4

Relationships Between Continuous Variables

Experienced analysts understand that the most important insights in marketing analysis often come from understanding relationships between variables. While it is helpful to understand single variables, such as how many products are sold at a store, more valuable insight emerges when we understand relationships such as "Customers who live closer to our store visit more often than those who live farther away," or "Customers of our online shop buy as much in person at the retail shop as do customers who do not purchase online."

Identifying these kinds of relationships helps marketers to understand how to reach customers more effectively. For example, if people who live closer to a store visit more frequently and buy more, then an obvious strategy would be to send advertisements to people who live in the area.

In this chapter we focus on understanding the relationships between pairs of variables in multivariate data, and examine how to visualize the relationships and compute statistics that describe their associations (correlation coefficients). These are the most important ways to assess relationships between continuous variables. While it might seem appealing to go straight into building regression models (see Chap. 7), we caution against that. The first step in any analysis is to explore the data and its basic properties. This chapter continues the data exploration and visualization process that we reviewed for single variables in Chap. 3. It often saves time and heartache to begin by examining the relationships among pairs of variables before building more complex models.

4.1 Retailer Data

We simulate a data set that describes customers of a multi-channel retailer and their transactions for 1 year. This data includes a subset of customers for whom we have survey data on product satisfaction.

C. Chapman, E.M. Feit, *R for Marketing Research and Analytics*, Use R!,
DOI 10.1007/978-3-319-14436-8_4

As in Chap. 3, we present the code that generates this data as a way to teach more about R syntax. However, if you prefer to jump right into the analysis, you could quickly run all the commands in Sect. 4.1.1 and then continue with Sect. 4.2 where we begin plotting the data.

Alternatively, the following will load the data from this book's website:

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

However, you will learn more about R if you work through the simulation code instead of downloading the data.

4.1.1 Simulating Customer Data

In this section, we create a data set for 1,000 customers of a retailer who sells products in stores and online. This data is typical of what one might sample from a company's customer relationship management (CRM) system. We begin by setting a random number seed to make the process repeatable (as described in Sect. 3.1.2) and creating a data frame to store the data:

```
> set.seed(21821)
> ncust <- 1000
> cust.df <- data.frame(cust.id=as.factor(c(1:ncust)))
```

We declare a variable `ncust` for the number of customers in the synthetic data set and use that variable wherever we need to refer to the number of customers. This is a good practice, as it allows you to change `ncust` in just one place in your code and then re-run the code to generate a new data set with a different number of customers.

Next we create a number of variables describing the customers, add those variables to the `cust.df` data frame, and inspect them with `summary()`:

```
> cust.df$age <- rnorm(n=ncust, mean=35, sd=5)
> cust.df$credit.score <- rnorm(n=ncust, mean=3*cust.df$age+620, sd=50)
> cust.df$email <- factor(sample(c("yes", "no"), size=ncust, replace=TRUE,
+                                prob=c(0.8, 0.2)))
> cust.df$distance.to.store <- exp(rnorm(n=ncust, mean=2, sd=1.2))

> summary(cust.df)
    cust.id          age         credit.score   email     distance.to.store
 1      :  1   Min.   :19.34   Min.   :543.0   no :186   Min.   :  0.2136
 2      :  1   1st Qu.:31.43   1st Qu.:691.7   yes:814   1st Qu.:  3.3383
 3      :  1   Median :35.10   Median :725.5             Median :  7.1317
 4      :  1   Mean   :34.92   Mean   :725.5             Mean   : 14.6553
 5      :  1   3rd Qu.:38.20   3rd Qu.:757.2             3rd Qu.: 16.6589
 6      :  1   Max.   :51.86   Max.   :880.8             Max.   :267.0864
 (Other):994
```

We add new variables to `cust.df` data frame using simple assignment (`<-`) to a name with $ notation. Columns in data frames can be easily created or replaced in this way, as long as the vector has the appropriate length (or is recycled to fit the length).

The customers' ages (`age`) are drawn from a normal distribution with mean 35 and standard deviation 5 using `rnorm(n, mean, sd)`. Credit scores (`credit.score`) are also simulated with a normal distribution, but in that case we specify that the mean of the distribution is related to the customer's age, with older customers having higher credit scores on average. We create a variable (`email`) indicating whether the customer has an email on file, using the `sample` function that was covered in Chap. 3.

Our final variable for the basic CRM data is `distance.to.store`, which we assume follows the exponential of the normal distribution. That gives distances that are all positive, with many distances that are relatively close to the nearest store and fewer that are far from a store. To see the distribution for yourself, try `hist(cust.df$distance.to.store)`. Formally, we say that `distance.to.store` follows a *lognormal* distribution. (This is sufficiently common that there is a built-in function called `rlnorm(n, meanlog, sdlog)` that does the same thing as taking the exponential of `rnorm()`.)

4.1.2 Simulating Online and In-Store Sales Data

Our next step is to create data for the online store: 1 year totals for each customer for online visits and transactions, plus total spending. We simulate the number of visits with a *negative binomial* distribution, a discrete distribution often used to model counts of events over time. Like the lognormal distribution, the negative binomial distribution generates positive values and has a long right-hand tail, meaning that in our data most customers make relatively few visits and a few customers make many visits. Data from the negative binomial distribution can be generated using `rnbinom()`:

```
> cust.df$online.visits <- rnbinom(ncust, size=0.3,
+                                  mu = 15 + ifelse(cust.df$email=="yes", 15, 0)
+                                  - 0.7 * (cust.df$age-median(cust.df$age)))
```

We model the mean (`mu`) of the negative binomial with a baseline value of 15. The `size` argument sets the degree of dispersion (variation) for the samples. We add an average 15 online visits for customers who have an email on file, using `ifelse()` to generate a vector of 0 or 15 as appropriate. Finally, we add or subtract visits from the target mean based on the customer's age relative to the sample median; customers who are younger are simulated to make more online visits. To see exactly how this works, try cutting and pasting pieces of the code above into the R console.

For each online visit that a customer makes, we assume there is a 30 % chance of placing an order and use `rbinom()` to create the variable `online.trans`. We assume that amounts spent in those orders (the variable `online.spend`) are lognormally distributed:

```
> cust.df$online.trans <- rbinom(ncust, size=cust.df$online.visits, prob=0.3)
> cust.df$online.spend <- exp(rnorm(ncust, mean=3, sd=0.1)) *
+                         cust.df$online.trans
```

The random value for amount spent per transaction—sampled with `exp(rnorm())` is multiplied by the variable for number of transactions to get the total amount spent.

Next we generate in-store sales data similarly, except that we don't generate a count of store visits; few customers visit a physical store without making a purchase and even if customers did visit without buying, the company probably couldn't track the visit. We assume that transactions follow a negative binomial distribution, with lower average numbers of visits for customers who live farther away. We model in-store spending as a lognormally distributed variable simply multiplied by the number of transactions:

```
> cust.df$store.trans <- rnbinom(ncust, size=5,
+                                mu=3 / sqrt(cust.df$distance.to.store))
> cust.df$store.spend <- exp(rnorm(ncust, mean=3.5, sd=0.4)) *
+                        cust.df$store.trans
```

As always, we check the data along the way:

```
> summary(cust.df)
    cust.id          age          credit.score   email      distance.to.store
 1      :  1   Min.   :19.34   Min.   :543.0   no :186   Min.   :  0.2136
 2      :  1   1st Qu.:31.43   1st Qu.:691.7   yes:814   1st Qu.:  3.3383
...
  online.spend      store.trans      store.spend
 Min.   :   0.00   Min.   : 0.000   Min.   :  0.00
 1st Qu.:   0.00   1st Qu.: 0.000   1st Qu.:  0.00
 Median :  37.03   Median : 1.000   Median : 30.05
...
```

4.1.3 Simulating Satisfaction Survey Responses

It is common for retailers to survey their customers and record responses in the CRM system. Our last simulation step is to create survey data for a subset of the customers.

To simulate survey responses, we assume that each customer has an unobserved overall satisfaction with the brand. We generate this overall satisfaction from a normal distribution:

```
> sat.overall <- rnorm(ncust, mean=3.1, sd=0.7)
> summary(sat.overall)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.617   2.632   3.087   3.100   3.569   5.293
```

We assume that overall satisfaction is a psychological construct that is not directly observable. Instead, the survey collects information on two items: satisfaction with service, and satisfaction with the selection of products. We assume that customers' responses to the survey items are based on unobserved levels of satisfaction *overall* (sometimes called the "halo" in survey response) plus the specific levels of satisfaction with the service and product selection.

To create such a score from a halo variable, we add `sat.overall` (the halo) to a random value specific to the item, drawn using `rnorm()`. Because survey responses are typically given on a discrete, ordinal scale (i.e., "very unsatisfied", "unsatisfied", etc.), we convert our continuous random values to discrete integers using the `floor()` function.

```
> sat.service <- floor(sat.overall + rnorm(ncust, mean=0.5, sd=0.4))
> sat.selection <- floor(sat.overall + rnorm(ncust, mean=-0.2, sd=0.6))
> summary(cbind(sat.service, sat.selection))
  sat.service    sat.selection
 Min.   :0.000   Min.   :-1.000
 1st Qu.:3.000   1st Qu.: 2.000
...
 Max.   :6.000   Max.   : 5.000
```

Note that we use `cbind()` to temporarily combine our two vectors of data into a matrix, so that we can get a combined summary with a single line of code. The summary shows that our data now ranges from -1 to 6. However, a typical satisfaction item might be given on a 5-point scale. To fit that, we replace values that are greater than 5 with 5, and values that are less than 1 with 1. This enforces the *floor* and *ceiling* effects often noted in survey response literature.

We set the ceiling by indexing with a vector that tests whether each element of `sat.service` is greater than 5): `sat.service[sat.service > 5]`. This might be read as "sat.service, where sat.service is greater than 5." For the elements that are selected—which means that the expression evaluates as `TRUE`—we replace the current values with the ceiling value of 5. We do the same for the floor effects (`< 1`, replacing with 1) and likewise for the ceiling and floor of `sat.selection`. While this sounds quite complicated, the code is simple:

```
> sat.service[sat.service > 5] <- 5
> sat.service[sat.service < 1] <- 1
> sat.selection[sat.selection > 5] <- 5
> sat.selection[sat.selection < 1] <- 1
> summary(cbind(sat.service, sat.selection))
  sat.service    sat.selection
 Min.   :1.000   Min.   :1.000
...
 Max.   :5.000   Max.   :5.000
```

Using this type of syntax to replace values in a vector or matrix is common in R, and we recommend that you try out some variations (being careful not to overwrite the `cust.df` data, of course).

4.1.4 Simulating Non-Response Data

Because some customers do not respond to surveys, we eliminate the simulated answers for a subset of respondents who are modeled as not answering. We do this by creating a variable of `TRUE` and `FALSE` values called `no.response` and then assigning a value of `NA` for the survey response for customers whose `no.response` is `TRUE`. As we have discussed, `NA` is R's built-in constant for missing data.

We model non-response as a function of age, with higher likelihood of not responding to the survey for older customers:

```
> no.response <- as.logical(rbinom(ncust, size=1, prob=cust.df$age/100))
> sat.service[no.response] <- NA
> sat.selection[no.response] <- NA
> summary(cbind(sat.service, sat.selection))
  sat.service   sat.selection
 Min.   :1.00   Min.   :1.000
 1st Qu.:3.00   1st Qu.:2.000
 Median :3.00   Median :2.000
 Mean   :3.07   Mean   :2.401
 3rd Qu.:4.00   3rd Qu.:3.000
 Max.   :5.00   Max.   :5.000
 NA's   :341    NA's   :341
```

`summary()` recognizes the 341 customers with `NA` values and excludes them from the statistics.

Finally, we add the survey responses to `cust.df` and clean up the workspace:

```
> cust.df$sat.service <- sat.service
> cust.df$sat.selection <- sat.selection
> summary(cust.df)
    cust.id         age          credit.score   email     distance.to.store
 1      :  1   Min.   :19.34   Min.   :543.0   no :186   Min.   :  0.2136
 2      :  1   1st Qu.:31.43   1st Qu.:691.7   yes:814   1st Qu.:  3.3383
 ...
   store.spend      sat.service    sat.selection
 Min.   :  0.00   Min.   :1.000   Min.   :1.000
...
 Max.   :705.66   Max.   :5.000   Max.   :5.000
                  NA's   :341     NA's   :341

> rm(ncust, sat.overall, sat.service, sat.selection, no.response)
```

The data set is now complete and ready for analysis.

4.2 Exploring Associations Between Variables with Scatterplots

Our analysis begins by checking the data with `str()` to review its structure:

```
> str(cust.df)
'data.frame': 1000 obs. of  12 variables:
 $ cust.id          : Factor w/ 1000 levels "1","2","3","4",..: 1 2 3 ...
 $ age              : num  22.9 28 35.9 30.5 38.7 ...
 $ credit.score     : num  631 749 733 830 734 ...
 $ email            : Factor w/ 2 levels "no","yes": 2 2 2 2 1 2 2 2 1 1 ...
 $ distance.to.store: num  2.58 48.18 1.29 5.25 25.04 ...
 $ online.visits    : num  20 121 39 1 35 1 1 48 0 14 ...
 $ online.trans     : int  3 39 14 0 11 1 1 13 0 6 ...
 $ online.spend     : num  58.4 756.9 250.3 0 204.7 ...
 $ store.trans      : num  4 0 0 2 0 0 2 4 0 3 ...
 $ store.spend      : num  140.3 0 0 95.9 0 ...
 $ sat.service      : num  3 3 NA 4 1 NA 3 2 4 3 ...
 $ sat.selection    : num  3 3 NA 2 1 NA 3 3 2 2 ...
```

As we noted above, in this data frame each row represents a different customer. For each, there is a flag indicating whether the customer has an email address on file (`email`), along with the customer's `age`, `credit.score`, and distance to the nearest physical store (`distance.to.store`).

Additional variables report 1-year total visits to the online site (`online.visits`) as well as online and in-store transaction counts (`online.trans` and `store.trans`) plus 1-year total spending online and in store (`online.spend` and `store.spend`). Finally, the data contains survey ratings of satisfaction with the service and product selection at the retail stores (`sat.service` and `sat.selection`). Some of the survey values are `NA` for customers without survey responses. All values are numeric, except that `cust.df$cust.id` and `cust.df$email` are factors (categorical). We'll say more shortly about why the details of the data structure are so important.

4.2.1 Creating a Basic Scatterplot with `plot()`

We begin by exploring the relationship between each customer's age and credit score using `plot(x, y)`, where `x` is the x-coordinate vector for the points and `y` is the y-coordinate vector:

```
> plot(x=cust.df$age, y=cust.df$credit.score)
```

The code above produces the graphic shown in the left panel of Fig. 4.1, a fairly typical scatterplot. There is a large mass of customers in the center of the plot with age around 35 and credit score around 725, and fewer customers at the margins. There are not many younger customers with very high credit scores, nor older customers with very low scores, which suggests an association between age and credit score.

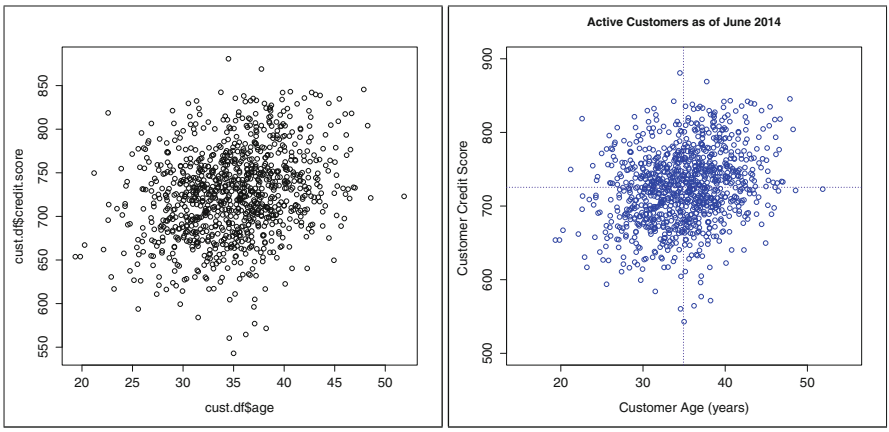

Fig. 4.1. Basic scatterplot of customer age versus credit score using default settings in `plot()` function (*left*), and a properly labeled version of the same plot (*right*).

The default settings in `plot()` produce a quick plot that is useful when you are exploring the data for yourself; `plot()` adjusts the x- and y-axes to accommodate the range of the data and labels the axes using variable names. But if we present the plot to others, we ought to provide more informative labels for the axes and chart title:

```
> plot(cust.df$age, cust.df$credit.score,
+      col="blue",
+      xlim=c(15, 55), ylim=c(500, 900),
+      main="Active Customers as of June 2014",
+      xlab="Customer Age (years)", ylab="Customer Credit Score ")
> abline(h=mean(cust.df$credit.score), col="dark blue", lty="dotted")
> abline(v=mean(cust.df$age), col="dark blue", lty="dotted")
```

We do not specifically name `x=` and `y=` here because, when names of arguments are omitted, a function such as `plot()` assumes that they line up in order as listed in a function's definition (and shown in help). We use the argument `col` to color the points blue. `xlim` and `ylim` set a range for each axis. `main`, `xlab`, and `ylab` provide a descriptive title and axis labels for the chart. The result on the right side of Fig. 4.1 is labeled well enough that someone viewing the chart can easily understand what it depicts.

After creating the plot, we use `abline()` to add lines to the plot, to indicate the average age and average credit score in the data. We add a horizontal line at `mean(cust.df$credit.score)` using `abline(h=)`, and a vertical line at the mean `age` with `abline(v=)`.

Often, plots are built up using a series of commands like this. The first step is to use `plot()` to set up the basic graphics; then add features with other graphics commands. Some of the most useful functions are `points()` to add specific points, `abline()` to add a line by slope and intercept, `lines()` to add a set of lines by

coordinates, and `legend()` to add a legend (see Sect. 4.2.3). Each of these adds elements to a plot that has already been created using `plot()`.

Before we move on, we should make an important note about how the `plot()` command works in R. When you type `plot()` into the console, R looks at what type of data you are trying to plot and, based on the data type, R will choose a specific lower-level plotting function, known as a *method*, that is appropriate to the data you are trying to plot. When we call `plot()` with vectors of x and y coordinates, R uses the `plot.default()` function. However, there are many other plotting functions for different data types. For example, if you plot the `cust.df` data frame by typing `plot(cust.df)` into the console, R will use `plot.data.frame()` instead of `plot.default()`. This produces one of several plot types depending on the number of dimensions in the data frame; in this case, it produces a scatterplot matrix, which we review in Sect. 4.4.2.

While this may seem like an obtuse detail of the language, it is important to general R users for two reasons. First, help files for generic functions like `plot()` and `summary()` may be rather *un*helpful because they describe the generic methods; often you need to navigate to the help file for the specific method that you are using. For instance, to learn more about the plotting function we are using in this chapter, you should type `?plot.default` into the console.

Second, when `plot()` produces something unexpected, it may be because R has selected a different method than you expect. If so, check the data types of the variables you're sending to `plot()` because R uses those to select a plot method. Despite this complexity, generic functions are convenient because you only have to remember one function name such as `plot()` instead of many. When you need to figure out more, you can check the methods available for `plot()`, depending on the packages you are using, by typing `methods(plot)`.

We next turn to an important marketing question: in our data, do customers who buy more online buy less in stores? We start by plotting online sales against in-store sales:

```
> plot(cust.df$store.spend, cust.df$online.spend,
+      main="Customers as of June 2014",
+      xlab="Prior 12 months in-store sales ($)",
+      ylab="Prior 12 months online sales ($)",
+      cex=0.7)
```

The resulting plot in Fig. 4.2 is typical of the skewed distributions that are common in behavioral data such as sales or transaction counts; most customers purchase rarely so the data is dense near zero. The resulting plot has a lot of points along the axes; we use the `cex` option, which scales down the plotted points to 0.7 of their default size so that we can see the points a bit more clearly. The plot shows that there are a large number of customers who didn't buy anything on one of the two channels (the points along the axes), along with a smaller number of customers who purchase fairly large amounts on one of the channels.

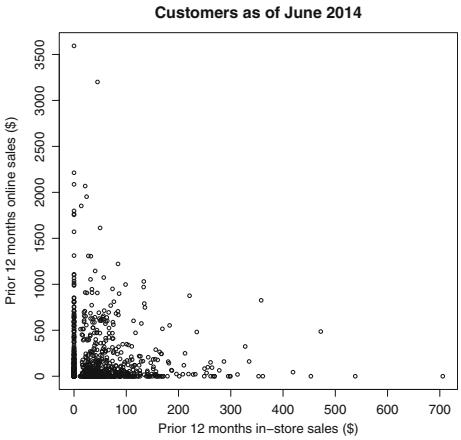

Fig. 4.2. Scatterplot of online sales versus in-store sales for the customers in our data set.

Because of the skewed data, Fig. 4.2 does not yet give a good answer to our question about the relationship between online and in-store sales. We investigate further with a histogram of just the in-store sales (see Sect. 3.4 for `hist()`):

```
 hist(cust.df$store.spend,
+      breaks=(0:ceiling(max(cust.df$store.spend)/10))*10,
+      main="Customers as of June 2014",
+      xlab="Prior 12 months online sales ($)",
+      ylab="Count of customers")
```

The histogram in Fig. 4.3 shows clearly that a large number of customers bought nothing in the online store (about 400 out of 1,000). The distribution of sales among those who do buy has a mode around $20 and a long right-hand tail with a few customers whose 12-month spending was high. Such distributions are typical of spending and transaction counts in customer data.

4.2.2 Color-Coding Points on a Scatterplot

Another question is whether the propensity to buy online versus in store is related to our email efforts (as reflected by whether or not a customer has an email address on file). We can add the `email` dimension to the plot in Fig. 4.2 by coloring in the points for customers whose email address is known to us. To do this, we use `plot()` arguments that allow us to draw different colors (`col=`) and symbols for the points (`pch=`). Each argument takes a vector that specifies the option—the color or symbol—that you want for each individual point. Thus, if we provide a vector of colors of the same length as the vectors of `x` and `y` values, `col=` will use the corresponding colors for each point. Constructing such vectors can be tricky, so we will build them up slowly.

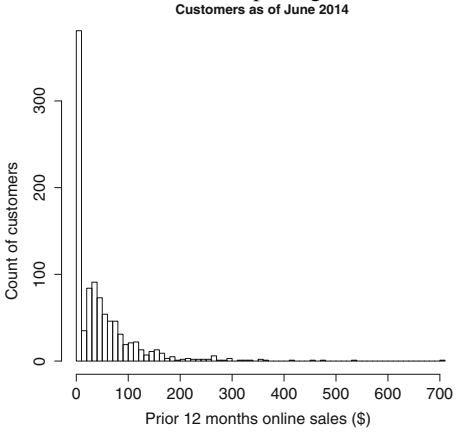

Fig. 4.3. A histogram of prior 12 months online sales reveals more clearly a large number of customers who purchase nothing along with a left-skewed distribution of sales among those who purchase something.

To begin, we first declare vectors for the color and point types that we want to use:

```
> my.col <- c("black", "green3")
> my.pch <- c(1, 19) # R's symbols for solid and open circles (see ?points)
```

We use `green3` as a slightly darker shade of green. It is often helpful to review all the color names in `colors()` to find such options.

With these defined, we can select the appropriate color and plotting symbol for each customer simply by using `cust.df$email` to index them. How does this work? The factor `email` is converted to a numeric value under the hood (1 for `no` and 2 for `yes`) and then that value is used to select colors.

Let's see how that works (using just the `head()` of the data for brevity). First we see that `email` is a factor, which we could coerce to numeric values:

```
> head(cust.df$email)
[1] yes yes yes yes no  yes
Levels: no yes
> as.numeric(head(cust.df$email))
[1] 2 2 2 2 1 2
```

If we use those numbers to index `my.col`, then we get the matching color for each value of `email`:

```
> my.col[as.numeric(head(cust.df$email))]
[1] "green3" "green3" "green3" "green3" "black"  "green3"
```

However, it's tedious (although error-resistant) to write `as.numeric()` all the time, and R understands what we want just by indexing with the factor directly:

```
> my.col[head(cust.df$email)]
[1] "green3" "green3" "green3" "green3" "black"  "green3"
```

Now that we have a vector of colors, we can pass it as the `col` option in `plot()` to get a plot where customers with emails on file are plotted in green and customers without email addresses on file are plotted in black. We use a similar strategy for setting the point styles using the `pch` option, such that customers without email addresses have open circles instead of solid. The complete code is:

```
> plot(cust.df$store.spend, cust.df$online.spend,
+      cex=0.7,
+      col=my.col[cust.df$email], pch=my.pch[cust.df$email],
+      main="Customers as of June 2014",
+      xlab="Prior 12 months in-store sales ($)",
+      ylab="Prior 12 months online sales ($)" )
```

The resulting plot appears in the left panel of Fig. 4.4.

When we created Fig. 4.1 earlier, we used an option `col="blue"` and it turned all of the points blue. This is because if the vector you pass for `col` is shorter than the length of `x` and `y`, then R recycles the values. Thus, if your `col` vector has one element, all the points will be that single color. Similarly, if you were to pass the vector `c("black", "green3")`, then `plot` would simply make alternating points black or green, which might not be what you want. Usually what you'll want is to create a vector that exactly matches the length of your data by starting with a shorter vector as we did here, and then indexing it with `[]` such that you extract a value for each one of your data points. That can be difficult to get right in practice, so we encourage you to experiment with these examples until you understand how it works.

4.2.3 Adding a Legend to a Plot

Given that we've colored some points in our chart, it would be helpful to add a legend that explains the colors. We can do this using `legend()`.

```
> legend(x="topright", legend=paste("email on file:", levels(cust.df$email)),
+        col=my.col, pch=my.pch)
```

The `legend()` function can be frustrating, but the idea is relatively simple. The first input to `legend()` is `x=LOCATION`, which sets the location of the legend on the plot. Then you specify the `legend` argument, which is a vector of labels that you want to include in the legend. In the present case, we use `paste()` to create the labels `"email on file: no"` and `"email on file: yes"` by adding the constant string `"email on file:"` to the factor levels of `email`. Next, you define the markers to associate with those labels in the legend. Because we defined these with `my.col` and `my.pch`, we reuse those here.

Although the code to create the legend is compact, it is a hassle to track the details of labels, colors, and symbols. Our recommendation is to define the argument values in a reusable way as we have done here using definition vectors such as `my.col`

and `my.pch`. An alternative would be to invest in learning a specialized graphics package such as `lattice` or `ggplot2`. Those packages handle legends in more sophisticated ways that we do not explore in depth here (see Sect. 3.5).

4.2.4 Plotting on a Log Scale

With raw values as plotted in the left panel of Fig. 4.4, it is still difficult to see whether there is a different relationship between in-store and online purchases for those with and without emails on file, because of the heavy skew in sales figures. A common solution for such scatterplots with skewed data is to plot the data on a *logarithmic* scale. This is easy to do with the `log=` argument of `plot()`: set `log="x"` to plot the x-axis on the log scale, `log="y"` for the y-axis, or `log="xy"` for both axes.

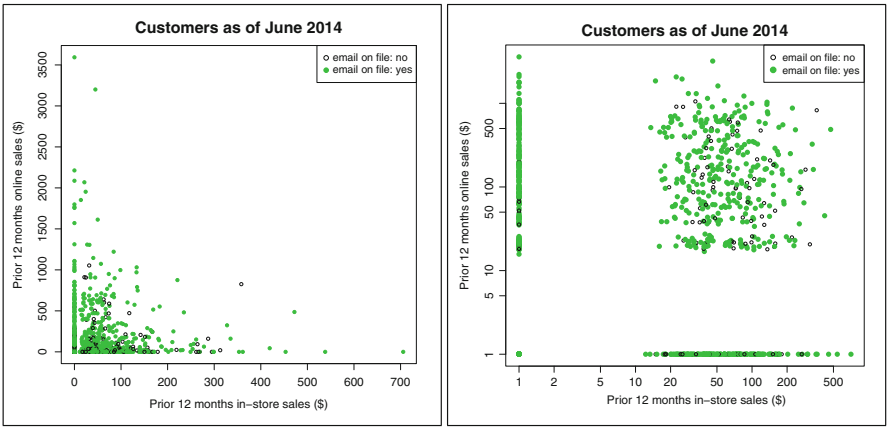

Fig. 4.4. Scatterplots of online sales vs. in-store sales by customer. On the *left*, we see a typical extremely skewed plot using raw sales values; data is grouped along the x and y axes because many customers purchase nothing. On the *right*, plotting the `log()` of sales separates zero and non-zero values more clearly, and reveals the association among those who purchase in the two channels (see Sect. 4.2.4).

For `cust.df`, because both online and in-store sales are skewed, we use a log scale for both axes:

```
> plot(cust.df$store.spend + 1, cust.df$online.spend + 1,
+      log="xy", cex=0.7,
+      col=my.col[cust.df$email], pch=my.pch[cust.df$email],
+      main="Customers as of June 2014",
+      xlab="Prior 12 months in-store sales ($)",
+      ylab="Prior 12 months online sales ($)" )
> legend(x="topright", legend=paste("email on file:", levels(cust.df$email)),
+        col=my.col, pch=my.pch)
```

In this code, we plot $\ldots spend + 1$ to avoid an error due to the fact that `log(0)` is not defined. In the right-hand side of Fig. 4.4, the axes are now logarithmic; for instance, the distance from 1 to 10 is the same as 10–100.

On the right-hand panel of Fig. 4.4, it is easy to see a large number of customers with no sales (the points at $x = 1$ or $y = 1$, which correspond to zero sales because we added `1`). It now appears that there is little or no association between online and in-store sales; the scatterplot among customers who purchase in both channels shows no pattern. Thus, there is no evidence here to suggest that online sales have cannibalized in-store sales (a formal test of that would be complex, but the present data do not argue for such an effect in any obvious way).

We also see in Fig. 4.4 that customers with no email address on file show slightly lower online sales than those with addresses; there are somewhat more black circles in the lower half of the plot than the upper half. If we have been sending email promotions to customers, then this suggests that the promotions might be working. An experiment to confirm that hypothesis could be an appropriate next step.

Did it take work to produce the final plot on the right side of Fig. 4.4? Yes, but the result shows how a well-crafted scatterplot can present a lot of information about relationships in data. Looking at the right-hand panel of Fig. 4.4, we have a much better understanding of how online and offline sales are related to each other, and whether each relates to having customers' email on-file.

4.3 Combining Plots in a Single Graphics Object

Sometimes we want to visualize several relationships at once. For instance, suppose we wish to examine whether customers who live closer to stores spend more in store, and whether those who live further away spend more online. Those involve different spending variables and thus need separate plots. If we plot several such things individually, we end up with many individual charts. Luckily, R can produce a single graphic that consists of multiple plots. You do this by telling R that you want multiple plots in a single graphical object with the `par(mfrow=...)` command; then simply plot each one with `plot()` as usual.

It is easiest to see how this works with an example:

```
> par(mfrow=c(2, 2))
> plot(cust.df$distance.to.store, cust.df$store.spend, main="store")
> plot(cust.df$distance.to.store, cust.df$online.spend, main="online")
> plot(cust.df$distance.to.store, cust.df$store.spend+1, log="xy",
+      main="store, log")
> plot(cust.df$distance.to.store, cust.df$online.spend+1, log="xy",
+      main="online, log")
```

Instead of four separate plots from the individual `plot()` commands, this code produces a single graphic with four panels as shown in Fig. 4.5. The first line sets

the graphical parameter `mfrow` to `c(2, 2)`, which instructs R to create a single graphic comprising a two-by-two arrangement of plots, which begins on the first row and moves from left to right.

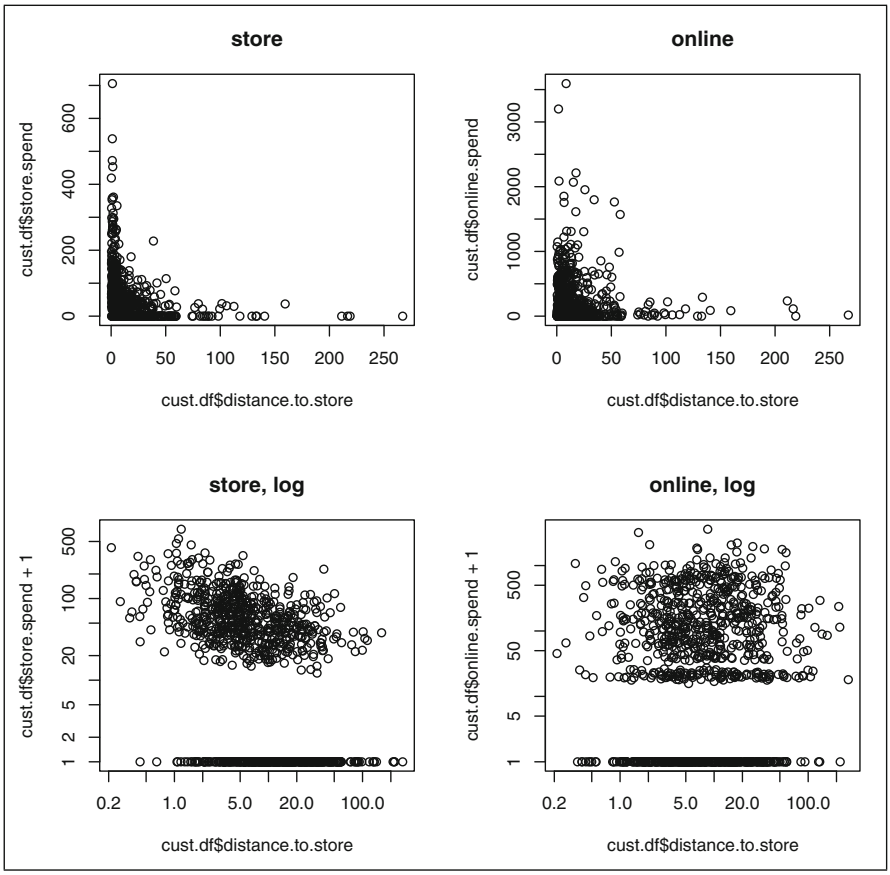

Fig. 4.5. A single graphic object consisting of multiple plots shows that distance to store is related to in-store spending, but seems to be unrelated to online spending. The relationships are easier to see when spending and distance are plotted on a log scale using `log="xy"` in the two lower panels.

Although the plots in Fig. 4.5 are not completely labelled, we see in the lower left panel that there may be a negative relationship between customers' distances to the nearest store and *in-store* spending. Customers who live further from their nearest store spend less in store. However, on the lower right, we don't see an obvious relationship between distance and *online* spending.

After using `par(mfrow=)`, you can return to a single plot layout with `par(mfrow=c(1,1))`.

4.4 Scatterplot Matrices

4.4.1 pairs()

In our customer data, we have a number of variables that might be associated with each other; age, distance.to.store, and email all might be related to online and offline transactions and to spending. When you have several variables such as these, it is good practice to examine scatterplots between all pairs of variables before moving on to more complex analyses.

To do this, R provides the convenient function pairs(formula, data), which makes a separate scatterplot for every combination of variables:

```
> pairs(formula = ~ age + credit.score + email +
+                   distance.to.store + online.visits + online.trans +
+                   online.spend + store.trans + store.spend,
+       data=cust.df)
```

The first input to pairs is a formula listing the variables to include from a data frame. Formulas are used in many R functions and we describe more about them in Chaps. 5, 7. For now it is sufficient to know that in pairs(), the formula is composed with a ~ followed by the variables to include, separated by +. If you want to transform a variable, include the math in the formula. For example, to plot the log() of online.spend, you would include log(online.spend) in the formula.

The second input is data=cust.df, which tells pairs that we want to use the cust.df data frame as the source of data for the plot.

The resulting plot is shown in Fig. 4.6 and is called a *scatterplot matrix*. Each position in this matrix shows a scatterplot between two variables as noted in the diagonal for each row and column. For example, the plot in the first row and forth column is a scatterplot of cust.df$age on the y-axis versus cust.df$distance.to.store on the x-axis.

We can see relationships between variables quickly in a scatterplot matrix. In the fifth row and sixth column we see a strong linear association between online.visits and online.trans; customers who visit the website more frequently make more online transactions. Looking quickly over the plot, we also see that customers with a higher number of online transactions have higher total online spending (not a surprise), and similarly, customers with more in-store transactions also spend more in-store. This simple command produced a lot of information to consider.

In addition to using the formula notation above, it is also possible to pass a data frame directly to pairs and when you do that, pairs() creates a scatterplot matrix including all the columns in your data frame. In the code below, we select columns 2–10 from cust.df and pass the resulting data frame to pairs, which gives us the same plot as shown in Fig. 4.6:

```
> pairs(cust.df[ , c(2:10)])   # output not repeated; same as above
```

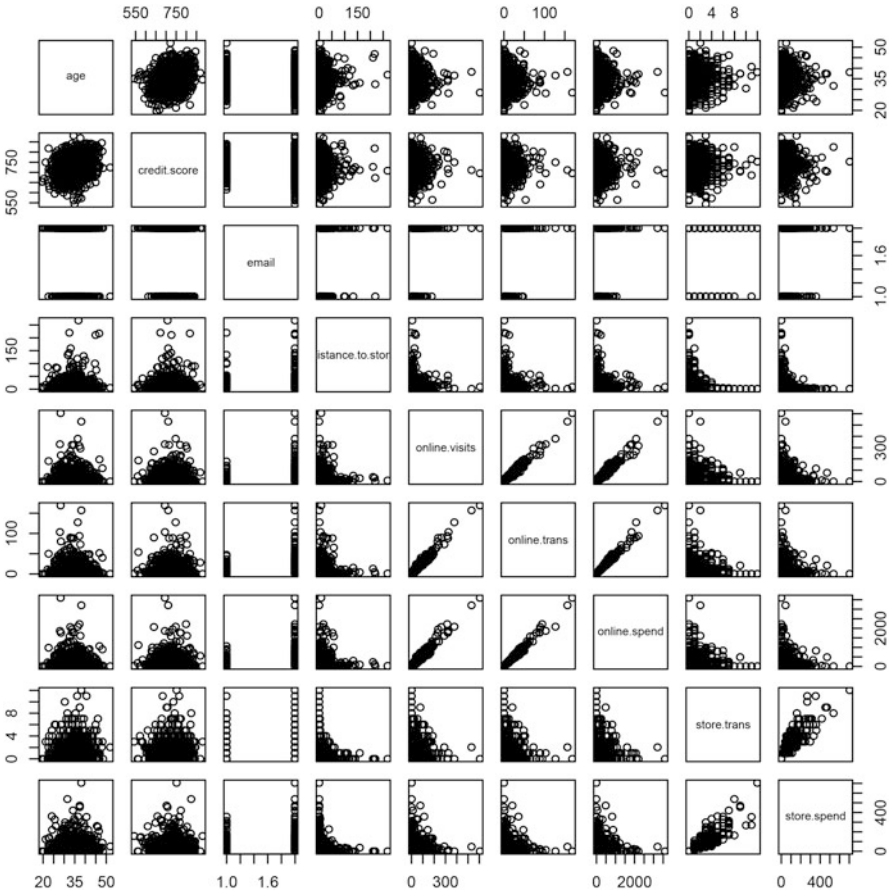

Fig. 4.6. A scatterplot matrix for the customer data set produced using `pairs()`.

While this results in compact code, we recommend instead to use the formula version as shown above; it is robust to future changes in `cust.df` that might re-order the columns. Over time, it becomes a habit to think about how your R code might be re-used in the future.

4.4.2 `scatterplotMatrix()`

Scatterplot matrices are so useful for data exploration that several add-on packages offer additional versions them. We want to point out two other scatterplot matrix functions that we find valuable. The `scatterplotMatrix()` function in the `car` package (abbreviating "*c*ompanion to *a*pplied *r*egression" [51]) adds a number of features over `pairs()`, including adding smoothed lines on scatterplots and

univariate histograms on the diagonal. The syntax for `scatterplotMatrix()` is similar to `pairs()`:

```
> library(car)    # install if needed
> scatterplotMatrix(formula = ~ age + credit.score + email +
+                             distance.to.store + online.visits + online.trans +
+                             online.spend + store.trans + store.spend,
+                   data=cust.df, diagonal="histogram")
Warning messages:
1: In smoother(x, y, col = col[2], log.x = FALSE, log.y = FALSE, ...
```

This produces warnings because the factor variable `email` cannot be smoothed.

In Fig. 4.7, we have histograms on the diagonal that show us the distribution of each variable, where it is easy to see that all of the variables except `age` and `credit.score` are highly left skewed. The green lines show linear fit lines (see Chap. 7), while the red lines show smoothed fit lines and their confidence intervals. The smoothed lines on the bivariate scatterplots suggest the extent to which associations are linear. For instance, the smoothed line on the plot of `age` versus `distance.to.store` is nearly flat and shows that there is no linear association between those variables.

A limitation of Figs. 4.6 and 4.7 concerns the display of the `email` variable. `email` is a binary factor with values `yes` and `no`, and a scatterplot is not ideal to visualize a discrete variable. For such variables, the `gpairs`, or Generalized Pair Plots, package [41] provides a function called `gpairs()` that produces a scatterplot matrix that includes better visualizations for both discrete and continuous variables. For example, if we want to look more closely at the relationship between `email` and `online.visits`, `online.trans` and `online.spend`, we can use `gpairs()` as follows:

```
> install.packages("gpairs")  # only run once
> library(gpairs)
> gpairs(cust.df[ , c(2:10)])
```

Unfortunately `gpairs()` does not accept formula input, so we select the columns to include by number. The resulting scatterplot matrix is shown in Fig. 4.8.

Like `pairs()` and `scatterplotMatrix()`, `gpairs()` produces scatterplots for pairs of continuous variables. However, for the factor `email`, `gpairs` includes a boxplot that compares the distribution of continuous variables for those who do and do not have email addresses in the data. A boxplot shows that the distributions of visits, transactions, and spending have longer tails among customers who have email addresses on file than those who don't. We discuss boxplots in depth in Chap. 5, which focuses on comparisons between groups.

Because it selects individual plots to fit the data types, `gpairs()` is useful for marketing data sets that include continuous and discrete variables. Note that `gpairs()` relies on the data types in R to determine how to construct its plots; if we had stored

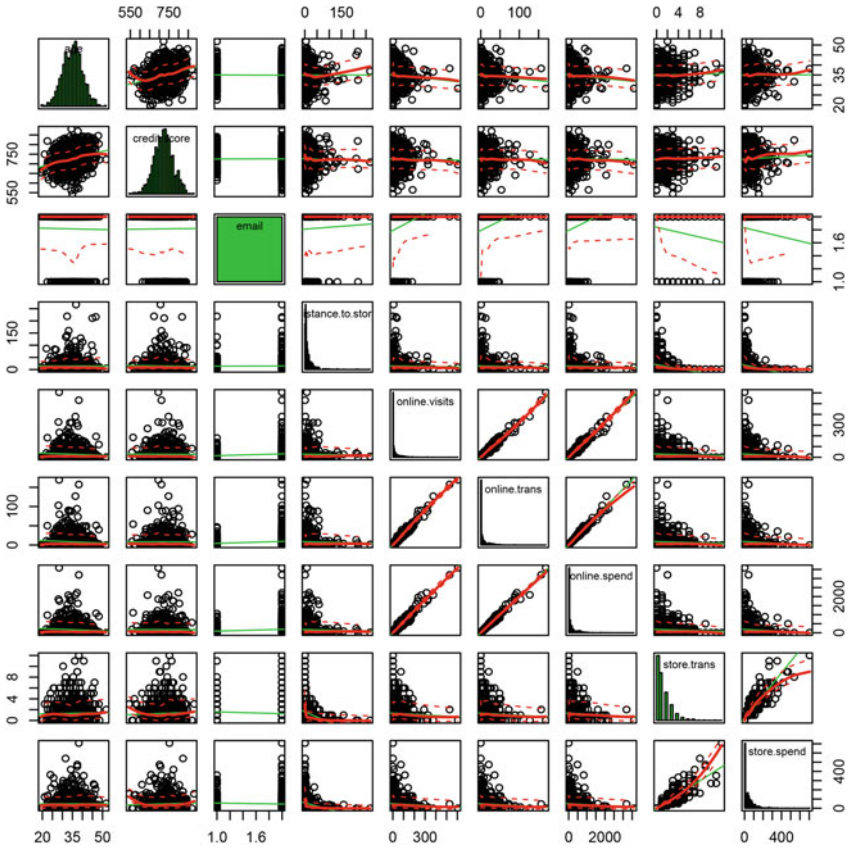

Fig. 4.7. A scatterplot matrix for the customer data set produced using `scatterplotMatrix()`.

`cust.df$email` as a numerical code rather than as a `factor`, `gpairs()` would have produced xy scatterplots instead of boxplots by factor. This is yet another reason why it is useful to set variable types appropriately.

4.5 Correlation Coefficients

Although scatterplots provide a lot of visual information, when there are more than a few variables, it can be helpful to assess the relationship between each pair with a single number. One measure of the relationship between two variables is the *covariance*, which can be computed for any two variables using the `cov` function:

```
> cov(cust.df$age, cust.df$credit.score)
[1] 63.23443
```

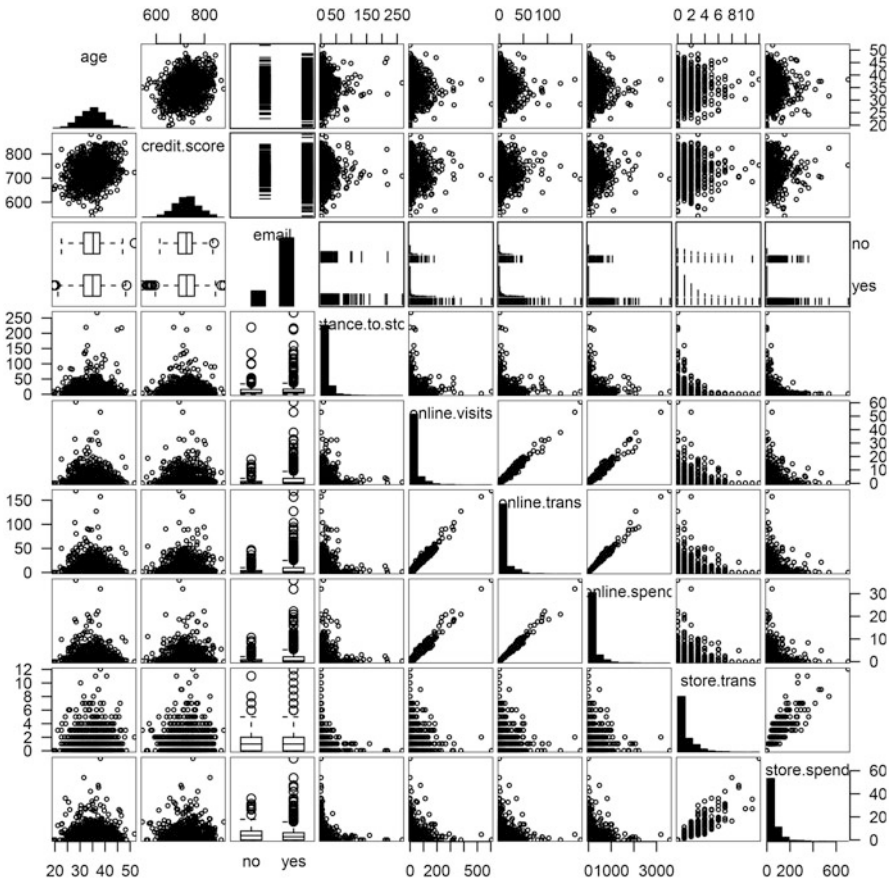

Fig. 4.8. A scatterplot matrix for the customer data set produced using `gpairs()`.

If values x_i and y_i tend to go in the same direction—to be both higher or both lower than their respective means—across observations, then they have a positive covariance. If $cov(x,y)$ is zero, then there is no (linear) association between x_i and y_i. Negative covariance means that the variables go in opposite directions relative to their means: when x_i is lower, y_i tends to be higher.

However, it is difficult to interpret the magnitude of covariance because the scale depends on the variables involved. Covariance will be different if the variables are measured in cents versus dollars or in inches versus centimeters. So, it is helpful to scale the covariance by the standard deviation for each variable, which results in a standardized, rescaled *correlation coefficient* known as the *Pearson product-moment correlation coefficient*, often abbreviated as the symbol r.

Pearson's r is a continuous metric that falls in the range $[-1, +1]$. It is $+1$ in the case of a perfect positive linear association between the two variables, and -1 for

perfect negative linear association. If there is little or no linear association, r will be near 0. On a scatterplot, data with $r = 1$ or $r = -1$ would have all points along a straight line (up or down, respectively). This makes r an easily interpreted metric to assess whether two variables have a close linear association or not.

In R, we compute correlation coefficient r with the `cor()` function:

```
> cor(cust.df$age, cust.df$credit.score)
[1] 0.2545045
```

r is identical to rescaling the covariance by the joint standard deviations (but more convenient):

```
> cov(cust.df$age, cust.df$credit.score) /
+ (sd(cust.df$age)*sd(cust.df$credit.score))
[1] 0.2545045
```

What value of r signifies an *important* correlation between two variables in marketing? In engineering and physical sciences, physical measurements may demonstrate extremely high correlations; for instance, r between the lengths and weights of pieces of steel rod might be 0.9, 0.95, or even 0.999, depending on the uniformity of the rods and the precision of measurement. However, in social sciences such as marketing, we are concerned with human behavior, which is less consistent and more difficult to measure. This results in lower correlations, but they are still important.

We often use *Cohen's Rules of Thumb*, which come out of the psychology tradition [27]. Cohen proposed that for correlations between variables describing people, $r = 0.1$ should be considered a *small* or *weak* association, $r = 0.3$ might be considered to be *medium* in strength, and $r = 0.5$ or higher could be considered to be *large* or *strong*. Cohen's interpretation of a *large* effect was that such an association would be easily noticed by casual observers. A *small* effect would require careful measurement to detect yet might be important to our understanding and to statistical models.

Importantly, interpretation of r according to Cohen's rules of thumb depends on the assumption that the variables are *normally distributed* (also known as *Gaussian*) or are approximately so. If the variables are not normal, but instead follow a logarithmic or other distribution that is skewed or strongly non-normal in shape, then these thresholds do not apply. In those cases, it can be helpful to transform your variables to normal distributions before interpreting, as we discuss in Sect. 4.5.3 below.

4.5.1 Correlation Tests

In the code above, `cor(age, credit.score)` shows $r = 0.25$, a medium-sized effect by Cohen's standard. Is this also statistically significant? We can use the function `cor.test()` to find out:

```
> cor.test(cust.df$age, cust.df$credit.score)

  Pearson's product-moment correlation

data:  cust.df$age and cust.df$credit.score
t = 8.3138, df = 998, p-value = 4.441e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.1955974 0.3115816
sample estimates:
      cor
0.2545045
```

This tells us that $r = 0.25$ and the 95 % confidence interval is $r = 0.196 - 0.312$. Because the confidence interval for r does not include 0 (and thus has p-value of $p < 0.05$), the association is statistically significant. Such a correlation, showing a medium-sized effect and statistical significance, probably should not be ignored in subsequent analyses.

4.5.2 Correlation Matrices

For more than two variables, you can compute the correlations between all pairs x, y at once as a *correlation matrix*. Such a matrix shows $r = 1.0$ on the diagonal because $cor(x,x) = 1$. It is also symmetric; $cor(x,y) = cor(y,x)$. We compute a correlation matrix by passing multiple variables to `cor()`:

```
> cor(cust.df[, c(2, 3, 5:12)])
                          age credit.score distance.to.store online.visits
age               1.000000000  0.254504457        0.00198741   -0.06138107
credit.score      0.254504457  1.000000000       -0.02326418   -0.01081827
distance.to.store 0.001987410 -0.023264183        1.00000000   -0.01460036
online.visits    -0.061381070 -0.010818272       -0.01460036    1.00000000
online.trans     -0.063019935 -0.005018400       -0.01955166    0.98732805
online.spend     -0.060685729 -0.006079881       -0.02040533    0.98240684
store.trans       0.024229708  0.040424158       -0.27673229   -0.03666932
store.spend       0.003841953  0.042298123       -0.24149487   -0.05068554
sat.service                NA           NA                NA            NA
sat.selection              NA           NA                NA            NA
                 online.trans online.spend store.trans  store.spend
age               -0.06301994 -0.060685729  0.02422971  0.003841953
credit.score      -0.00501840 -0.006079881  0.04042416  0.042298123
...
```

In the second column of the first row, we see that `cor(age, credit.store) = 0.254` as above. We can easily scan to find other large correlations; for instance, the correlation between `store.trans`, `distance.to.store` $= -0.277$, showing that people who live further from a store tend to have fewer in-store transactions. `cor()` did not compute correlations for `sat.selection`

and `sat.service` because they have some `NA` values. The argument `use="complete.obs"` would instruct R to use only cases without `NA` values; try it for practice.

Rather than requiring one to scan a matrix of numbers, the `corrplot` package charts correlation matrices nicely with `corrplot()` and `corrplot.mixed()`:

```
> library(corrplot)     # for correlation plot, install if needed
> library(gplots)       # color interpolation, install if needed
> corrplot.mixed(corr=cor(cust.df[ , c(2, 3, 5:12)], use="complete.obs"),
+                upper="ellipse", tl.pos="lt",
+                col = colorpanel(50, "red", "gray60", "blue4"))
```

The resulting graphic is shown in Fig. 4.9. We will explain the code and features of the plot. The main argument to `corrplot.mixed` is a correlation matrix and we use `cor(..., use="complete.obs")` to provide this, excluding the `NA` values.

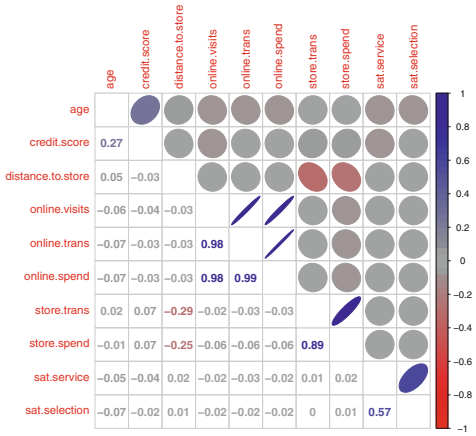

Fig. 4.9. A correlation plot produced using `corrplot.mixed()` from the `corrplot` package is an easy way to visualize all of the correlations in the data. Correlations close to zero are plotted as *circular* and *gray* (using the color scheme we specified), while magnitudes away from zero produce ellipses that are increasingly tighter and *blue* for positive correlation and *red* for negative.

In Fig. 4.9, numeric values of r are shown in the lower triangle of the matrix. The upper triangle displays ellipses (because we used the argument `upper="ellipse"`). These ellipses are tighter, progressively closer to being lines, for larger values of r, and are rounder, more like circles for r near zero. They are also shaded blue for positive direction, and red for negative (and show corresponding positive or negative slope).

This makes it easy to find the larger correlations in the data: `age` is positively correlated with `credit.score`; `distance.to.store` is negatively correlated with `store.trans` and `store.spend`; `online.visits`, `online.trans`, and `online.spend` are all strongly correlated with one another, as are `store.trans` and `store.spend`. In the survey items, `sat.service` is positively correlated with `sat.selection`.

`corrplot.mixed()` has numerous options that let you customize a chart. For this plot, we use the options `upper="ellipse"` to visualize the correlations as ellipses and `tl.pos="lt"` to place the variable name labels on the left and top of the matrix. The correlations in this case are mostly small in magnitude, which produces a very light chart with the default colors. We use `colorpanel()` from the `gplots` package to generate a set of colors anchored at three points ("red", "gray60", and "blue4") and tell `corrplot.mixed()` to use that set of colors instead of its default. You could try other colors and see how the plot is affected; the `colors()` command will list all the names of colors that R understands.

While it is impossible to draw strong conclusions based on associations such as Fig. 4.9, finding large correlations should inform subsequent analysis or suggest hypotheses.

4.5.3 Transforming Variables before Computing Correlations

Correlation coefficient r measures the *linear* association between two variables. If the relationship between two variables is not linear, it would be misleading to interpret r. For example, if we create a random variable that falls in the range $[-10, 10]$—using `runif()` to sample random uniform values—and then compute the correlation between that variable and its square, we get a correlation close to zero:

```
> set.seed(49931)
> x <- runif(1000, min=-10, max=10)
> cor(x, x^2)
[1] -0.003674254
```

r is near zero despite the fact that there is a perfect *nonlinear* relationship between x and x^2. So, it is important that we consider transformations before assessing the correlation between two variables. (It might be helpful to plot x and x^2 by typing `plot(x, x^2)`, so that you can see the relationship.)

Many relationships in marketing data are nonlinear. For example, as we see in the `cust.df` data, the number of trips a customer makes to a store may be *inversely* related to distance from the store. When we compute the correlation between the raw values of`distance.to.store` and `store.spend`, we get a modest negative correlation:

```
> cor(cust.df$distance.to.store, cust.df$store.spend)
[1] -0.2414949
```

However, if we transform `distance.to.store` to its *inverse* $(1/distance)$, we find a much stronger association:

```
> cor(1/cust.df$distance.to.store, cust.df$store.spend)
[1] 0.4329997
```

In fact, the inverse square root of distance shows an even greater association:

```
> cor(1/sqrt(cust.df$distance.to.store), cust.df$store.spend)
[1] 0.4843334
```

How do we interpret this? Because of the inverse *square root* relationship, someone who lives 1 mile from the nearest store will spend quite a bit more than someone who lives 5 miles away, yet someone who lives 20 miles away will only buy a little bit more than someone who lives 30 miles away.

These transformations are important when creating scatterplots between variables as well. For example, examine the scatterplots in Fig. 4.10 for raw `distance.to.store` versus `store.spend`, as compared to the inverse square root of `distance.to.store` versus `store.spend`. We create those two charts as follows:

```
> plot(cust.df$distance.to.store, cust.df$store.trans)
> plot(1/sqrt(cust.df$distance.to.store), cust.df$store.trans)
```

The association between distance and spending is much clearer with the transformed data as shown in the right-hand panel of Fig. 4.10.

To review, it is important to consider transforming variables to approximate normality before computing correlations or creating scatterplots; the appropriate transformation may help you to see associations more clearly. As we noted in Sect. 4.5, interpretation of r with rules of thumb requires data to be approximately normal.

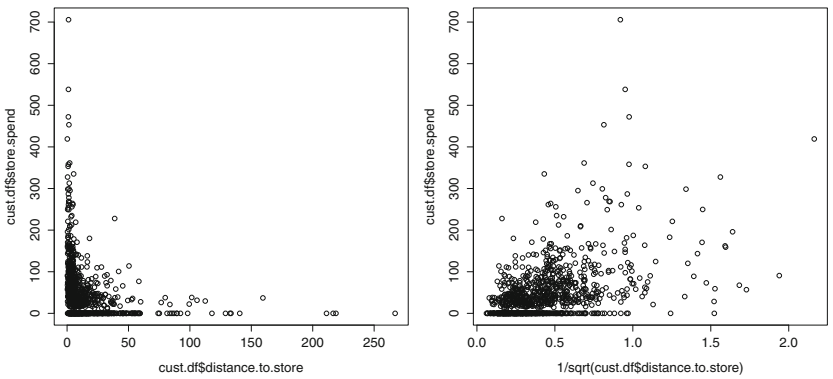

Fig. 4.10. A transformation of `distance.to.store` to its inverse square root makes the association with `store.trans` more apparent in the right-hand chart, as compared to the original values on the left.

4.5.4 Typical Marketing Data Transformations

Considering all the possible transforms may seem impossible, but because marketing data often concerns the same kinds of data in different data sets—counts, sales, revenue, and so forth—there are a few common transformations that often apply. For example, as we discussed when simulating the data for Chap. 3, unit sales are often related to the logarithm of price.

In Table 4.1, we list common transformations that are often helpful with different types of marketing variables.

Table 4.1. Common transformations of variables in marketing

Variable	Common transform
Unit sales, revenue, household income, price	$\log(x)$
Distance	$1/x$, $1/x^2$, $\log(x)$
Market or preference share based on a utility value (Sect. 9.2.1)	$\frac{e^x}{1+e^x}$
Right-tailed distributions (generally)	$\sqrt{x}$ or $\log(x)$ (watch out for $\log(x \leq 0)$)
Left-tailed distributions (generally)	x^2

For most purposes, these standard transformations are appropriate and theoretically sound. However, when these transformations don't work or you want to determine the very best transformation, there is a general-purpose transformation function that can be used instead, and we describe that next.

4.5.5 Box–Cox Transformations*

The remaining sections in the chapter are optional, although important. If you're new to this material, you might skip to the Key Points at the end of this chapter (Sect. 4.8). Remember to return to these sections later and learn more about correlation analysis!

Many of the transformations in Table 4.1 involve taking a power of x: x^2, $1/x = x^{-1}$, and $\sqrt{x} = x^{-0.5}$. The *Box–Cox transformation* generalizes this use of power functions and is defined as:

$$y_i^{(lambda)} \begin{cases} = \frac{y_i^{lambda} - 1}{lambda} & \text{if } lambda \neq 0 \\ = \log(y_i) & \text{if } lambda = 0 \end{cases} \tag{4.1}$$

where *lambda* can take any value and log is the natural logarithm. One could try different values of *lambda* to see which transformation makes the distribution best fit

the normal distribution. (We will see in Chap. 7 that it is also common to use transformed data that makes a linear regression have normally distributed residuals.) Because transformed data is more approximately normal, it is more suitable to assess the strength of association using the rules of thumb for *r* (Sect. 4.5).

Instead of trying values of *lambda* by hand, there is an automatic way to find the optimal value: use the `powerTransform(object=DATA)` function. We find the best Box–Cox transformation for `distance.to.store` using `powerTransform()` as follows:

```
> library(car)
> powerTransform(cust.df$distance.to.store)
Estimated transformation parameters
cust.df$distance.to.store
          -0.003696395
```

This tells us that the value of *lambda* to make distance as similar as possible to a normal distribution is −0.003696. We extract that value of `lambda` using the `coef()` function and create the transformed variable using `bcPower(U=DATA, lambda)`:

```
> lambda <- coef(powerTransform(1/cust.df$distance.to.store))
> bcPower(cust.df$distance.to.store, lambda)
   [1]  0.950421270  3.902743543  0.251429693  1.664085284  3.239908993
   [6]  2.931485684  2.243992143  1.940984081  2.565290889  1.896458754
  [11]  1.898262423  0.411047042  4.101597125  1.359172873  3.8973383223
...
```

To see how this changes `cust.df$distance.to.store`, we plot two histograms comparing the transformed and untransformed variables:

```
> par(mfrow=c(1,2))
> hist(cust.df$distance.to.store,
+      xlab="Distance to Nearest Store", ylab="Count of Customers",
+      main="Original Distribution")
> hist(bcPower(cust.df$distance.to.store, lambda),
+      xlab="Box-Cox Transform of Distance", ylab="Count of Customers",
+      main="Transformed Distribution")
```

The resulting graphs in Fig. 4.11 show the highly skewed original distribution on the left and the transformed distribution on the right, which is much approximately normally distributed.

If you attempt to transform a variable that is already close to normally distributed, `powerTransform()` will report a value of `lambda` that is close to 1. For example, if we find the Box–Cox transform for `age`, we get `lambda` very close to 1, suggesting that a transformation is not required:

```
> powerTransform(cust.df$age)
Estimated transformation parameters
cust.df$age
1.036142
```

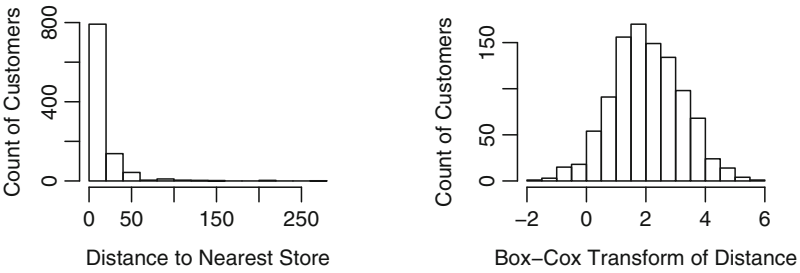

Fig. 4.11. A Box–Cox transformation of `distance.to.store` makes the distribution closer to Normal.

Finally, we can compute correlations for the transformed variable. These correlations will often be larger in magnitude than correlations among raw, untransformed data points. We check *r* between distance and in-store spending, transforming both of them first:

```
> l.dist  <- coef(powerTransform(cust.df$distance.to.store))
> l.spend <- coef(powerTransform(cust.df$store.spend+1))
>
> cor(bcPower(cust.df$distance.to.store, l.dist),
+     bcPower(cust.df$store.spend+1, l.spend))
[1] -0.4683126
```

The relationship between distance to the store and spending can be interpreted as strong and negative.

In practice, you could consider Box–Cox transformations on all variables with skewed distributions before computing correlations or creating scatterplots. This will increase the chances that you will find and interpret important associations between variables.

4.6 Exploring Associations in Survey Responses*

Many marketing data sets include variables where customers provide ratings on a discrete scale, such as a 5- or 7-point rating scale. These are *ordinal* (ranked) variables and it can be a bit tricky to assess associations among them. For instance, in the `cust.df` data, we have response on a 5-point scale for two satisfaction items, satisfaction with the retailer's service and with the retailer's product selection.

What is the problem? Consider a simple `plot()` of the two 5-point items:

```
> plot(cust.df$sat.service, cust.df$sat.selection,
+      xlab="Customer Satisfaction with Service",
+      ylab="Customer Satisfaction with Selection",
+      main="Customers as of June 2014")
```

The resulting plot shown in the left-hand panel of Fig. 4.12 is not very informative. Because `cust.df$sat.service` and `cust.df$sat.selection` only take integer values from 1 to 5, the points for customers who gave the same responses are drawn on top of each other. The main thing we learn from this plot is that customers reported most of the possible pairs of values, except that ratings rarely showed a difference between the two items of 3 or more points (there were no pairs for (1, 4), (1, 5), (5, 2), or a few other combinations).

Fig. 4.12. A scatterplot of responses on a survey scale (*left*) is not very informative. Using jitter (*right*) makes the plot more informative and reveals the number of observations for each pair of response values.

This poses a problem both for visualization and, as it turns out, for assessing the strength of association. We'll see next how to improve the visualization.

4.6.1 `jitter()`*

One way to make a plot of ordinal values more informative is to *jitter* each variable, adding a small amount of random noise to each response. This moves the points away from each other and reveals how many responses occur at each combination of (x, y) values.

R provides the function `jitter()` to do this:

```
> plot(jitter(cust.df$sat.service), jitter(cust.df$sat.selection),
+      xlab="Customer Satisfaction with Service",
+      ylab="Customer Satisfaction with Selection",
+      main="Customers as of June 2014")
```

The result is shown in the right-hand panel of Fig. 4.12, where it is easier to see that the ratings (3, 2) and (3, 3) were the most common responses. It is now clear that there is a positive relationship between the two satisfaction variables. People who are more satisfied with selection tend to be more satisfied with service.

4.6.2 `polychoric()`*

The constrained observations from ratings scales affect assessment of correlation with metrics such as Pearson's *r* because the number of available scale points constrains the potential range and specificity of *r*. An alternative to the simple computation of *r* is a *polychoric* correlation coefficient, which is designed specifically for ordinal responses.

The concept of a polychoric correlation is that respondents have continuous values in mind when they answer on a rating scale. However, because the scales are limited to a small number of points, respondents must select discrete values and choose points on the scale that are closest to the unobserved latent continuous values. The polychoric estimate attempts to recover the correlations between the hypothetical latent (unobserved) continuous variables.

We examine whether the `sat.service` survey item is associated with `sat.selection`. Because we have responses for only some customers, we set an index vector `resp` to identify the customers with responses to examine. Then we look at the *r* correlation coefficient from `cor()`:

```
> resp <- !is.na(cust.df$sat.service)
> cor(cust.df$sat.service[resp], cust.df$sat.selection[resp])
[1] 0.5878558
```

To compute the polychoric correlation coefficient, we use `polychoric()` from the `psych` package:

```
> library(psych)
> polychoric(cbind(cust.df$sat.service[resp],
+                  cust.df$sat.selection[resp]))
Call: polychoric(x = cbind(cust.df$sat.service[resp], cust.df$sat.selection[resp]))
Polychoric correlations
   C1   C2
R1 1.00
R2 0.67 1.00

 with tau of
         1     2    3   4
[1,] -1.83 -0.72 0.54 1.7
[2,] -0.99  0.12 1.26 2.4
# warnings omitted (caused by simulated data's lack of error)
```

This is somewhat more complex information than the simple output of `cor()`. At the top of the output, `polychoric()` reports the polychoric correlation matrix. The values range [−1, 1] and are interpreted in the same way as Pearson's *r*. (In fact,

they are the values of Pearson's *r* between the estimated latent continuous variables.) In our satisfaction data, we can see that the polychoric correlation is quite high at *rho* = 0.67. Like `cor()`, `polychoric()` can produce a correlation matrix for multiple variables.

The second output section under "`with a tau of`" describes how the estimated latent scores are mapped to the discrete item values. For each variable (in our case just two), there are four *cut points*: if a customer's latent satisfaction is below the first cut point, the survey response is the first value on the scale (i.e., 1). For latent scores between the first and second cut points, the survey response is the second value (2), and so forth. Reviewing the cut points can be informative about how the scale is performing and whether it has adequate discrimination of responses versus the estimated latent scores.

4.7 Learning More*

Plotting. As we mentioned at the end of Chap. 3, plotting in R is a complete topic and the subject of several books. We've demonstrated fundamental plotting methods that work for many analyses. Those who do a great deal of plotting or need to produce high-quality graphics for presentation might consider learning `ggplot2` [162] or `lattice` [141].

Correlation analysis. The analysis of variable associations is important for several reasons: it often reveals interesting patterns, it is relatively straightforward to interpret, and it is the simplest case of multivariate analysis. Despite the apparent simplicity there are numerous issues to consider, some of which we have considered here. A classic text for learning about correlation analysis in depth and how to perform it well while avoiding pitfalls is Cohen, Cohen and West [29], *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*, although it is not specific to R.

Analyzing survey scale responses. Much of the data in that we analyze in marketing involves customers' responses to survey ratings scales, and in Sect. 4.6.2 we mentioned some of the challenges with such ordinal response data. Although `polychor()` is a useful tool when analyzing survey data, there are other advanced options. For example, the `bayesm` package [136] provides the function `rscaleUsage()`, which estimates differences in how each customer uses a scale (see also the material on scale usage in Rossi, Allenby and McCulloch [137]). Using `bayesm` requires knowledge of Bayesian methods, which we introduce in Chap. 5.

4.8 Key Points

Following are some of the important points to consider when analyzing relationships between variables.

Visualization

- `plot(x, y)` creates scatterplots where `x` is a vector of x-values to be plotted and `y` is a vector of the same length with y-values (Sect. 4.2.1).
- When preparing a plot for others, the plot should be labeled carefully using arguments such as `xlab`, `ylab`, and `main`, so that the reader can easily understand the graphic (Sect. 4.2.1).
- You can color-code a plot by passing a vector of color names or color numbers as the `col` parameter in `plot()` (Sect. 4.2.2).
- Use the `legend()` command to add a legend so that readers will know what your color coding means (Sect. 4.2.3).
- The `cex=` argument is helpful to adjust point sizes on a scatterplot (Sect. 4.2.1).
- A scatterplot matrix is a good way to visualize associations among several variables at once; options include `pairs()` (Sect. 4.4.1), `scatterplotMatrix()` from the `cars` package, and `gpairs()` from the `gpairs` package (Sect. 4.4.2).
- Many functions such as `plot()` call a *generic function* that determines what to do based on the type of data. When a plotting function does something unexpected, checking data types with `str()` will often reveal the problem (Sect. 4.2.1).
- When variables are highly skewed, it is often helpful to draw the axes on a logarithmic scale by setting the `log` argument of the `plot()` function to `log="x"`, `log="y"`, or `log="xy"` (Sect. 4.2.4). Alternatively, the variables might be transformed to a more interpretable distribution (Sect. 4.5.3).

Statistics

- `cor(x, y)` computes the Pearson correlation coefficient *r* between variables `x` and `y`. This measures the strength of the linear relationship between the variables (Sect. 4.5).
- `cor()` will produce a correlation matrix when it is passed several or many variables. A handy way to visualize these is with the `corrplot` package (Sect. 4.5.2).
- `cor.test()` assesses statistical significance and reports the confidence interval for *r* (Sect. 4.5.1).

- For many kinds of marketing data, the magnitude of r may be interpreted by Cohen's rules of thumb ($r = 0.1$ is a weak association, $r = 0.3$ is medium, and $r = 0.5$ is strong), although this assumes that the data are approximately normal in distribution (Sect. 4.5).
- When the relationship between two variables is nonlinear, r does not give an accurate assessment of the association. Computing r between transformed variables may make associations more apparent (Sect. 4.5.3).
- There are common distributions that often occur in marketing, such as unit sales being related to $\log(price)$. Before modeling associations, plot histograms of your variables and assess potential transformations of them (Sect. 4.5.4).
- An automated way to select an optimal transformation is to use a Box–Cox transform (Sect. 4.5.5).
- The function `polychor()` from the `psych` package is useful to compute correlations between survey responses on ordinal ratings scales (Sect. 4.6.2).