidyr` to gather, try to plot the data similar to what you saw for `iris` earlier.