Programmers' Niche

A simple class, in S3 and S4

by Thomas Lumley

ROC curves

Receiver Operating Characteristic curves (ROC curves) display the ability of an ordinal variable to discriminate between two groups. Invented by radio engineers, they are perhaps most used today in describing medical diagnostic tests. Suppose T is the result of a diagnostic test, and D is an indicator variable for the presence of a disease.

The ROC curve plots the true positive rate (sensitivity), $\Pr(T > c|D = 1)$, against the false positive rate (1−specificity), $\Pr(T > c|D = 0)$ for all values of c. Because the probabilities are conditional on disease status the ROC curve can be estimated from either an ordinary prospective sample or from separate samples of cases ($D = 1$) or controls ($D = 0$).

I will start out with a simple function to draw ROC curves, improve it, and then use it as the basis of S3 and S4 classes.

Simply coding up the definition of the ROC curve gives a function that is efficient enough for most practical pruposes. The one necessary optimisation is to realise that $\pm\infty$ and the the unique values of T are the only cutpoints needed. Note the use of `sapply` to avoid introducing an index variable for elements of `cutpoints`.

```
drawROC <-function(T,D){
  cutpoints<-c(-Inf, sort(unique(T)), Inf)
  sens<-sapply(cutpoints,
      function(c) sum(D[T>c])/sum(D))
  spec<-sapply(cutpoints,
      function(c) sum((1-D)[T<=c]/sum(1-D)))

  plot(1-spec, sens, type="l")
}
```

It would usually be bad style to use `T` and `c` as variable names, because of confusion with the S-compatible `T==TRUE` and the built-in function `c()`. In this case the benefit of preserving standard notation is arguably worth the potential confusion.

There is a relatively simple optimisation of the function that increases the speed substantially, though at the cost of requiring T to be a number, rather than just an object for which > and <= are defined.

```
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")
}
```

One pedagogical virtue of this code is that it makes it obvious that the ROC curve must be monotone: the true positive and false positive rates are cumulative sums of non-negative numbers.

Classes and methods

Creating an S3 class for ROC curves is an easy incremental step. The computational and graphical parts of `drawROC` are separated, and an object of class "ROC" is created simply by setting the `class` attribute of the result.

The first thing a new class needs is a `print` method. The `print` function is *generic*, when given an object of class "ROC" it automatically looks for a `print.ROC` function to take care of the work. Failing that, it calls `print.default`, which spews the internal representation of the object all over the screen.

```
ROC<-function(T,D){
   TT<-rev(sort(unique(T)))
   DD<-table(-T,D)

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

  rval<-list(sens=sens, mspec=mspec, test=TT,
             call=sys.call())
  class(rval)<-"ROC"
  rval
}

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

The programmer is responsible for ensuring that the object has all the properties assumed by functions that use ROC objects. If an object has class "duck", R will assume it can `look.duck`, `walk.duck` and `quack.duck`.

Since the main purpose of ROC curves is to be graphed, a `plot` method is also needed. As with `print`, `plot` will look for a `plot.ROC` method when handed an object of class "ROC" to plot.

```
plot.ROC<-function(x, type="b", null.line=TRUE,
     xlab="1-Specificity", ylab="Sensitivity",
     main=NULL, ...){
  par(pty="s")
  plot(x$mspec, x$sens, type=type,
         xlab=xlab, ylab=ylab, ...)
```

```
  if(null.line)
    abline(0, 1, lty=3)
  if(is.null(main))
    main<-x$call
  title(main=main)
}
```

A `lines` method allows ROC curves to be graphed on the same plot and compared. It is a stripped-down version of the `plot` method:

```
lines.ROC<-function(x,...){
  lines(x$mspec, x$sens, ...)
}
```

A method for `identify` allows the cutpoint to be found for any point on the curve:

```
identify.ROC<-function(x, labels=NULL,
                       ...,digits=1)
{
  if (is.null(labels))
    labels<-round(x$test,digits)
  identify(x$mspec, x$sens, labels=labels,...)
}
```

An `AUC` function, to compute the area under the ROC curve, is left as an exercise for the reader.

Generalising the class

The test variable T in an ROC curve may be a model prediction rather than a single biomarker, and ROC curves have been used to summarise the discriminatory power of logistic regression and survival models. A generic constructor function would allow methods for single variables and for models. Here the default method is the same as the original `ROC` function.

```
ROC <- function(T,...) UseMethod("ROC")
ROC.default<-function(T,D,...){
  TT<-rev(sort(unique(T)))
  DD<-table(-T,D)

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

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

The `ROC.glm` method extracts the fitted values from a binomial regression model and uses them as the test variable.

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

  test<-fitted(T)
  disease<-(test+resid(T, type="response"))
  disease<-disease*weights(T)
  if (max(abs(disease %% 1))>0.01)
    warning("Y values suspiciously
                    far from integers")

  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(sens=sens, mspec=mspec,
             test=TT,call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

S4 classes and method

In the new S4 class system provided by the "methods" package, classes and methods must be registered. This ensures that an object has the components required by its class, and avoids the ambiguities of the S3 system. For example, it is not possible to tell from the name that `t.test.formula` is a method for `t.test` while `t.data.frame` is a method for `t` (and `t.test.cluster`, in the "Design" package, is neither).

As a price for this additional clarity, the S4 system takes a little more planning, and can be clumsy when a class needs to have components that are only sometimes present.

The definition of the ROC class is very similar to the calls used to create the ROC objects in the S3 functions.

```
setClass("ROC",
  representation(sens="numeric",mspec="numeric",
                 test="numeric",call="call"),
  validity=function(object)  {
     length(object@sens)==length(object@mspec) &&
     length(object@sens)==length(object@test)
  }
)
```

The first argument to `setClass` gives the name of the new class. The second describes the components (slots) that contain the data. Optional arguments include a validity check. In this case the ROC curve contains three numeric vectors and a copy of the call that created it. The validity check makes sure that the lengths of the three vectors agree. It could also check that the vectors were appropriately ordered, or that the true and false positive rates were between 0 and 1.

In contrast to the S3 class system, the S4 system requires all creation of objects to be done by the `new`

function. This checks that the correct components are present and runs any validity checks. Here is the translation of the ROC function.

```
ROC<-function(T,D){
  TT<-rev(sort(unique(T)))
  DD<-table(-T,D)

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

  new("ROC", sens=sens, mspec=mspec,
             test=TT,call=sys.call())

}
```

Rather than print, S4 objects use show for display. Here is a translation of the print method from the S3 version of the code

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

The first argument of setMethod is the name of the generic function (show). The second is the *signature*, which specifies when the method should be called. The signature is a vector of class names, in this case a single name since the generic function show has only one argument. The third argument is the method itself. In this example it is an anonymous function; it could also be a named function. The notation object@call is used to access the slots defined in setClass, in the same way that $ is used for list components. The @ notation should be used only in methods, although this is not enforced in current versions of R.

The plot method shows some of the added flexibility of the S4 method system. The generic function plot has two arguments, and the chosen method can be based on the classes of either or both. In this case a method is needed for the case where the first argument is an ROC object and there is no second argument, so the signature of the method is c("ROC","missing"). In addition to the two arguments x and y of the generic function, the method has all the arguments needed to customize the plot, including a ... argument for further graphical parameters.

```
setMethod("plot", c("ROC","missing"),
    function(x, y, type="b", null.line=TRUE,
             xlab="1-Specificity",
             ylab="Sensitivity",
             main=NULL, ...){
       par(pty="s")
       plot(x@mspec, x@sens, type=type,
          xlab=xlab, ylab=ylab, ...)
       if(null.line)
          abline(0,1,lty=3)
       if(is.null(main))
          main<-x@call
       title(main=main)
       }
)
```

Creating a method for lines appears to work the same way

```
setMethod("lines","ROC",
    function(x,...)
        lines(x@mspec, x@sens,...)
)
```

In fact, things are more complicated. There is no S4 generic function for lines (unlike plot and show). When an S4 method is set on a function that is not already an S4 generic, a generic function is created. If you knew that lines was not already an S4 generic it would be good style to include a specific call to setGeneric, to make clear what is happening. Looking at the function lines before

```
> lines
function (x, ...)
UseMethod("lines")
<environment: namespace:graphics>
```

and after the setMethod call

```
> lines
standardGeneric for "lines" defined from
 package "graphics"

function (x, ...)
standardGeneric("lines")
<environment: 0x2fc68a8>
Methods may be defined for arguments: x
```

shows what happens.

Note that the creation of this S4 generic does not affect the workings of S3 methods for lines. Calling methods("lines") will still list the S3 methods, and calling getMethods("lines") will list both S3 and S4 methods.

Adding a method for creating ROC curves from binomial generalised linear models provides an example of setGeneric. Calling setGeneric creates a generic ROC function and makes the existing function the default method. The code is the same as the S3 ROC.glm except that new is used to create the ROC object.

```
setGeneric("ROC")

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

```
    test<-fitted(T)
    disease<-(test+resid(T,type="response"))
    disease<-disease*weights(T)
    if (max(abs(disease %% 1))>0.01)
        warning("Y values suspiciously far
                          from integers")

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

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

    new("ROC",sens=sens, mspec=mspec,
              test=TT,call=sys.call())
  }
)
```

Finally, a method for `identify` shows one additional feature of `setGeneric`. The `signature` argument to `setGeneric` specifies which arguments are permitted in the signature of a method and thus are used for method dispatch. Method dispatch for `identify` will be based only on the first argument, which saves having to specify a `"missing"` second argument in the method.

```
setGeneric("identify",signature=c("x"))

setMethod("identify", "ROC",
  function(x, labels=NULL,
                  ...,digits=1){
    if (is.null(labels))
      labels<-round(x@test, digits)
    identify(x@mspec, x@sens,
             labels=labels,...)
    }
)
```

Discussion

Creating a simple class and methods requires very similar code whether the S3 or S4 system is used, and a similar incremental design strategy is possible. The S3 and S4 method system can coexist peacefully, even when S4 methods need to be defined for a function that already has S3 methods.

This example has not used inheritance, where the S3 and S4 systems differ more dramatically. Judging from the available examples of S4 classes, inheritance seems most useful in defining data structures, rather than objects representing statistical calculations. This may be because inheritance extends a class by creating a special case, but statisticians more often extend a class by creating a more general case. Reusing code from, say, linear models in creating generalised linear models is more an example of delegation than inheritance. It is not that a generalised linear model "is" a linear model, more that it "has" a linear model (from the last iteration of iteratively reweighted least squares) associated with it.

Thomas Lumley
Department of Biostatistics
University of Washington, Seattle

Changes in R

by the R Core Team

New features in 1.9.1

- `as.Date()` now has a method for "POSIXlt" objects.
- `mean()` has a method for "difftime" objects and so `summary()` works for such objects.
- `legend()` has a new argument `pt.cex`.
- `plot.ts()` has more arguments, particularly `yax.flip`.
- `heatmap()` has a new `keep.dendro` argument.
- The default `barplot` method now handles vectors and 1-d arrays (e.g., obtained by `table()`) the same, and uses grey instead of heat color palettes in these cases. (Also fixes PR#6776.)
- `nls()` now looks for variables and functions in its formula in the environment of the formula before the search path, in the same way `lm()` etc look for variables in their formulae.

User-visible changes in 1.9.0

- Underscore _ is now allowed in syntactically valid names, and `make.names()` no longer changes underscores. Very old code that makes use of underscore for assignment may now give confusing error messages.
- Package 'base' has been split into packages 'base', 'graphics', 'stats' and 'utils'. All four are loaded in a default installation, but the separation allows a 'lean and mean' version of R to be used for tasks such as building indices.

  Packages ctest, eda, modreg, mva, nls, stepfun and ts have been merged into stats, and lqs has