5 Comparing Groups: Tables and Visualizations

Marketing analysts often investigate differences between groups of people. Do men or women subscribe to our service at a higher rate? Which demographic segment can best afford our product? Does the product appeal more to homeowners or renters? The answers help us to understand the market, to target customers effectively, and to evaluate the outcome of marketing activities such as promotions.

Such questions are not confined to differences among people; similar questions are asked of many other kinds of groups. One might be interested to group data by geography: does Region A perform better than Region B? Or time period: did same-store sales increase after a promotion such as a mailer or a sale? In all such cases, we are comparing one group of data to another to identify an effect.

In this chapter, we examine the kinds of comparisons that often arise in marketing, with data that illustrates a consumer segmentation project. We review R procedures to find descriptive summaries by groups, and then visualize the data in several ways.

5.1 Simulating Consumer Segment Data

We begin by creating a data set that exemplifies a consumer segmentation project. For this example, we are offering a subscription-based service (such as cable television or membership in a warehouse club) and have collected data from $N = 300$ respondents on *age*, *gender*, *income*, *number of children*, whether they *own or rent* their homes, and whether they currently *subscribe* to the offered service or not. We use this data in several later chapters as well.

In this data, each respondent has been assigned to one of four consumer segments: "Suburb mix," "Urban hip," "Travelers," or "Moving up." (In this chapter we do not

© Springer International Publishing Switzerland 2015

C. Chapman, E.M. Feit, *R for Marketing Research and Analytics*, Use R!,
DOI 10.1007/978-3-319-14436-8_5

address *how* such segments might be created; we just presume to know them. We look at how to cluster respondents in Chap. 11.)

Segmentation data is moderately complex and we separate our code into three parts:

1. Definition of the data structure: the demographic variables (age, gender, and so forth) plus the segment names and sizes.
2. Parameters for the distributions of demographic variables, such as the mean and variance of each.
3. Code that iterates over the segments and variables to draw random values according to those definitions and parameters.

By organizing the code this way, it becomes easy to change some aspect of the simulation to draw data again. For instance, if we wanted to add a segment or change the mean of one of the demographic variables, only minor change to the code would be required. We also use this structure to teach new R commands that appear in the third step to generate the data.

If you wish to load the data directly, it is available from the book's website:

```
> seg.df <- read.csv("http://goo.gl/qw303p")
> summary(seg.df)
      age            gender          income              kids          ownHome
 Min.   :19.26   Female:157   Min.   : -5183   Min.   :0.00   ownNo :159
 1st Qu.:33.01   Male  :143   1st Qu.: 39656   1st Qu.:0.00   ownYes:141
...
```

However, we recommend that you at least read the data generation sections. We teach important R language skills—looping and `if()` statements—in Sects. 5.1.2 and 5.1.3.

5.1.1 Segment Data Definition

Our first step is to define general characteristics of the data set: the variable names, data types, segment names, and sample size for each segment:

```
> segVars <- c("age", "gender", "income", "kids", "ownHome", "subscribe")
> segVarType <- c("norm", "binom", "norm", "pois", "binom", "binom")
> segNames <- c("Suburb mix", "Urban hip", "Travelers", "Moving up")
> segSize <- c(100, 50, 80, 70)
```

The first variable `segVars` specifies and names the variables to create. `segVarType` defines what kind of data will be present in each of those variables: normal data (continuous), binomial (yes/no), or Poisson (counts). Next we name the four segments with the variable `segNames` and specify the number of observations to generate in each segment (`segSize`). For instance, looking at the first entry in segNames and segSize, the code says that we will create $N = 100$ observations (as specified by segSize[1]) for the "Suburb mix" segment (named by segNames[1]).

Although those variables are enough to determine the *structure* of the data set—the number of rows (observations) and columns (demographic variables and segment assignment)—they do not yet describe the *values* of the data. The second step is to define those values. We do this by specifying distributional parameters such as the mean for each variable within each segment.

There are four segments and six demographic variables, so we create a 4×6 matrix to hold the *mean* of each. The first row holds the mean values of each of the six variables for the first segment; the second row holds the mean values for the second segment, and so forth. We do this as follows:

```
> segMeans <- matrix( c(
+   40, 0.5, 55000, 2, 0.5, 0.1,
+   24, 0.7, 21000, 1, 0.2, 0.2,
+   58, 0.5, 64000, 0, 0.7, 0.05,
+   36, 0.3, 52000, 2, 0.3, 0.2  ), ncol=length(segVars), byrow=TRUE)
```

How does this work? It specifies, for example, that the first variable (which we defined above as age) will have a mean of 40 for the first segment, 24 for the second segment, and so forth. When we draw the random data later in this section, our routine will look up values in this matrix and sample data from distributions with those parameters.

In the case of binomial and Poisson variables, we only need to specify the mean. In these data gender, ownHome, and subscribe will be simulated as binomial (yes/no) variables, which requires specifying the probability for each draw. kids is represented as a Poisson (count) variable, whose distribution is specified by its mean. (Note that we use these distributions for simplicity and do not mean to imply that they are necessarily the *best* distributions to fit real observations of these variables. For example, real observations of income are better represented with a skewed distribution.)

However, for normal variables—in this case, age and income, the first and third variables—we additionally need to specify the *variance* of the distribution, the degree of dispersion around the mean. So we create a second 4×6 matrix that defines the standard deviation for the variables that require it:

```
> # standard deviations for each segment (NA = not applicable for the variable)
> segSDs <- matrix( c(
+   5, NA, 12000, NA, NA, NA,
+   2, NA,  5000, NA, NA, NA,
+   8, NA, 21000, NA, NA, NA,
+   4, NA, 10000, NA, NA, NA  ), ncol=length(segVars), byrow=TRUE)
```

Putting those two matrices together, we have fully defined the distributions of the segments. For instance, look at the third line of each matrix, which corresponds to the "Travelers" segment. The matrices specify that the mean `age` of that segment will be 58 years (looking at the first matrix) and that it will have a standard deviation of 8 years (second matrix). Also it will be approximately 50 % male (looking at the second column), with an average income of $64,000 and $21,000 standard deviation. By storing values in *look-up tables* this way, we can easily change the definitions for future purposes without digging through detailed code. Such separation between data definition and procedural code is a good programming practice.

With these data definitions in place, we are ready to generate data. This uses `for()` loops and `if()` blocks, so we review those before continuing with the simulation process in Sect. 5.1.4.

5.1.2 Language Brief: `for()` Loops

Our data set involves six random variables for age, gender, and so forth, and four segments. So, we need to draw random numbers from $6 \times 4 = 24$ different distributions. Luckily, the structure of each of those random number draws is very similar, and we can use `for()` loops to *iterate* over the variables and the segments.

The `for()` command iterates over a vector of values such as `1:10`, assigning successive values to an *index* variable and running a statement block on each iteration. Here is a simple example:

```
> for (i in 1:10) { print(i) }
[1] 1
[1] 2
[1] 3
...
[1] 10
```

The value of `i` takes on the values from 1 to 10, and the loop executes 10 times in all, running the `print()` command for each successive value of `i`.

If you've programmed before, this will be quite familiar—but there are a couple of twists. The index variable in R `for()` loops can take on any scalar value, not just integers, and it can operate on any vector including those that are defined elsewhere or are unordered. Consider the following where we define a vector of real numbers, `i.seq`, and iterate over its values:

```
> (i.seq <- rep(sqrt(seq(from=2.1, to=6.2, by=1.7)), 3))
[1] 1.449138 1.949359 2.345208 1.449138 1.949359 2.345208 1.449138 1.949359 ...
> for (i in i.seq ) { print(i) }
[1] 1.449138
[1] 1.949359
...
[1] 2.345208
```

An index vector may comprise character elements instead of numeric:

```
> for (i in c("Hello ","world, ","welcome to R!")) { cat(i) }
Hello world, welcome to R!
```

We use the `cat()` command for output here instead of `print()` because of its greater flexibility.

The brackets ("{" and "}") enclose the statements that you want to loop over and are only strictly required when a loop executes a block of more than one statement. However, we recommend for clarity to use brackets with all loops and control structures as we've done here.

By tradition the most common index variable is named "`i`" (and inner loops commonly use "`j`") but you may use any legal variable name. It is a nice practice to give your index variable a descriptive-but-short name like `seg` for segments or `cust` for customers.

There is one thing to *avoid* with `for()` loops in R: indexing on `1:length (someVariable)`. Suppose for the example above, we wanted not the *value* of each element in `i.seq` but its *position* (1, 2, 3, etc.). It would seem natural to write something like this:

```
> for (i in 1:length(i.seq)) { cat("Entry", i, "=", i.seq[i], "\n") }
Entry 1 = 1.449138
Entry 2 = 1.949359
...
```

Don't! This works in many cases but R has a better solution: `seq_along (someVariable)`, which gives a vector of 1, 2, 3, etc. of the same length as `someVariable`. Write the following instead:

```
> for (i in seq_along(i.seq)) { cat("Entry", i, "=", i.seq[i], "\n") }
Entry 1 = 1.449138
Entry 2 = 1.949359
...
```

Why? Because `seq_along()` protects against common errors when the index vector has zero length or is inadvertently reversed. To see what happens when `for()` has a zero-length vector, look at the following buggy code:

```
> i.seq <- NULL
> # maybe we have a bunch of other code, and then ...
> for (i in 1:length(i.seq)) { print (i) }
[1] 1
[1] 0
```

What happened? If `i.seq` is `NULL`, why does it appear to have length 2? The answer is that it doesn't. We told R to do this: start a `for()` loop with the value of 1, and then continue until you reach an index value that matches the length of the vector `i.seq`, which happens to be 0. R complied precisely and iterated over the vector `1:0`. That is, it started with `1` on the first iteration, and then `0` on the second iteration, which then matched the length of `i.seq`.

The proper way to write this is to use `seq_along()`:

```
> i.seq <- NULL
> for (i in seq_along(NULL)) { print (i) }     # better
>
```

This time the index vector has zero length, so nothing is printed. Whenever you find yourself or a colleague writing "`for (i in 1:length ...)`," stop right there and fix it. One day this will save you from a hard-to-find bug . . . and every day you'll be writing better R!

Many R and functional language programmers prefer to avoid `for()` loops and instead use `apply()` functions and the like (Sect. 3.3.4), which automatically work across vector or list objects. We suggest a mixed approach: do whichever makes more sense to you. For many R newcomers, the logic of a `for()` loop is obvious and easier to write reliably than an equivalent such as a list apply (`lapply()`). As you gain experience in R and become comfortable with functions, we recommend to reduce the reliance on `for()`.

5.1.3 Language Brief: `if()` Blocks

Like most programming languages, R provides `if()` statements to handle conditional code execution. The formal syntax is `if (statement1) statement2 else statement3` but we suggest the following code template:

```
# dummy code, not executed
if (condition1) {
  statements
} else if (condition2) {
  statements
} else {
  statements
}
```

A *condition* is the part that evaluates to be `TRUE` or `FALSE`, and it goes inside parentheses. For example, we might write `if (segment==1) { ... }` to do something when the value of `segment` is `1`. (Advanced note: An `if()` condition may in fact be any R statement and will be coerced to a logical value but you should take care to make sure it resolves to be a *single*, *logical* `TRUE` or `FALSE` value.)

The `else if()` blocks and final `else` block in the template are optional and are evaluated only when the preceding `if()` statement evaluates as `FALSE`. The `else if()` blocks are just sequenced `if()` statements and may be chained indefinitely.

In case you're wondering, there is no requirement for an `else` block to handle cases that `if()` does not match; there is an implicit "`else` do nothing and just continue."

We strongly encourage to use brackets ("{" and "}") around all conditional statement blocks as we've done in the template above. This makes code more readable, avoids syntactical ambiguities, and helps prevent bugs when lines are added or deleted in code. Note that when brackets are followed by an `else` block, the closing bracket ("}") generally *must* be on the same line as `else`.

There is a common error with `if()` statements: accidentally or mistakenly using a logical *vector* instead of a single logical value for the condition. Consider:

```
> x <- 1:5
> if (x > 1) {
+   print ("hi")
+ } else {
+   print ("bye")
+ }
[1] "bye"
Warning message:
In if (x > 1) { :
  the condition has length > 1 and only the first element will be used
```

R warns us that the condition is not a single value. It then evaluates `x[1] > 1` as `FALSE`, so it skips to the `else` statement and evaluates that. This is probably not what the programmer intended with this code. There are two possibilities that are likely responsible for this code problem. First, the programmer might have forgotten that `x` is a vector instead of a single value. The warning about "length $>$ 1" tells us to examine our code for that problem.

A second possibility is that the programmer wanted to evaluate *all* of the values of `x` and act on each one of them. However, the `if()` statement is about program flow—in R jargon, it is not vectorized—and thus it evaluates only a single condition per `if()`. To perform conditional evaluation on every element of a vector, use `ifelse(test, yes, no)` instead:

```
> ifelse(x > 1, "hi", "bye")
[1] "bye" "hi"  "hi"  "hi"  "hi"
```

In this case, the condition `x > 1` is evaluated for each element of `x`. That is, it tests whether `x[1] > 1` and then `x[2] > 1` and so forth. When a test evaluates as `TRUE`, the function returns the first ("yes") value; for others it returns the second ("no") value. The "yes" and "no" values may be functions as needed. For instance, in a silly case:

```
> fn.hi  <- function() { "hi" }
> fn.bye <- function() { "bye" }
> ifelse(x > 1, fn.hi(), fn.bye() )
[1] "bye" "hi"  "hi"  "hi"  "hi"
```

Experienced programmers: applying functions conditionally along a vector in this way is one way to avoid `for()` loops in R as we mentioned in Sect. 5.1.2.

5.1.4 Final Segment Data Generation

Armed with `for()` and `if()` and the data definitions above, we are ready to generate the segment data. The logic we follow is to use nested `for()` loops, one for the segments and another within that for the set of variables. (As mentioned in Sects. 5.1.2 and 5.1.3, one could do this without `for()` loops in keeping with the functional programming paradigm of R. However, we use `for()` loops here for clarity and simplicity, and recommend that you code similarly; write whatever code is clearest and easiest to maintain.)

To outline how this will work, consider the following *pseudocode* (sentences organized like code):

```
Set up data frame "seg.df" and pseudorandom number sequence
For each SEGMENT i in "segNames" {
  Set up a temporary data frame "this.seg" for this SEGMENT's data
  For each VARIABLE j in "segVars" {
    Use nested if() on "segVarType[j]" to determine data type for VARIABLE
    Use segMeans[i, j] and segSDs[i, j] to
    ... Draw random data for VARIABLE (within SEGMENT) with
    ... "segSize[i]" observations
  }
  Add this SEGMENT's data ("this.seg") to the overall data ("seg.df")
}
```

Pseudocode is a good way to outline and debug code conceptually before you actually write it. In this case, you can compare the pseudocode to the actual R code to see how we accomplish each step. Translating the outline into R, we write:

```
> seg.df <- NULL
> set.seed(02554)

> # iterate over segments and create data for each
> for (i in seq_along(segNames)) {
+   cat(i, segNames[i], "\n")
+
+   # empty matrix to hold this particular segment's data
+   this.seg <- data.frame(matrix(NA, nrow=segSize[i], ncol=length(segVars)))
+
+   # within segment, iterate over variables and draw appropriate random data
+   for (j in seq_along(segVars)) {     # and iterate over each variable
+     if (segVarType[j] == "norm") {    # draw random normals
+       this.seg[,j] <- rnorm(segSize[i], mean=segMeans[i,j], sd=segSDs[i,j])
+     } else if (segVarType[j] == "pois") {     # draw counts
+       this.seg[, j] <- rpois(segSize[i], lambda=segMeans[i, j])
+     } else if (segVarType[j] == "binom") {    # draw binomials
+       this.seg[, j] <- rbinom(segSize[i], size=1, prob=segMeans[i, j])
+     } else {
+       stop("Bad segment data type: ", segVarType[j])
+     }
+   }
+   # add this segment to the total dataset
+   seg.df <- rbind(seg.df, this.seg)
+ }
```

The core commands occur inside the `if()` statements: according to the data type we want ("norm"[al], "pois"[son], or "binom"[ial]), use the appropriate pseudorandom function to draw data (the function `rnorm(n, mean, sd)`, `rpois(n, lambda)`, or `rbinom(n, size, prob)`, respectively). We draw all of the values for a given variable within a given segment with a single command (drawing all the observations at once, with length specified by `segSize[i]`).

There are a few things to note about this code. As in Sect. 5.1.2 we use `seq_along()` to set up the `for()` loops. To see that the code is working and to show progress, we use `cat("some output message", counter, "\n")` inside the loop (`\n` ends a line so the next iteration will be on a new line of output). That results in the following output as the code runs:

```
1 Suburb mix
2 Urban hip
3 Travelers
4 Moving up
```

Inside the first loop (the `i` loop), we predefine `this.seg` as a data frame with the desired number of rows and columns, but full of missing values (`NA`). Why? Whenever R grows an object in memory—such as adding a row—it makes a copy of the object. This uses twice the memory and slows things down; by preallocating, we avoid that. In small data sets like this one, it hardly matters, but with larger data sets, it can make a huge difference in speed. Also, R can easily draw random values for all respondents in a segment at once and this makes it easier to do so. Finally, it adds a bit of error checking: if a result doesn't fit into the data frame where it *should* fit, we will get a warning or error.

By filling temporary and placeholder objects with missing values (`NA`) instead of 0 or blank values, we add another layer of error-checking: if we `describe()` the object and discover missing values where we expect data, we know there is a code error.

We finish the `if()` blocks in our code with a `stop()` command that executes in the case that a proposed data type doesn't match what we expect. There are three `if()` tests for the expected data types, and a final `else` block in case none of the `if`s matches. This protects us in the case that we mistype a data type or if we try to use a distribution that hasn't been defined in the random draw code, such as a gamma distribution. This `stop()` condition would cause the code to exit immediately and print an error string.

Notice that we are doing a lot of thinking ahead about how our code might change and potentially break in the future to ensure that we would get a warning when something goes wrong. Our code also has another advantage that you may not notice right away: we call each random data function such as `rnorm` in exactly one place.

If we discover that there was something wrong with that call—say we wanted to change one of the parameters of the call—we only need to make the correction in one place. This sort of planning is a hallmark of good programming in R or any other language. While it might seem overly complex at first, many of these ideas will become habitual as you write more programs.

To finish up the data set, we perform a few housekeeping tasks: we name the columns, add segment membership, and convert each binomial variable to a labeled factor:

```
# make the data frame names match what we defined
names(seg.df) <- segVars
# add segment membership for each row
seg.df$Segment   <- factor(rep(segNames, times=segSize))
# convert the binomial variables to nicely labeled factors
seg.df$ownHome   <- factor(seg.df$ownHome, labels=c("ownNo", "ownYes"))
seg.df$gender    <- factor(seg.df$gender, labels=c("Female", "Male"))
seg.df$subscribe <- factor(seg.df$subscribe, labels=c("subNo", "subYes"))
```

We may now inspect the data. As always, we recommend a data inspection plan as noted in Sect. 3.6, although we only show one of those steps here:

```
> summary(seg.df)
      age           gender        income            kids        ownHome
 Min.   :19.26   Female:157   Min.   : -5183   Min.   :0.00   ownNo :159
 1st Qu.:33.01   Male  :143   1st Qu.: 39656   1st Qu.:0.00   ownYes:141
 Median :39.49                Median : 52014   Median :1.00
 Mean   :41.20                Mean   : 50937   Mean   :1.27
 3rd Qu.:47.90                3rd Qu.: 61403   3rd Qu.:2.00
 Max.   :80.49                Max.   :114278   Max.   :7.00
...
```

The data frame is now suitable for exploration. And we have reusable code: we could create data with more observations, different segment sizes, or segments with different distributions or means by simply adjusting the matrices that define the segments and running the code again.

As a final step we save the data frame as a backup and to use again in later chapters (Sects. 11.2 and 12.4). Change the destination if you have created a folder for this book or prefer a different location:

```
> save(seg.df, file="~/segdf-Rintro-Ch5.RData")
```

5.2 Finding Descriptives by Group

For our consumer segmentation data, we are interested in how measures such as household income and gender vary for the different segments. With this insight, a firm might develop tailored offerings for the segments or engage in different ways to reach them.

An ad hoc way to do this is with data frame indexing: find the rows that match some criterion, and then take the mean (or some other statistic) for the matching observations on a variable of interest. For example, to find the mean income for the "Moving up" segment:

```
> mean(seg.df$income[seg.df$Segment == "Moving up"])
[1] 53090.97
```

This says "from the income observations, take all cases where the Segment column is 'Moving up' and calculate their mean." We could further narrow the cases to "Moving up" respondents who also do not subscribe using Boolean logic:

```
> mean(seg.df$income[seg.df$Segment == "Moving up" &
+                      seg.df$subscribe=="subNo"])
[1] 53633.73
```

This quickly becomes tedious when you wish to find values for multiple groups.

As we saw briefly in Sect. 3.4.5, a more general way to do this is with `by(data, INDICES, FUN)`. The result of `by()` is to divide `data` into groups for each of the unique values in `INDICES` and then apply the `FUN` function to each group:

```
> by(seg.df$income, seg.df$Segment, mean)
seg.df$Segment: Moving up
[1] 53090.97
------------------------------------------------
seg.df$Segment: Suburb mix
[1] 55033.82
...
```

With `by()`, keep in mind that `data` is the first argument and the splitting factors `INDICES` come second. You can break out the results by multiple factors if you supply factors in a `list()`. For example, we can break out by segment and subscription status:

```
> by(seg.df$income, list(seg.df$Segment, seg.df$subscribe), mean)
: Moving up
: subNo
[1] 53633.73
----------------------------------------------------------------
: Suburb mix
: subNo
[1] 54942.69
...
----------------------------------------------------------------
: Urban hip
: subYes
[1] 20081.19
```

Our favorite command for this is `aggregate()` as we introduced in Sect. 3.4.5. `aggregate()` works almost identically to `by` in its *list* form (we'll see another form of `aggregate()` momentarily), except that it takes a list for even a single factor:

```
> aggregate(seg.df$income, list(seg.df$Segment), mean)
     Group.1        x
1  Moving up 53090.97
2 Suburb mix 55033.82
3  Travelers 62213.94
4  Urban hip 21681.93
```

A first advantage of `aggregate()` is this: the result is a data frame. As we saw in Sect. 3.4.5, you can save the results of `aggregate()` to an object, which you can then index, subject to further computation, write to a file, or manipulate in other ways.

Here's an example: suppose we wish to add a "segment mean" column to our data set, a new observation for each respondent that contains the mean income for their respective segment so we can compare respondents' incomes to those typical for their segments. We can do this by first aggregating the mean incomes into a table, and then indexing that by segment to look up the appropriate value for each row of our data:

```
> seg.income.mean <- aggregate(seg.df$income, list(seg.df$Segment), mean)
> seg.df$segIncome <- seg.income.mean[seg.df$Segment, 2]
```

When we check the data, we see that each row has an observation that matches its segment mean (`some()` does a random sample of rows, so your output may vary):

```
> library(car)
> some(seg.df)
         age gender   income kids ownHome subscribe    Segment segIncome
58  34.46528   Male 60971.76    2   ownNo     subNo Suburb mix  55033.82
79  42.31337   Male 49674.79    0  ownYes     subNo Suburb mix  55033.82
124 22.30333 Female 24541.24    1   ownNo     subNo  Urban hip  21681.93
136 23.08861   Male 33909.50    3   ownNo     subNo  Urban hip  21681.93
158 43.35230   Male 51787.88    0   ownNo     subNo  Travelers  62213.94
...
```

It is worth thinking about how this works. In the following command:

```
> seg.df$segIncome <- seg.income.mean[seg.df$Segment, 2]
```

...we see this index for the rows: `seg.df$Segment`. If we evaluate that on its own, we see that it is a vector with one entry for each row of `seg.df`:

```
> seg.df$Segment
  [1] Suburb mix Suburb mix Suburb mix Suburb mix Suburb mix Suburb mix Suburb
     mix
...
[295] Moving up  Moving up  Moving up  Moving up  Moving up  Moving up
Levels: Moving up Suburb mix Travelers Urban hip
```

Now let's see what happens when we index `seg.income.mean` with that vector:

```
> seg.income.mean[seg.df$Segment, ]
         Group.1        x
2    Suburb mix 55033.82
2.1  Suburb mix 55033.82
...
1.68  Moving up 53090.97
1.69  Moving up 53090.97
```

The result is a data frame in which each row of `seg.income.mean` occurs many times in the order requested.

Finally, selecting the second column of that gives us the value to add for each row of `seg.df`:

```
> seg.income.mean[seg.df$Segment, 2]
  [1] 55033.82 55033.82 55033.82 55033.82 55033.82 55033.82 55033.82 55033.82
...
[297] 53090.97 53090.97 53090.97 53090.97
```

We generally do not like adding derived columns to primary data because we like to separate data from subsequent computation, but we did so here for illustration. We now remove that column by setting its value to `NULL`:

```
> seg.df$segIncome <- NULL
```

This use of `aggregate()` exemplifies the power of R to extract and manipulate data with simple and concise commands. You may recall that we said this was the *first* advantage of `aggregate()`. The second advantage is even more important and we describe it next.

5.2.1 Language Brief: Basic Formula Syntax

R provides a standard way to describe relationships among variables through *formula* specification. A formula uses the tilde ($\sim$) operator to separate *response variables* on the left from *explanatory variables* on the right. The basic form is:

$$y \sim x \qquad \text{(Simple formula)}$$

This is used in many contexts in R, where the meaning of *response* and *explanatory* depend on the situation. For example, in linear regression, the simple formula above would model y as a linear function of x. In the case of the `aggregate()` command, the effect is to aggregate y according to the levels of x.

Let's see that in practice. Instead of aggregate(seg.df$income, list (seg.df$Segment), mean) we can write:

```
> aggregate(income ~ Segment, data=seg.df, mean)
     Segment   income
1  Moving up 53090.97
...
```

The general form is aggregate(formula, data, FUN). In our example, we tell R to "take income by Segment within the data set seg.df, and apply mean to each group."

The formula "$y \sim x$" might be pronounced in various contexts as "y in response to x," "y is modeled by x," "y varies with x," and so forth. R programmers often become so accustomed to this syntax that they just say "y tilde x." This syntax may seem like nothing special at first, but formulas are used in many different contexts throughout R. We will encounter many uses for formulas later in this book, and discuss additional forms of them in Sect. 7.5.1.

5.2.2 Descriptives for Two-Way Groups

A common task in marketing is cross-tabulating, separating customers into groups according to two (or more) factors. Formula syntax makes it easy to compute a cross tab just by specifying multiple explanatory variables:

$$y \sim x1 + x2 + \cdots \qquad \text{(Multiple variable formula)}$$

Using this format with aggregate(), we write:

```
> aggregate(income ~ Segment + ownHome, data=seg.df, mean)
     Segment ownHome   income
1  Moving up   ownNo 54497.68
2 Suburb mix   ownNo 54932.83
...
7  Travelers  ownYes 61889.12
8  Urban hip  ownYes 23059.27
```

We now have a separate group for each combination of Segment and ownHome and can begin to see how income is related to both the Segment and the ownHome variables.

A formula can be extended to include as many grouping variables as needed:

```
> aggregate(income ~ Segment + ownHome + subscribe, data=seg.df, mean)
      Segment ownHome subscribe   income
1   Moving up   ownNo     subNo 55402.89
...
8   Urban hip  ownYes     subNo 23993.93
9   Moving up   ownNo    subYes 50675.70
...
16  Urban hip  ownYes    subYes 19320.64
```

As we saw for one-way aggregate, the result can be assigned to a data frame object and indexed:

```
> agg.data <- aggregate(income ~ Segment + ownHome, data=seg.df, mean)
> agg.data[2, ]
     Segment ownHome   income
2 Suburb mix   ownNo 54932.83
> agg.data[2, 3]
[1] 54932.83
```

The `aggregate` command allows us to compute functions of continuous variables, such as the `mean` of `income` or `age`) for any combination of factors (`Segment`, `ownHome` and so forth). This is such a common task in marketing research that there are entire companies who specialize in producing cross tabs. As we've just seen, these are not difficult to compute in R.

We might also want to know the *frequency* with which different combinations of `Segment` and `ownHome` occur. We can compute frequencies using `table(factor1, factor2, ...)` to obtain one-way or multi-way counts:

```
> with(seg.df, table(Segment, ownHome))
            ownHome
Segment       ownNo ownYes
  Moving up      47     23
  Suburb mix     52     48
  Travelers      20     60
  Urban hip      40     10
```

There are 10 observed customers in the "Urban hip" segment who own their own homes, and 60 in the "Travelers" segment.

Suppose we want a breakdown of the number of kids in each household (`kids`) by `segment`:

```
> with(seg.df, table(kids, Segment))
    Segment
kids Moving up Suburb mix Travelers Urban hip
   0        13         11        80        17
   1        17         36         0        17
   2        18         22         0        11
   3        13         19         0         4
   4         5          7         0         1
   5         3          3         0         0
   6         0          2         0         0
   7         1          0         0         0
```

This tells us that we have 17 "Urban hip" respondents with 0 kids, 22 "Suburb mix" respondents with 2 kids, and so forth. It represents purely the count of incidence for each crossing point between the two factors, `kids` and `Segment`. In this case we are treating `kids` as a factor and not a number.

However, `kids` is actually a count variable; if a respondent reported 3 kids, that is a count of 3 and we could add together the counts to get the total number of children reported in each segment. `xtabs(formula, data)` provides a handy way to do this. It works with counts to find their total:

```
> xtabs(kids ~ Segment, data=seg.df)
Segment
 Moving up Suburb mix  Travelers  Urban hip
       134        192          0         55
```

Now we know that our "Urban hip" respondents reported a total of 55 kids, while the "Travelers" reported none. You might think of other ways this could be done in R as well. One alternative is `aggregate( ..., sum)`:

```
> aggregate(kids ~ Segment, data=seg.df, sum)
     Segment kids
1  Moving up  134
2 Suburb mix  192
3  Travelers    0
4  Urban hip   55
```

Another option is to multiply the frequency table by marginal number of kids and add it up:

```
> seg.tab <- with(seg.df, table(kids, Segment))
> apply(seg.tab*0:7, 2, sum)
 Moving up Suburb mix  Travelers  Urban hip
       134        192          0         55
```

`apply( , 2, sum)` is better expressed using `colSums()`:

```
> seg.tab <- with(seg.df, table(kids, Segment))
> colSums(seg.tab*0:7)
 Moving up Suburb mix  Travelers  Urban hip
       134        192          0         55
```

We have belabored this in order to show that R typically has many ways to arrive at the same result. This may seem overly complex, yet it is a good thing. One reason is that there are multiple options to match your style and situation. Each method produces results in a different format, and one format might work better in some situation than another. For instance, we've argued that the format from `aggregate()` is often more useful than `by()`. Another reason is that you can do the same thing in two different ways and compare the answers, thus testing your analyses and uncovering potential errors.

5.2.3 Visualization by Group: Frequencies and Proportions

Suppose we plot the proportion of subscribers for each segment to understand better which segments use the subscription service. Apart from making four separate plots, it isn't obvious how to do this with the tools we have learned so far. We could use `table()` along with `barplot()` (from Sect. 3.2.1) to get a plot showing the number of subscribers and non subscribers overall, but breaking this out by `segment` would require lots of work to separate the data and label the plots correctly.

Happily, the `lattice` package provides a useful solution: `histogram (formula, data, type)` is similar to `hist()` but understands formula notation including *conditioning* on a factor, which means to separate the plot into multiple panes based on that factor. Conditioning is indicated with the symbol "|". This is easiest to understand in an example:

```
> require(lattice)
> histogram(~subscribe | Segment, data=seg.df)
```

You will notice that there is no response variable before the tilde (~) in this formula, only the explanatory variable (`subscribe`) after it. `histogram()` automatically assumes that we want to plot the proportion of people at each level of `subscribe`. We condition the plot on `Segment`, telling `histogram` to produce a separate histogram for each segment. The result is shown in Fig. 5.1.

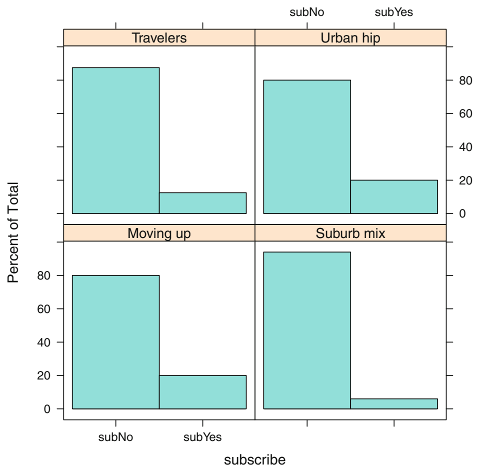

Fig. 5.1. Conditional histogram for proportion of subscribers within each segment, using `lattice`.

In Fig. 5.1, we see that the "Suburban mix" segment is least likely to subscribe to our service. While this data doesn't tell us why that might be, it does suggest that the company might investigate and perhaps either improve the product to make it more appealing to this group or else stop marketing to them.

The default in `histogram()` is to plot *proportions* within each group so that the values are relative to the group size. If we wanted actual *counts* instead, we could include the argument `type="count"`. We do that, adding options for color and changing the `layout` to 4 columns and 1 row:

```
> histogram(~subscribe | Segment, data=seg.df, type="count",
+           layout=c(4,1), col=c("burlywood", "darkolivegreen"))
```

This produces Fig. 5.2. By plotting the counts, we can see which segments are larger, but it is difficult and potentially misleading to compare the count of subscribers across groups of different sizes.

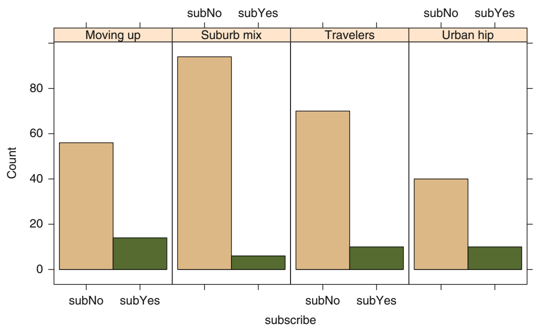

Fig. 5.2. Conditional histogram for *count* of subscribers within each segment.

You can condition on more than one factor; just include it in the conditioning part of the formula with "+". For example, what is the proportion of subscribers within each segment, by home ownership? We add `ownHome` to the formula in `histogram()`:

```
> histogram(~subscribe | Segment + ownHome, data=seg.df)
```

The result is Fig. 5.3. In this plot, the top and bottom rows of Fig. 5.3 are similar, and we conclude that differences in subscription rate according to home ownership within segment are small. An implication is that the company should continue to market to both homeowners and non-homeowners.

Finally, we could plot just "yes" proportions instead of both "yes" and "no" bars. There are several ways to do this; we'll do so by introducing the `prop.table (table, margin)` command. If you wrap `prop.table(..., margin= ...)` around a regular `table()` command, it will give you the proportions for each cell with respect to the entire table (by default), or just the rows (`margin=1`), or the columns (`margin=2`).

We would like to know the proportion of subscribers within each segment, which are the columns in `table(...$subscribe, $Segment)`, so we use `prop.table(..., margin=2)` as follows:

```
> prop.table(table(seg.df$subscribe, seg.df$Segment), margin=2)

         Moving up Suburb mix Travelers Urban hip
  subNo      0.800      0.940     0.875     0.800
  subYes     0.200      0.060     0.125     0.200
```

To plot just the "yes" values, we use `barchart()` and select only the second row of the `prop.table()` result:

```
> barchart(prop.table(table(seg.df$subscribe, seg.df$Segment), margin=2)[2, ],
+          xlab="Subscriber proportion by Segment", col="darkolivegreen")
```

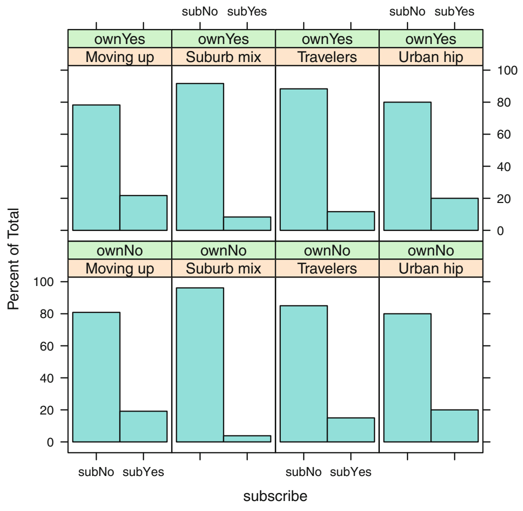

Fig. 5.3. Conditional histogram for subscribers, broken out by segment (in the four columns) and home ownership (in the two rows).

The result is Fig. 5.4, which strongly communicates that the Suburb mix segment has an apparent low subscription rate. Note that this visual impression is amplified by the fact that `barchart()` started the X axis at 0.05, not at 0, which is rather misleading. In practice, you might adjust that using the `xlim=c(low, high)` argument to `barchart()`; we leave that as an exercise. We will see more examples of barcharts in the next section.

5.2.4 Visualization by Group: Continuous Data

In the previous section we saw how to plot counts and proportions. What about continuous data? How would we plot `income` by `segment` in our data? A simple way is to use `aggregate()` to find the mean income, and then use `barchart()` from the `lattice` package to plot the computed values:

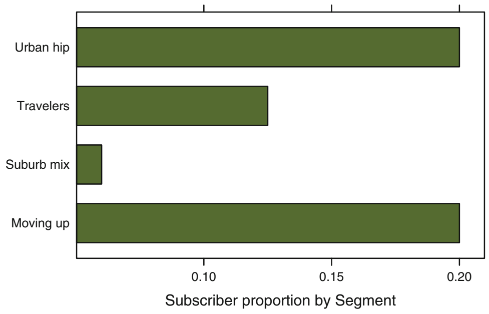

Fig. 5.4. Proportion of subscribers by segment using `prop.table` and `barchart`.

```
> seg.mean <- aggregate(income ~ Segment, data=seg.df, mean)
> library(lattice)
> barchart(income~Segment, data=seg.mean, col="grey")
```

The result is Fig. 5.5.

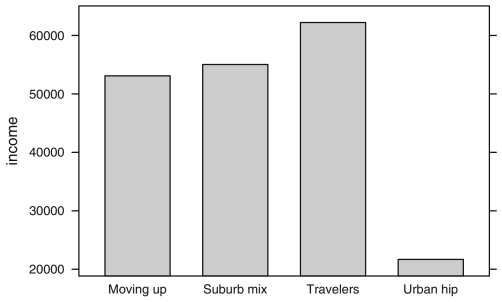

Fig. 5.5. Average income by segment using `prop.table` and `barchart`.

How do we split this out further by home ownership? First we have to aggregate the data to include both factors in the formula. Then we tell `barchart()` to use `ownHome` as a grouping variable by adding the argument `groups=factor`. Doing that, and also adding a `simpleTheme` option to improve the chart colors, we have:

```
> seg.income.agg <- aggregate(income ~ Segment + ownHome, data=seg.df, mean)
> barchart(income ~ Segment, data=seg.income.agg,
+          groups=ownHome, auto.key=TRUE,
+          par.settings = simpleTheme(col=terrain.colors(2)) )
```

This produces a passable graphic as shown in Fig. 5.6 although it still looks as if it came from a spreadsheet program. We can do better in R.

A more informative plot for comparing values of continuous data, like `income` for different groups is a *box-and-whiskers* plot or *boxplot*, which we first encountered in Sect. 3.4.2. A boxplot is better than a barchart because it shows more about the *distributions* of values.

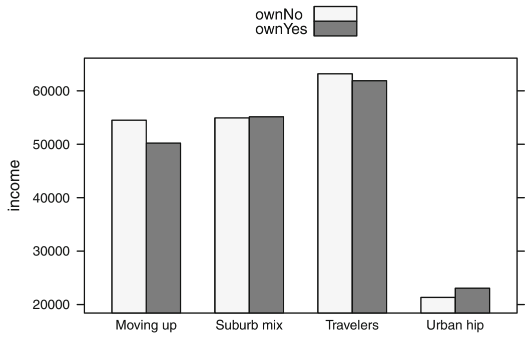

Fig. 5.6. Average income by segment and home ownership, using `aggregate` and `barchart`.

`boxplot()` works with formula syntax to plot a box-and-whiskers plot by factor. Adding improved labels for the *Y* axis (see Sect. 3.4), we write:

```
> boxplot(income ~ Segment, data=seg.df, yaxt="n", ylab="Income ($k)")
> ax.seq <- seq(from=0, to=120000, by=20000)
> axis(side=2, at=ax.seq, labels=paste(ax.seq/1000, "k", sep=""), las=1)
```

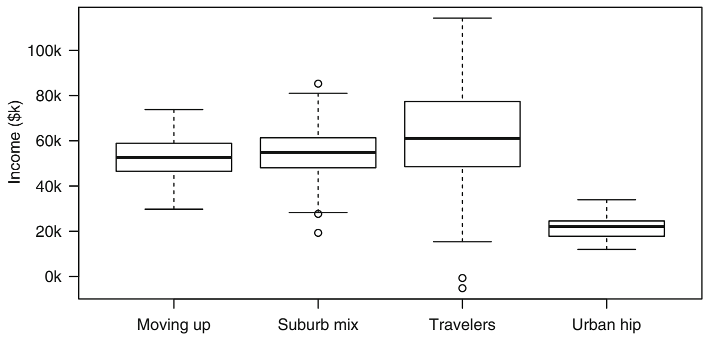

Fig. 5.7. Box-and-whiskers plot for income by segment using `boxplot`.

We can now see in Fig. 5.7 that the income for "Travelers" is higher and also has a greater range, with a few "Travelers" reporting very low incomes. The range of income for "Urban hip" is much lower and tighter. Although box-and-whisker plots are not common in business reporting, we think they should be. They are fairly simple to understand and encode a lot more information than the averages shown in Fig. 5.4.

An even better option for box-and-whiskers plots is the `bwplot()` command from the `lattice` package, which produces better looking charts and allows multi-factor conditioning. One point of caution is that `bwplot()` uses the model formula in a direction opposite than you might expect; you write `Segment ~ income`. We plot a horizontal box-and-whiskers for income by segment as follows:

```
> bwplot(Segment ~ income, data=seg.df, horizontal=TRUE, xlab = "Income")
```

The `lattice` box-and-whiskers is shown in Fig. 5.8.

We can break out home ownership as a conditioning variable using " | `ownHome`" in the formula:

```
> bwplot(Segment ~ income | ownHome, data=seg.df, horizontal=TRUE,
+        xlab="Income")
```

The conditioned plot for income by segment and home ownership is shown in Fig. 5.9. In this chart we discover—among other things—that in our simulated data the Travelers segment has a much wider distribution of income among those who own their homes than those who don't.

5.3 Learning More*

The topics in this chapter are foundational both for programming skills in R and for applied statistics. To gain skill in aspects of R programming that we introduce in this chapter, we recommend Matloff's *The Art of R Programming* [110].

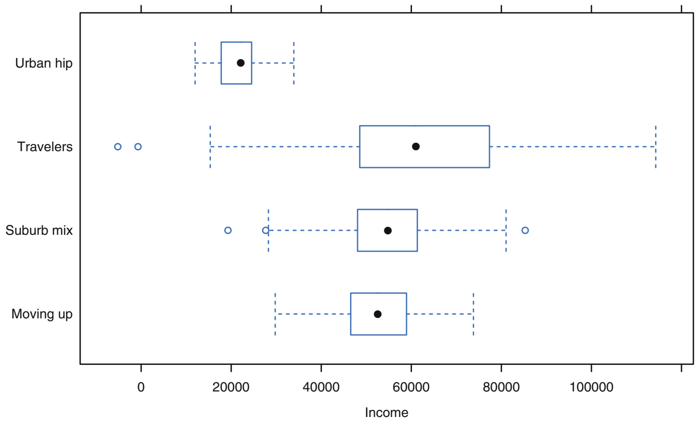

Fig. 5.8. Box-and-whiskers plot for income by segment using `bwplot`.

In Chap. 6 we continue our investigation with methods that formalize group comparisons and estimate the statistical strength of differences between groups.

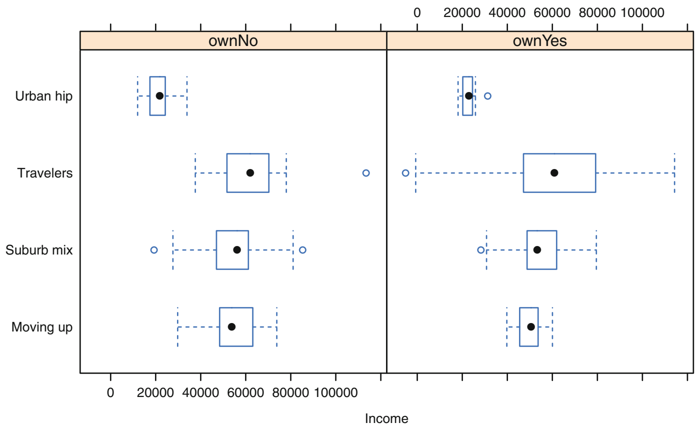

Fig. 5.9. Box-and-whiskers plot for income by segment and home ownership using `bwplot`.

5.4 Key Points

This was a crucial chapter for doing everyday analytics with R. Following are some of the lessons.

In R code in general:

- When writing `for()` loops, use `seq_along()` instead of `1:length()` (Sect. 5.1.2)
- For `if()` and `for()` blocks, always use brackets ("{" and "}") for improved readability and reliability (Sect. 5.1.3)
- When creating a data object from scratch, pre-populate it with missing data (`NA`) and then fill it in, for speed and reliability (Sect. 5.1.1)

When describing and visualizing data for groups:

- The `by()` command can split up data and automatically apply functions such as `mean()` and `summary()` (Sect. 5.2)
- `aggregate()` is even more powerful: it understands formula models and produces a reusable, indexable object with its results (Sects. 5.2 and 5.2.1)
- Frequency of occurrence can be found with `table()`. For count data, especially when using formulas, `xtabs()` is useful (Sect. 5.2.2)
- Charts of proportions and occurrence by a factor are well suited to the `lattice` package `histogram()` command (Sect. 5.2.2)
- Plots for continuous data by factor may use `barchart()`, or even better, box-and-whiskers plots with `boxplot()`. The `lattice` package extends such plots to multiple factors using formula specification and the `bwplot()` command (Sect. 5.2.4)