7.2: Transforming and Recoding a Variable

It's not uncommon in real world data analysis to find that one of your variables isn't quite equivalent to the variable that you really want. For instance, it's often convenient to take a continuous-valued variable (e.g., age) and break it up into a smallish number of categories (e.g., younger, middle, older). At other times, you may need to convert a numeric variable into a different numeric variable (e.g., you may want to analyse at the absolute value of the original variable). In this section I'll describe a few key tricks that you can make use of to do this.

7.2.1 Creating a transformed variable

The first trick to discuss is the idea of ***transforming*** a variable. Taken literally, *anything* you do to a variable is a transformation, but in practice what it usually means is that you apply a relatively simple mathematical function to the original variable, in order to create new variable that either (a) provides a better way of describing the thing you're actually interested in or (b) is more closely in agreement with the assumptions of the statistical tests you want to do. Since – at this stage – I haven't talked about statistical tests or their assumptions, I'll show you an example based on the first case.

To keep the explanation simple, the variable we'll try to transform (`likert.raw`) isn't inside a data frame, though in real life it almost certainly would be. However, I think it's useful to start with an example that doesn't use data frames because it illustrates the fact that you already know how to do variable transformations. To see this, let's go through an example. Suppose I've run a short study in which I ask 10 people a single question:

On a scale of 1 (strongly disagree) to 7 (strongly agree), to what extent do you agree with the proposition that "Dinosaurs are awesome"?

Now let's load and look at the data. The data file `likert.Rdata` contains a single variable that contains the raw Likert-scale responses:

```
load("./rbook-master/data/likert.Rdata")
likert.raw
```

```
##  [1] 1 7 3 4 4 4 2 6 5 5
```

However, if you think about it, this isn't the best way to represent these responses. Because of the fairly symmetric way that we set up the response scale, there's a sense in which the midpoint of the scale should have been coded as 0 (no opinion), and the two endpoints should be +3 (strong agree) and −3 (strong disagree). By recoding the data in this way, it's a bit more reflective of how we really think about the responses. The recoding here is trivially easy: we just subtract 4 from the raw scores:

```
likert.centred <- likert.raw - 4
likert.centred
```

```
##  [1] -3  3 -1  0  0  0 -2  2  1  1
```

One reason why it might be useful to have the data in this format is that there are a lot of situations where you might prefer to analyse the *strength* of the opinion separately from the *direction* of the opinion. We can do two different transformations on this `likert.centred` variable in order to distinguish between these two different concepts. Firstly, to compute an `opinion.strength` variable, we want to take the absolute value of the centred data (using the `abs()` function that we've seen previously), like so:

```
opinion.strength <- abs( likert.centred )
opinion.strength
```

```
##  [1] 3 3 1 0 0 0 2 2 1 1
```

Secondly, to compute a variable that contains only the direction of the opinion and ignores the strength, we can use the `sign()` function to do this. If you type `?sign` you'll see that this function is really simple: all negative numbers are converted to −1, all positive numbers are converted to 1 and zero stays as 0. So, when we apply the `sign()` function we obtain the following:

```
opinion.dir <- sign( likert.centred )
opinion.dir
```

```
##  [1] -1  1 -1  0  0  0 -1  1  1  1
```

And we're done. We now have three shiny new variables, all of which are useful transformations of the original `likert.raw` data. All of this should seem pretty familiar to you. The tools that you use to do regular calculations in R (e.g., Chapters 3 and 4) are very much the same ones that you use to transform your variables! To that end, in Section 7.3 I'll revisit the topic of doing calculations in R because there's a lot of other functions and operations that are worth knowing about.

Before moving on, you might be curious to see what these calculations look like if the data had started out in a data frame. To that end, it may help to note that the following example does all of the calculations using variables inside a data frame, and stores the variables created inside it:

```
df <- data.frame( likert.raw )                  # create data frame
df$likert.centred <- df$likert.raw - 4          # create centred data
df$opinion.strength <- abs( df$likert.centred )  # create strength variable
df$opinion.dir <- sign( df$likert.centred )      # create direction variable
df
```

```
##    likert.raw likert.centred opinion.strength opinion.dir
## 1           1             -3                3          -1
## 2           7              3                3           1
## 3           3             -1                1          -1
## 4           4              0                0           0
## 5           4              0                0           0
## 6           4              0                0           0
## 7           2             -2                2          -1
## 8           6              2                2           1
## 9           5              1                1           1
## 10          5              1                1           1
```

In other words, the commands you use are basically ones as before: it's just that every time you want to read a variable from the data frame or write to the data frame, you use the `$` operator. That's the easiest way to do it, though I should make note of the fact that people sometimes make use of the `within()` function to do the same thing. However, since (a) I don't use the `within()` function anywhere else in this book, and (b) the `$` operator works just fine, I won't discuss it any further.

7.2.2 Cutting a numeric variable into categories

One pragmatic task that arises more often than you'd think is the problem of cutting a numeric variable up into discrete categories. For instance, suppose I'm interested in looking at the age distribution of people at a social gathering:

```
age <- c( 60,58,24,26,34,42,31,30,33,2,9 )
```

In some situations it can be quite helpful to group these into a smallish number of categories. For example, we could group the data into three broad categories: young (0-20), adult (21-40) and older (41-60). This is a quite coarse-grained classification, and the labels that I've attached only make sense in the context of this data set (e.g., viewed more generally, a 42 year old wouldn't

consider themselves as "older"). We can slice this variable up quite easily using the `cut()` function.[105] To make things a little cleaner, I'll start by creating a variable that defines the boundaries for the categories:

```
age.breaks <- seq( from = 0, to = 60, by = 20 )
age.breaks
```

```
## [1]  0 20 40 60
```

and another one for the labels:

```
age.labels <- c( "young", "adult", "older" )
age.labels
```

```
## [1] "young" "adult" "older"
```

Note that there are four numbers in the `age.breaks` variable, but only three labels in the `age.labels` variable; I've done this because the `cut()` function requires that you specify the *edges* of the categories rather than the mid-points. In any case, now that we've done this, we can use the `cut()` function to assign each observation to one of these three categories. There are several arguments to the `cut()` function, but the three that we need to care about are:

- `x` . The variable that needs to be categorised.
- `breaks` . This is either a vector containing the locations of the breaks separating the categories, or a number indicating how many categories you want.
- `labels` . The labels attached to the categories. This is optional: if you don't specify this R will attach a boring label showing the range associated with each category.

Since we've already created variables corresponding to the breaks and the labels, the command we need is just:

```
age.group <- cut( x = age,               # the variable to be categorised
                  breaks = age.breaks,   # the edges of the categories
                  labels = age.labels )  # the labels for the categories
```

Note that the output variable here is a factor. In order to see what this command has actually done, we could just print out the `age.group` variable, but I think it's actually more helpful to create a data frame that includes both the original variable and the categorised one, so that you can see the two side by side:

```
data.frame(age, age.group)
```

```
##    age age.group
## 1   60     older
## 2   58     older
## 3   24     adult
## 4   26     adult
## 5   34     adult
## 6   42     older
## 7   31     adult
## 8   30     adult
## 9   33     adult
## 10   2     young
## 11   9     young
```

It can also be useful to tabulate the output, just to see if you've got a nice even division of the sample:

```
table( age.group )
```

```
## age.group
## young adult older
##     2     6     3
```

In the example above, I made all the decisions myself. Much like the `hist()` function that we saw in Chapter 6, if you want to you can delegate a lot of the choices to R. For instance, if you want you can specify the *number* of categories you want, rather than giving explicit ranges for them, and you can allow R to come up with some labels for the categories. To give you a sense of how this works, have a look at the following example:

```
age.group2 <- cut( x = age, breaks = 3 )
```

With this command, I've asked for three categories, but let R make the choices for where the boundaries should be. I won't bother to print out the `age.group2` variable, because it's not terribly pretty or very interesting. Instead, all of the important information can be extracted by looking at the tabulated data:

```
table( age.group2 )
```

```
## age.group2
## (1.94,21.3] (21.3,40.7] (40.7,60.1]
##           2           6           3
```

This output takes a little bit of interpretation, but it's not complicated. What R has done is determined that the lowest age category should run from 1.94 years up to 21.3 years, the second category should run from 21.3 years to 40.7 years, and so on. The formatting on those labels might look a bit funny to those of you who haven't studied a lot of maths, but it's pretty simple. When R describes the first category as corresponding to the range (1.94,21.3] what it's saying is that the range consists of those numbers that are larger than 1.94 but less than *or equal to* 21.3. In other words, the weird asymmetric brackets is R s way of telling you that if there happens to be a value that is exactly equal to 21.3, then it belongs to the first category, not the second one. Obviously, this isn't actually possible since I've only specified the ages to the nearest whole number, but R doesn't know this and so it's trying to be precise just in case. This notation is actually pretty standard, but I suspect not everyone reading the book will have seen it before. In any case, those labels are pretty ugly, so it's usually a good idea to specify your own, meaningful labels to the categories.

Before moving on, I should take a moment to talk a little about the mechanics of the `cut()` function. Notice that R has tried to divide the `age` variable into three roughly equal sized bins. Unless you specify the particular breaks you want, that's what it will do. But suppose you want to divide the `age` variable into three categories of different size, but with approximately identical numbers of people. How would you do that? Well, if that's the case, then what you want to do is have the breaks correspond to the 0th, 33rd, 66th and 100th percentiles of the data. One way to do this would be to calculate those values using the `quantiles()` function and then use those quantiles as input to the `cut()` function. That's pretty easy to do, but it does take a couple of lines to type. So instead, the `lsr` package has a function called `quantileCut()` that does exactly this:

```
age.group3 <- quantileCut( x = age, n = 3 )
table( age.group3 )
```

```
## age.group3
## (1.94,27.3] (27.3,33.7] (33.7,60.1]
##           4           3           4
```

Notice the difference in the boundaries that the `quantileCut()` function selects. The first and third categories now span an age range of about 25 years each, whereas the middle category has shrunk to a span of only 6 years. There are some situations where this is genuinely what you want (that's why I wrote the function!), but in general you should be careful. Usually the numeric variable that you're trying to cut into categories is already expressed in meaningful units (i.e., it's interval scale), but if you cut it into unequal bin sizes then it's often very difficult to attach meaningful interpretations to the resulting categories.

More generally, regardless of whether you're using the original `cut()` function or the `quantileCut()` version, it's important to take the time to figure out whether or not the resulting categories make any sense at all in terms of your research project. If they don't make any sense to you as meaningful categories, then any data analysis that uses those categories is likely to be just as meaningless. More generally, in practice I've noticed that people have a very strong desire to carve their (continuous and messy) data into a few (discrete and simple) categories; and then run analysis using the categorised data instead of the original one.[106] I wouldn't go so far as to say that this is an inherently bad idea, but it does have some fairly serious drawbacks at times, so I would advise some caution if you are thinking about doing it.
