10 Advanced data handling

In the preceding text, we have covered a basic set of elementary statistical procedures. In the chapters that follow, we begin to discuss more elaborate statistical modelling.

This is also a natural point to discuss some data handling techniques that are useful in the practical analysis of data but were too advanced to cover in the first two chapters of the book.

10.1 Recoding variables

This section describes some techniques that are used to construct derived variables: grouping quantitative data, combining and renaming factor levels, and handling date values.

10.1.1 The `cut` function

You may need to convert a quantitative variable to a grouping factor. For instance, you may wish to present your data in terms of age in 5-year groups, but age is in the data set as a quantitative variable, recorded as whole years or perhaps to a finer resolution. This is what the `cut` function is for. The basic principles are quite simple, although there are some fine points to be aware of.

The function has two basic arguments: a numeric vector and a vector of *breakpoints*. The latter defines a set of intervals into which the variable is grouped. You have to specify both ends of all intervals; that is, the total number of break points must be one more than the number of intervals. It is a common mistake to believe that the outer breakpoints can be omitted but the result for a value outside all intervals is set to `NA`. The outer breakpoints can be chosen as `-Inf` and `Inf`, though.

The intervals are left-open, right-closed by default. That is, they include the breakpoint at the right end of each interval. The lowest breakpoint is *not* included unless you set `include.lowest=TRUE`, making the first interval closed at both ends.

In (e.g.) epidemiology, you are more likely to want groupings like "40–49 years of age". This opposite convention can be obtained by setting `right=FALSE`.

Of course, as you switch to left-closed, right-open intervals, the issue of losing the extreme interval endpoint shifts to the other end of the scale. In that case, `include.lowest` actually includes the *highest* value! In the example below, the difference lies in the inclusion of two subjects who were exactly 16 years old.

```
> age <- subset(juul, age >= 10 & age <= 16)$age
> range(age)
[1] 10.01 16.00
> agegr <- cut(age, seq(10,16,2), right=F, include.lowest=T)
> length(age)
[1] 502
> table(agegr)
agegr
[10,12) [12,14) [14,16]
    190     168     144
> agegr2 <- cut(age, seq(10,16,2), right=F)
> table(agegr2)
agegr2
[10,12) [12,14) [14,16)
    190     168     142
```

It is sometimes desired to split data into roughly equal-sized groups. This can be achieved by using breakpoints computed by `quantile`, which was described in Section 4.1. For instance, you could do

```
> q <- quantile(age, c(0, .25, .50, .75, 1))
> q
     0%     25%     50%     75%    100%
10.0100 11.3825 12.6400 14.2275 16.0000
```

```
> ageQ <- cut(age, q, include.lowest=T)
> table(ageQ)
ageQ
  [10,11.4] (11.4,12.6] (12.6,14.2]   (14.2,16]
        126         125         125         126
```

The level names resulting from `cut` turn out rather ugly at times. Fortunately they are easily changed. You can modify each of the factors created above as follows:

```
> levels(ageQ) <- c("1st", "2nd", "3rd", "4th")
> levels(agegr) <- c("10-11", "12-13", "14-15")
```

Frank Harrell's `Hmisc` package contains the `cut2` function, which simplifies some of these matters.

10.1.2 Manipulating factor levels

In Section 1.2.8, we used `levels(f)<- ....` to change the level set of a factor. Some related tasks will be discussed in this section.

First, notice that the conversion from numeric input and renaming of levels can be done in one operation:

```
> pain <- c(0,3,2,2,1)
> fpain <- factor(pain,levels=0:3,
+        labels=c("none","mild","medium","severe"))
```

Beware the slightly confusing distinction between `levels` and `labels`. The latter end up being the levels of the result, whereas the former refers to the coding of the input vector (`pain` in this case). That is, `levels` refers to the input and `labels` to the output.

If you do not specify a `levels` argument, the levels will be the sorted, unique values represented in the vector. This is not always desirable when dealing with text variables since the sorting is *alphabetical*. Consider, for instance,

```
> text.pain <-  c("none","severe", "medium", "medium", "mild")
> factor(text.pain)
[1] none   severe medium medium mild
Levels:  medium mild none severe
```

Another reason for specifying `levels` is that the default `levels`, obviously, do not include values that are not present in data. This may or may not be a problem, but it has consequences for later analyses; for instance, whether tables contain zero entries or whether barplots leave space for the empty columns.

The `factor` function works on factors as if they were character vectors, so you can reorder the levels as follows

```
> ftpain <- factor(text.pain)
> ftpain2 <- factor(ftpain,
+                   levels=c("none", "mild", "medium", "severe"))
```

Another typical task is to combine two or more levels. This is often done when groups would otherwise be too small for valid statistical analysis. Say you wish to combine the levels `"medium"` and `"mild"` into a single `"intermediate"` level. For this purpose, the assignment form of `levels` allows the right-hand side to be a list:

```
> ftpain3 <- ftpain2
> levels(ftpain3) <- list(
+         none="none",
+         intermediate=c("mild","medium"),
+         severe="severe")
> ftpain3
[1] none         severe       intermediate intermediate
[5] intermediate
Levels: none intermediate severe
```

However, it is often easier just to change the level names and give the same name to several groups:

```
> ftpain4 <- ftpain2
> levels(ftpain4) <- c("none","intermediate","intermediate","severe")
> ftpain4
[1] none         severe       intermediate intermediate
[5] intermediate
Levels: none intermediate severe
```

The latter method is not quite as general as the former, though. It gives less control over the final ordering of levels.

10.1.3 Working with dates

In epidemiology and survival data, you often deal with time in the form of dates in calendar format. Different formats are used in different places of the world, and the files you have to read were not necessarily written in the same region as the one you are currently in. The `"Date"` class and associated conversion routines exist to help you deal with the complexity.

As an example, consider the Estonian stroke study, a preprocessed version of which is contained in the data frame `stroke`. The raw data files can be found in the `rawdata` directory of the `ISwR` package and read using the following code:

```
> stroke <- read.csv2(
+   system.file("rawdata","stroke.csv", package="ISwR"),
+   na.strings=".")
> names(stroke) <- tolower(names(stroke))
> head(stroke)
  sex       died       dstr age dgn coma diab minf han
1   1  7.01.1991  2.01.1991  76 INF    0    0    1   0
2   1       <NA>  3.01.1991  58 INF    0    0    0   0
3   1  2.06.1991  8.01.1991  74 INF    0    0    1   1
4   0 13.01.1991 11.01.1991  77 ICH    0    1    0   1
5   0 23.01.1996 13.01.1991  76 INF    0    1    0   1
6   1 13.01.1991 13.01.1991  48 ICH    1    0    0   1
```

(You can of course also just substitute the full path to `stroke.csv` instead of using the `system.file` construction.)

In this data set, the two date variables `died` and `dstr` (date of stroke) appear as factor variables, which is the standard behaviour of `read.table`. To convert them to class `"Date"`, we use the function `as.Date`. This is straightforward but requires some attention to the date format. The format used here is (day, month, year) separated by a period (dot character), with year given as four digits. This is not a standard format, so we need to specify it explicitly.

```
> stroke <- transform(stroke,
+    died = as.Date(died, format="%d.%m.%Y"),
+    dstr = as.Date(dstr, format="%d.%m.%Y"))
```

Notice the use of "percent-codes" to represent specific parts of the date: `%d` indicates the day of the month, `%m` means the month as a number, and `%Y` means that a four-digit year is used (notice the uppercase `Y`). The full set of codes is documented on the help page for `strptime`.

Internally, dates are represented as the number of days before or after a given point in time, known as the *epoch*. Specifically, the epoch is January 1, 1970, although this is an implementation detail that should not be relied upon.

It is possible to perform arithmetic on dates; that is, they behave mostly like numeric vectors:

```
> summary(stroke$died)
        Min.      1st Qu.       Median         Mean      3rd Qu.
"1991-01-07" "1992-03-14" "1993-01-23" "1993-02-15" "1993-11-04"
        Max.
"1996-02-22"
> summary(stroke$dstr)
        Min.      1st Qu.       Median         Mean      3rd Qu.
"1991-01-02" "1991-11-08" "1992-08-12" "1992-07-27" "1993-04-30"
        Max.
"1993-12-31"
```

```
> summary(stroke$died - stroke$dstr)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
    0.0     8.0    28.0   225.7   268.5  1836.0   338.0
> head(stroke$died - stroke$dstr)
Time differences in days
[1]    5   NA  145    2 1836    0
```

Notice that means and quantiles are displayed in date format (even if they are nonintegers). The count of NA values is not displayed for date variables even though the date of death is unknown for quite a few patients; this is a bit unfortunate, but it would conflict with a convention that numerical summaries have the same class as the object that is summarized (so you would get the count displayed as a date!).

The vector of differences between the two dates is actually an object of class "difftime". Such objects can have different units — when based on dates, it will always be "days", but for other kinds of time variables it can be "hours" or "seconds". Accordingly, it is somewhat bad practice just to treat the vector of differences as a numeric variable. The recommended procedure is to use as.numeric with an explicit units argument.

In the data file, NA for a death date means that the patient did not die before the end of the study on January 1, 1996. Six patients were recorded as having died after this date, but since there may well be unrecorded deaths among the remaining patients, we have to discard these death dates and just record the patients as alive at the end of the study.

We shall transform the data so that all patients have an end date plus an indicator of what happened at the end date: died or survived.

```
> stroke <- transform(stroke,
+   end = pmin(died,  as.Date("1996-1-1"), na.rm = T),
+   dead = !is.na(died) & died < as.Date("1996-1-1"))
> head(stroke)
  sex       died       dstr age dgn coma diab minf han
1   1 1991-01-07 1991-01-02  76 INF    0    0    1   0
2   1       <NA> 1991-01-03  58 INF    0    0    0   0
3   1 1991-06-02 1991-01-08  74 INF    0    0    1   1
4   0 1991-01-13 1991-01-11  77 ICH    0    1    0   1
5   0 1996-01-23 1991-01-13  76 INF    0    1    0   1
6   1 1991-01-13 1991-01-13  48 ICH    1    0    0   1
         end  dead
1 1991-01-07  TRUE
2 1996-01-01 FALSE
3 1991-06-02  TRUE
4 1991-01-13  TRUE
5 1996-01-01 FALSE
6 1991-01-13  TRUE
```

The pmin function calculates the minimum, but unlike the min function, which returns a single number, it does so in *p*arallel across multiple vectors. The na.rm argument allows NA values to be ignored, so the result is that wherever died is missing or later than 1996-01-01, the end date becomes 1996-01-01 and the actual date of death otherwise.

The expression for dead is straightforward, although you should check that missing values are treated correctly. (They are. The & operator handles missingness such that if one argument is FALSE the result is FALSE, even if the other is NA.)

Finally, to obtain the observation time for all individuals, we can do

```
> stroke <- transform(stroke,
+   obstime = as.numeric(end - dstr, units="days")/365.25)
```

in which we pragmatically convert to "epidemiological years" of average length. (This cannot be done just by setting units="years". Objects of class "difftime" can only have units of "weeks" or less.)

Notice that we performed the transformations in three separate calls to transform. This was not just for the flow of the presentation; each of the last two calls refers to variables that were not defined previously. The transform function does not allow references to variables defined in the same call (we could have used within, though; see Section 2.1.8).

Further time classes

R also has classes that represent time to a granularity finer than 1 day. The "POSIXct" class (calendar time according to the POSIX standards) is similar to "Date" except that it counts seconds rather than days, and "POSIXlt" (local time) represents date and time using a structure that consists of fields for various components: year, month, day of month, hours, minutes, seconds, and more. Working with such objects involves, by and large, the same issues as for the "Date" class, although with a couple of extra twists related to time zones and Daylight Savings Time. We shall not go deeper into this area here.

10.1.4 Recoding multiple variables

In the previous sections, we had some cases where essentially the same transformation had to be applied to several variables. The solution in those cases was simply to repeat the operation, but it can happen that a data set contains many similar variables that all need to be recoded (questionnaire data may, for instance, have dozens of items rated on the same five-point scale). In such cases, you can make use of the fact that data frames are fundamentally lists and that lapply and indexing work on them. For instance, in dealing with the raw stroke data, we could have done the date handling as follows:

```
> rawstroke <- read.csv2(
+   system.file("rawdata","stroke.csv", package="ISwR"),
+   na.strings=".")
> ix <- c("DSTR", "DIED")
> rawstroke[ix] <- lapply(rawstroke[ix],
+                          as.Date, format="%d.%m.%Y")
> head(rawstroke)
  SEX       DIED       DSTR AGE DGN COMA DIAB MINF HAN
1   1 1991-01-07 1991-01-02  76 INF    0    0    1   0
2   1       <NA> 1991-01-03  58 INF    0    0    0   0
3   1 1991-06-02 1991-01-08  74 INF    0    0    1   1
4   0 1991-01-13 1991-01-11  77 ICH    0    1    0   1
5   0 1996-01-23 1991-01-13  76 INF    0    1    0   1
6   1 1991-01-13 1991-01-13  48 ICH    1    0    0   1
```

Similarly, the four binary variables could be converted to “No/Yes” factors in a single operation.

```
> ix <- 6:9
> rawstroke[ix] <- lapply(rawstroke[ix],
+                          factor, levels=0:1, labels=c("No","Yes"))
```

10.2 Conditional calculations

The ifelse function lets you apply different calculations to different parts of data. For illustration, we use a subset of the stroke data discussed in Section 10.1.3, but we use the “cooked” version contained in the ISwR package.

```
> strokesub <- ISwR::stroke[1:10,2:3]
> strokesub
         died       dstr
1  1991-01-07 1991-01-02
2        <NA> 1991-01-03
3  1991-06-02 1991-01-08
4  1991-01-13 1991-01-11
5        <NA> 1991-01-13
6  1991-01-13 1991-01-13
7  1993-12-01 1991-01-14
8  1991-12-12 1991-01-14
9        <NA> 1991-01-15
10 1993-11-10 1991-01-15
```

To compute the time on study and the event/censoring indicator needed for survival models, we can do as follows:

```
> strokesub <- transform(strokesub,
+   event = !is.na(died))
> strokesub <- transform(strokesub,
+  obstime = ifelse(event, died-dstr, as.Date("1996-1-1") - dstr))
> strokesub
         died       dstr event obstime
1  1991-01-07 1991-01-02  TRUE       5
2        <NA> 1991-01-03 FALSE    1824
3  1991-06-02 1991-01-08  TRUE     145
4  1991-01-13 1991-01-11  TRUE       2
5        <NA> 1991-01-13 FALSE    1814
6  1991-01-13 1991-01-13  TRUE       0
7  1993-12-01 1991-01-14  TRUE    1052
8  1991-12-12 1991-01-14  TRUE     332
9        <NA> 1991-01-15 FALSE    1812
10 1993-11-10 1991-01-15  TRUE    1030
```

The way `ifelse` works is that it takes three arguments: `test`, `yes`, and `no`. All three are vectors of the same length (if not, they will be made so by recycling). The answer is "stitched together" of pieces of `yes` and `no` in the sense that the `yes` element is selected wherever `test` is `TRUE` and the `no` element where it is `FALSE`. When the condition is `NA`, so is the result.

Notice that both alternatives are computed (exceptions are made for the cases where the condition is all `TRUE` or all `FALSE`). This is not usually a problem in terms of speed, but it does mean that `ifelse` is not the right tool to use if you want to avoid, for example, taking the logarithm of negative values. Also notice that `ifelse` discards attributes, including the class, so that `obstime` is not of class `"difftime"` even though both the `yes` and the `no` part are. This sometimes makes using `ifelse` more trouble than it is worth, and it can be preferable simply to use explicit subsetting operations.

10.3 Combining and restructuring data frames

In this section, we discuss ways of joining data frames either "vertically" (adding records) or "horizontally" (adding variables). We also look at the issue of converting data with repeated measurements of the same variables between the "long" and the "wide" formats.

10.3.1 *Appending frames*

Sometimes data are received from multiple sources and you need to combine them to form one bigger data set. In this subsection, we consider the case where data are combined by "vertical stacking"; that is, you start out with data frames which refer to separate rows of the result — typically different subjects. It is required that the data frames contain the same variables, although not necessarily in the same order (this is unlike some other statistical systems, which will simply insert missing values for variables that are absent in a data set).

To simulate such a situation, suppose that the `juul` data set had been collected separately for boys and girls. In that case, the data frames might not contain the variable `sex`, since this is the same for everyone in the same data frame, and variables that only make sense for one gender may also have been omitted for the other group.

```
> juulgrl <- subset(juul, sex==2, select=-c(testvol,sex))
> juulboy <- subset(juul, sex==1, select=-c(menarche,sex))
```

Notice the use of the `select` argument to `subset`. The processing of this argument replaces column names by column numbers, and the resulting expression is used to index the data frame. The net effect of the negative indices is to remove, for example, `testvol` and `sex` from `juulgrl`.

To put the data frames back together, you must first add in the missing variables

```
> juulgrl$sex <- factor("F")
> juulgrl$testvol <- NA
> juulboy$sex <- factor("M")
> juulboy$menarche <- NA
```

and then it is just a matter of using the `rbind` method for data frames:

```
> juulall <- rbind(juulboy, juulgrl)
> names(juulall)
[1] "age"      "igf1"     "tanner"   "testvol"  "sex"
[6] "menarche"
```

Notice that `rbind` uses the column names (so that it does not concatenate unrelated variables even though the order of columns differs in the two data frames) and that the order of variables in the first data frame "wins": The result has the variables in the same order as `juulboy`. Notice also that `rbind` is being smart about factor levels:

```
> levels(juulall$sex)
[1] "M" "F"
```

10.3.2 Merging data frames

Just as you may have different groups of subjects collected in separate data sets, you may also have different sorts of data on the same patients collected separately. For example, you could have one data frame with registry data, one with clinical biochemistry data, and one with questionnaire data. It may work to use `cbind` to stick the data frames together side-by-side, but it could be dangerous: What if the data are not complete in all data frames or out of sequence? You typically have to work with a unique subject identification code to avoid mistakes of this sort.

The `merge` function deals with these issues. It works by matching on one or several variables from each data frame. By default, this is the set of variables that have the same name in both frames (typically, there is a variable called something like `ID`, which holds the subject identification). Assuming that this default works and that the two data frames are called respectively `dfx` and `dfy`, the merged frame is computed simply as

```
  merge(dfx, dfy)
```

However, there may be variables of the same name in both frames. In such cases, you can add a `by` argument, which contains the variable name or names to match on as in

```
  merge(dfx, dfy, by="ID")
```

Any other variables that appear in both frames will have `.x` or `.y` appended to their name in the result. It is recommended to use this format in any case as a safeguard and for readability and explicitness. If the matching variable(s) have different names in the two data frames, you can use `by.x` and `by.y`.

Matching is not necessarily one-to-one. One of the data sets might for instance hold tabular material corresponding to the study population. The common example is mortality tables. In such cases, there is generally a many-to-one relationship between the data frames. More than one subject in the study population will belong to the table entry for 40–49 year-olds, and the rows of the table will have to be duplicated accordingly during the merge.

To illustrate these concepts, we use the data set `nickel`. This describes a cohort of nickel smelting workers in South Wales. The data set `ewrates` contains a table of the population mortality by year and age group in five-year intervals.

```
> head(nickel)
  id icd exposure      dob  age1st   agein  ageout
1  3   0        5 1889.019 17.4808 45.2273 92.9808
```

```
2  4 162        5 1885.978 23.1864 48.2684 63.2712
3  6 163       10 1881.255 25.2452 52.9917 54.1644
4  8 527        9 1886.340 24.7206 47.9067 69.6794
5  9 150        0 1879.500 29.9575 54.7465 76.8442
6 10 163        2 1889.915 21.2877 44.3314 62.5413
> head(ewrates)
  year age lung nasal other
1 1931  10    1     0  1269
2 1931  15    2     0  2201
3 1931  20    6     0  3116
4 1931  25   14     0  3024
5 1931  30   30     1  3188
6 1931  35   68     1  4165
```

Suppose we wish to merge these two data sets according to the values at entry into the study population. This age is contained in `agein`, and the date of entry is computed as `dob + agein`. You can compute group codes corresponding to `ewrates` as follows:

```
> nickel <- transform(nickel,
+   agr = trunc(agein/5)*5,
+   ygr = trunc((dob+agein-1)/5)*5+1)
```

The `trunc` function rounds values towards zero. Notice that the age groups start on values that are evenly divisible by 5, whereas the year groups end on such values; this is why the expression for `ygr` subtracts 1 and adds it back after truncation. (Actually this does not matter because all enrollment dates were April 1 of 1934, 1939, 1944, or 1949.) Notice also that we do not use the same variable names as in `ewrates`. We could have done so, but the names `age` and `year` would be unintuitive in the context of the `nickel` data.

With the age and year groups defined, it is an easy matter to perform the merge. We just need to account for the fact that we have used different variable names in the two data frames.

```
> mrg <- merge(nickel, ewrates,
+   by.x=c("agr","ygr"), by.y=c("age","year"))
> head(mrg,10)
   agr  ygr  id icd exposure      dob  age1st   agein  ageout
1   20 1931 273 154        0 1909.500 14.6913 24.7465 55.9302
2   20 1931 213 162        0 1910.129 14.2018 24.1177 63.0493
3   20 1931 546   0        0 1909.500 14.4945 24.7465 72.5000
4   20 1931 574 491        0 1909.729 14.0356 24.5177 70.6592
5   20 1931 110   0        0 1909.247 14.0302 24.9999 72.7534
6   20 1931 325 434        0 1910.500 14.0737 23.7465 43.0343
7   25 1931  56 502        2 1904.500 18.2917 29.7465 51.5847
8   25 1931 690 420        0 1906.500 17.2206 27.7465 55.1219
9   25 1931 443 420        0 1905.326 14.5562 28.9204 65.7616
10  25 1931 137 465        0 1905.386 19.0808 28.8601 74.2794
   lung nasal other
```

```
1     6     0  3116
2     6     0  3116
3     6     0  3116
4     6     0  3116
5     6     0  3116
6     6     0  3116
7    14     0  3024
8    14     0  3024
9    14     0  3024
10   14     0  3024
```

We have only described the main function of `merge`. There are also options to include rows that only exist in one of the two frames (`all`, `all.x`, `all.y`), and it may also be useful to know that the pseudo-variable `row.names` will allow matching on row names.

We have discussed the cases of one-to-one and many-to-one matching. Many-to-many is possible but rarely useful. What happens in that case is that the "Cartesian product" is formed by generating all combinations of rows from the two frames within each matching set. The extreme case of many-to-many matching occurs if the `by` set is empty, which gives a result with as many rows as the *product* of the row counts. This sometimes surprises people who expect that the row number will act as an implicit ID.

10.3.3 Reshaping data frames

Longitudinal data come in two different forms: a "wide" format, where there is a separate column for each time point but only one record per case; and a "long" format, where there are multiple records for each case, one for each time point. The long format is more general since it does not need to assume that the cases are recorded at the same set of times, but when applicable it may be easier to work with data in the wide format, and some statistical functions expect it that way. Other functions expect to find data in the long format. Either way, there is a need to convert from one format to another. This is what the `reshape` function does.

Consider the following data from a randomized study of bone metabolism data during Tamoxifen treatment after breast cancer. The concentration of alkaline phosphatase is recorded at baseline and 3, 6, 9, 12, 18, and 24 months after treatment start.

```
> head(alkfos)
  grp  c0  c3  c6  c9 c12 c18 c24
1   1 142 140 159 162 152 175 148
2   1 120 126 120 146 134 119 116
3   1 175 161 168 164 213 194 221
```

```
4   1 234 203 174 197 289 174 189
5   1  94 107 146 124 128  98 114
6   1 128  97 113 203  NA  NA  NA
```

In the simplest uses of `reshape`, the function will assume that the variable names encode the information necessary for reshaping to the long format. By default, it assumes that variable names are separated from time of measurement by a `"."` (dot), so we might oblige by modifying the name format.

```
> a2 <- alkfos
> names(a2) <- sub("c", "c.", names(a2))
> names(a2)
[1] "grp"  "c.0"  "c.3"  "c.6"  "c.9"  "c.12" "c.18" "c.24"
```

The `sub` function does substitutions within character strings, in this case replacing the string `"c"` with `"c."`. Alternatively, the original name format (`c0`, ..., `c24`) can be handled by adding `sep=""` to the `reshape` call.

Once we have the variable naming in place, the only things we need to specify are the direction of the reshape and the set of variables to be considered time-varying. As a convenience feature, the latter can be specified by index rather than by name.

```
> a.long <- reshape(a2, varying=2:8, direction="long")
> head(a.long)
    grp time   c id
1.0   1    0 142  1
2.0   1    0 120  2
3.0   1    0 175  3
4.0   1    0 234  4
5.0   1    0  94  5
6.0   1    0 128  6
> tail(a.long)
      grp time   c id
38.24   2   24  95 38
39.24   2   24  NA 39
40.24   2   24 192 40
41.24   2   24  94 41
42.24   2   24 194 42
43.24   2   24 129 43
```

Notice that the sort order of the result is that `id` varies within `time`. This is the most convenient format to generate technically, but if you prefer the opposite sort order, just use

```
> o <- with(a.long, order(id, time))
> head(a.long[o,], 10)
     grp time   c id
```

```
1.0    1    0 142  1
1.3    1    3 140  1
1.6    1    6 159  1
1.9    1    9 162  1
1.12   1   12 152  1
1.18   1   18 175  1
1.24   1   24 148  1
2.0    1    0 120  2
2.3    1    3 126  2
2.6    1    6 120  2
```

To demonstrate the reverse procedure, we use the same data, in the long format. Actually, this is a bit too easy because `reshape` has inserted enough information in its output to let you convert to the wide format just by saying `reshape(a.long)`. To simulate the situation where the original data are given in the long format, we remove the `"reshapeLong"` attribute, which holds these data. Furthermore, we remove the records for which we have missing data by using `na.omit`.

```
> a.long2 <- na.omit(a.long)
> attr(a.long2, "reshapeLong") <- NULL
```

To convert `a.long2` to the wide format, use

```
> a.wide2 <- reshape(a.long2, direction="wide", v.names="c",
+                   idvar="id", timevar="time")
> head(a.wide2)
    grp id c.0 c.3 c.6 c.9 c.12 c.18 c.24
1.0   1  1 142 140 159 162  152  175  148
2.0   1  2 120 126 120 146  134  119  116
3.0   1  3 175 161 168 164  213  194  221
4.0   1  4 234 203 174 197  289  174  189
5.0   1  5  94 107 146 124  128   98  114
6.0   1  6 128  97 113 203   NA   NA   NA
```

Notice that `NA` values are filled in for patient no. 6, for whom only the first four observations are available.

The arguments `idvar` and `timevar` specify the names of the variables that contain the ID and the time for each observation. It is not strictly necessary to specify them if they have their default names, but it is good practice to do so. The argument `v.names` specifies the time-varying variables; notice that if it were omitted, then the `grp` variable would also be treated as time-varying.

10.4 Per-group and per-case procedures

A specific data management task involves operations within subsets of a data frame, particularly those where there are multiple records for each individual. Examples include calculation of cumulative dosage in a pharmacokinetic experiment and various methods of normalization and standardization.

A nice general approach to such tasks is first to split the data into a list of groups, operate on each group, and then put the pieces back together.

Consider the task of normalizing the values of alkaline phosphatase in `a.long` to their baseline values. The `split` function can be used to generate a list of the individual time courses:

```
> l <- split(a.long$c, a.long$id)
> l[1:3]
$`1`
[1] 142 140 159 162 152 175 148

$`2`
[1] 120 126 120 146 134 119 116

$`3`
[1] 175 161 168 164 213 194 221
```

Next, we apply a function to each element of the list and collect the results using `lapply`.

```
> l2 <- lapply(l, function(x) x / x[1])
```

Finally, we put the pieces back together using `unsplit`, which is the reverse operation of `split`. Notice that `a.long` has `id` varying within `time`, so this is not just a matter of concatenating the elements of `l2`. The data for the first patient are now

```
> a.long$c.adj <- unsplit(l2, a.long$id)
> subset(a.long, id==1)
     grp time   c id     c.adj
1.0    1    0 142  1 1.0000000
1.3    1    3 140  1 0.9859155
1.6    1    6 159  1 1.1197183
1.9    1    9 162  1 1.1408451
1.12   1   12 152  1 1.0704225
1.18   1   18 175  1 1.2323944
1.24   1   24 148  1 1.0422535
```

In fact, there is a function that formalizes this sort of split-modify-unsplit operation. It is called `ave` because the default use is to replace data with group averages, but it can also be used for more general transformations. The following is an alternative way of doing the same computation as above:

```
> a.long$c.adj <- ave(a.long$c, a.long$id,
+     FUN = function(x) x / x[1])
```

In the preceding code, we worked on the single vector `a.long$c`. Alternatively, we can split the entire data frame and use code like

```
> l <- split(a.long, a.long$id)
> l2 <- lapply(l, transform, c.adj = c / c[1])
> a.long2 <- unsplit(l2, a.long$id)
```

Notice how the last argument to `lapply` is passed on to `transform`, so that you effectively call `transform(x, c.adj = c / c[1])` for each data frame `x` in the list `l`. This procedure is somewhat less efficient than the first one because there is more copying of data, but it generalizes to more complex transformations.

10.5 Time splitting

This section is rather advanced, and the beginner may want to skip it on the first read. Understanding the contents is not crucial for the later parts of the book. On the other hand, apart from solving the particular problem, this is also a rather nice first example of the use of ad hoc programming in R and also of the "lateral thinking" that is sometimes required.

The merge operation of the `nickel` and `ewrates` data in Section 10.3.2 does not really make sense statistically: We merged in the mortality table corresponding to the age at the time of entry into the study population. However, the data set is about cancer, a slow disease, and an exposure that perhaps leads to an increased risk 20 or more years later. If the subjects typically die around age 50, the population mortality for people of age 30 is hardly relevant.

A sensible statistical analysis needs to consider the population mortality during the entire follow-up period. One way to handle this issue is to split the individuals into multiple "sub-individuals".

In the data set, the first six observations are (after the `merge` in Section 10.3.2)

```
> head(nickel)
  id icd exposure      dob  age1st   agein  ageout agr  ygr
1  3   0        5 1889.019 17.4808 45.2273 92.9808  45 1931
```

```
2  4 162        5 1885.978 23.1864 48.2684 63.2712  45 1931
3  6 163       10 1881.255 25.2452 52.9917 54.1644  50 1931
4  8 527        9 1886.340 24.7206 47.9067 69.6794  45 1931
5  9 150        0 1879.500 29.9575 54.7465 76.8442  50 1931
6 10 163        2 1889.915 21.2877 44.3314 62.5413  40 1931
```

Consider the individual with `id == 4`; this person entered the study at the age of 48.2684 and died (from lung cancer) at the age of 63.2712 (apologies for the excess precision). The time-splitting method treats this subject as four separate subjects, one entering the study at age 48.2684 and leaving at age 50 (on his 50th birthday) and the others covering the intervals 50–55, 55–60, and 60–63.2712. The first three are censored observations, as the subject did not die.

If we merge these data with the population tables, then we can compute the expected number of deaths in a given age interval and compare that with the actual number of deaths.

Taking advantage of the vectorized nature of computations in R, the nice way of doing this is to loop over age intervals, "trimming" every observation period to each interval.

To trim the observation periods to ages between (say) 60 and 65, the entry and exit times should be adjusted to the interval if they fall outside of it, cases that are unobserved during the interval should be removed, and if the subject did not die inside the interval, `icd` should be set to 0.

The easiest procedure is to "shoot first and ask later". The adjusted entry and exit times are

```
> entry <- pmax(nickel$agein, 60)
> exit <- pmin(nickel$ageout, 65)
```

or rather they would be if there were always a suitable overlap between the observation period and the target age interval. However, there are people leaving the study population before age 60 (by death or otherwise) and people entering the study after age 65. In either case, what goes wrong is that `entry >= exit`, and we can check for such cases by calculating

```
> valid <- (entry < exit)
> entry <- entry[valid]
> exit  <- exit[valid]
```

The censoring indicator for valid cases is

```
> cens <- (nickel$ageout[valid] > 65)
```

(We might have used `cens <- (exit == 65)`, but it is a good rule to avoid testing floating point data for equality.)

The trimmed data set can then be obtained as

```
> nickel60 <- nickel[valid,]
> nickel60$icd[cens] <- 0
> nickel60$agein <- entry
> nickel60$ageout <- exit
> nickel60$agr <- 60
> nickel60$ygr <- with(nickel60, trunc((dob+agein-1)/5)*5+1)
```

and the first lines of the result are

```
> head(nickel60)
  id icd exposure      dob  age1st agein  ageout agr  ygr
1  3   0        5 1889.019 17.4808    60 65.0000  60 1946
2  4 162        5 1885.978 23.1864    60 63.2712  60 1941
4  8   0        9 1886.340 24.7206    60 65.0000  60 1946
5  9   0        0 1879.500 29.9575    60 65.0000  60 1936
6 10 163        2 1889.915 21.2877    60 62.5413  60 1946
7 15 334        0 1890.500 23.2836    60 62.0000  60 1946
```

A couple of fine points: If someone dies exactly at age 65, they are counted as dying inside the age interval. Conversely, we do not include people dying exactly at age 60; they belong in the interval 55–60 (for purposes like those of Chapter 15, one should avoid observation intervals of length zero). It was also necessary to recompute `ygr` since this was based on the original `agein`.

To get the fully expanded data set, you could repeat the above for each age interval (20–25, . . . , 95–100) and append the resulting 16 data frames with `rbind`. However, this gets rather long-winded, and there is a substantial risk of copy-paste errors. Instead, you can do a little programming. First, wrap up the procedure for one group as a function:

```
> trim <- function(start)
+ {
+   end   <- start + 5
+   entry <- pmax(nickel$agein, start)
+   exit  <- pmin(nickel$ageout, end)
+   valid <- (entry < exit)
+   cens  <- (nickel$ageout[valid] > end)
+   result <- nickel[valid,]
+   result$icd[cens] <- 0
+   result$agein <- entry[valid]
+   result$ageout <- exit[valid]
+   result$agr <- start
+   result$ygr <- with(result, trunc((dob+agein-1)/5)*5+1)
+   result
+ }
```

(In practice, you should not type all this at the command line but use a script window or an editor; see Section 2.1.3.)

This is typical ad hoc programming. The function is far from general since it relies on knowing various names, and it also hardcodes the interval length as 5. However, more generality is not required for a one-off calculation. The important thing for the purpose at hand is to make the dependence on `start` explicit so that we can loop over it.

With this definition, `trim(60)` is equivalent to the `nickel60` we computed earlier:

```
> head(trim(60))
  id icd exposure      dob  age1st agein  ageout agr  ygr
1  3   0        5 1889.019 17.4808    60 65.0000  60 1946
2  4 162        5 1885.978 23.1864    60 63.2712  60 1941
4  8   0        9 1886.340 24.7206    60 65.0000  60 1946
5  9   0        0 1879.500 29.9575    60 65.0000  60 1936
6 10 163        2 1889.915 21.2877    60 62.5413  60 1946
7 15 334        0 1890.500 23.2836    60 62.0000  60 1946
```

To get results for all intervals, do the following:

```
> nickel.expand <- do.call("rbind", lapply(seq(20,95,5), trim))
> head(nickel.expand)
     id icd exposure      dob  age1st   agein ageout agr  ygr
84  110   0        0 1909.247 14.0302 24.9999     25  20 1931
156 213   0        0 1910.129 14.2018 24.1177     25  20 1931
197 273   0        0 1909.500 14.6913 24.7465     25  20 1931
236 325   0        0 1910.500 14.0737 23.7465     25  20 1931
384 546   0        0 1909.500 14.4945 24.7465     25  20 1931
400 574   0        0 1909.729 14.0356 24.5177     25  20 1931
```

The `do.call` construct works by creating a call to `rbind` with a given argument list, which in this case is the return value from `lapply`, which in turn has applied the `trim` function to each of the values 20, 25, ...95. That is, the whole thing is equivalent to

```
  rbind(trim(20), trim(25), ......, trim(95))
```

Displaying the result for a single subject yields, for example,

```
> subset(nickel.expand, id==4)
     id icd exposure      dob  age1st   agein  ageout agr  ygr
2     4   0        5 1885.978 23.1864 48.2684 50.0000  45 1931
2100  4   0        5 1885.978 23.1864 50.0000 55.0000  50 1931
2102  4   0        5 1885.978 23.1864 55.0000 60.0000  55 1936
2104  4 162        5 1885.978 23.1864 60.0000 63.2712  60 1941
```

(The strange row names occur because multiple data frames with the same row names are being `rbind`-ed together and data frames must have unique row names.)

A weakness of the ygr computation is that since ygr refers to the calendar time group at agein, it may be off by up to 5 years. However, lung cancer death rates by age do not change that quickly, so we leave it at this. A more careful procedure, and in fact the common practice in epidemiology, is to split on both age and calendar time. The Epi package contains generalized time-splitters splitLexis and cutLexis, which are useful for this purpose and also for handling the related case of splitting time based on individual events (e.g., childbirth).

As a final step, we can merge in the mortality table as we did in Section 10.3.2.

```
> nickel.expand <- merge(nickel.expand, ewrates,
+   by.x=c("agr","ygr"), by.y=c("age","year"))
> head(nickel.expand)
  agr  ygr  id icd exposure      dob  age1st   agein ageout lung
1  20 1931 325   0        0 1910.500 14.0737 23.7465     25    6
2  20 1931 273   0        0 1909.500 14.6913 24.7465     25    6
3  20 1931 110   0        0 1909.247 14.0302 24.9999     25    6
4  20 1931 574   0        0 1909.729 14.0356 24.5177     25    6
5  20 1931 213   0        0 1910.129 14.2018 24.1177     25    6
6  20 1931 546   0        0 1909.500 14.4945 24.7465     25    6
  nasal other
1     0  3116
2     0  3116
3     0  3116
4     0  3116
5     0  3116
6     0  3116
```

For later use, the expanded data set is made available "precooked" in the ISwR package under the name nickel.expand. We return to the data set in connection with the analysis of rates in Chapter 15.

10.6 Exercises

10.1 Create a factor in which the blood.glucose variable in the thuesen data is divided into the intervals $(4,7]$, $(7,9]$, $(9,12]$, and $(12,20]$. Change the level names to "low", "intermediate", "high", and "very high".

10.2 In the bcmort data set, the four-level factor cohort can be considered the product of two two-level factors, say period and area. How can you generate them?

10.3 Convert the `ashina` data to the long format. Consider how to encode whether the `vas` measurement is from the first or the second measurement session.

10.4 Split the `stroke` data according to `obsmonths` into time intervals 0–0.5, 0.5–2, 2–12, and 12+ months after stroke.