Statistics 771

R Object Systems

Managing R Projects
Creating R Packages

Douglas Bates

Object systems available in R

- R has two object systems available, known informally as the S3 and the S4 systems.
- S3 objects, classes and methods have been available in R since its inception. They correspond to the system described in Statistical Models in S (1990).
- S4 objects, classes and methods have been added recently. They correspond to the system described in Programming with Data (1998).
- Both systems are based on generic functions and method dispatch according to the class of one or more arguments.
- Many common functions in R are defined as (S3) generic functions. (Existing functions automatically become S4 generic functions as soon as an S4 method is defined.)

Why use classes?

Classes allow you to

- Encapsulate the representation of an object (information hiding)
- Specialize the behavior of your functions to your objects
- Specialize the behavior of system functions to your objects
- Dependably access slots in your objects from within C code

Information hiding

The major reason for using classes is to hide implementation details from the user. The user sees only the output from methods for print, summary, plot and other generic functions and doesn't need to know the internal structure.

Standard statistical packages manage this by not having any accessible internal structure: a linear model, for example, cannot be stored in a variable and acted on by the language. There is no user access to how SPSS, for example, stores a linear model. In S everything can be stored and manipulated, so some form of information hiding must be added.

An example - variance components

- Suppose that we are simulating results from maximum likelihood (ML) or restricted maximum likelihood (REML) estimation of variance components in mixed-effects models.
- These estimates can be zero. In practice what happens is that the 'zero' estimates are very small compared to the 'non-zero' estimates.
- For a summary you may want just a count the number of zero values (defined perhaps as any value whose magnitude is less than 1.0E-7 * the maximum) and the summary of those that are non-zero.
- Similarly for a plot you want a qqnorm plot of the non-zero values.
- We will use a class called varest (S3) or varEst (S4) with a single numeric component or "slot" to represent these estimates.

S3 implementation

- A object may have a `"class"` attribute with a list of one or more classes: eg `c("glm","lm")`. This is set and read by the `class()` function.
- A generic function, such as `plot` contains a call to `UseMethod`
- R replaces the call to `plot` with a call to the method `plot.glm`. The generic and the first class name are just pasted together.
- A method can call `NextMethod`, which looks for the next classe name (`"lm"`) and calls the appropriate method.
- If a generic has no method for a class the next class is tried and so on. Finally the default method is called (eg `plot.default`) or an error is reported.

varest as an S3 class

Suppose that we have the result of such a simulation as the object tt. Without a class its summary is difficult to read

```
> summary(tt)
```

```
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
2.539e-12 1.214e-01 6.715e-01 2.647e+00 1.935e+00 4.445e+01
```

varest as an S3 class (cont'd)

If we set the class and use a method, the result is more informative

```
> summary.varest = function(object, ...) {
+     object = as.numeric(object)
+     small = object < (1e-07) * max(object)
+     val = summary(object[!small])
+     if (any(small))
+         val = c(val, Zero = sum(small))
+     val
+ }
> class(tt) = "varest"

> summary(tt)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     Zero
 0.07821  0.50350  1.30800  3.45200  2.36300 44.45000  7.00000
```

S4 implementation

To use S4 classes you must attach the methods package. All classes and methods must be formally declared.

```
> library("methods")
> setClass("varEst", "numeric")
> setMethod("summary", "varEst", function(object, ...) {
+     object = as.numeric(object)
+     small = object < (1e-07) * max(object)
+     val = summary(object[!small])
+     if (any(small))
+         val = c(val, Zero = sum(small))
+     val
+ })
> class(tt) = "varEst"

> summary(tt)

    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     Zero
 0.07821  0.50350  1.30800  3.45200  2.36300 44.45000  7.00000
```

How to manage your R use

- Ways to store things in R
- Multiple projects
- Printing results
- Using R output

Storage in R

Options for storage

Workspace When R starts it will read in the file `.RData`, and when it exits you are given the chance to save the workspace to that file (you can save it at any time with `save.image()`). This saves everything except loaded packages and is the equivalent of the `.Data` directory in S-PLUS.

Binary files The `save()` command puts specified functions and data in a binary file. This file can be `attach()`ed (like a directory in S-PLUS) or `load()`ed.

Source code . Instead of saving results and data they can be recreated as needed from source code.

Multiple projects

There are two extreme ways to handle multiple projects in R

- Store each project in a separate directory and use the `.RData` file to hold everything. If you want to share functions or objects with another project then explicitly export and import them. The `.RData` file is primary; any transcripts or source files are documentation.
- Store everything as source code. For every piece of analysis have a file that reads in data, processes it, and possibly `save`s a modified version to a new file. The source is primary, any saved files are just a labour-saving device.

The same choices apply to programming as well as data analysis, and to other dialects of S.

Workspace is primary

The first method is common among new users of S-PLUS. Many of us subsequently find that we aren't sufficiently organised to be sure that we keep records of how every analysis was done.

This approach is riskier in R than in S-PLUS.

- In R the workspace is only saved when you explicitly ask for it; in S-PLUS it is saved frequently
- In R a corrupted `.RData` is likely to be completely unreadable, in S-PLUS many objects will still be recoverable.

It makes sense for short-term projects, especially if data loss is not critical, or for very careful people.

Source is primary

Managing projects is easy when everything can be reconstructed from source files. These files and intermediate data files can be stored in a project directory where they are easy to find and are automatically time-stamped by the operating system.

Emacs Speaks Statistics (ESS) is particularly useful for this style of R use. With R running in an Emacs window you can run sections of code with a few keystrokes.

Printing results

To save and print R output either

- Run R under Emacs: the R session is then sitting in an editing buffer
- Use `sink()` to temporarily redirect output to a file
- Under Windows, the contents of the console can be saved or printed.

Saving and printing graphics

- The current graphics device can be printed with `dev.print()` or saved as postscript with `dev.copy2eps`
- A file-based graphics device can be opened before creating the plot and closed afterwards (eg `pdf`,`postscript`,`WMF`,`png`)
- Under Windows there is a menu option to print or save the graphics device.

Using R output

Retyping results from R is potentially inaccurate as well as time-wasting, so it is useful to produce output that can be incorporated in reports. R doesn't do this very well

- `write.table` produces formatted text, and can do most of the work of producing LaTeX tables.
- In principle the DCOM link could be used to automate reports in MS Word, and the XML package should become useful for transferring formatted results as XML becomes more popular.

R packaging system

- Why package?
- Structure of R packages
- Documentation
- `package.skeleton()`
- `CMD check` and `CMD build`
- Distributing packages.

Why package?

R packages provide a way to manage collections of functions or data and their documentation.

- Dynamically loaded and unloaded: the package only occupies memory when it is being used.
- Easily installed and updated: the functions, data and documentation are all installed in the correct places by a single command that can be executed either inside or outside R.
- Customisable by users or administrators: in addition to a site-wide library, users can have one or more private libraries of packages.
- Validated: R has commands to check that documentation exists, to spot common errors, and to check that examples actually run

Why package? (2)

- Most users first see the packages of functions distributed with R or from CRAN. The package system allows many more people to contribute to R while still enforcing some standards.
- Data packages are useful for teaching: datasets can be made available together with documentation and examples. For example, Doug Bates translated data sets and analysis exercises from an engineering statistics textbook into the `Devore5` package
- Private packages are useful to organise and store frequently used functions or data. I have packaged ICD9 codes, for example.

Structure of R packages

The basic structure of package is a directory, commonly containing

- A `DESCRIPTION` file with descriptions of the package, author, and license conditions in a structured text format that is readable by computers and by people
- An `INDEX` file listing all the functions and data (and optionally with other descriptive information). This can be generated automatically.
- A `man/` subdirectory of documentation files
- An `R/` subdirectory of R code
- A `data/` subdirectory of datasets
- A `src/` subdirectory of C, Fortran or C++ source

Structure of R packages (cont)

Less commonly it contains

- `tests/` for validation tests
- `exec/` for other executables (eg Perl or Java)
- `inst/` for miscellaneous other stuff.
- A `configure` script to check for other required software or handle differences between systems.

Apart from `DESCRIPTION` and `INDEX` these are mostly optional, though any useful package will have `man/` and at least one of `R/` and `data/`.

Everything about packages is described in more detail in the Writing R Extensions manual distributed with R.

Data formats

The `data()` command loads datasets from packages. These can be

- Rectangular text files, either whitespace or comma-separated
- S source code, produced by the `dump()` function in R or S-PLUS.
- R binary files produced by the `save()` function.

The file type is chosen automatically, based on the file extension.

Documentation

The R documentation format looks rather like LaTeX.

```
\name{birthday}  % name of the file
\alias{qbirthday} % the functions it documents
\alias{pbirthday}
% one-line title of the documentation page
\title{Probability of coincidences}
% short description
\description{
  Computes approximate answers to a generalised "birthday
  paradox" problem. \code{pbirthday} computes the probability of a
  coincidence and \code{qbirthday} computes the number of
  observations needed to have a specified probability of coincidence.
}
\usage{ % how to invoke the function
pbirthday(prob=0.5,classes=365, coincident=2)
qbirthday(n,classes=365,  coincident=2)
}
```

Documentation (2)

The file continues with sections

- `\arguments`, listing the arguments and their meaning
- `\value`, describing the returned value
- `\details`, a longer description of the function, if necessary.
- `\references`, giving places to look for detailed information
- `\seealso`, with links to related documentation
- `\examples`, with examples of how to use the functions.
- `\keyword` for indexing

There are other possible sections, and ways of specifying equations, urls, links to other R documentation, and more.

Documentation (3)

The documentation files can be converted into HTML, plain text, GNU info format, PostScript, and PDF. They can also be converted into the old nroff-based S help format.

The packaging system can check that all objects are documented, that the `usage` corresponds to the actual definition of the function, and that the `examples` will run. This enforces a minimal level of accuracy on the documentation.

There is an Emacs mode for editing R documentation, and a function `prompt()` to help produce it.

Setting up a package

The `package.skeleton()` function partly automates setting up a package with the correct structure and documentation.

The `usage` section from `help(package.skeleton)` looks like

```
> package.skeleton(name = "anRpackage", list, environment = .GlobalEnv,
+     path = ".", force = FALSE)
```

Given a collection of R objects (data or functions) specifed by a `list` names or an `environment` it creates a package called `name` in the directory specified by `path`.

The objects are sorted into data (put in `data/`) or functions (`R/`), skeleton help files are created for them using `prompt()` and a `DESCRIPTION` file is created. The function then prints out a list of things for you to do next.

Building a package

`R CMD build` (`Rcmd build` on Windows) will create a compressed package file from your package directory.

It does this in a reasonably intelligent way, omitting object code, emacs backup files, and other junk. The resulting file is easy to transport across systems and can be `INSTALL`ed without decompressing.

There are options to store help and data files in permanently compressed form. This is particularly useful on older Windows systems where packages with many small files waste a lot of disk space.

Binary and source packages

`CMD build` makes source packages. If you want to distribute a package that contains C or Fortran for Windows users, they may well need a binary package, as compiling under Windows requires downloading exactly the right versions of quite a number of tools. The R for Windows FAQ describes this process.

Binary packages are created by `INSTALL`ing the source package and then making a zip file of the installed version.

Checking a package

`R CMD check` (`Rcmd check` in Windows) helps you do QA/QC on packages.

- The directory structure and the format of `DESCRIPTION` and `INDEX` are checked.
- The documentation is converted into text, HTML, and LaTeX, and run through `latex` if available.
- The examples are run
- Any tests in the `tests/` subdirectory are run
- Undocumented objects, and those whose `usage` and definition disagree are reported.

Distributing packages

If you have a package that does something useful and is well-tested and documented, you might want other people to use it too. Contributed packages have been very important to the success of R (and before that of S).

Packages can be submitted to CRAN by ftp.

- The CRAN maintainers will make sure that the package passes `CMD check` (and will keep improving `CMD check` to find more things for you to fix in future versions).
- Other users will complain if it doesn't work on more esoteric systems and no-one will tell you how helpful it has been.
- But it will be appreciated. Really.