Playing nicely together

Thomas Lumley

R Core Development Team
and University of Washington

Auckland — 2009–11–28

Objects

Many functions in R return objects, which are collections of information that can be operated on by other functions.

In more extreme object-oriented languages objects have no user-serviceable parts. In R you can always get direct access to the internals of an object. You shouldn't use this access if there is another way to get the information: the developer may change the internal structure and break your code.

Use `str` and `names` to guess the internal structure.

Generics and methods

Many functions in R are generic. This means that the function itself (eg `plot`, `summary`, `mean`) doesn't do anything. The work is done by methods that know how to plot, summarize or average particular types of information. Earlier I said this was done by magic. Here is the magic.

If you call `summary` on a `data.frame`, R works out that the correct function to do the work is `summary.data.frame` and calls that instead. If there is no specialized method to summarize the information, R will call `summary.default`

You can find out all the types of data that R knows how to summarize with two functions

```
> methods("summary")
 [1] summary.Date              summary.POSIXct      summary.POSIXlt
 [4] summary.aov               summary.aovlist      summary.connection
 [7] summary.data.frame        summary.default      summary.ecdf*
[10] summary.factor            summary.glm          summary.infl
[13] summary.lm                summary.loess*       summary.manova
[16] summary.matrix            summary.mlm          summary.nls*
[19] summary.packageStatus* summary.ppr*         summary.prcomp*
[22] summary.princomp*         summary.stepfun      summary.stl*
[25] summary.table             summary.tukeysmooth*

    Non-visible functions are asterisked
> getMethods("summary")
NULL
```

There are two functions because S has two object systems, for historical reasons (called S3 and S4)

Methods

The class and method system makes it easy to add new types of information (eg survey designs) and have them work just like the built-in ones.

Some standard methods are

- `print`, `summary`: everything should have these
- `plot` or `image`: if you can work out an obvious way to plot the thing, one of these functions should do it.
- `coef`, `vcov`: Anything that has parameters and variance matrices for them should have these.
- `anova`, `logLik`, `AIC`: a model fitted by maximum likelihood should have these.
- `residuals`: anything that has residuals should have this.

Use the methods, rather than direct access to components (eg `coef(model)` not `model$coefficients`)

New classes

Creating a new class is easy

```
class(x) <- "duck"
```

R will now automatically look for the `print.duck` method for the generic `print`.

R doesn't know anything about the structure of the class: if there are generics like `look`, `walk`, and `quack` you need to be sure that `x` looks like a duck, walks like a duck, and quacks like a duck.

New generics

New generics are almost as easy:

```
> print
function (x, ...)
UseMethod("print")
```

A generic is just a function with a call to `UseMethod()`.

New generics

By default, the method is chosen based on the class of the first argument, but you can specify a different argument

```
> library(survey)
> svymean
function (x, design, na.rm = FALSE, ...)
{
    .svycheck(design)
    UseMethod("svymean", design)
}
```

The method is chosen based on `design`, but the argument that specifies which variables to mean is first, for consistency with the rest of R.

[The `.svycheck()` call detects designs created with an obsolete version of the package.]

Example: ROC curves

The Receiver Operating Characteristic curve describes the ability of a ordinal variable T to predict a binary variable D.

For any threshold c we can compute

- Sensitivity of $T \geq c$ for D: $P(T \geq c|D = 1)$
- Specificity of $T \geq c$ for D: $P(T < c|D = 0)$

We plot sensitivity vs $1-$specificity for all values of c to give the ROC curve.

Example: ROC curves

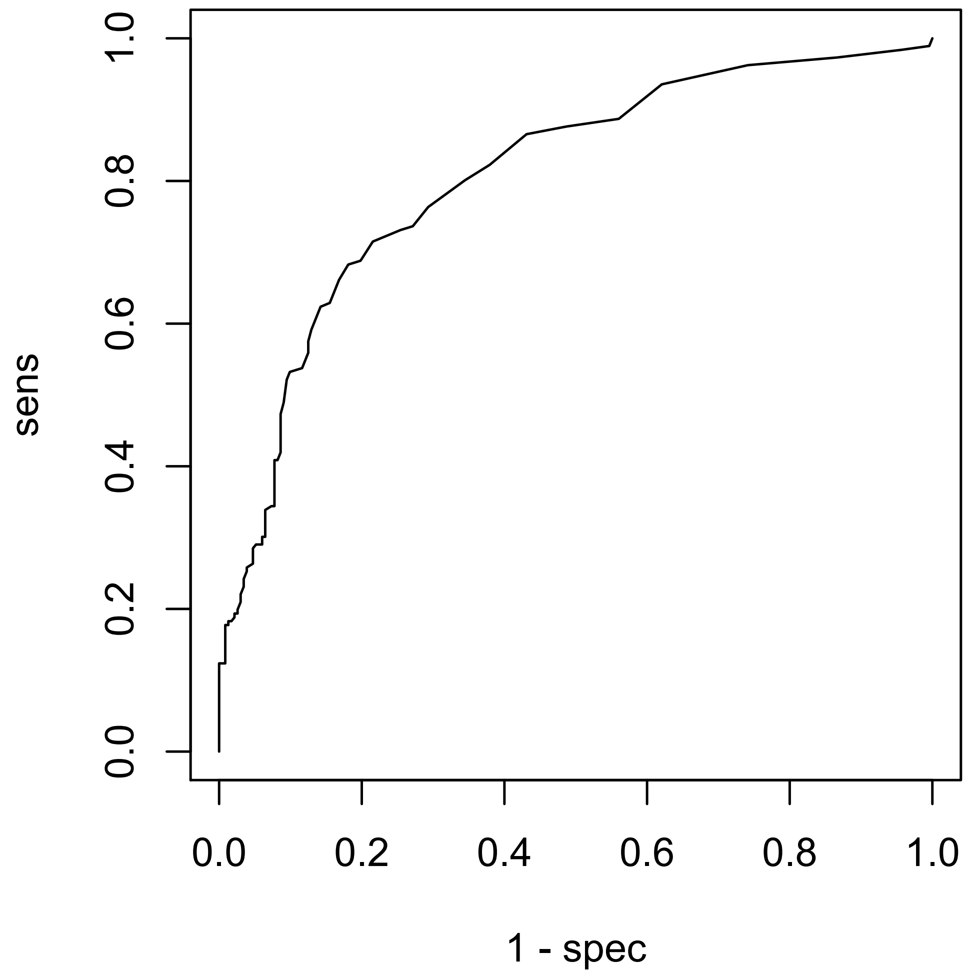
Simple code

```
drawROC<-function(T,D){
   cutpoints <- c(-Inf, sort(unique(T)), Inf)
   sens <- sapply(cutpoints, function(c) sum(T>=c & D==1)/sum(D==1))
   spec <- sapply(cutpoints, function(c) sum(T<c & D==0)/sum(D==0))
   plot(1-spec, sens, type="l")
}
```

For example

```
data(pbc, package="survival")
par(pty="s")
with(pbc, drawROC(bili, status>0))
```

Example: ROC curves



Example: ROC curves

A slightly more efficient version using use the vectorized `cumsum` rather than the implied loop of `sapply`.

```
drawROC<-function(T,D){
  DD <- table(-T,D)
  sens <- cumsum(DD[,2])/sum(DD[,2])
  mspec <- cumsum(DD[,1])/sum(DD[,1])
  plot(mspec, sens, type="l")
}
```

We want to make this return an **ROC object** that can be plotted and operated on in other ways

ROC curve object

```
ROC<-function(T,D){
  DD <- table(-T,D)
  sens <- cumsum(DD[,2])/sum(DD[,2])
  mspec <- cumsum(DD[,1])/sum(DD[,1])
  rval <- list(tpr=sens, fpr=mspec,
               cutpoints=rev(sort(unique(T))),
               call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Instead of plotting the curve we return the data needed for the plot, plus some things that might be useful later. `sys.call()` is a copy of the call.

Methods

We need a `print` method to stop the whole contents of the object being printed

```
print.ROC<-function(x,...){
   cat("ROC curve: ")
   print(x$call)
}
```

Methods

A plot method

```
plot.ROC <- function(x, xlab="1-Specificity",
                        ylab="Sensitivity", type="l",...){
        plot(x$fpr, x$tpr, xlab=xlab, ylab=ylab, type=type, ...)
}
```

We specify some graphical parameters in order to set defaults for them. Others are automatically included in `...` and will be passed down until they reach a function that knows how to handle them.

Methods

We want to be able to add lines to an existing plot

```
lines.ROC <- function(x, ...){
lines(x$fpr, x$tpr, ...)
}
```

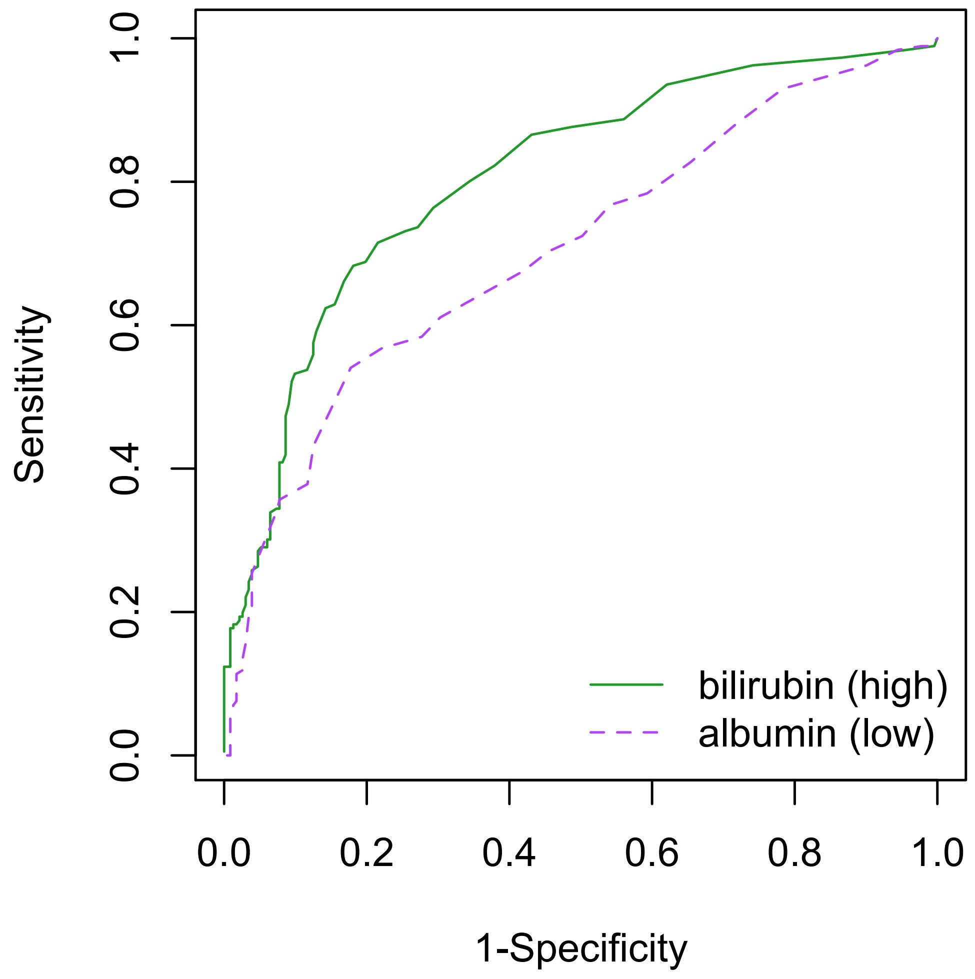
and also be able to identify cutpoints

```
identify.ROC<-function(x, labels=NULL, ...,digits=1)
{
  if (is.null(labels))
    labels<-round(x$cutpoints,digits)
  identify(x$fpr, x$tpr, labels=labels,...)
}
```

Methods

```
> roc1 <- with(pbc, ROC(bili, status>0))
> roc1
ROC curve: ROC(bili, status > 0)
> roc2 <- with(pbc, ROC(protime, status>0))
> roc2
ROC curve: ROC(protime, status > 0)
> plot(roc1, col="forestgreen")
> lines(roc2, col="purple", lty=2)
> legend("bottomright",lty=1:2,col=c("forestgreen","purple"),
     legend=c("bilirubin (high)", "albumin (low)"), bty="n")
```

Methods


Sensitivity
1-Specificity
0.0
0.2
0.4
0.6
0.8
1.0
bilirubin (high)
albumin (low)

S4 classes

These provide formal definitions for classes and methods, so all objects of the same class have the same structure and so methods don't have to be guessed by name.

- `setClass()` defines a class
- `new()` creates a new object of a specified class
- `setMethod()` defines a method

S4 classes

```
setClass("ROC",
   representation(fpr="numeric",tpr="numeric",
       cutpoints="numeric", call="call")
   )

ROC<-function(T,D){
  DD <- table(-T,D)
  sens <- cumsum(DD[,2])/sum(DD[,2])
  mspec <- cumsum(DD[,1])/sum(DD[,1])
  rval <- new("ROC", tpr=sens, fpr=mspec,
               cutpoints=rev(sort(unique(T))),
               call=sys.call())
  rval
}
```

Methods

The `print` generic is renamed `show` in S4

```
setMethod("show","ROC",
   function(object){
    cat("ROC curve: ")
    print(object@call)
  }
)
```

Use `@` rather than `$` to refer to slots in an S4 class: 'information hiding'

Methods

If there isn't a built-in S4 generic (eg `plot`), `setMethod()` will create one automatically, or you can do it yourself with `setGeneric`. The previous S3 generic will become the default method.

```
setMethod("plot" , c("ROC","missing"),
   function(x, y, xlab="1-Specificity",
                     ylab="Sensitivity", type="l",...){
        plot(x@fpr, x@tpr, xlab=xlab, ylab=ylab, type=type, ...)
})
```

`plot()` has two arguments, x and y. The method is chosen based on the classes of both arguments.

This method is called when the x an ROC object and y is not given.

Methods

```
> setGeneric("lines")
[1] "lines"
> lines
standardGeneric for "lines" defined from package "graphics"

function (x, ...)
standardGeneric("lines")
<environment: 0x180ceaf4>
Methods may be defined for arguments: x
Use  showMethods("lines")  for currently available ones.

setMethod("lines", "ROC",
   function(x,  ...){
        lines(x@fpr, x@tpr,  ...)
})
```

Model objects

Typical statistics packages fit a model and output the results. In S a model object is created that stores all the information about the fitted model. Coefficients, diagnostics, and other model summaries are produced by methods for this object.

Objects again

We might want to construct an ROC curve as a summary of a logistic regression. We need the `ROC` function to be generic, so it can work for regression models or vectors

```
ROC <- function(T,...)  UseMethod("ROC")
```

The previous `ROC` function now becomes the default method

```
ROC.default <-function(T,D){
  DD <- table(-T,D)
  sens <- cumsum(DD[,2])/sum(DD[,2])
  mspec <- cumsum(DD[,1])/sum(DD[,1])
  rval <- list(tpr=sens, fpr=mspec,
               cutpoints=rev(sort(unique(T))),
               call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Objects again

and we need a new `ROC.glm` function that computes T and D from a fitted logistic regression model.

```
ROC.glm<-function(T,...){
  if (!(T$family$family %in%
            c("binomial", "quasibinomial")))
    stop("ROC curves for binomial glms only")

  test<-fitted(T)
  disease<-T$response

  TT<-rev(sort(unique(test)))
  DD<-table(-test,disease)

  sens<-cumsum(DD[,2])/sum(DD[,2])
  mspec<-cumsum(DD[,1])/sum(DD[,1])

  rval<-list(tpr=sens, fpr=mspec,
            cutpoints=TT,call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Formulas and data frames

Many R functions for statistical modelling or graphics specify the variables to use with

- a model formula giving the names of variables (and other aspects of model structure)
- a data frame to look up the variables in

You can do this too.

Review: formula syntax

In regression models

- ~ separates response from predictors
- + specifies predictor terms
- * requests the interaction of two terms (and the two main effects)

For graphics (and for other uses such as the survey package) the formula just specifies a list of variables and * is not used.

The formula has length 3 if it has both left and right sides, with the LHS as the second and RHS as third element.

Using formulas (simple version)

```
f <- function(formula, data){

   tms <- terms(formula)
   mf <- model.frame(tms, data)
   mm <- model.matrix(tms, mf)

  colSums(mm)
}

> data(api, package="survey")
> f(api00~api99*stype, data=apipop)
 (Intercept)        api99       stypeH       stypeM api99:stypeH
        6194      3914069          755         1018       468895
api99:stypeM
      645968
```

Steps

- `terms()` parses a formula to work out what variables will be needed and how they will go into the model
- `model.frame()` gathers together all the necessary variables
- `model.matrix()` makes a design matrix, with interactions, factor contrasts, and other fun stuff.

In a perfect world, all the variables in the formula are in the data frame. In reality, some may be floating free in the workspace.

`model.frame()` looks first in the data frame, then in the place where the formula was defined.

Missing data

By default, `model.frame()` drops observations with missing values (and attaches an attribute to the model frame saying what it dropped).

Sometimes you want to pass through missing values to make it easier to match up observations in different objects:

```
model.frame(formula, data, na.action=na.pass)
```

ROC example

Let's write a function that fits an ROC curve to a single predictor or to a logistic regression with multiple predictors (this uses the S3 version of the functions)

```
ROC.formula <- function(formula, data){
     if (length(formula[[3]])==1){
          tms <-terms(formula)
          mf <- model.frame(tms, data)
          rval <- ROC(mf[,2], mf[,1])
          rval$call <- sys.call()
     } else {
       model <- glm(formula, data, family=binomial)
       rval <- ROC(model)
       rval$call<-sys.call()
     }
     rval
}
```

Making a formula

Creating a formula from a list of variables is one of the few good uses for writing code by string manipulation

```
> make.formula <- function (names)
formula(paste("~", paste(names, collapse = "+")))
> make.formula(names(apipop))
~cds + stype + name + sname + snum + dname + dnum + cname + cnum +
 flag + pcttest + api00 + api99 + target + growth + sch.wide +
 comp.imp + both + awards + meals + ell + yr.rnd + mobility +
 acs.k3 + acs.46 + acs.core + pct.resp + not.hsg + hsg + some.col +
 col.grad + grad.sch + avg.ed + full + emer + enroll + api.stu
```

Look what I made!

Thomas Lumley

R Core Development Team
and University of Washington

Auckland — 2009–11–28

R Packages

The most important single innovation in R is the package system, which provides a cross-platform system for distributing and testing code and data.

The Comprehensive R Archive Network (`http://cran.r-project.org`) distributes (thousands of) public packages, but packages are also useful for internal distribution.

A package consists of a directory with a `DESCRIPTION` file and subdirectories with code, data, documentation, etc. The Writing R Extensions manual documents the package system, and `package.skeleton()` simplifies package creation.

Packaging commands

- `R CMD INSTALL packagename` installs a package.
- `R CMD check packagename` runs the QA tools on the package.
- `R CMD build packagename` creates a package file.

These use Perl and a few other tools not built-in on Windows. The Rtools bundle contains all of these `http://www.murdoch-sutherland.com/Rtools/`

For Mac OS X the basic tools are automatically available.

The DESCRIPTION file

From the survey package

```
Package: survey
Title: analysis of complex survey samples
Description: Summary statistics, generalised linear models,
cumulative link models, Cox models, loglinear models, and
general maximum pseudolikelihood estimation for multistage stratified,
 cluster-sampled, unequally weighted survey samples. Variances by
 Taylor series linearisation or replicate weights. Post-stratification, calibratio
 and raking. Two-phase subsampling designs. Graphics. Predictive margins
 by direct standardization. PPS sampling without replacement.
Version: 3.19
Author: Thomas Lumley
Maintainer: Thomas Lumley <tlumley@u.washington.edu>
License: GPL-2 | GPL-3
Depends: R (>= 2.2.0)
Suggests: survival, MASS, KernSmooth, hexbin, mitools, lattice, RSQLite,
RODBC, quantreg, splines, Matrix, multicore
Enhances: odfWeave.survey
URL: http://faculty.washington.edu/tlumley/survey/
```

The DESCRIPTION file

`Depends:` lists R packages or the version of R needed to build this one.

`Suggests:` lists packages needed eg to run examples.

`Enhances:` are packages that can use this one.

`License:` is important for CRAN packages, and if possible should be one of a standard set.

`Version:` should be X.YYY-ZZZZ and should be higher than the previous version (so `update.packages()` doesn't get confused)

The INDEX file

This also goes in the package directory and contains information about every sufficiently interesting function in the package.

If an `INDEX` file is not present it will be created from the titles of all the help pages. The `INDEX` file is displayed by

```
library(help=packagename)
```

Interpreted code

R code goes in the `R` subdirectory, in files with extension `.s`, `.S`, `.r`, `.R` or `.q`.

The filenames are sorted in ASCII order and then concatenated (one of the few places that R doesn't observe locale sorting conventions).

`R CMD check` detects a number of common errors such as using `T` instead of `TRUE`.

Documentation

Documentation in `.Rd` format (which looks rather like LaTeX) is the the `man` subdirectory.

`R CMD Sd2Rd` will convert S-PLUS documentation (either the old troff format or the new SGML) and `R CMD Rdconv` will do the reverse.

The QA tools check that every object is documented and that the arguments a function is documented to have are the same as the ones it actually has, and that all the examples run. They check for some common coding errors such as `T` for `TRUE`. They find unused or uninitialized variables in your code.

Data

Data go in the `data` subdirectory and are read with the `data()` function.

- ASCII tables with `.tab`, `.txt` or `.TXT`, read using `read.table(,header=TRUE)`
- R source code with `.R` or `.r` extensions, read using `source`
- R binary format with `.Rdata` or `.rda` extensions, read using `load`.

The directory has an index file (`00INDEX`) to provide descriptions of the data files.

Compiled code

C or Fortran code (or other code together with a `Makefile`) goes in the `src` subdirectory.

It is compiled and linked to a DLL, which can be loaded with the `useDynLib` directive in the NAMESPACE file

Obviously this requires suitable compilers. The nice people at CRAN compile Windows and Macintosh versions of packages for you, but only if it can be done without actual human intervention.

The Windows compilers used to build R are in the windows Rtools collection.

The Mac compilers are on the OS X DVD but are not installed by default.

Namespaces

A package namespace allows the package to have private functions that are not visible to the user.

As usual, the invisibility is limited to preventing accidental access: `survey:::nlcon` shows the private function `nlcon` in the survey package.

The `NAMESPACE` file declares the functions that should be visible to the user, with `export()` to make a function visible and `S3method()` to make a method accessible to `UseMethod()` but not visible as a function.

Namespaces

NAMESPACE from leaps package

```
useDynLib(leaps)
export(regsubsets,leaps)
S3method(regsubsets, biglm)
S3method(regsubsets,formula)
S3method(regsubsets,default)
S3method(summary,regsubsets)
S3method(print,summary.regsubsets)
S3method(print,regsubsets)
S3method(plot,regsubsets)
S3method(coef,regsubsets)
S3method(vcov,regsubsets)
```

Similar syntax to export S4 classes and methods.

inst/ and Vignettes

The contents of the `inst` subdirectory are copied on installation. A `CITATION` file can be supplied in `inst` to give information on how to cite the package. These are read by the `citation()` function. Please cite R and packages that you use.

Vignettes, Sweave documents that describe how to carry out particular tasks, go in the `inst/doc/` subdirectory. The Bioconductor project in bioinformatics is requiring vignettes for its packages.

A `NEWS` or `Changelog` file in `inst/` will be used by CRAN on the package web page.

You can put anything else in `inst/` as well.

Tests

Additional validation tests go in the `tests` subdirectory. Any `.R` file will be run, with output going to a corresponding `.Rout` file. Errors will cause `R CMD check` to fail.

If there is a `.Rout.save` file it will be compared to the `.Rout` file, with differences listed to the screen.

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.

Packages can be submitted to CRAN by ftp to `cran.r-project.org`

- 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.

ROC curves (yet again)

We can now put the ROC curve functions into a simple package.

- Use `package.skeleton` to start
- Edit `DESCRIPTION`
- Edit help files
- What should be in the NAMESPACE?