3

Describing Data

In this chapter, we tackle our first marketing analytics problem: summarizing and exploring a data set with descriptive statistics (mean, standard deviation, and so forth) and visualization methods. Such investigation is the simplest analysis one can do yet also the most crucial. It is important to describe and explore any data set before moving on to more complex analysis. This chapter will build your R skills and provide a set of tools for exploring your own data.

3.1 Simulating Data

We start by creating data to be analyzed in later parts of the chapter. Why simulate data and not work entirely with real data sets? There are several reasons. The process of creating data lets us practice and deepen R skills from Chap. 2. It makes the book less dependent on vagaries of finding and downloading online data sets. And it lets you manipulate the synthetic data, run analyses again, and examine how the results change.

Perhaps most importantly, data simulation highlights a strength of R: because it is easy to simulate data, R analysts often use simulated data to prove that their methods are working as expected. When we know what the data *should* say (because we created it), we can test our analyses to make sure they are working correctly before applying them to real data. If you have real data sets that you work with regularly, we encourage you to use those for the same analyses alongside our simulated data examples. (See Sect. 2.6 for more information on how to load data files.)

We encourage you to create data in this section step-by-step because we teach R along the way. However, if you are in a hurry to learn how to compute means, standard deviations, and other summary statistics, you could quickly run the commands

C. Chapman, E.M. Feit, *R for Marketing Research and Analytics*, Use R!,
DOI 10.1007/978-3-319-14436-8_3

in this section to generate the simulated data. Alternatively, the following will load the data from the book's website, and you can then go to Sect. 3.2:

```
> store.df <- read.csv("http://goo.gl/QPDdMl")
```

But if you're new to R, don't do that! Instead, work through the following section to create the data from scratch. If you accidentally ran the command above, you can use `rm(store.df)` to remove the data before proceeding.

3.1.1 Store Data: Setting the Structure

Our first data set represents observations of total sales by week for two products at a chain of stores. We begin by creating a data structure that will hold the data, a simulation of sales for the two products in 20 stores over 2 years, with price and promotion status. We remove most of the R output here to focus on the input commands. Type the following lines, but feel free to omit the comments (following "#"):

```
> k.stores <- 20    # 20 stores, using "k." for "constant"
> k.weeks <- 104    # 2 years of data each

# create a data frame of initially missing values to hold the data
> store.df <- data.frame(matrix(NA, ncol=10, nrow=k.stores*k.weeks))
> names(store.df) <- c("storeNum", "Year", "Week", "p1sales", "p2sales",
+                      "p1price", "p2price", "p1prom", "p2prom", "country")
```

We see the simplest summary of the data frame using `dim()`:

```
> dim(store.df)
[1] 2080   10
```

As expected, `store.df` has 2,080 rows and 10 columns. We create two vectors that will represent the store number and country for each observation:

```
> store.num <- 101:(100+k.stores)
> (store.cty <- c(rep("US", 3), rep("DE", 5), rep("GB", 3), rep("BR", 2),
+                 rep("JP", 4), rep("AU", 1), rep("CN", 2)))
 [1] "US" "US" "US" "DE" "DE" "DE" "DE" "DE" "GB" "GB" "GB" "BR" "BR" "JP" ...
> length(store.cty)    # make sure the country list is the right length
[1] 20
```

Now we replace the appropriate columns in the data frame with those values, using `rep()` to expand the vectors to match the number of stores and weeks:

```
> store.df$storeNum <- rep(store.num, each=k.weeks)
> store.df$country  <- rep(store.cty, each=k.weeks)
> rm(store.num, store.cty)    # clean up
```

Next we do the same for the `Week` and `Year` columns:

```
> (store.df$Week <- rep(1:52, times=k.stores*2))
   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 ...
> # try the inner parts of the next line to figure out how we use rep()
> (store.df$Year  <- rep(rep(1:2, each=k.weeks/2), times=k.stores))
   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
```

We check the overall data structure with `str()`:

```
> str(store.df)
'data.frame': 2080 obs. of  10 variables:
 $ storeNum: int  101 101 101 101 101 101 101 101 101 101 ...
 $ Year    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Week    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ p1sales : logi  NA NA NA NA NA NA ...
 $ p2sales : logi  NA NA NA NA NA NA ...
 $ p1price : logi  NA NA NA NA NA NA ...
 $ p2price : logi  NA NA NA NA NA NA ...
 $ p1prom  : logi  NA NA NA NA NA NA ...
 $ p2prom  : logi  NA NA NA NA NA NA ...
 $ country : chr  "US" "US" "US" "US" ...
```

The data frame has the right number of observations and variables, and proper column names.

R chose types for all of the variables in our data frame. For example, `store.df$country` is of type `chr` (character) because we assigned a vector of strings to it. However, country labels are actually discrete values and not just arbitrary text. So it is better to represent country explicitly as a categorical variable, known in R as a *factor*. Similarly, `storeNum` is a label, not a number as such. By converting those variables to factors, R knows to treat them as a categorical in subsequent analyses such as regression models. It is good practice to set variable types correctly as they are created; this will help you to avoid errors later.

We redefine `store.df$storeNum` and `store.df$country` as categorical using `factor()`:

```
> store.df$storeNum <- factor(store.df$storeNum)
> store.df$country  <- factor(store.df$country)
> str(store.df)
'data.frame': 2080 obs. of  10 variables:
 $ storeNum: Factor w/ 20 levels "101","102","103",..: 1 1 1 1 1 1 1 1 1 1 ...
... [rows omitted] ...
 $ country : Factor w/ 7 levels "AU","BR","CN",..: 7 7 7 7 7 7 7 7 7 7 ...
```

`storeNum` and `country` are now defined as factors with 20 and 7 levels, respectively.

It is a good idea to inspect data frames in the first and last rows because mistakes often surface there. You can use `head(x=DATA, n=NUMROWS)` and `tail()` commands to inspect the beginning and end of the data frame (we omit long output from the last two commands):

```
> head(store.df)   # defaults to 6 rows
  storeNum Year Week p1sales p2sales p1price p2price p1prom p2prom country
1      101    1    1      NA      NA      NA      NA     NA     NA      US
2      101    1    2      NA      NA      NA      NA     NA     NA      US
3      101    1    3      NA      NA      NA      NA     NA     NA      US
...
> head(store.df, 120)  # 120 rows is enough to check 2 stores; not shown
> tail(store.df, 120)  # make sure end looks OK too; not shown
```

All of the specific measures (sales, price, promotion) are shown as missing values (indicated by `NA`) because we haven't assigned other values to them yet, while the store numbers, year counters, week counters, and country assignments look good. It's always useful to debug small steps like this as you go.

3.1.2 Store Data: Simulating Data Points

We complete `store.df` with random data for *store-by-week* observations of the sales, price, and promotional status of 2 products.

Before simulating random data, it is important to set the random number generation *seed* to make the process replicable. After setting a seed, when you draw random samples in the same sequence again, you get exactly the same (*pseudo-*)random numbers. Pseudorandom number generators (PRNGs) are a complex topic whose issues are out of scope here. If you are using PRNGs for something important, you should review the literature; it has been said that whole shelves of journals could be thrown away due to poor usage of random numbers. (R has support for a wide array of pseudorandom sequences; see `?set.seed` for details. A starting point to learn more abut PRNGs is Knuth [93].)

If you don't set a PRNG seed, R will select one for you, but you will get different random numbers each time you repeat the process. If you set the seed and execute commands in the order shown in this book, you will get the results that we show.

```
> set.seed(98250)   # a favorite US postal code
```

Now we can draw the random data. In each row of data—that is, one week of 1 year, for one store—we set the status of whether each product was promoted (value `1`) by drawing from the *binomial distribution* that counts the number of "heads" in a collection of coin tosses (where the coin can have any proportion of heads, not just 50 %).

To detail that process: we use the `rbinom(n, size, p)` (decoded as "*r*andom *binom*ial") function to draw from the binomial distribution. For every row of the store data, as noted by `n=nrow(store.df)`, we draw from a distribution representing the number of heads in a single coin toss (`size=1`) with a coin that has probability `p=0.1` for product 1 and `p=0.15` for product 2. In other words, we arbitrarily assign a 10 % likelihood of promotion for product 1, and 15 % likelihood for product 2 and then randomly determine which weeks have promotions.

```
> store.df$p1prom <- rbinom(n=nrow(store.df), size=1, p=0.1)  # 10% promoted
> store.df$p2prom <- rbinom(n=nrow(store.df), size=1, p=0.15) # 15% promoted
> head(store.df)  # how does it look so far? (not shown)
```

Next we set a price for each product in each row of the data. We suppose that each product is sold at one of five distinct price points ranging from $2.19 to $3.19 overall. We randomly draw a price for each week by defining a vector with the five price points and using `sample(x, size, replace)` to draw from it as many times as we have rows of data (`size=nrow(store.df)`). The five prices are sampled many times, so we sample with replacement (`replace=TRUE`):

```
> store.df$p1price <- sample(x=c(2.19, 2.29, 2.49, 2.79, 2.99),
+                            size=nrow(store.df), replace=TRUE)
> store.df$p2price <- sample(x=c(2.29, 2.49, 2.59, 2.99, 3.19),
+                            size=nrow(store.df), replace=TRUE)
> head(store.df)  # now how does it look?
  storeNum Year Week p1sales p2sales p1price p2price p1prom p2prom country
1      101    1    1      NA      NA    2.29    2.29      0      0      US
2      101    1    2      NA      NA    2.49    2.49      0      0      US
3      101    1    3      NA      NA    2.99    2.99      1      0      US
...
```

Question: if *price* occurs at five discrete levels, does that make it a factor variable? That depends on the analytic question, but in general probably not. We often perform math on price, such as subtracting cost in order to find gross margin, multiplying by units to find total sales, and so forth. Thus, even though it may have only a few unique values, price is a number, not a factor.

Our last step is to simulate the sales figures for each week. We calculate sales as a function of the relative prices of the two products along with the promotional status of each.

Item sales are in unit counts, so we use the Poisson distribution to generate count data: `rpois(n, lambda)`, where `n` is the number of draws and `lambda` is the mean value of units per week. We draw a random Poisson count for each row (`nrow(store.df)`), and set the mean sales (`lambda`) of Product 1 to be higher than that of Product 2:

```
# sales data, using poisson (counts) distribution, rpois()
# first, the default sales in the absence of promotion
> tmp.sales1 <- rpois(nrow(store.df), lambda=120)
> tmp.sales2 <- rpois(nrow(store.df), lambda=100)
```

Now we scale those counts up or down according to the relative prices. Price effects often follow a logarithmic function rather than a linear function, so we use `log(price)` here:

```
# scale sales according to the ratio of log(price)
> tmp.sales1 <- tmp.sales1 * log(store.df$p2price) / log(store.df$p1price)
> tmp.sales2 <- tmp.sales2 * log(store.df$p1price) / log(store.df$p2price)
```

We have assumed that sales vary as the *inverse* ratio of prices. That is, sales of Product 1 go up to the degree that the `log(price)` of Product 1 is lower than the `log(price)` of Product 2.

Finally, we assume that sales get a 30 % or 40 % lift when each product is promoted in store. We simply multiply the promotional status vector (which comprises all $\{0, 1\}$ values) by 0.3 or 0.4, respectively, and then multiply the sales vector by that. We use the `floor()` function to drop fractional values and ensure integer counts for weekly unit sales, and put those values into the data frame:

```
# final sales get a 30% or 40% lift when promoted
> store.df$p1sales <- floor(tmp.sales1 * (1 + store.df$p1prom*0.3))
> store.df$p2sales <- floor(tmp.sales2 * (1 + store.df$p2prom*0.4))
```

Inspecting the data frame, we see that the data look plausible on the surface:

```
> head(store.df)
  storeNum Year Week p1sales p2sales p1price p2price p1prom p2prom country
1      101    1    1     127     106    2.29    2.29      0      0      US
2      101    1    2     137     105    2.49    2.49      0      0      US
3      101    1    3     156      97    2.99    2.99      1      0      US
...
```

A final command is useful to inspect data because it selects rows at random and thus may find problems buried inside a data frame away from the `head` or `tail`: `some()` from the `car` package [51]:

```
> install.packages("car")   # if needed
> library(car)
> some(store.df, 10)
    storeNum Year Week p1sales p2sales p1price p2price p1prom p2prom country
27       101    1   27     135      99    2.29    2.49      0      0      US
144      102    1   40     123     113    2.79    2.59      0      0      US
473      105    2    5     127      96    2.99    3.19      0      0      DE
...
```

Thanks to the power of R, we have created a simulated data set with 20,800 values (2,080 rows $\times$ 10 columns) using a total of 22 assignment commands. In the next section we explore the data that we created.

3.2 Functions to Summarize a Variable

Observations may comprise either *discrete* data that occurs at specific levels or *continuous* data with many possible values. We look at each type in turn.

3.2.1 Discrete Variables

A basic way to describe discrete data is with frequency counts. The `table()` function will count the observed prevalence of each value that occurs in a variable (i.e., a vector or a column in a data frame). In `store.df`, we may count how many times Product 1 was observed to be on sale at each price point:

```
> table(store.df$p1price)

2.19 2.29 2.49 2.79 2.99
 395  444  423  443  375
```

If your counts vary from those above, that may be due to running commands in a different order or setting a different random number seed. The counts shown here assume that the commands have been run in the exact sequence shown in this chapter. There is no problem if your data is modestly different; just remember that it won't match the output here, or try Sect. 3.1.1 again.

One of the most useful features of R is that most functions produce an object that you can save and use for further commands. So, for example, if you want to save the table that was created by `table()`, you can just assign the same command to a named object:

```
> p1.table <- table(store.df$p1price)
> p1.table

2.19 2.29 2.49 2.79 2.99
 395  444  423  443  375
> str(p1.table)
 'table' int [1:5(1d)] 395 444 423 443 375
...
```

The `str()` command shows us that the object produced by `table()` is a special type called `table`. You will find many functions in R produce objects of special types. We can also easily pass `p1.table` to the `plot()` function to produce a quick plot.

```
> plot(p1.table)
```

You can see from the resulting bar plot in Fig. 3.1 that the product was on sale at each price point roughly the same number of times. R chose a type of plot suitable for our `table` object, but it is fairly ugly and the labels could be clearer. Later in this chapter we show how to modify a plot to get better results.

An analyst might want to know how often each product was promoted at each price point. The `table()` command produces two-way *cross tabs* when a second variable is included:

```
> table(store.df$p1price, store.df$p1prom)

        0   1
  2.19 354  41
  2.29 398  46
```

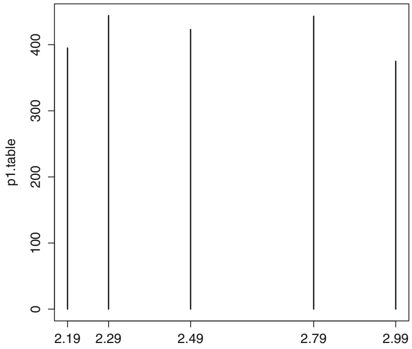

Fig. 3.1. A simple bar plot produced by passing a table object to `plot()`. Default charts are sometimes unattractive, but there are many options to make them more attractive and useful.

```
  2.49 381  42
  2.79 396  47
  2.99 343  32
```

At each price level, Product 1 is observed to have been promoted approximately 10 % of the time (as expected, given how we created the data in Sect. 3.1.1). In fact, we can compute the exact fraction of times product 1 is on promotion at each price point, if we assign the table to a variable and then divide the second column of the table by the sum of the first and second columns:

```
> p1.table2 <- table(store.df$p1price, store.df$p1prom)
> p1.table2[, 2] / (p1.table2[, 1] + p1.table2[, 2])
      2.19       2.29       2.49       2.79       2.99
0.10379747 0.10360360 0.09929078 0.10609481 0.08533333
```

The second command takes the second column of table `p1.table`—the column with counts of how often the product is promoted—and divides by the total count to get the proportion of times the product was promoted at each price point. R automatically applies math operators + and / across the entire columns.

By combining results in this way, you can easily produce exactly the results you want along with code that can repeat the analysis on demand. This is very helpful to marketing analysts who produce weekly or monthly reports for sales, web traffic, and the like.

3.2.2 Continuous Variables

Counts are useful when we have a small number of categories, but with continuous data it is more helpful to summarize the data in terms of its distribution. The most common way to do that is with mathematical functions that describe the range of the data, its center, the degree to which it is concentrated or dispersed, and specific points that may be of interest (such as the 90th percentile). Table 3.1 lists some R functions to calculate statistics for numeric vector data, such as numeric columns in a data frame.

Table 3.1. Distribution functions that operate on a numeric vector

Describe	Function	Value
Extremes	`min(x)` `max(x)`	Minimum value Maximum value
Central tendency	`mean(x)` `median(x)`	Arithmetic mean Median

Dispersion	`var(x)`
	`sd(x)`
	
	`IQR(x)`
	
	`mad(x)`
		Variance around the mean
		Standard deviation (`sqrt(var(x))`)
		Interquartile range, 75th–25th percentile
		Median absolute deviation (a robust variance estimator)

Points	`quantile(x, probs=c(...))`	Percentiles

Following are examples of those common functions:

```
> min(store.df$p1sales)
[1] 73
> max(store.df$p2sales)
[1] 225
> mean(store.df$p1prom)
[1] 0.1
> median(store.df$p2sales)
[1] 96
> var(store.df$p1sales)
[1] 805.0044
> sd(store.df$p1sales)
[1] 28.3726
> IQR(store.df$p1sales)
[1] 37
> mad(store.df$p1sales)
[1] 26.6868
> quantile(store.df$p1sales, probs=c(0.25, 0.5, 0.75))
25% 50% 75%
113 129 150
```

In the case of `quantile()` we have asked for the 25th, 50th, and 75th percentiles using the argument `probs=c(0.25, 0.5, 0.75)`, which are also known as the *median* (50th percentile, same as the `median()` function) and the edges of the *interquartile range*, the 25th and 75th percentiles.

For skewed and asymmetric distributions that are common in marketing, such as unit sales or household income, the arithmetic `mean()` and standard deviation `sd()` may be misleading; in those cases, the `median()` and interquartile range (`IQR()`, the range of the middle 50 % of data) are often more useful to summarize a distribution.

Change the `probs=` argument in `quantile()` to find other quantiles:

```
> quantile(store.df$p1sales, probs=c(0.05, 0.95))  # central 90% of data
 5% 95%
 93 184
> quantile(store.df$p1sales, probs=0:10/10)
   0%   10%   20%   30%   40%   50%   60%   70%   80%   90%  100%
 73.0 100.0 109.0 117.0 122.6 129.0 136.0 145.0 156.0 171.0 263.0
```

The second example here shows that we may use sequences in many places in R; in this case, we find every 10th percentile by creating a simple sequence of `0:10` and dividing by 10 to yield the vector `0, 0.1, 0.2 ... 1.0`. You could also do this using the sequence function (`seq(from=0, to=1, by=0.1)`), but `0:10/10` is shorter and more commonly used.

Suppose we wanted a summary of the sales for product 1 and product 2 based on their median and interquartile range. We might assemble these summary statistics into a data frame that is easer to read than the one-line-at-a-time output above. We create a data frame to hold our summary statistics and then populate it using functions from Table 3.1. We name the columns and rows, and fill in the cells with function values:

```
> mysummary.df <- data.frame(matrix(NA, nrow=2, ncol=2))
> names(mysummary.df) <- c("Median Sales", "IQR")
> rownames(mysummary.df) <- c("Product 1", "Product 2")
> mysummary.df["Product 1", "Median Sales"] <- median(store.df$p1sales)
> mysummary.df["Product 2", "Median Sales"] <- median(store.df$p2sales)
> mysummary.df["Product 1", "IQR"] <- IQR(store.df$p1sales)
> mysummary.df["Product 2", "IQR"] <- IQR(store.df$p2sales)
> mysummary.df
          Median Sales IQR
Product 1          129  37
Product 2           96  29
```

With this custom summary we can easily see that median sales are higher for product 1 (129 versus 96) and that the variation in sales of product 1 (the IQR across observations by week) is also higher. Once we have this code, we can easily run it the next time we have new sales data to produce a revised version of our table of summary statistics. Such code might be a good candidate for a custom function you can reuse (see Sects. 2.7 and 11.3.1.1). We'll see a shorter way to create this summary in Sect. 3.3.4.

3.3 Summarizing Data Frames

As useful as functions such as `mean()` and `quantile()` are, it is tedious to apply them one at a time to columns of a large data frame, as we did with the summary table above. R provides a variety of ways to summarize data frames without writing extensive code. We describe three approaches: the basic `summary()` command, the `describe()` command from the `psych` package, and the R approach to iterating over variables with `apply()`.

3.3.1 summary()

As we saw in Sect. 2.5, `summary()` is a good way to do a preliminary inspection of a data frame or other object. When you use `summary()` on a data frame, it reports a few descriptive statistics for every variable:

```
> summary(store.df)
   storeNum          Year            Week           p1sales         p2sales     
 101    : 104   Min.   :1.0   Min.   : 1.00   Min.   : 73   Min.   : 51.0  
 102    : 104   1st Qu.:1.0   1st Qu.:13.75   1st Qu.:113   1st Qu.: 84.0  
 103    : 104   Median :1.5   Median :26.50   Median :129   Median : 96.0  
 104    : 104   Mean   :1.5   Mean   :26.50   Mean   :133   Mean   :100.2  
 105    : 104   3rd Qu.:2.0   3rd Qu.:39.25   3rd Qu.:150   3rd Qu.:113.0  
 106    : 104   Max.   :2.0   Max.   :52.00   Max.   :263   Max.   :225.0  
 (Other):1456                                                              
   p1price         p2price          p1prom          p2prom          country
 Min.   :2.190   Min.   :2.29   Min.   :0.0   Min.   :0.0000   AU:104  
 1st Qu.:2.290   1st Qu.:2.49   1st Qu.:0.0   1st Qu.:0.0000   BR:208  
 Median :2.490   Median :2.59   Median :0.0   Median :0.0000   CN:208  
 Mean   :2.544   Mean   :2.70   Mean   :0.1   Mean   :0.1385   DE:520  
 3rd Qu.:2.790   3rd Qu.:2.99   3rd Qu.:0.0   3rd Qu.:0.0000   GB:312  
 Max.   :2.990   Max.   :3.19   Max.   :1.0   Max.   :1.0000   JP:416  
                                                               US:312  
```

`summary()` works similarly for single vectors, with a horizontal rather than vertical display:

```
> summary(store.df$Year)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    1.0     1.0     1.5     1.5     2.0     2.0 
```

The `digits=` argument is helpful if you wish to change the precision of the display:

```
> summary(store.df, digits=2)
   storeNum          Year            Week        p1sales         p2sales    
 101    : 104   Min.   :1.0   Min.   : 1   Min.   : 73   Min.   : 51  
 102    : 104   1st Qu.:1.0   1st Qu.:14   1st Qu.:113   1st Qu.: 84  
...
   p1price        p2price         p1prom         p2prom        country
 Min.   :2.2   Min.   :2.3   Min.   :0.0   Min.   :0.00   AU:104  
 1st Qu.:2.3   1st Qu.:2.5   1st Qu.:0.0   1st Qu.:0.00   BR:208  
...
```

R generally uses *digits* to mean *significant digits* regardless of absolute magnitude or the decimal position. Thus, `digits=3` does not mean "three decimal places" but instead "three significant positions." Output conforming to `digits=` is not guaranteed; the format may be different in various cases such as reporting integer values and for factors.

Perhaps the most important use for `summary()` is this: *after importing data, use* `summary()` *to do a quick quality check*. Check the `min` and `max` for outliers or miskeyed data, and check to see that the `mean` and `median` are reasonable and similar to one another (if you expect them to be similar, of course). This simple inspection often turns up errors in the data!

3.3.2 describe()

Another useful command is `describe()` from the `psych` package [132]. To use `describe()`, install the `psych` package if you haven't done so already and make it available with `library()`:

```
> install.packages("psych")
Installing package ...
> library(psych)
```

`describe()` reports a variety of statistics for each variable in a data set, including *n*, the count of observations; *trimmed mean*, the mean after dropping a small proportion of extreme values; and statistics such as *skew* and *kurtosis* that are useful when interpreting data with regard to normal distributions.

```
> describe(store.df)
          vars    n   mean    sd median trimmed   mad   min    max range  skew
storeNum*    1 2080  10.50  5.77  10.50   10.50  7.41  1.00  20.00  19.0  0.00
Year         2 2080   1.50  0.50   1.50    1.50  0.74  1.00   2.00   1.0  0.00
Week         3 2080  26.50 15.01  26.50   26.50 19.27  1.00  52.00  51.0  0.00
p1sales      4 2080 133.05 28.37 129.00  131.08 26.69 73.00 263.00 190.0  0.74
...
country*    10 2080   4.55  1.72   4.50    4.62  2.22  1.00   7.00   6.0 -0.29
          kurtosis   se
storeNum*    -1.21 0.13
Year         -2.00 0.01
Week         -1.20 0.33
p1sales       0.66 0.62
...
country*     -0.81 0.04
```

By comparing the trimmed mean to the overall mean, one might discover when outliers are skewing the mean with extreme values. `describe()` is especially recommended for summarizing survey data with discrete values such as 1–7 Likert scale items from surveys (items that use a scale with ordered values such as "Strongly disagree (1)" to "Strongly agree (7)" or similar).

Note that there is an `*` next to the labels for `storeNum` and `country` in the output above. This is a warning; `storeNum` and `country` are factors and these summary statistics may not make sense for them. `describe()` treats each store number as an integer and computes statistics based on those integers. This may be useful when your factors are in a meaningful order. When data include character strings or other non-numeric data, `describe()` gives an error, "`non-numeric argument`." These problems may be solved by selecting only the variables (columns) that are numeric with matrix indices. For example, if we wished to describe only columns 2 and 4 through 9, then we could use the following:

```
> describe(store.df[ , c(2, 4:9)])
        vars    n   mean    sd median trimmed   mad   min    max range skew
Year       1 2080   1.50  0.50   1.50    1.50  0.74  1.00   2.00   1.0 0.00
p1sales    2 2080 133.05 28.37 129.00  131.08 26.69 73.00 263.00 190.0 0.74
p2sales    3 2080 100.16 24.42  96.00   98.05 22.24 51.00 225.00 174.0 0.99
```

```
p1price    4 2080   2.54  0.29   2.49    2.53  0.44  2.19   2.99   0.8 0.28
p2price    5 2080   2.70  0.33   2.59    2.69  0.44  2.29   3.19   0.9 0.32
...
```

3.3.3 Recommended Approach to Inspecting Data

We can now recommend a general approach to inspecting a data set after compiling or importing it; replace "`my.data`" and "`DATA`" with the names of your objects:

1. Import your data with `read.csv()` or another appropriate function and check that the importation process gives no errors.
2. Convert it to a data frame if needed (`my.data <- data.frame(DATA)` and set column names (`names(my.data) <- c(...)`) if needed.
3. Examine `dim()` to check that the data frame has the expected number of rows and columns.
4. Use `head()` and `tail(my.data)` to check the first few and last few rows; make sure that header rows at the beginning and blank rows at the end were not included accidentally. Also check that no good rows were skipped at the beginning.
5. Use `some()` from the `car` package to examine a few sets of random rows.
6. Check the data frame structure with `str()` to ensure that variable types and values are appropriate. Change the type of variables—especially to `factor` types—as necessary.
7. Run `summary()` and look for unexpected values, especially `min` and `max` that are unexpected.
8. Load the `psych` library and examine basic descriptives with `describe()`. Reconfirm the observation counts by checking that `n` is the same for each variable, and check trimmed mean and skew (if relevant).

3.3.4 `apply()`*

An advanced and powerful tool in R is the `apply()` command. `apply(x=DATA, MARGIN=MARGIN, FUN=FUNCTION)` runs any function that you specify on each of the rows and/or columns of an object. If that sounds cryptic, well…it is. In R the term *margin* is a two-dimensional metaphor that denotes which "direction" you want to do something: either along the rows (`MARGIN=1`) or columns (`MARGIN=2`), or both simultaneously (`MARGIN=c(1, 2)`).

Here's an example: suppose we want to find the mean of every column of `store.df`, except for `store.df$Store`, which isn't a number and so doesn't have a mean. We can `apply()` the `mean()` function to the *column* margin of the data like this:

```
> apply(store.df[,2:9], MARGIN=2, FUN=mean)
       Year        Week     p1sales     p2sales     p1price     p2price
  1.5000000  26.5000000 133.0485577 100.1567308   2.5443750   2.6995192
     p1prom      p2prom
  0.1000000   0.1384615
```

As it happens, `colMeans()` does the same thing as the command above, but `apply` gives you the flexibility to apply any function you like. If we want the *row* means instead, we simply change the margin to `1`:

```
> apply(store.df[,2:9], 1, mean)
   [1] 29.9475 31.2475 32.9975 29.2725 31.2600 31.7850 27.5225 30.7850 28.0725
  [10] 31.5600 30.5975 32.5850 25.6350 29.3225 27.9225 30.5350 31.4475 ...
```

Although row means make little sense for this data set, they can be useful for other kinds of data.

Similarly, we might find the `sum()` or `sd()` for multiple columns with `margin=2`:

```
> apply(store.df[,2:9], 2, sum)
    Year     Week  p1sales  p2sales  p1price  p2price   p1prom   p2prom
  3120.0  55120.0 276741.0 208326.0   5292.3   5615.0    208.0    288.0
> apply(store.df[,2:9], 2, sd)
      Year       Week    p1sales    p2sales    p1price    p2price     ...
 0.5001202 15.0119401 28.3725990 24.4241905  0.2948819  0.3292181   ...
```

What if we want to know something more complex? In our discussion of functions in Sect. 2.7, we noted the ability to define an ad hoc *anonymous function*. Imagine that we are checking data and wish to know the difference between the mean and median of each variable, perhaps to flag skew in the data. Anonymous function to the rescue! We can `apply` that calculation to multiple columns using an anonymous function:

```
> apply(store.df[,2:9], 2, function(x) { mean(x) - median(x) } )
     Year      Week   p1sales   p2sales   p1price   p2price    p1prom    p2prom
0.0000000 0.0000000 4.0485577 4.1567308 0.0543750 0.1095192 0.1000000 0.1384615
```

This analysis shows that the mean of `p1sales` and the mean of `p2sales` are larger than the median by about four sales per week, which suggests there is a right-hand tail to the distribution. That is, there are some weeks with very high sales that pull the mean up. (Note that we only use this to illustrate an anonymous function; there are better, more specialized tests of skew, such as those in the `psych` package.)

Experienced programmers: your first instinct, based on experience with procedural programming languages, might be to solve the preceding problem with a `for()` loop that iterates the calculation across columns. That is possible in R but less efficient and less "R-like". Instead, try to think in terms of functions that are applied across data as we do here.

There are specialized versions of `apply()` that work similarly with lists and other object types besides data frames. If interested, check `?tapply` and `?lapply`.

All of these functions, including `apply()`, `summary()`, and `describe()` return values that can be assigned to an object. For example, using `apply`, we can produce our customized summary data frame from Sect. 3.2.2 in five lines of code rather than seven:

```
> mysummary2.df <- data.frame(matrix(NA, nrow=2, ncol=2))
> names(mysummary2.df) <- c("Median Sales", "IQR")
> rownames(mysummary2.df) <- names(store.df)[4:5] # names from the data frame
> mysummary2.df[, "Median Sales"] <- apply(store.df[, 4:5], 2, median)
> mysummary2.df[, "IQR"]          <- apply(store.df[, 4:5], 2, IQR)
> mysummary2.df
        Median Sales IQR
p1sales          129  37
p2sales           96  29
```

If there were many products instead of just two, the code would still work if we changed the number of allocated rows, and `apply()` would run automatically across all of them.

Now that we know how to summarize data with statistics, it is time to visualize it.

3.4 Single Variable Visualization

We start by examining plots that are part of the base R system. We examine histograms, density plots, and box plots, and take an initial look at more complex graphics including maps. Later chapters build on these foundational plots and introduce more that are available in other packages. R has many options for graphics including dedicated plotting packages such as `ggplot2` and `lattice`, and specialized plots that are optimized for particular data such as correlation analysis.

3.4.1 Histograms

A fundamental plot for a single continuous variable is the *histogram*. Such a plot can be produced in R with the `hist()` function:

```
> hist(store.df$p1sales)
```

The result, which will appear in the graphical display of base R or RStudio, is shown in Fig. 3.2. It is not a bad start. We see that the weekly sales for product 1 range from a little less than 100 to a bit more than 250. Because axes should always be labeled, R tried to provide reasonable labels based on the variables we passed to `hist()`.

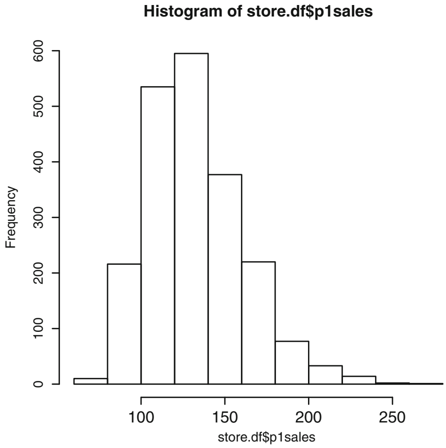

Fig. 3.2. A basic histogram using `hist()`.

That plot was easy to make but the visual elements are less than pleasing, so we will improve it. For future charts, we will show either the basic chart or the final one, and will not demonstrate the successive steps to build one up. However, we go through the intermediate steps here so you can see the process of how to evolve a graphic in R.

As you work through these steps, there are four things you should understand about graphics in R:

- R graphics are produced through commands that often seem tedious and require trial and iteration.
- Always use a text editor when working on plot commands; they rapidly become too long to type, and you will often want to try slight variants and to copy and paste them for reuse.
- Despite the difficulties, R graphics can be very high quality, portable in format, and even beautiful.
- Once you have code for a useful graphic, you can reuse it with new data. It is often helpful to tinker with previous plotting code when building a new plot, rather than recreating it.

Our first improvement to Fig. 3.2 is to change the title and axis labels. We do that by adding arguments to the `hist()` command:

`main="..."` : sets the main title

`xlab="..."` : sets the X axis label

`ylab="..."` : sets the Y axis label

We add the title and axis labels to our plot command:

```
> hist(store.df$p1sales,
+      main="Product 1 Weekly Sales Frequencies, All Stores",
+      xlab="Product 1 Sales (Units)",
+      ylab="Count" )
```

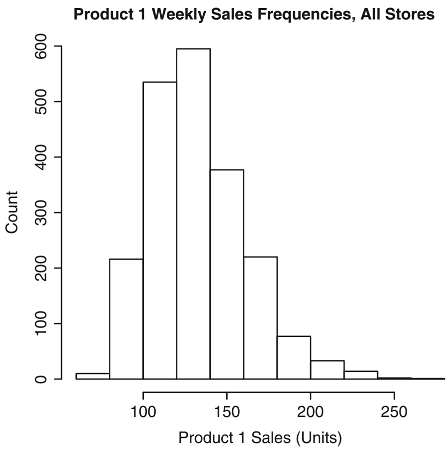

Fig. 3.3. The same histogram, with improved labels.

The result is shown in Fig. 3.3 and is improved but not perfect; it would be nice to have more granularity (more bars) in the histogram. While we're at it, let's add a bit of color. We adjust the graphic by asking for more bins (*breaks*) and color the histogram bars light blue. Here are the arguments involved:

`breaks=NUM` : suggest NUM bars in the result

`col="..."` : color the bars

When specifying colors, R knows many by name, including the most common ones in English ("red", "blue", "green", etc.) and less common (such as "coral" and "burlywood"). Many of these can be modified by adding the prefix "light" or "dark" (thus "lightgray", "darkred", and so forth). For a list of built-in color names, run the `colors()` command.

We add `breaks=` and `col=` arguments to our code, with the result shown in Fig. 3.4:

```
> hist(store.df$p1sales,
+      main="Product 1 Weekly Sales Frequencies, All Stores",
+      xlab="Product 1 Sales (Units)",
+      ylab="Count",
+      breaks=30,                # more columns
+      col="lightblue")          # color the bars
```

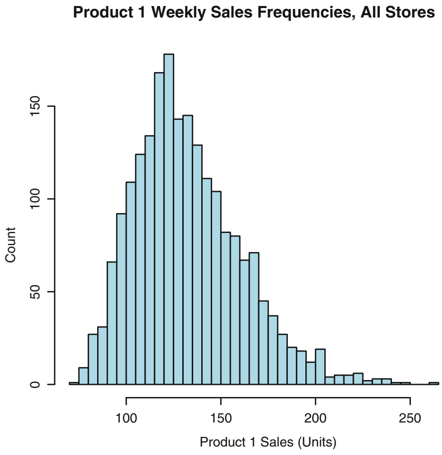

Fig. 3.4. The histogram after adding color and dividing the counts into a larger number of bins (`breaks`).

Comparing Figs. 3.4 with 3.3 we notice a new problem: the y-axis value for the height of the bars changes according to count. The count depends on the number of bins and on the sample size. We can make it absolute by using *relative frequencies* (technically, the *density* estimate) instead of counts for each point. This makes the Y axis comparable across different sized samples.

Figure 3.4 also has ugly and oddly centered numbering on the X axis. Instead of using `hist()`'s default *tick marks* (axis numbers), we remove the axis in order to replace it with one more to our liking. The arguments for relative frequency and removing the X axis are:

`freq=FALSE` : use density instead of counts on Y axis

`xaxt="n"` : X axis text is set to "none"

Now we need to create the replacement axis. This can be done with `axis(side=MARGIN,at=VECTOR)`. Note that `axis()` is a second command and not an argument to `hist()`; `hist()` creates the plot and then `axis()` modifies it.

Here is the amended code. First we call `hist()` to create a new plot without an X axis :

```
> hist(store.df$p1sales,
+      main="Product 1 Weekly Sales Frequencies, All Stores",
+      xlab="Product 1 Sales (Units)",
+      ylab="Relative frequency",
+      breaks=30,
+      col="lightblue",
+      freq=FALSE,                      # freq=FALSE means plot density, not counts
+      xaxt="n")                        # xaxt="n" means "x axis tick marks == no"
```

With `axis()`, we specify which axis to change using an argument: `side=1` alters the X axis, while `side=2` alters the Y axis (the top and right axes are `side=3` and `side=4`, respectively). We have to tell it where to put the labels, and the argument at=VECTOR specifies the new tick marks for the axis. These are easily made with the seq() function to generate a sequence of numbers:

```
> axis(side=1, at=seq(60, 300, by=20))   # add "60", "80", ...
```

The updated histogram is shown in Fig. 3.5. It is looking good now!

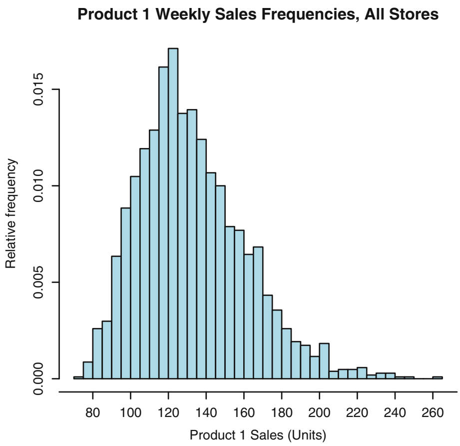

Fig. 3.5. Histogram with relative frequencies (density estimates) and improved axis tick mark labels.

Finally, we add a smoothed estimation line. To do this, we use the density() function to estimate density values for the p1sales vector, and add those to the chart with the lines() command. The lines() command adds elements to the current plot in the same way we saw above for the axis command.

```
> lines(density(store.df$p1sales, bw=10),    # "bw= ..." adjusts the smoothing
+       type="l", col="darkred", lwd=2)      # lwd = line width
```

Figure 3.6 is now very informative. Even someone who is unfamiliar with the data can easily tell that this plot describes weekly sales for product 1 and that the typical sales range from about 80 to 200.

The process we have shown to produce this graphic is representative of how analysts use R for visualization. You start with a default plot, change some of the options, and use functions like axis() and density() to alter features of the plot with complete control. Although at first this will seem cumbersome compared to the drag-and-drop methods of other visualization tools, it really isn't much more time consuming if you use a code editor and become familiar with the plotting functions' examples and help files. It has the great advantage that once you've written the code, you can reuse it with different data.

Exercise: modify the code to create the same histogram for product 2. It requires only minor change to the code whereas with a drag-and-drop tool, you would start all over. If you produce a plot often, you could even write it as a custom function.

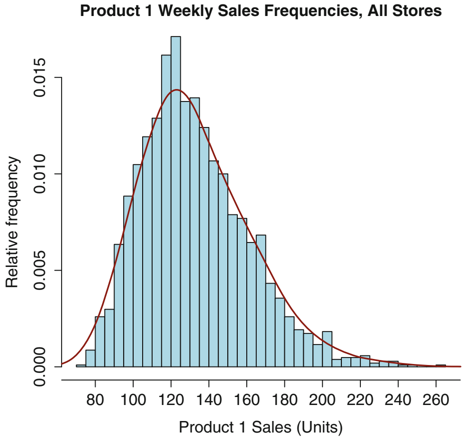

Fig. 3.6. Final histogram with density curve.

3.4.2 Boxplots

Boxplots are a compact way to represent a distribution. The R `boxplot()` command is straightforward; we add labels and use the option `horizontal=TRUE` to rotate the plot 90° to look better:

```
> boxplot(store.df$p2sales, xlab="Weekly sales", ylab="P2",
          main="Weekly sales of P2, All stores", horizontal=TRUE)
```

Figure 3.7 shows the resulting graphic. The boxplot presents the distribution more compactly than a histogram. The median is the center line while the 25th and 75th percentiles define the *box*. The outer lines are *whiskers* at the points of the most extreme values that are no more than 1.5 times the width of the box away from the box. Points beyond the whiskers are outliers drawn as individual circles. This is also known as a *Tukey boxplot* (after the statistician, Tukey) or as a *box-and-whiskers* plot.

Fig. 3.7. A simple example of `boxplot()`.

Boxplots are even more useful when you compare distributions by some other factor. How do different stores compare on sales of product 2? The `boxplot()` command makes it easy to compare these by specifying a response *formula* using *tilde notation*, where the tilde ("∼") separates the *response variable* (sometimes called a *dependent* variable) from the *explanatory variable* (sometimes rather misleadingly called an *independent variable*). In this case, our response variable is `p2sales` and we want to plot it with regard to the explanatory variable `storeNum`. This may be easiest to understand with the R code:

```
> boxplot(store.df$p2sales ~ store.df$storeNum, horizontal=TRUE,
+         ylab="Store", xlab="Weekly unit sales", las=1,
+         main="Weekly Sales of P2 by Store")
```

The first portion of the command may be read as "boxplot p2sales by Store." Formulas like this are pervasive in R and are used both for plotting and for estimating models. We discuss formulas in detail in Sect. 5.2.1 and Chap. 7.

We added one other argument to the plot: `las=1`. That forces the axes to have text in the horizontal direction, making the store numbers more readable. The result is Fig. 3.8, where stores are roughly similar in sales of product 2 (this is not a statistical test of difference, just a visualization).

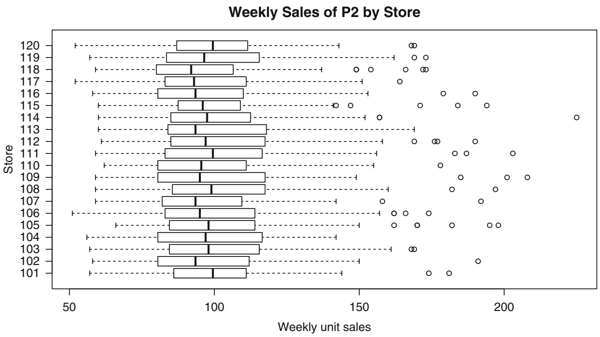

Fig. 3.8. `boxplot()` of sales by store.

We see in Fig. 3.8 that the stores are similar in unit sales of P2, but do P2 sales differ in relation to in-store *promotion*? In this case, our explanatory variable would be the promotion variable for P2, so we use `boxplot()` with the response formula again, replacing `storeNum` with the promotion variable `p2prom`.

This is a good time to introduce two shortcut commands that make life easier. Many commands for statistics and plotting understand the `data=DATAFRAME` argument, and will use variables from `data` without specifying the full name of the data frame. This makes it easy to repeat analyses on different data sets that include the same variables. All you have to do is change the argument for `data=`.

```
> boxplot(p2sales ~ p2prom, data=store.df, horizontal=TRUE, yaxt="n",
+         ylab="P2 promoted in store?", xlab="Weekly sales",
+         main="Weekly sales of P2 with and without promotion")
> axis(side=2, at=c(1,2), labels=c("No", "Yes"))
```

In this plot we also used `axis()` to replace the default Y axis with one that is more informative.The result is shown in Fig. 3.9. There is a clear visual difference in sales on the basis of in-store promotion!

To wrap up: boxplots are powerful tools to visualize a distribution and make it easy to explore how an outcome variable is related to another factor. In Chaps. 4 and 5 we explore many more ways to examine data association and statistical tests of relationships.

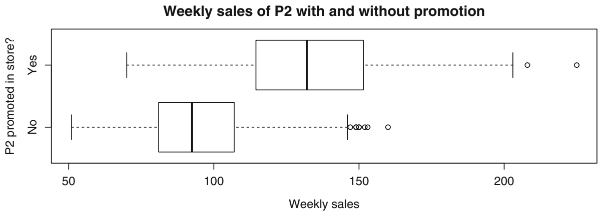

Fig. 3.9. Boxplot of product sales by promotion status.

3.4.3 QQ Plot to Check Normality*

This is an optional section on a graphical method to evaluate a distribution more formally. You may wish to skip to Sect. 3.4.4 on cumulative distributions or Sect. 3.4.5 that describes how to compute aggregate values in R.

Quantile–quantile (QQ) plots are a good way to check one's data against a distribution that you think it should come from. Some common statistics such as the correlation coefficient *r* (to be precise, the *Pearson product-moment correlation coefficient*) are interpreted under an assumption that data are normally distributed. A QQ plot can confirm that the distribution is, in fact, normal by plotting the *observed* quantiles of your data against the quantiles that would be *expected* for a normal distribution.

To do this, the `qqnorm()` command compares data vs. a normal distribution; you can use `qqline()` to add a diagonal line for easier reading. We check `p1sales` to see whether it is normally distributed:

```
> qqnorm(store.df$p1sales)
> qqline(store.df$p1sales)
```

The QQ plot is shown in Fig. 3.10. The distribution of `p1sales` is far from the line at the ends, suggesting that the data is not normally distributed. The upward curving shape is typical of data with high positive skew.

What should you do in this case? If you are using models or statistical functions that assume normally distributed data, you might wish to transform your data. As we've already noted, a common pattern in marketing data is a logarithmic distribution. We examine whether `p1sales` is more approximately normal after a `log()` transform:

```
> qqnorm(log(store.df$p1sales))
> qqline(log(store.df$p1sales))
```

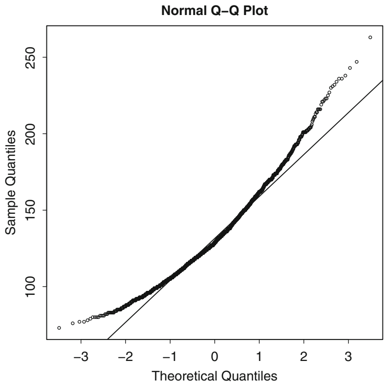

Fig. 3.10. QQ plot to check distribution. The tails of the distribution bow away from the line that represents an exact normal distribution, showing that the distribution of `p1sales` is skewed.

The QQ plot for `log(p1sales)` is shown in Fig. 3.11. The points are much closer to the solid line, indicating that the distribution of `log(store.df$p1sales)` is more consistent with the normal distribution than the untransformed variable.

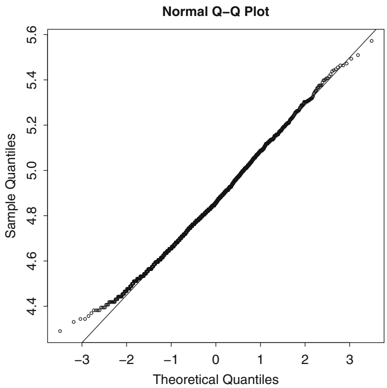

Fig. 3.11. QQ plot for the data after `log()` transformation. The sales figures are now much better aligned with the *solid line* that represents an exact normal distribution.

We recommend that you use `qqnorm()` (and the more general `qqplot()` command) regularly to test assumptions about your data's distribution. Web search will reveal further examples of common patterns that appear in QQ plots and how to interpret them.

3.4.4 Cumulative Distribution*

This is another optional section, but one that can be quite useful. If you wish to skip ahead to cover just the fundamentals, you should continue with Sect. 3.4.5.

Another useful univariate plot involves the impressively named *empirical cumulative distribution function* (ECDF). It is less complex than it sounds and is simply a plot that shows the cumulative proportion of data values in your sample. This is an easy way to inspect a distribution and to read off percentile values.

Before that we should explain an important thing to know about the R `plot()` command: `plot()` can make only a few plot types on its own and otherwise must be given an *object* that includes more information such as *X* and *Y* values. Many R functions produce objects automatically that are suitable as input for `plot()`. A typical pattern looks like this:

```
> my.object <- FUNCTION(my.data)      # not real code
> plot(my.object)
```

...or combined into a single line as:

```
> plot(FUNCTION(my.data))              # not real code
```

We plot the ECDF of `p1sales` by combining a few steps. First, we use the `ecdf()` function to find the ECDF of the data. Then we wrap `plot()` around that, adding options such as titles. Next we put some nicer-looking labels on the Y axis that relabel the proportions as percentiles. The `paste()` function combines a number vector (0, 10, 20, ...) with the "%" symbol to make each label.

Suppose we also want to know where we should expect 90 % of sales figures to occur, i.e., the 90th percentile for weekly sales of P1. We can use the function `abline()` to add vertical and horizontal lines at the 90th percentile. We do not have to tell R the exact value at which to draw a line for the 90th percentile; instead, we use `quantile( , pr=0.9)` to find it:

```
> plot(ecdf(store.df$p1sales),
+      main="Cumulative distribution of P1 Weekly Sales",
+      ylab="Cumulative Proportion",
+      xlab=c("P1 weekly sales, all stores", "90% of weeks sold <= 171 units"),
+      yaxt="n")
> axis(side=2, at=seq(0, 1, by=0.1), las=1,
+      labels=paste(seq(0,100,by=10), "%", sep=""))
> abline(h=0.9, lty=3)            # "h=" for horizontal line; "lty=3" for dotted
> abline(v=quantile(store.df$p1sales, pr=0.9), lty=3)   # "v=" for vertical line
```

The resulting plot is shown in Fig. 3.12. We often use cumulative distribution plots both for data exploration and for presenting data to others. They are a good way to highlight data features such as discontinuities in the data, long tails, and specific points of interest.

3.4.5 Language Brief: `by()` and `aggregate()`

What should we do if we want to break out data by factors and summarize it, a process you might know as "cross-tabs" or "pivot tables"? For example, how can we compute the mean sales by store? We have voluminous data (every store by every week by each product) but many marketing purposes only need an aggregate figure such as a total or mean. We saw in Sect. 3.3.4 how to summarize data with

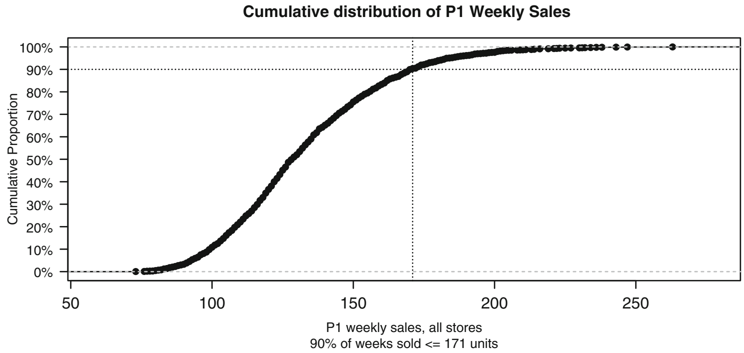

Fig. 3.12. Cumulative distribution plot with lines to emphasize the 90th percentile. The chart identifies that 90 % of weekly sales are lower than or equal to 171 units. Other values are easy to read off the chart. For instance, roughly 10 % of weeks sell less than 100 units, and fewer than 5 % sell more than 200 units.

various statistics and plots, and to summarize across columns with the `apply()` function. Now we will see how to summarize by a factor within the data itself using the commands `by()` and `aggregate()`.

Let's look first at `by(data=DATA, INDICES=INDICES, FUN=FUNCTION)`. `by()` uses `INDICES` as grouping factors to divide `DATA` into subgroups. Then it applies the function `FUN` to each subgroup.

This is easier to understand in the context of an example. Suppose we wish to find the average sales of P1 by store. The `DATA` would be the weekly sales for each store, `store.df$p1sales`. We wish to split this by store, so the `INDICES` (actually, "index" in this case) would be `store.df$storeNum`. Finally, we get the average of each of those groups by using the `mean` function. Here is the complete command to break out mean sales of P1 by store:

```
> by(store.df$p1sales, store.df$storeNum, mean)
store.df$storeNum: 101
[1] 130.5385
------------------------------------------------
store.df$storeNum: 102
[1] 134.7404
...
```

To group it by more than one factor, use a `list()` of factors. For instance, we can obtain the mean of `p1sales` by store and by year:

```
> by(store.df$p1sales, list(store.df$storeNum, store.df$Year), mean)
: 101
: 1
[1] 127.7885
---------------------------------------------------------------
: 102
```

```
: 1
[1] 129.7115
...
```

A limitation of `by()` is that the result is easy to read but not structured for reuse. How can we save the results as data to use for other purposes such as plotting?

The answer is `aggregate()` which operates almost identically to `by()` but returns a nicely formatted data frame. The following computes the total (`sum()`) sales of P1 by country:

```
> aggregate(store.df$p1sales, by=list(country=store.df$country), sum)
  country     x
1      AU 14544
2      BR 27836
3      CN 27381
4      DE 68876
5      GB 40986
6      JP 55381
7      US 41737
```

How does this work? Just as with `by()`, `aggregate(x=DATA, by=BY, FUN=FUNCTION)` applies a particular function (`FUN`) according to divisions of the data specified by a factor (`by`). We want to find the total sales by country, so we apply the `mean` function by `store.df$country`.

If we want to save the result as a new data frame, we simply assign it somewhere—as we do now because we will use it in Sect. 3.4.6 to make a map:

```
> p1sales.sum <- aggregate(store.df$p1sales,
+                          by=list(country=store.df$country), sum)

> p1sales.sum
  country     x
1      AU 14544
2      BR 27836
3      CN 27381
...
```

`aggregate()` gave us a nicely structured data frame with our summary. We will see further options for `aggregate()` in Sect. 5.2.1.

3.4.6 Maps

We often need to plot marketing data on a map. A common variety is a *choropleth* map, which uses graphics or color to indicate values of a variable such as income or sales. We consider how to do this for a world map using the `rworldmap` package [146].

Here is a routine example. Suppose that we want to chart the total sales by country. We use `aggregate()` as in Sect. 3.4.5 to find the total sales of P1 by country:

```
p1sales.sum <- aggregate(store.df$p1sales,
                         by=list(country=store.df$country), sum)
```

To make a map, we'll use the `rworldmap` package for plotting routines [146], plus the `RColorBrewer` package [121] to generate some better-looking colors.

```
> install.packages(c("rworldmap", "RColorBrewer"))  # if needed
> library(rworldmap)
> library(RColorBrewer)
```

First, we have to associate the aggregated data with specific map regions using the country codes. This can be done with the `joinCountryData2Map()` function, which matches country locations (`store.df$country`) for data points with the corresponding international standard names (*ISO* names) and returns a map object:

```
> p1sales.map <- joinCountryData2Map(p1sales.sum, joinCode = "ISO2",
                                     nameJoinColumn = "country")
```

Let's inspect that command more closely. The data object that we wish to map is the `p1sales.sum` aggregated data frame. We place that on a map according to the 2-letter country names (`joinCode="ISO2"`) which are present in the data object as the `"country"` column.

Next we draw the resulting map object using `mapCountryData()`, selecting colors from the `RColorBrewer` package "Greens" palette. We plot the column named `x` because that is the default name that the `aggregate()` function gives in the aggregated data fame:

```
> mapCountryData(p1sales.map, nameColumnToPlot="x",
+                mapTitle="Total P1 sales by Country",
+                colourPalette=brewer.pal(7, "Greens"),
+                catMethod="fixedWidth", addLegend=FALSE)
```

The result is shown in Fig. 3.13, known as a *choropleth* chart.

Although such maps are popular, they can be misleading. In *The Wall Street Journal Guide to Information Graphics*, Wong explains that choropleth charts are problematic because they confuse geographic area with scaled quantities [168, p. 90]. For instance, in Fig. 3.13, China is more prominent than Japan not because it has a higher value but because it is larger in size. We acknowledge the need for caution despite the popularity of such maps.

For more complex charts, there are options in `?rworldmap` for drawing regional maps, more granular areas, setting color palettes, using locations other than country codes, and so forth. For other mapping options, see the suggestions in Sect. 3.5 below.

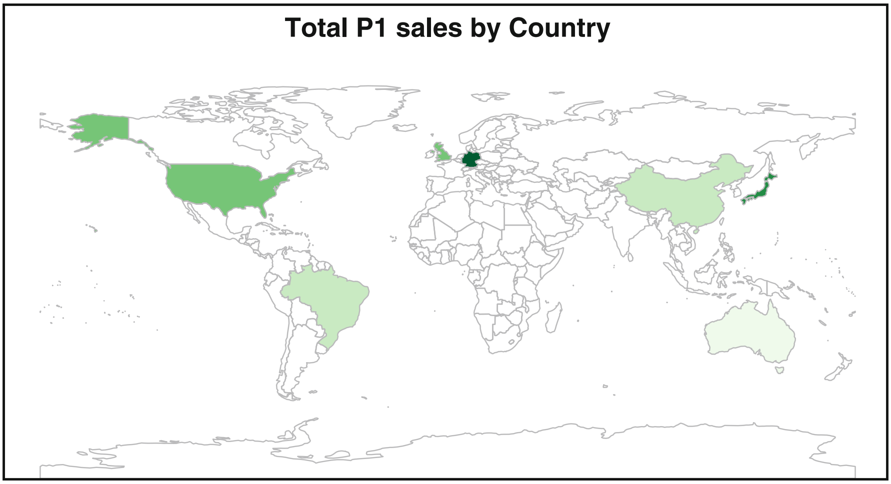

Fig. 3.13. World map for P1 sales by country, using `rworldmap`.

3.5 Learning More*

Plotting. We demonstrate plotting in R throughout this book. R has multiple, often disjoint solutions for plotting and in this text we use plots as appropriate without going deeply into their details. The *base* plotting system comes standard in R and appears in commands such as `hist()` and `plot()`.

Two popular and powerful packages that produce more complex graphics are `lattice` [141] and `ggplot2` [162]. The choice between `lattice` and `ggplot2` is largely a matter of personal preference and style. We sometimes suspect that `lattice` appeals more to scientists and engineers while `ggplot2` appeals to computer scientists and social scientists. Chang's *R Graphics Cookbook* [24] is a single volume overview of many kinds of plots available in R, focused on the `ggplot2` package.

Wong's *The Wall Street Journal Guide to Information Graphics* [168] presents fundamentals of good style for effective graphics in any business context (not specific to R).

Maps. Producing maps in R is an especially complex topic. Maps require three essential components: *shape files* that define the borders of areas (such as country or city boundaries); *spatial translation* of one's data (for instance, a database to match Zip codes in your data to the relevant areas on a map); and *plotting software* to perform the actual plotting. R packages such as `rworldmap` usually provide access to all three of those elements.

As of this writing, the landscape of available packages and tools for mapping in R was changing rapidly. We use the `rworldmap` package here for its simplicity.

For more complex tasks, the `ggplot2` package [162] serves as the basis for a sophisticated mapping tool, the `ggmap` package [90].

3.6 Key Points

The following guidelines and pointers will help you to describe data accurately and quickly:

- Consider simulating data before collecting it, in order to test your assumptions and develop initial analysis code (Sect. 3.1).
- Always check your data for proper structure and data quality using `str()`, `head()`, `summary()`, and other basic inspection commands (Sect. 3.3.3).
- Describe discrete (categorical) data with `table()` (Sect. 3.2.1) and inspect continuous data with `describe()` from the `psych` package (Sect. 3.3.2).
- Histograms (Sect. 3.4.1) and boxplots (Sect. 3.4.2) are good for initial data visualization.
- Use `by()` and `aggregate()` to break out your data by grouping variables (Sect. 3.4.5).
- Advanced visualization methods include cumulative distribution (Sect. 3.4.4), normality checks (Sect. 3.4.3), and mapping (Sect. 3.4.6).