Chapter 2
Reading and Transforming Data Format

2.1 Reading and Transforming Data

2.1.1 Data Layout

R, like Splus and S, represents an entire conceptual system for thinking about data. You may need to learn some new ways of thinking. One way that is new for users of Systat, SAS, and (probably) SPSS concerns two different ways of laying out a data set. In the Systat way, each subject is a row (which may be continued on the next row if too long, but still conceptually a row) and each variable is a column. You can do this in R too, and most of the time it is sufficient.

But some the features of R will not work with this kind of representation, in particular, repeated-measures analysis of variance or hierarchical linear modeling. So you need a second way of representing data, which is that each row represents a single datum, e.g., one subject's answer to one question. The row also contains an identifier for all the relevant classifications, such as the question number, the subscale that the question is part of, and the subject. Thus, "subject" becomes a category with no special status, technically a factor (and remember to make sure it is a factor, lest you find yourself studying the effect of the subject's number).

The former is referred to as the *wide* layout and the latter the *long* layout to be consistent with the terms used by the `reshapde()` function.

2.1.2 A Simple Questionnaire Example

Let us start with an example of the old-fashioned way. In the file `ctest3.data`, each subject is a row, and there are 134 columns. The first four are age, sex, student status, and time to complete the study. The rest are the responses to four questions

Y. Li and J. Baron, *Behavioral Research Data Analysis with R*, Use R,
DOI 10.1007/978-1-4614-1238-0_2, © Springer Science+Business Media, LLC 2012

about each of 32 cases. Each group of four is preceded by the trial order, but this is ignored for now.

```
> c0 <- read.table(file = "ctest3.data")
```

The file can be downloaded from the data file has no labels, so we can read it with `read.table`. You can also try `read.csv` or `read.delim`. The `file` parameter can be ignored. If the data file is found online then the `file` parameter can be the complete URL address to that file.

```
> age1 <- c0[,1]
> sex1 <- c0[,2]
> student1 <- c0[,3]
> time1 <- c0[,4]
> nsub1 <- nrow(c0)
```

We can refer to elements of `c0` by `c0[row,column]`. For example, `c0[1,2]` is the sex of the first subject. We can leave one part blank and get all of it, e.g., `c0[,2]` is a vector (column of numbers) representing the sex of all the subjects. The last line defines `nsub1` as the number of subjects.

```
> c1 <- as.matrix(c0[,4+1:128])
```

Now `c1` is the main part of the data, the matrix of responses. The expression 1:128 is a vector, which expands to 1 2 3 ... 128. By adding 4, it becomes 5 6 7 ... 132.

2.1.2.1 Extracting Subsets of Data

```
> rsp1 <- c1[,4*c(1:32)-2]
> rsp2 <- c1[,4*c(1:32)-1]
```

The above two lines illustrate the extraction of sub-matrices representing answers to two of the four questions making up each item. The matrix `rsp1` has 32 columns, corresponding to columns 2 6 10 ... 126 of the original 128-column matrix `c1`. The matrix `rsp2` corresponds to 3 7 11 ... 127.

Another way to do this is to use an array. We could say `a1 <- array(c1, c(ns, 4, 32))`. Then `a1[,1,]` is the equivalent of `rsp1`, and `a1[20,1,]` is `rsp1` for subject 20. To see how arrays print out, try the following:

```
> m1 <- matrix(1:60,5,)
> a1 <- array(m1,c(5,2,6))
> m1
> a1
```

You will see that the rows of each table are the first index and the columns are the second index. Arrays seem difficult at first, but they are very useful for this sort of analysis.

2.1.2.2 Finding Means (or Other Things) of Sets of Variables

```
> r1mean <- apply(rsp1,1,mean)
> r2mean <- apply(rsp2,1,mean)
```

The above lines illustrate the use of `apply` for getting means of subscales. In particular, `abrmean` is the mean of the subscale consisting of the answers to the second question in each group. The `apply` function works on the data in its first argument, then applies the function in its third argument, which, in this case, is `mean`. (It can be `max` or `min` or any defined function.) The second argument is 1 for rows, 2 for columns (and so on, for arrays). We want the function applied to rows.

```
> r4mean <- apply(c1[,4*c(1:32)], 1, mean)
```

The expression here represents the matrix for the last item in each group of four. The first argument can be any matrix or data frame. (The output for a data frame will be labeled with row or column names.) For example, suppose you have a list of variables such as `q1, q2, q3`, etc. Each is a vector, whose length is the number of subjects. The average of the first three variables for each subject is `apply(cbind(q1,q2,q3),1,mean)`. (This is the equivalent of the Systat expression `avg(q1, q2, q3)`. A little more verbose, to be sure, but much more flexible.)

You can use apply to tabulate the values of each column of a matrix `m1`: `apply(m1, 2, table)`. Or, to find column means, `apply(m1, 2, mean)`.

There are many other ways to make tables. Some of the relevant functions are `table`, `tapply`, `sapply`, `ave`, and `by`. Here is an illustration of the use of `by`. Suppose you have a matrix `m1` like this:

```
1 2 3 4
4 4 5 5
5 6 4 5
```

The columns represent the combination of two variables, `y1` is `0 0 1 1`, for the four columns, respectively, and `y2` is `0 1 0 1`. To get the means of the columns for the two values of `y1`, say `by(t(m1), y1, mean)`. You get 3.67 and 4.33 (labeled appropriately by values of y1). You need to use `t(m1)` because `by` works by rows. If you say `by(t(m1), data.frame(y1,y2), mean)`, you get a cross tabulation of the means by both factors. (This is, of course, the means of the four columns of the original matrix.)

Of course, you can also use `by` to classify rows; in the usual examples, this would be groups of subjects rather than classifications of variables.

2.1.2.3 One Row Per Observation

The next subsection shows how to transform the data from the *wide* layout (one row per subject) to the *long* layout (one row per observation). We will use the matrix

rsp1, which has 32 columns and one row per subject. Here are the data from five subjects:

```
1 1 2 2 1 2 3 5 2 3 2 4 2 5 7 7 6 6 7 5 7 8 7 9 8 8 9 9 8 9 9 9
1 2 3 2 1 3 2 3 2 3 2 3 2 3 2 4 1 2 4 5 4 5 5 6 5 6 6 7 6 7 7 8
1 1 2 3 1 2 3 4 2 3 3 4 2 4 3 4 4 4 5 5 5 6 6 7 6 7 7 8 7 7 8 8
1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9 9 9
1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9
```

We will create a matrix with one row per observation. The first column will contain the observations, one variable at a time, and the remaining columns will contain numbers representing the subject and the level of the observation on each variable of interest. There are two such variables here, r2 and r1. The variable r2 has four levels, 1 2 3 4, and it cycles through the 32 columns as 1 2 3 4 1 2 3 4 . . . The variable r1 has the values (for successive columns) 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4. These levels are ordered. They are not just arbitrary labels. (For that, we would need the factor function.)

```
> r2 <- rep(1:4,8)
> r1 <- rep(rep(1:4,rep(4,4)),2)
```

The above two lines create vectors representing the levels of each variable for each subject. The rep command for r2 says to repeat the sequence 1 2 3 4, eight times. The rep command for r1 says take the sequence 1 2 3 4, then repeat the first element four times, the second element four times, etc. It does this by using a vector as its second argument. That vector is rep(4,4), which means repeat the number 4, four times. So rep(4,4) is equivalent to c(4 4 4 4). The last argument, 2, in the command for r1 means that the whole sequence is repeated twice. Notice that r1 and r2 are the codes for one row of the matrix rsp1.

```
> nsub1 <- nrow(rsp1)
> subj1 <- as.factor(rep(1:nsub1,32))
```

nsub1 is just the number of subjects (5 in the example), the number of rows in the matrix rsp1. The vector subj1 is what we will need to assign a subject number to each observation. It consists of the sequence 1 2 3 4 5, repeated 32 times. It corresponds to the columns of rsp1.

```
> abr1 <- data.frame(ab1 = as.vector(rsp1),
+ sub1 = subj1, dcost1 = rep(r1,rep(nsub1,32)),
+ abcost1 = rep(r2,rep(nsub1,32)))
```

The data.frame function puts together several vectors into a data frame, which has rows and columns like a matrix.[1] Each vector becomes a column. The as.vector function reads down by columns, that is, the first column, then the

[1]The cbind function does the same thing but makes a matrix instead of a data frame.

second, and so on. So ab is now a vector in which the first nsub1 elements are the same as the first column of rsp1, that is, 1 1 1 1 1. The first 15 elements of ab are: 1 1 1 1 1 1 2 1 2 1 2 3 2 2 1. Notice how we can define names within the arguments to the data.frame function. Of course, sub1 now represents the subject number of each observation. The first ten elements of sub1 are 1 2 3 4 5 1 2 3 4 5. The variable abcost1 now refers to the value of r2. Notice that each of the 32 elements of r2 is repeated nsub1 times. Thus, the first 15 values of abcost1 are 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3. Here are the first ten rows of abr1:

```
      ab1 sub1 dcost1 abcost1
1     1   1      1       1
2     1   2      1       1
3     1   3      1       1
4     1   4      1       1
5     1   5      1       1
6     1   1      1       2
7     2   2      1       2
8     1   3      1       2
9     2   4      1       2
10    1   5      1       2
```

The following line makes a table of the means of abr1, according to the values of dcost1 (rows) and abcost1 (columns).

```
> ctab1 <- tapply(abr1[,1],list(abr1[,3],abr1[,4]),
   mean)
```

It uses the function tapply, which is like the apply function except that the output is a table. The first argument is the vector of data to be used. The second argument is a list supplying the classification in the table. This list has two columns corresponding to the columns of abr representing the classification. The third argument is the function to be applied to each grouping, which in this case is the mean. Here is the resulting table:

```
     1   2   3   4
1 2.6 3.0 3.7 3.8
2 3.5 4.4 4.4 5.4
3 4.5 5.2 5.1 5.9
4 5.1 6.1 6.2 6.8
```

The following line provides a plot corresponding to the table.

```
> matplot(ctab1, type = "l")
```

Type l means lines. Each line plots the four points in a column of the table. If you want it to go by rows, use t(ctab1) instead of ctab1. The function t() transposes rows and columns.

Finally, the following line does a regression of the response on the two classifiers, actually an analysis of variance.

```
> summary(aov(ab1 ~ dcost1 + abcost1 +
+       Error(sub1/(dcost1 + abcost1)), data = abr))
```

The function `aov`, like `lm`, fits a linear model, because `dcost1` and `abcost1` are numerical variables, not factors (although `sub1` is a factor). The model is defined by its first argument (to the left of the comma), where `~` separates the dependent variable from the predictors. The second element defines the data frame to be used. The `summary` function prints a summary of the regression. (The `lm` and `aov` objects themselves contains other things, such as residuals, many of which are not automatically printed.) We explain the `Error` term later in Sect. 5.1, but the point of it is to make sure that we test against random variation due to subjects, that is, test "across subjects." Here is some of the output, which shows significant effects of both predictors:

```
Error: sub1
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals  4 52.975  13.244

Error: sub1:dcost1
          Df  Sum Sq Mean Sq F value    Pr(>F)
dcost1     1 164.711 164.711  233.63 0.0001069 ***
Residuals  4   2.820   0.705
---

Error: sub1:abcost1
          Df Sum Sq Mean Sq F value   Pr(>F)
abcost1    1 46.561  46.561    41.9 0.002935 **
Residuals  4  4.445   1.111
---

Error: Within
           Df Sum Sq Mean Sq F value Pr(>F)
Residuals 145 665.93    4.59
```

Note that, in many examples in this section, we used `rep()` to generate repeated values. We can also use the `gl()` function for this. For example, instead of `subj1 <- as.factor(rep(1:nsub1,32))`, we could say `subj1 <- gl(nsub1,1,nsub1*32)`. The first argument specifies the number of levels, which in this case is the number of subjects. The second argument specifies the number of immediate repetitions of each level (within each cycle, when there are cycles – not the total number of repetitions), and the third argument specifies the total length, which is here the number of subjects times the number of items. If we wanted a code for each item, we could say `gl(32,nsub1,nsub1*32)`. But here

we do not need the last argument because there is only one cycle. Each item number is immediately repeated `nsub1` times. Thus, we could say `gl(32,nsub1)`. The `gl()` function is useful because it avoids having to say `as.factor`, which is often forgotten.

2.1.3 Other Ways to Read in Data

First example. Here is another example of creating a matrix with one row per observation.

```
> symp1 <- read.table("symp1.data",header=T)
> sy1 <- as.matrix(symp1[,c(1:17)])
```

The first 17 columns of `symp1` are of interest. The file `symp1.data` contains the names of the variables in its first line. The `header=T` (an abbreviation for `header=TRUE`) makes sure that the names are used; otherwise the variables will be names `V1, V2,` etc.

```
> gr1 <- factor(symp1$group1)
```

The variable `group1`, which is in the original data, is a factor that is unordered.

The next four lines create the new matrix, defining identifiers for subjects and items in a questionnaire.

```
> syv1 <- as.vector(sy1)
> subj1 <- factor(rep(1:nrow(sy1),ncol(sy1)))
> item <- factor(rep(1:ncol(sy1),rep(nrow(sy1),
+       ncol(sy1))))
> grp <- rep(gr1,ncol(sy1))
> cgrp <- ((grp==2) | (grp==3))+0
```

The variable `cgrp` is a code for being in `grp` 2 or 3. The reason for adding 0 is to make the logical vector of `T` and `F` into a numeric vector of 1 and 0.

The following three lines create a table from the new matrix, plot the results, and report the results of an analysis of variance.

```
> sytab <- tapply(syv,list(item,grp),mean)
> matplot(sytab,type="l")
> svlm <- aov(syv ~ item + grp + item*grp)
```

Second example. In the next example, the data file has labels. We want to refer to the labels as if they were variables we had defined, so we use the `attach` function.

```
> t9 <- read.table("tax9.data",header=T)
> attach(t9)
```

Third example. In the next example, the data file has no labels, so we can read it with `scan`. The `scan` function just reads in the numbers and makes them into a vector, that is, a single column of numbers.

```
> abh1 <- matrix(scan("abh1.data"),,224,byrow=T))
```

We then apply the `matrix` command to make it into a matrix. (There are many other ways to do this.) We know that the matrix should have 224 columns, the number of variables, so we should specify the number of columns. If you say help(matrix) you will see that the matrix command requires several arguments, separated by commas. The first is the vector that is to be made into a matrix, which in this case is `scan("abh1.data")`. We could have given this vector a name, and then used its name, but there is no point. The second and third arguments are the number of rows and the number of columns. We can leave the number of rows blank. (That way, if we add or delete subjects, we do not need to change anything.) The number of columns is 224. By default, the matrix command fills the matrix by columns, so we need to say `byrow=TRUE` or `byrow=T` to get it to fill by rows, which is what we want. (Otherwise, we could just leave that field blank.)

We can refer to elements of `abh1` by `abh1[row,column]`. For example, `abh1[1,2]` is the sex of the first subject. We can leave one part blank and get all of it, e.g., `abh1[,2]` is a vector (column of numbers) representing the sex of all the subjects.

2.1.4 Other Ways to Transform Variables

2.1.4.1 Contrasts

Suppose you have a matrix `t1` with four columns. Each row is a subject. You want to contrast the mean of columns 1 and 3 with the mean of columns 2 and 4. A *t*-test would be fine. (Otherwise, this is the equivalent of the `cmatrix` command in Systat.) Here are three ways to do it. The first way calculates the mean of the columns 1 and 3 and subtracts the mean of columns 2 and 4. The result is a vector. When we apply `t.test()` to a vector, it tests whether the mean of the values is different from 0.

```
> t1 <- matrix(rnorm(40), ncol = 4)
> t.test(apply(t1[c(1,3),], 2, mean) -
+ apply(t1[c(2, 4), ], 2, mean))
```

The second way multiplies the matrix by a vector representing the contrast weights, `1, -1, 1, -1`. Ordinary multiplication of a matrix by a vector multiplies the rows, but we want the columns, so we must apply `t()` to transform the matrix, and then transform it back.

```
> t.test(t(t(t1)*c(1,-1,1,-1)))
```

or

```
> contr1 <- c(1,-1,1,-1)
> t.test(t(t(t1)*contr1))
```

The third way is the most elegant. It uses matrix multiplication to accomplish the same thing.

```
> contr1 <- c(1,-1,1,-1)
> t.test(t1 %*% contr1)
```

2.1.4.2 Averaging Items in a Within-Subject Design

Suppose we have a matrix `t2`, with 32 columns. Each row is a subject. The 32 columns represent a 8x4 design. The first eight columns represent eight different levels of the first variable, at the first level of the second variable. The next eight columns are the second level of the second variable, etc. Suppose we want a matrix in which the columns represent the eight different levels of the first variable, averaged across the second variable.

First method: loop

One way to do it – inelegantly but effectively – is with a loop. First, we set up the resulting matrix. (We cannot put anything in it this way if it doesn't exist yet.)

```
> m2 <- t2[,c(1:8)]*0
```

The idea here is just to make sure that the matrix has the right number of rows, and all 0's. Now here is the loop:

```
> for (i in 1:8) m2[,i] <- apply(t2[,i+c(8*0:3)],1,
    mean)
```

Here, the index `i` is stepped through the columns of `m2`, filling each one with the mean of four columns of `t2`. For example, the first column of `m2` is the mean of columns 1, 9, 17, and 25 of `t2`. This is because the vector `c(8*0:3)` is 0, 8, 16, 24. The `apply` function uses `1` as its second argument, which means to apply the function `mean` across *rows*.

Second method: matrix multiplication

Now here is a more elegant way, but one that requires an auxiliary matrix, which may use memory if that is a problem. This time we want the means according to the second variable, which has four levels, so we want a matrix with four columns. We will multiply the matrix `t2` by an auxiliary matrix `c0`.

The matrix `c0` has 32 rows and four columns. The first column is 1,1,1,1,1,1,1,1 followed by 24 0's. This is the result of `rep(c(1,0,0,0),rep(8,4))`, which repeats each of the elements of `1,0,0,0` eight times (since `rep(8,4)` means `8,8,8,8`). The second column is 8 0's, 8 1's, and 16 0's.

```
> c0 <- cbind(rep(c(1,0,0,0), rep(8,4)), rep(c(0,1,0,0),
+             rep(8,4)), rep(c(0,0,1,0), rep(8,4)),
+             rep(c(0,0,0,1), rep(8,4)))
> c2 <- t2 %*% c0
```

The last line above uses matrix multiplication to create the matrix `c2`, which has four columns and one row per subject. Note that the order here is important; switching `t2` and `c0` will not work.

2.1.4.3 Selecting Cases or Variables

There are several other ways for defining new matrices or data frames as subsets of other matrices or data frames.

One very useful function is `which()`, which yields the indices for which its argument is true. For example, the output of `which(3:10 > 4)` is the vector `3 4 5 6 7 8`, because the vector `3:10` has a length of 8, and the first two places in it do not meet the criterion that their value is greater than 4. With `which()`, you can use a vector to select rows or columns from a matrix (or data frame). For example, suppose you have nine variables in a matrix `m9` and you want to select three submatrices, one consisting of variables 1, 4, 7, another with 2, 5, 8, and another with 3, 6, 9. Define `mvec` so that it is the vector `1 2 3 1 2 3 1 2 3`.

```
> m9 <- matrix(rnorm(90), ncol = 9)
> mvec9 <- rep(1:3,3)
> m9a <- m9[,which(mvec9 == 1)]
> m9b <- m9[,which(mvec9 == 2)]
> m9c <- m9[,which(mvec9 == 3)]
```

You can use the same method to select subjects by any criterion, putting the `which()` expression before the comma rather than after it, so that it indicates rows.

2.1.4.4 Recoding and Replacing Data

Suppose you have `m1` a matrix of data in which 99 represents missing data, and you want to replace each 99 with `NA`. Simply say `m1[m1==99] <- NA`. Note that this will work only if `m1` is a matrix (or vector), not a data frame (which could result from a `read.table()` command). You might need to use the `as.matrix()` function first.

Sometimes you want to recode a variable, e.g., a column in a matrix. If `q1[,3]` is a 7-point scale and you want to reverse it, you can say

```
> q1[,3] <- 8 - q1[,3]
```

In general, suppose you want to recode the numbers 1,2,3,4,5 so that they come out as 1,5,3,2,4, respectively. You have a matrix `m1`, containing just the numbers 1 through 5. You can say

```
> c1 <- c(1,5,3,2,4)
> apply(m1,1:2,function(x) c1[x])
```

In this case `c1[x]` is just the value at the position indicated by `x`.

Suppose that, instead of 1 through 5, you have A through E, so that you cannot use numerical positions. You want to convert A,B,C,D,E to 1,5,3,2,4, respectively. You can use two vectors:

```
> c1 <- c(1,5,3,2,4)
> n1 <- c("A","B","C","D","E")
> apply(m1,1:2,function(x) c1[which(n1)==x])
```

Or, alternatively, you can give names to `c1` instead of using a second vector:

```
> c1 <- c(1,5,3,2,4)
> names(c1) <- c("A","B","C","D","E")
> apply(m1,1:2,function(x) c1[x])
```

The same general idea will work for arrays, vectors, etc., instead of matrices.

Here are some other examples, which may be useful in simple cases, or as illustrations of various tricks.

In this example, `q2[,c(2,4)]` are two columns that must be recoded by switching 1 and 2 but leaving responses of 3 or more intact. To do this, say

```
> q2[,c(2,4)] <- (q2[,c(2,4)] < 3) * (3 - q2[,c(2,4)]) +
+     (q2[,c(2,4)] >= 3) * q2[,c(2,4)]
```

Here the expression `q2[,c(2,4)] < 3` is a two-column matrix full of `TRUE` and `FALSE`. By putting it in parentheses, you can multiply it by numbers, and `TRUE` and `FALSE` are treated as 1 and 0, respectively. Thus, `(q2[,c(2,4)] < 3) * (3 - q2[,c(2,4)])` switches 1 and 2, for all entries less than 3. The expression `(q2[,c(2,4)] >= 3) * q2[,c(2,4)]` replaces all the other values, those greater than or equal to 3, with themselves.

Here is an example that will switch 1 and 3, 2 and 4, but leave 5 unchanged, for columns 7 and 9

```
> q3[,c(7,9)] <- (q3[,c(7,9)]==1)*3 +
+   (q3[,c(7,9)]==2)*4 + (q3[,c(7,9)]==3)*1 +
+   (q3[,c(7,9)]==4)*2 + (q3[,c(7,9)]==5)*5
```

Notice that this works because everything on the right of `<-` is computed on the values in `q3` before any of these values are replaced.

2.1.4.5 Replacing Characters with Numbers

Sometimes you have questionnaire data in which the responses are represented as (for example) "y" and "n" (for yes and no). Suppose you want to convert these to numbers so that you can average them. The following command does this for a matrix `q1`, whose entries are `y`, `n`, or some other character for "unsure." It converts `y` to 1 and `n` to −1, leaving 0 for the "unsure" category.

```
> q1 <- (q1[,]=="y") - (q1[,]=="n")
```

In essence, this works by creating two new matrices and then subtracting one from the other, element by element.

A related issue is how to work with date and time variables. A timestamp value like "2009-02-01 15:22:35" is typically shown as a character string in a spreadsheet program. Character variables of date and time can be converted into `DateTimeClasses`.

```
> x <- c("2008-02-28 15:22:35", "2008-03-01 15:30:35")
> fmt <- "%Y-%m-%d %H:%M:%S"
> y <- strptime(x, format = fmt)
> y[2] - y[1]
Time difference of 2.0056 days
```

Note the time difference of approximately 2 days because there are 29 days in February 2008. The `strptime()` function filters the character variable `x` by a specific timestamp format. A "mm/dd/yyyy" date would need a format of `"%m/%d/%Y"`, and a "mm/dd/yy" date would need `"%m/%d/%y"`.

Timestamp variables are often imported from the text output of a spreadsheet program. Text variables imported through `read.csv()` and `read.table()` are automatically converted into factors when the imported data are turned into a data frame. `strptime()` does not accept factors. One workaround is to deactivate the automatic conversion by setting `read.csv(..., as.is = TRUE)`.

2.1.5 Using R to Compute Course Grades

Here is an example that might be useful and instructive. Suppose you have a set of grades including a midterm with two parts `m1` and `m2`, a final with two parts, and two assignments. You told the students that you would standardize the midterm scores, the final scores, and each of the assignment scores, then compute a weighted sum to determine the grade. Here, with comments, is an R file that does this. The critical line is the one that standardizes and computes a weighted sum, all in one command.

```
> g1 <- read.csv("grades.csv",header=F)
> a1 <- as.vector(g1[,4])
```

```
> m1 <- as.vector(g1[,5])
> m2 <- as.vector(g1[,6])
> a2 <- as.vector(g1[,7])
> f1 <- as.vector(g1[,8])
> f2 <- as.vector(g1[,9])
> a1[a1=="NA"] <- 0 # missing assignment 1 gets a 0
> m <- 2*m1+m2 # compute midterm score from the parts
> f <- f1+f2
> gdf <- data.frame(a1,a2,m,f)
> gr <- apply(t(scale(gdf))*c(.10,.10,.30,.50),2,sum)
 # The last line standardizes the scores and computes
 # their weighted sum.
 # The weights are .10, .10, .30, and .50 for
 # a1, a2, m, and f
> gcut <- c(-2,-1.7,-1.4,-1.1,-.80,-.62,-.35,-.08,.16,
+         .40,.72,1.1,2)
 # The last line defines cutoffs for letter grades.
> glabels <- c("f","d","d+","c-","c","c+","b-","b",
+            "b+","a-","a","a+")
> gletter <- cut(gr,gcut,glabels) # letter grades
> grd <- cbind(g1[,1:2],round(gr,digits=4),gletter)
 # gl[,1:2] are students' names
> grd[order(gr),] # sorts & prints matrix in rank order
> round(table(gletter)/.83,1) # prints, with rounding
 # the .83 is because there are 83 students
> gcum <- as.vector(round(cumsum(table(gletter)/.83),1))
> names(gcum) <- glabels
> gcum # cumulative sum of students w/ different grades
```

2.2 Reshape and Merge Data Frames

The `reshape()` function reshapes a data frame between the wide and long layouts.

```
> data1 <- c(
+    49,47,46,47,48,47,41,46,43,47,46,45,
+    48,46,47,45,49,44,44,45,42,45,45,40,
+    49,46,47,45,49,45,41,43,44,46,45,40,
+    45,43,44,45,48,46,40,45,40,45,47,40)
> data1 <- data.frame(subj = paste("s", 1:12, sep=""),
+  matrix(data1, ncol = 4))
> names(data1) <- c("subj","sq.red", "circ.red",
+       "sq.blue", "circ.blue")
> data1
```

```
    subj sq.red circ.red sq.blue circ.blue
1     s1     49       48      49        45
2     s2     47       46      46        43
3     s3     46       47      47        44
4     s4     47       45      45        45
5     s5     48       49      49        48
6     s6     47       44      45        46
7     s7     41       44      41        40
8     s8     46       45      43        45
9     s9     43       42      44        40
10   s10     47       45      46        45
11   s11     46       45      45        47
12   s12     45       40      40        40
```

The data come from a hypothetical study of reaction time in working with control panels of different shape (square and circle) and color (red and blue). Each subject works with all all types of controls and the reaction time is collected. Details of this example are described in Sect. 5.1.

You can tell `reshape` to convert columns 2 through 5 into a single long variable called `rt`, with 4 records per `subj`.

```
> data1.long <- reshape(data1, direction = "long",
+  idvar = "subj", varying= 2:5, v.names = "rt")
```

The command takes columns 2 through 5 (`varying = 2:5`) and collapses them into a single variable called `v.names = "rt"` in the long format (`direction = "long"`). The `varying` option can be variable names, e.g., `varying = c("sq.red", "circ.red", "sq.blue", "circ.blue")`. The `subj` ids are repeated in the long format. A new variable (named `time` by default) is created to index the collapsed columns. The index values of 1, 2, 3, and 4 represent the second (`sq.red`) through the 5th columns (`circ.blue`), respectively. The default variable name `time` can be changed by specifying `timevar = "groups"` if you want the new variable be named as `groups`. To convert `data1.long` back into the wide format, type `reshape(data1.long, direction = "wide", ids = "subj")`. We will see this data frame again in Sect. 5.1 when we deal with repeated-measures ANOVA.

Another useful function is `merge()`, which joins data frames. Suppose you have in a separate data frame the gender information of subjects 1 through 9. By default the two data frames are matched by common variable(s), in this case one single variable `subj`.

```
subj.char <- data.frame(subj = paste("s", 1:9,
   sep = ""), sex = c("F","F","M","F","F","F","M",
   "M","M"))
merge(x = data1.long, y = subj.char, all = TRUE)
```

```
    subj time rt  sex
1     s1    1 49    F
2     s1    2 48    F
3     s1    3 49    F
4     s1    4 45    F
5    s10    2 45 <NA>
6    s10    1 47 <NA>
7    s10    4 45 <NA>
8    s10    3 46 <NA>
...
15   s12    4 40 <NA>
16   s12    3 40 <NA>
...
47    s9    2 42    M
48    s9    1 43    M
```

Note that `all = TRUE` retains all subject ids from both data frames. The default is `all = FALSE`, which would drop subjects 10 through 12. Set only `all.x = TRUE` if you want to keep all subjects in `data1.long` but you are fine with subjects in `subj.char` being dropped. The `all.y` option works the opposite way. Note also that R sorts character strings by one character at a time, so that subject id "s10" comes after id "s1." We can force the subject ids to contain one "s" and two digits by `paste("s", sprintf("%02d", 1:12), sep = "")` when `data1` and `subj.char` are created. (although `subj.char` only contains subjects 1 through 9) You get "s01," "s02," ..., "s10," and so on.

2.3 Data Management with a SQL Database

The last sections of this chapter deals with data management with a SQL database. These advanced data management topics can be skipped without loss of continuity or context.

Researchers working with Ecological Momentary Assessment (EMA) data (Shiffman et al. 2008) may find this section especially useful. In this section we cover how to work with PostgreSQL, an open-source database program that can be freely downloaded and installed on computers running Unix/Linux, Mac OS, and Windows. To fully appreciate how this works, you need to install a PostgreSQL server program on your computer, run the SQL commands in Appendix A to build a database, and run the R query commands below to retrieve data from the PostgreSQL database program. It is a different method of data management. You have the option to retrieve only a handful of variables you need to run an analysis. You no longer need to use R to manage many variables in one large data frame, most of which are anyways not needed in a specific analysis.

Subjchar

id	sex	edu	race
s001	F	3	W
s002	F	2	A
s003	M	1	W
s004	M	4	B
s005	F	2	B

Baseassess

id	bsi	bdi	basedate
s001	10	13	2009–06–28
s002	12	15	2009–06–17
s003	12	10	2009–07–09
s004	14	16	2009–07–12
s005	11	10	2009–07–12

EMA

id	tstamp	smoke
s001	2009–06–29 09:20:25	1
s001	2009–06–29 09:35:35	1
...		
s001	2009–06–29 10:35:55	1
s002	2009–06–19 07:35:35	1
s002	2009–06–19 08:05:15	1
...		
s002	2009–06–19 09:42:32	0
...		
s005	2009–07–14 11:07:03	1
s005	2009–07–14 11:32:23	1
...		
s005	2009–07–14 12:42:19	0
s005	2009–07–14 13:29:07	1

Fig. 2.1 An example SQL database with three data tables

Figure 2.1 shows the design of a hypothetical database with three data tables. Each table can be thought of as a spreadsheet. The `subchar` table contains information on subject characteristics. The `baseassess` table contains baseline assessments. These two tables are simple. The `ema` table contains repeated measures of intensive EMA data. For example, subject 001 was asked whether or not she was smoking on June 29, 2009 at 9:20 and she responded "yes" (coded 1). Another entry is timestamped at 9:35, and another at 10:35. EMA is typically collected through an electronic device such as a hand-held computer or a cellular phone to capture behaviors as they happen in real time. An obvious advantage of EMA is that it minimizes recall bias or noncompliance. A data analysis challenge is that the `ema` table can be very long. Another complication is that different subjects can produce different numbers of assessments. It would not make sense to format the data in a *wide* layout.

Appendix A describes how to create this hypothetical database called `test` on a PostgreSQL server program. The syntax in Appendix A should also work with other database programs such as MySQL. Once created, the database `test` can be linked to R by the `library(RPostgreSQL)` package.

```
> library(RPostgreSQL)
Loading required package: DBI
> conn <- dbConnect(PostgreSQL(), user = "usr1",
         password = "**********", dbname = "test")
```

A connection is first established between R and the PostgreSQL server program. In this example we use a user name and a password to provide data safety protection.

Next, a Standard Query Language (SQL) query is sent to the server through `conn` to retrieve a result set.

```
> rs <- dbSendQuery(conn, "SELECT subjchar.id, sex,
+ edu, race, bsi, bdi, bdate, tstamp, smoke
+ FROM subjchar, baseassess, ema
+ WHERE subjchar.id = baseassess.id AND
+ subjchar.id = ema.id
+ ORDER BY subjchar.id, tstamp;")
> dat <- fetch(rs, n = -1)
> dbDisconnect(conn)
```

A result set `rs` is retrieved and `fetch()` actually gets all data in the result set into `dat`. A great convenience is that any timestamp variable such as `tstamp` in this example is automatically and seamlessly converted by the `RPostgreSQL` package into `DateTimeClasses` in R. There is no need to do the often tedious manual conversion. The `tapply()` command shows that the five consecutive assessments for subject 001 are separated by approximately 15–25 minutes apart.

```
> tapply(dat.del$tstamp, list(dat.del$id),
+    function(x) {
+    x[2:length(x)] - x[1:(length(x)-1)] } )
$s001
Time differences in mins
[1] 15.167 15.000 24.500 20.833
attr(,"tzone")
[1] ""
....
$s005
Time differences in mins
[1] 25.333 30.167 39.767 46.800 34.783
attr(,"tzone")
[1] ""
```

R can also work with an ACCESS database or any other ODBC-compliant database programs. An example on how to set up and ODBC connection on a standalone PC running Windows XP is provided in Appendix A.3.

2.4 SQL Database Considerations

Data management by a SQL-based database requires some preparations and basic knowledge of SQL. Would it not be much easier just to save the data in spreadsheet files, export each file into a comma-separated file (CSV) and use `read.csv()` and `merge()` to combine them?

The answer to this question depends on a few things. A database management system has several advantages over a spreadsheet program. A database management system also deals with different variable types more efficiently, especially variables

marked by timestamps. There is limited gain in efficiency if you only have a small dataset in a fixed format, with mostly numeric, binary, and categorical variables. Managing complex data with spreadsheet programs can be frustrating. For example, reading character string variables into R and converting them into `DateTimeClass` format is tedious and error prone. Timestamps variables have to be converted one by one from text strings by `strptime()`. Sometimes the user of a spreadsheet program inadvertently changes the format of a date variable so that some entries are entered as `"mm/dd/yyyy"` and others as `"mm/dd/yy"`. A `"%d/%m/%Y"` string in `strptime()` requires a `"mm/dd/yyyy"` format so that it fails with entries in `"mm/dd/yy"`. Furthermore, sometimes the person who enters the data accidentally type a space in one of the cells in a blank column. The resulting CSV file may contain many blank variables. It certainly takes time to set up a SQL-based database, but the prevention of common problems in managing data with a spreadsheet program may more than compensate for the upfront cost in setting up a SQL-based database.

Exercises

2.1. Importing data from a website

In the first exercise of this chapter, we will try importing data directly from a website. Online data repositories make it easy to share de-identified data. The `read.table()` and `read.csv()` functions in R can directly import data from a file on the internet. For example, the `ctest3.data` file in this chapter can be directly accessed from http://idecide.mskcc.org/yl_home/.

(a) Try the command below to read the `ctest3.data` file.

```
c0 <- read.table("http://idecide.mskcc.org/yl_home/
                  rbook/ctest3.data")
```

2.2. Importing data from an online data repository

Online data repository is common. Many authors now make their data available online. One example is the online data repository for the book by Fitzmaurice et al. (2004a). Its URL is http://www.biostat.harvard.edu/∼fitzmaur/ala/ (last accessed April 20, 2011).

(a) Click on the "Datasets" icon, and you will find a link called "Television School and Family Smoking Prevention and Cessation Project." That link points to a raw data file called `tvsfp.txt`.
(b) Click on the link to the TVSFP dataset to view its contents.
(c) What is the complete URL that goes into the `file` option in your `read.csv()` function?

(d) The first 44 lines of text in that file are the authors' notes. They will have to be skipped by the `skip` option. How can this be done?
(e) Write the complete `read.csv()` command with the `skip` option set.
(f) Would you set the `header` option to `TRUE` or `FALSE`?
(g) Convert the retrieved data into a `data.frame` in R.
(h) Add variable names to the final data frame if necessary.

2.3. Read and merge two data files
Read two data files from http://idecide.mskcc.org/yl_home/rbook/. The first is `subjchar.dat`, the second is `ema.dat`. The first row of each file contains the variable names.

(a) The `ema.dat` file should be imported with an `as.is=T` to keep the timestamp variable as a character string.
(b) Try the commands below.

```
url <- paste("http://idecide.mskcc.org/yl_home/",
       "rbook/ema.dat", sep="")
ema <- read.table(url, as.is=T, sep="\t",
       header=TRUE)
t1 <- strptime(ema$tstamp,
      format="%Y-%m-%d %H:%M:%S")
ema <- data.frame(id=ema$id, tstamp=t1,
       smoke=ema$smoke)
```

(c) Explain why we need the `strptime()` function for `ema$tstamp`?
(d) Use `merge()` to combine the two data frames into one.

2.4. Change data layout through `reshape()`
In this exercise we practice how to use `reshape()` to convert the `ema` data in the previous problem into a *wide* format.

(a) First, create a new variable called `time` that contains the chronological order of each subject's `tstamp` variable. Try the command below and explain what it does.

```
ema$time <- unlist(tapply(ema$tstamp, list(ema$id),
    function(x) { order(x) }))
```

(b) Next, build the `reshape()` command that converts `ema` into the wide format. (hint: You will need `timevar="time"`, `idvar="id"`, `v.names="smoke"`, and an optional `drop="tstamp"`).
(c) Note that the reshaped wide layout contains additional variable(s), particularly the one associated with the reshaped `smoke` variable. Explain why extra variable(s) were created automatically as part of `reshape()`?
(d) Explain what the `drop="tstamp"` option does?
(e) Explain why you do not need to set the `varying` option?

http://www.springer.com/978-1-4614-1237-3

Behavioral Research Data Analysis with R

Li, Y.; Baron, J.

2012, XII, 245 p. 32 illus., 11 illus. in color., Softcover
ISBN: 978-1-4614-1237-3