6

Data Manipulation and Management

Data analysis has as its end point the use of forms of data summary that will convey, fairly and succinctly, the information that is in the data. The fitting of a model is itself a form of data summary.

Be warned of the opportunities that simple forms of data summary, which seem superficially harmless, can offer for misleading inferences. These issues affect, not just data summary per se, but all modeling. Data analysis is a task that should be undertaken with critical faculties fully engaged.

Data summaries that can lead to misleading inferences arise often, from a unbalance in the data and/or failure to account properly for important variables or factors.

Alternative types of data objects

Column objects: These include (atomic) vectors, factors, and dates.

Date and date-time objects: The creation and manipulations of date objects will be described below.

Data Frames: These are rectangular structures. Columns may be "atomic" vectors, or factors, or other objects (such as dates) that are one-dimensional.

A data frame is a list of column objects, all of the same length.

Matrices and arrays: Matrices[1] are rectangular arrays in which all elements have the same mode. An array is a generalization of a matrix to allow an arbitrary number of dimensions.

[1] Internally, matrices are one long vector in which the columns follow one after the other.

Tables: A table is a specialized form of array.

Lists: A list is a collection of objects that can be of arbitrary class. List elements are themselves lists. In more technical language, lists are *recursive* data structures.

S3 model objects: These are lists that have a defined structure.

S4 objects: These are specialized data structures with tight control on the structure. Unlike S3 objects, they cannot be manipulated as lists. Modeling functions in certain of the newer packages[2] return S4 objects.

[2] These include *lme4*, the Bioconductor packages, and the spatial analysis packages.

6.1 Manipulations with Lists, Data Frames and Arrays

Recall that data frames are lists of columns that all have the same length. They are thus a specialised form of list. Matrices are two-dimensional arrays. Tables are in essence arrays that hold numeric values.

6.1.1 Tables and arrays

The dataset `UCBAdmissions` is stored as a 3-dimensional table. If we convert it to an array, very little changes:

It changes from a table object to a numeric object, which affects the way that it is handled by some functions. In either case, what we have is a numeric vector of length 24 (= $2 \times 2 \times 6$) that is structured to have dimensions 2 by 2 by 6.

6.1.2 Conversion between data frames and tables

The three-way table `UCBAdmissions` are admission frequencies, by Gender, for the six largest departments at the University of California at Berkeley in 1973. For a reference to a web page that has the details; see the belp page for `UCBAdmissions`. Type

```
help(UCBAdmissions)      # Get details of the data
example(UCBAdmissions)
```

Note the margins of the table:

```
str(UCBAdmissions)
```

```
 'table' num [1:2, 1:2, 1:6] 512 313 89 19 353 207 17 8 120 205 ...
 - attr(*, "dimnames")=List of 3
  ..$ Admit : chr [1:2] "Admitted" "Rejected"
  ..$ Gender: chr [1:2] "Male" "Female"
  ..$ Dept  : chr [1:6] "A" "B" "C" "D" ...
```

In general, operations with a table or array are easiest to conceptualise if the table is first converted to a data frame in which the separate dimensions of the table become columns. Thus, the `UCBAdmissions` table will be converted to a data frame that has columns `Admit`, `Gender` and `Dept`. Either use the `as.data.frame.table()` command from base R, or use the `adply()` function from the *plyr* package.

The following uses the function `as.data.frame.table()` to convert the 3-way table `UCBAdmissions` into a data frame in which the margins are columns:

As `UCBAdmissions` is a table (not an array), `as.data.frame(UCBAdmissions)` will give the same result.

```
UCBdf <- as.data.frame.table(UCBAdmissions)
head(UCBdf, 5)
```

```
     Admit Gender Dept Freq
1 Admitted   Male    A  512
2 Rejected   Male    A  313
3 Admitted Female    A   89
4 Rejected Female    A   19
5 Admitted   Male    B  353
```

Alternatively, use the function `adply()` from the *plyr* package that is described in Section 6.2. Here the `identity()` function does the manipulation, working with all three dimensions of the array:

```
library(plyr)
UCBdf <- adply(.data=UCBAdmissions,
               .margins=1:3,
               .fun=identity)
names(UCBdf)[4] <- "Freq"
```

First, calculate overall admission percentages for females and males. The following calculates also the total accepted, and the total who applied:

```
library(dplyr)
gpUCBgender <- dplyr::group_by(UCBdf, Gender)
AdmitRate <- dplyr::summarise(gpUCBgender,
                              Accept=sum(Freq[Admit=="Admitted"]),
                              Total=sum(Freq),
                              pcAccept=100*Accept/Total)
AdmitRate
```

```
# A tibble: 2 x 4
  Gender Accept Total pcAccept
  <fct>   <dbl> <dbl>    <dbl>
1 Male     1198  2691     44.5
2 Female    557  1835     30.4
```

Now calculate admission rates, total number of females applying, and total number of males applying, for each department:

```
gpUCBgd <- dplyr::group_by(UCBdf, Gender, Dept)
rateDept <- dplyr::summarise(gpUCBgd,
    Total=sum(Freq),
    pcAccept=100*sum(Freq[Admit=="Admitted"])/Total)
```

Results can conveniently be displayed as follows. First show admission rates, for females and males separately:

```
xtabs(pcAccept~Gender+Dept, data=rateDept)
```

```
        Dept
Gender        A      B      C      D      E      F
  Male   62.061 63.036 36.923 33.094 27.749  5.898
  Female 82.407 68.000 34.064 34.933 23.919  7.038
```

Now show total numbers applying:

```
xtabs(Total~Gender+Dept, data=rateDept)
```

```
        Dept
Gender     A   B   C   D   E   F
  Male   825 560 325 417 191 373
  Female 108  25 593 375 393 341
```

As a fraction of those who applied, females were strongly favored in department A, and males somewhat favored in departments C and

The overall bias arose because males favored departments where admission rates were relatively high.

E. Note however that relatively many males applied to A and B, where admission rates were high. This biased overall male rates upwards. Relatively many females applied to C, D and F, where rates were low. This biased the overall female rates downwards.

6.1.3 *Table margins*

For working directly on tables, note the function `margin.table()`. The following retains margin 1 (`Admit`) and margin 2 (`Gender`), adding over `Dept` (the remaining margin):

```
## Tabulate by Admit (margin 2) & Gender (margin 1)
(margin21 <- margin.table(UCBAdmissions,
                          margin=2:1))
```

Take margin 2, first, then margin 1, gving a table where rows correspond to levels of `Gender`.

```
        Admit
Gender   Admitted Rejected
  Male       1198     1493
  Female      557     1278
```

Use the function `margin.table()` to turn this into a table that has the proportions in each row:

```
prop.table(margin21, margin=1)
```

```
        Admit
Gender   Admitted Rejected
  Male     0.4452   0.5548
  Female   0.3035   0.6965
```

6.1.4 *Categorization of continuous data*

The data frame `bronchit`, in the *DAAGviz* package, has observations on 212 men in a sample of Cardiff (Wales, UK) enumeration districts. Variables are `r` (1 if respondent suffered from chronic bronchitis and 0 otherwise), `cig` (number of cigarettes smoked per day) and `poll` (the smoke level in the locality).

The dataset `bronchit` may alternatively be found in the *SMIR* package.

It will be convenient to define a function `props` that calculates the proportion of the total in the first (or other nominated element) of a vector:

```
props <- function(x, elem=1)sum(x[elem])/sum(x)
```

Now use the function `cut()` to classify the data into four categories, and form tables:

```
library(DAAGviz)
catcig <- with(bronchit,
               cut(cig, breaks=c(0,1,10,30),
                   include.lowest=TRUE))
tab <- with(bronchit, table(r, catcig))
round(apply(tab, 2, props, elem=2), 3)
```

```
 [0,1]  (1,10] (10,30]
 0.072   0.281   0.538
```

The argument `breaks` can be either the number of intervals, or it can be a vector of break points such that all data values lie within the range of the breaks. If the smallest of the break points equals the smallest data value, supply the argument `include.lowest=TRUE`.

There is a clear increase in the risk of bronchitis with the number of cigarettes smoked.

This categorization was purely for purposes of preliminary analysis. Categorization for purposes of analysis is, with the methodology and software that are now available, usually undesirable. Tables that are based on categorization can nevertheless be useful in data exploration.

It was at one time common practice to categorize continuous data, in order to allow analysis methods for multi-way tables. There is a loss of information, which can at worst be serious.

6.1.5 *Matrix Computations

Let `X` (n by p), `Y` (n by p) and B (p by k) be numeric matrices. Some of the possibilities are:

```
X + Y              # Elementwise addition
X * Y              # Elementwise multiplication
X %*% B            # Matrix multiplication
solve(X, Y)        # Solve X B = Y for B
svd(X)             # Singular value decomposition
qr(X)              # QR decomposition
t(X)               # Transpose of X
```

Note that if `t()` is used with a data frame, a matrix is returned. If necessary, all values are coerced to the same mode.

Calculations with data frames that are slow and time consuming will often be much faster if they can be formulated as matrix calculations. This is in general become an issue only for very large datasets, with perhaps millions of observations. Section 6.4 has examples. For small or modest-sized datasets, convenience in formulating the calculations is likely to be more important than calculation efficiency.

Section 4.3.7 will discuss the use of `apply()` for operations with matrices, arrays and tables.

6.2 plyr, dplyr & reshape2 *Data Manipulation*

The *plyr* package has functions that together:

- provide a systematic approach to computations that perform a desired operation across one or more dimensions of an array, or of a data frame, or of a list;
- allow the user to choose whether results will be returned as an array, or as a data frame, or as a list.

The *dplyr* package has functions for performing various summary and other operations on data frames. For many purposes, it supersedes the *plyr* package.

The *reshape2* package is, as its name suggests, designed for moving between alternative data layouts.

6.2.1 plyr

The *plyr* package has a separate function for each of the nine possible mappings. The first letter of the function name (one of `a` = array, `d` = data frame, `l` = list) denotes the class of the input object, while the second letter (the same choice of one of three letters) denotes the class of output object that is required. This pair of letters is then followed by `ply`.

Here is the choice of functions:

	Class of Output Object		
	`a` (array)	`d` (data frame)	`l` (list)
Class of Input Object			
a (array)	aaply	adply	alply
d (data frame)	daply	ddply	dlply
l (list)	laply	ldply	llply

First observe how the function `adply` can be used to change from a tabular form of representation to a data frame. The dimension names will become columns in the data frame.

```
detach("package:dplyr")
library(plyr)
```

```
dreamMoves <-
   matrix(c(5,3,17,85), ncol=2,
          dimnames=list("Dreamer"=c("Yes","No"),
                        "Object"=c("Yes","No")))
(dfdream <- plyr::adply(dreamMoves, 1:2,
                        .fun=identity))
```

```
  Dreamer Object V1
1     Yes    Yes  5
2      No    Yes  3
3     Yes     No 17
4      No     No 85
```

To get the table back, do:

```
plyr::daply(dfdream, 1:2, function(df)df[,3])
```

```
       Object
Dreamer Yes No
```

```
  Yes    5 17
  No     3 85
```

The following calculates sums over the first two dimensions of the table `UCBAdmissions`:

Here, `aaply()` behaves exactly like `apply()`.

```
plyr::aaply(UCBAdmissions, 1:2, sum)
```

```
          Gender
Admit       Male Female
  Admitted  1198    557
  Rejected  1493   1278
```

The following calculates, for each level of the column `trt` in the data frame `nswdemo`, the number of values of `re74` that are zero:

```
library(DAAG, quietly=TRUE)
plyr::daply(nswdemo, .(trt),
      function(df)sum(df[,"re74"]==0, na.rm=TRUE))
```

```
  0   1
195 131
```

To calculate the proportion that are zero, for each of control and treatment and for each of non-black and black, do:

Notice the use of the syntax `.(trt, black)` to identify the columns `trt` and `black`. This is an alternative to `c("trt", "black")`.

```
options(digits=3)
plyr::daply(nswdemo, .(trt, black),
      function(df)sum(df[,"re75"]==0)/nrow(df))
```

```
   black
trt     0     1
  0 0.353 0.435
  1 0.254 0.403
```

The function `colwise()` takes as argument a function that operates on a column of data, returning a function that operates on all nominated columns of a data frame. To get information on the proportion of zeros for both of the columns `re75` and `re78`, and for each of non-black and black, do:

Here, `colwise()` operates on the objects that are returned by splitting up the data frame `nswdemo` according to levels of `trt` and `black`. Note the use of `ddply()`, not `daply()`.

```
plyr::ddply(nswdemo, .(trt, black),
      colwise(function(x)sum(x==0)/length(x),
              .cols=.(re75, re78)))
```

```
  trt black  re75   re78
1   0     0 0.353 0.1529
2   0     1 0.435 0.3412
3   1     0 0.254 0.0847
4   1     1 0.403 0.2605
```

6.2.2 *Use of* dplyr *with Word War 1 cricketer data*

Data in the data frame `cricketer`, extracted by John Aggleton (now at Univ of Cardiff), are from records of UK first class cricketers born 1840 – 1960. Variables are

- Year of birth
- Years of life (as of 1990)
- 1990 status (dead or alive)
- Cause of death: killed in action / accident / in bed
- Bowling hand – right or left

The following creates a data frame in which the first column has the year, the second the number of right-handers born in that year, and the third the number of left-handers born in that year. .

Both *plyr* and *dplyr* have functions `summarise()`. As in the code shown, detach *plyr* before proceeding. Alternatively, or additionally, specify `dplyr::summarise()` rather than `summarise()`

```
library(DAAG)
detach("package:plyr")
library(dplyr)
```

```
names(cricketer)[1] <- "hand"
gpByYear <- group_by(cricketer, year)
lefrt <- dplyr::summarise(gpByYear,
                          left=sum(hand=='left'),
                          right=sum(hand=='right'))
## Check first few rows
lefrt[1:4, ]
```

```
# A tibble: 4 x 3
   year  left right
  <int> <int> <int>
1  1840     1     6
2  1841     4    16
3  1842     5    16
4  1843     3    25
```

The data frame is split by values of `year`. Numbers of left and right handers are then tabulated.

From the data frame `cricketer`, we determine the range of birth years for players who died in World War 1. We then extract data for all cricketers, whether dying or surviving until at least the final year of Workd War 1, whose birth year was within this range of years. The following code extracts the relevant range of birth years.

Note that a cricketer who was born in 1869 would be 45 in 1914, while a cricketer who was born in 1896 would be 18 in 1914.

```
## Use subset() from base R
ww1kia <- subset(cricketer,
                 kia==1 & (year+life)%in% 1914:1918)
range(ww1kia$year)
```

```
[1] 1869 1896
```

Alternatively, use `filter()` from *dplyr*:

```
ww1kia <- filter(cricketer,
                 kia==1, (year+life)%in% 1914:1918)
```

For each year of birth between 1869 and 1896, the following expresses the number of cricketers killed in action as a fraction of the total number of cricketers (in action or not) who were born in that year:

```
## Use filter(), group_by() and summarise() from dplyr
crickChoose <- filter(cricketer,
                      year%in%(1869:1896), ((kia==1)|(year+life)>1918))
gpByYearKIA <- group_by(crickChoose, year)
crickKIAyrs <- dplyr::summarise(gpByYearKIA,
                                kia=sum(kia), all=length(year), prop=kia/all)
crickKIAyrs[1:4, ]
```

```
# A tibble: 4 x 4
   year   kia   all   prop
  <int> <int> <int>  <dbl>
1  1869     1    37 0.0270
2  1870     2    36 0.0556
3  1871     1    45 0.0222
4  1872     0    39 0
```

For an introduction to *dplyr*, enter:

```
vignette("introduction", package="dplyr")
```

6.2.3 reshape2: *`melt()`, `acast()` & `dcast()`*

The *reshape2* package has functions that move between a dataframe layout where selected columns are unstacked, and a layout where they are stacked. In moving from an unstacked to a stacked layout, column names become levels of a factor. In the move back from stacked to unstacked, factor levels become column names.

Here is an example of the use of `melt()`:

```
## Create dataset Crimean, for use in later calculations
library(HistData)    # Nightingale is from this package
library(reshape2)    # Has the function melt()
Crimean <- melt(Nightingale[,c(1,8:10)], "Date")
names(Crimean) <- c("Date", "Cause", "Deaths")
Crimean$Cause <- factor(sub("\\.rate", "", Crimean$Cause))
Crimean$Regime <- ordered(rep(c(rep('Before', 12), rep('After', 12)), 3),
                          levels=c('Before', 'After'))
formdat <- format.Date(sort(unique(Crimean$Date)), format="%d %b %y")
Crimean$Date <- ordered(format.Date(Crimean$Date,
                        format="%b %y"), levels=formdat)
```

The dataset is now in a suitable form for creating a Florence Nightingale style wedge plot, in Figure C.2.

The dataset `Crimean` has been included in the *DAAGviz* package.

Reshaping data for Motion Chart display – an example

The following inputs and displays World Bank Development Indicator data that has been included with the package *DAAGviz*:

```
## DAAGviz must be installed, need not be loaded
path2file <- system.file("datasets/wdiEx.csv", package="DAAGviz")
wdiEx <- read.csv(path2file)
print(wdiEx, row.names=FALSE)
```

```
 Country.Name Country.Code      Indicator.Name Indicator.Code    X2010    X2000
    Australia          AUS Labor force, total SL.TLF.TOTL.IN 1.17e+07 9.62e+06
    Australia          AUS  Population, total    SP.POP.TOTL 2.21e+07 1.92e+07
        China          CHN Labor force, total SL.TLF.TOTL.IN 8.12e+08 7.23e+08
        China          CHN  Population, total    SP.POP.TOTL 1.34e+09 1.26e+09
```

A *googleVis* Motion Chart does not make much sense for this dataset as it stands, with data for just two countries and two years. Motion charts are designed for showing how scatterplot relationships, here between forest area and population, have changed over a number of years. The dataset will however serve for demonstrating the reshaping that is needed.

For input to Motion Charts, we want indicators to be columns, and years to be rows. The `melt()` and `dcast()`[3] functions from the *reshape2* package can be used to achieve the desired result. First, create a single column of data, indexed by classifying factors:

[3] Note also `acast()`, which outputs an array or a matrix.

```
library(reshape2)
wdiLong <- melt(wdiEx, id.vars=c("Country.Code",
                "Indicator.Name"),
                measure.vars=c("X2000", "X2010"))
## More simply: wdiLong <- melt(wdiEx[, -c(2,4)])
wdiLong
```

```
  Country.Code     Indicator.Name variable    value
1          AUS Labor force, total    X2000 9.62e+06
2          AUS  Population, total    X2000 1.92e+07
3          CHN Labor force, total    X2000 7.23e+08
4          CHN  Population, total    X2000 1.26e+09
5          AUS Labor force, total    X2010 1.17e+07
6          AUS  Population, total    X2010 2.21e+07
7          CHN Labor force, total    X2010 8.12e+08
8          CHN  Population, total    X2010 1.34e+09
```

Now use `dcast()` to “cast” the data frame into a form where the indicator variables are columns:

If a matrix or array is required, use `acast()` in place of `dcast()`.

```
names(wdiLong)[3] <- "Year"
wdiData <- dcast(wdiLong,
                 Country.Code+Year ~ Indicator.Name,
                 value.var="value")
wdiData
```

```
  Country.Code  Year Labor force, total Population, total
1          AUS X2000           9.62e+06          1.92e+07
2          AUS X2010           1.17e+07          2.21e+07
3          CHN X2000           7.23e+08          1.26e+09
4          CHN X2010           8.12e+08          1.34e+09
```

A final step is to replace the factor `Year` by a variable that has the values 2000 and 2010.

```
wdiData <- within(wdiData, {
   levels(Year) <- substring(levels(Year),2)
   Year <- as.numeric(as.character(Year))
})
wdiData
```

```
  Country.Code Year Labor force, total Population, total
1          AUS 2000           9.62e+06          1.92e+07
2          AUS 2010           1.17e+07          2.21e+07
3          CHN 2000           7.23e+08          1.26e+09
4          CHN 2010           8.12e+08          1.34e+09
```

6.3 *Session and Workspace Management*

6.3.1 *Keep a record of your work*

A recommended procedure is to type commands into an editor window, then sending them across to the command line. This makes it possible to recover work on those hopefully rare occasions when the session aborts.

Be sure to save the script file from time to time during the session, and upon quitting the session.

6.3.2 *Workspace management*

For tasks that make heavy memory demands, it may be important to ensure that large data objects do not remain in memory once they are no longer needed. There are two complementary strategies:

- Objects that cannot easily be reconstructed or copied from elsewhere, but are not for the time being required, are conveniently saved to an image file, using the `save()` function.
- Use a separate working directory for each major project.

Note the utility function `dir()` (get the names of files, by default in the current working directory).

Use `getwd()` to check the name and path of the current working directory. Use `setwd()` to change to a new working directory, while leaving the workspace contents unchanged.

Several image files ("workspaces") that have distinct names can live in the one working directory. The image file, if any, that is called **.RData** is the file whose contents will be loaded at the beginning of a new session in the directory.

The removal of clutter: Use a command of the form `rm(x, y, tmp)` to remove objects (here `x`, `y`, `tmp`) that are no longer required.

As noted in Section 2.2.2, a good precaution can be to make an archive of the workspace before such removal.

Movement of files between computers: Files that are saved in the default binary save file format, as above, can be moved between different computer systems.

Further possibilities – saving objects in text form: An alternative to saving objects[4] in an image file is to dump them, in a text format, as dump files, e.g.

[4] Dumps of S4 objects and environments, amongs others, cannot currently be retrieved using `source()`. See `help(dump)`.

```
volume <- c(351, 955, 662, 1203, 557, 460)
weight <- c(250, 840, 550, 1360, 640, 420)
dump(c("volume", "weight"), file="books.R")
```

The objects can be recreated [5] from this "dump" file by inputting the lines of **books.R** one by one at the command line. This is what, effectively, the command `source()` does.

[5] The same checks are performed on dump files as if the text had been entered at the command line. These can slow down entry of the data or other object. Checks on dependencies can be a problem. These can usually be resolved by editing the R source file to change or remove offending code.

```
source("books.R")
```

For long-term archival storage, dump (**.R**) files may be preferable to image files. For added security, retain a printed version. If a problem arises (from a system change, or because the file has been corrupted), it is then possible to check through the file line by line to find what is wrong.

6.4 *Computer Intensive Computations*

Computations may be computer intensive because of the size of datasets. Or the computations may themselves be time-consuming, even for data sets that are of modest size.

Note that using all of the data for an analysis or for a plot is not always the optimal strategy. Running calculations separately on different subsets may afford insights that are not otherwise available. The subsets may be randomly chosen, or they may be chosen to reflect, e.g., differences in time or place.

Where it is necessary to look for ways to speed up computations, it is important to profile computations to find which parts of the code are taking the major time. Really big improvements will come from implementing key parts of the calculation in C or Fortran rather than in an application oriented language such as R or Python. Python may do somewhat better than R.

The relatively new Julia language appears to offer spectacular improvements on both R and Python, with times that are within a factor of 2 of the Fortran or C times. See `http://julialang.org/`.

There can be big differences between the alternatives that may be available in R for handling a calculation. Some broad guidelines will now be provided, with examples of how differences in the handling of calculations can affect timings.

6.4.1 Considerations for computations with large datasets

Consider supplying, matrices in preference to data frames: Most of R's modeling functions (regression, smoothing, discriminant analysis, etc.) are designed to accept data frames as input. The computational and associated memory requirements of the steps needed to form the matrices used for the numerical computations can. for large datasets, generate large overheads. The matrix computations that follow use highly optimized compiled code, and are much more efficient than if directly implemented in R code. Where it is possible to directly input the matrices that will be required for the calculations, this can greatly reduce the time and memory requirements.

Biological expression array applications are among those that are commonly designed to work with data that is in a matrix format. The matrix or matrices may be components of a more complex data structure.

Matrix arithmetic can be faster than the equivalent computations that use `apply()`. Here are timings for alternatives that find the sums of rows of the matrix `xy` that was generated thus:

```
xy <- matrix(rnorm(5*10^7), ncol=100)
dim(xy)
```

Timings are on a mid 2012 1.8 Ghz Intel i5 Macbook Air laptop with 8 gigabytes of random access memory.

```
system.time(xy+1)
```

```
   user  system elapsed
  0.155   0.242   0.396
```

```
xy.df <- data.frame(xy)
system.time(xy.df+1)
```

```
   user  system elapsed
  0.151   0.117   0.268
```

Use efficient coding: Matrix arithmetic can be faster than the equivalent computations that use `apply()`. Here are timings for some alternatives that find the sums of rows of the matrix `xy` above:

	user	system	elapsed
`apply(xy,1,sum)`	0.528	0.087	0.617
`xy %*% rep(1,100)`	0.019	0.001	0.019
`rowSums(xy)`	0.034	0.001	0.035

The bigmemory project: For details, go to `http://www.bigmemory.org/`. The *bigmemory* package for R "supports the creation, storage, access, and manipulation of massive matrices". Note also the associated packages *biganalytics*, *bigtabulate*, *synchronicity*, and *bigalgebra*.

The data.table package: This allows the creation of `data.table` objects from which information can be quickly extracted, often in a fraction of the time required for extracting the same information from a data frame. The package has an accompanying vignette. To display it (assuming that the package has been installed), type

On 64-bit systems, massive data sets, e.g., with tens or hundreds of millions of rows, are possible. For such large data objects, the time saving can be huge.

```
vignette("datatable-intro", package="data.table")
```

6.5 Summary

`apply()`, and `sapply()` can be useful for manipulations with data frames and matrices. Note also the functions `melt()`, `dcast()` and `acast()` from the *reshape2* package.

Careful workspace management is important when files are large. It pays to use separate working directories for each different project, and to save important data objects as image files when they are, for the time being, no longer required.

In computations with large datasets, operations that are formally equivalent can differ greatly in their use of computational resources.