Programmer's Niche

Mind Your Language

Bill Venables

Introduction

John Chambers once told me that with S (including R) you have the choice of typing a command and implicitly presenting it to the evaluator or getting the engine to construct the command and explicitly presenting to the evaluator. The first is what everyone does when they use R and the second seemed to me at the time to be an esoteric possibility that might only be used in the most arcane situations imaginable. However it is not so. Language manipulation within R is a live possibility to consider for everyday data analysis and can be a boon to lazy typists. There is also one situation where without trying to do anything extreme or extraordinary you *must* use the second of John's alternatives to achieve the result you need.

In *S Programming*Venables and Ripley (2000) there is a discussion on language manipulation and its uses (see §3.4) which I do not want to repeat that here. Rather, what I would like to do is give a real example of a data analysis problem where a language manipulation solution offers an elegant, natural and practical solution. If you have the book you may wish to go back afterwards and read more, but the example should still make sense even if you have never heard of that book and intend never to go anywhere near it.

The problem

In biodiversity studies samples are taken from many different sites and, in the simplest case, a record is taken of whether or not each member of a suite of biological taxa is present or not. At the same time environmental variables are measured at each site to be used as predictor variables in building models for the probability that a taxon, say, is present at any site. In the particular case I will discuss here there are 1194 sites on the Great Barrier Reef from a study conducted by the Queensland Department of Primary Industry. The data frame containing the variables is called `QDPI`. There are 60 variables in the frame, the first three are location variables, the next 30 are possible environmental predictors and the final 27 are 0/1 variables indicating the presence/absence of 27 marine taxa.

We will consider separate models for each taxon. It would be natural to consider either logistic regression models or perhaps generalized additive models (in package **mgcv**) for example. For our purposes here tree models (using package **rpart**) have several advantages. So the problem is to construct separate tree models for each of 27 taxa where each model will have the full suite of 30 possible predictor variables from which to choose.

A strategy

Let's agree that we will put the fitted tree model objects in a list, say `tList`. We set up a dummy list to take them first

```
> nam <- names(QDPI)
> namX <- nam[4:33]
> namY <- nam[34:60]
> tList <- vector("list", length(namY))
> names(tList) <- namY
```

So far so good. Now for a bit of language manipulation. First we construct a character string version of an assignment to an element of `tList` of an `rpart` object, which will be evaluated at each cycle of a `for` loop:

```
> tAsgn <- paste("tList[[n]] <- try(rpart(X ~",
   paste(namX, collapse = " + "),
   ", QDPI, method = 'class'))", sep = "")
> tAsgn <- parse(text = tAsgn)[[1]]
```

(Notice how quotes within character strings can be handled by single quotes enclosed within doubles.) Turning character versions of commands into language objects is done by `parse(text =` *string*`)` but the result is an `expression` which in this case is rather like a list of language objects of length 1. It is not strictly necessary to extract the first object as we have done above, but it makes things look a little neater, at least. Here is the assignment object that we have constructed:

```
> tAsgn
tList[[n]] <- try(rpart(X ~ Depth +
     GBR.Bathyetry + GBR.Aspect +
     GBR.Slope + M.BenthicStress +
     SW.ChlorophyllA +
     SW.ChlorophyllA.SD + SW.K490 +
     SW.K490.SD + SW.BenthicIrradiance +
     OSI.CaCO3 + OSI.GrainSize +
     OSI.Rock + OSI.Gravel + OSI.Sand +
     OSI.Mud + CARS.Nitrate +
     CARS.Nitrate.SD + CARS.Oxygen +
     CARS.Oxygen.SD + CARS.Phosphate +
     CARS.Phosphate.SD + CARS.Salinity +
     CARS.Salinity.SD + CARS.Silica +
     CARS.Silica.SD + CARS.Temperature +
     CARS.Temperature.SD + Effort +
     TopographyCode, QDPI, method = "class"))
```

You can probably see now why I was not keen to type it all out. The index `n` will be supplied as a loop variable but the `X` dummy response variable will need to be replaced the by the *name* of each taxon in turn. Note that here *name* is a technical term. Language elements are composed of objects of a variety of special modes (and some, line numbers, not so special) and an object of mode "name" is the appropriate constituent of a language object to stand for a variable name.

Now for the main loop:

```
> for(n in namY) {
  TAsgn <- do.call("substitute",
  list(tAsgn, list(n = n, X = as.name(n))))
  eval(TAsgn)
  }
Error in matrix(c(rp$isplit[, 2:3], rp$dsplit),
  ncol = 5, dimnames = list(tname[rp$isplit[,:
  length of dimnames[1] not equal to array extent
```

There has been a problem with one of the modelling computations. We can find out which one it is by seeing which object does not have the correct mode at the finish:

```
> namY[which(sapply(tList, class) != "rpart")]
[1] "Anemone"
```

So Anemones were un able to have a tree model constructed for their presence/absence. [I am using an older version of R here and this seems to be a bug in pkgrpart that has since been corrected. It serves as an example here, though.]

Notes on `do.call`, `substitute` and `eval`

The three crucial functions we have used above, so far with no explanation are `do.call`, `substitute` and `eval`. We will not go into an extended explanation here but some pointers may be useful. First of all `do.call`. This used to be a well-kept secret but in recent years it has become fairly familiar to readers of R-news. It is perhaps the simplest to use function that constructs and evaluates a language object, so it is usually the first one people meet in their travels in R programming.

The function `do.call` is used to evaluate a function call where the name of the function is given as a character string as the first argument and the arguments to be used for the call to that function are given as a list in the second argument. It is probably useful to know that this second argument can be a list of objects, or a list of *names* of objects, or a mixture of both.

Nearly everyone knows of `substitute` through the idiom `deparse(substitute(arg))` for getting a character string version of the actual bit of code used for a formal argument, `arg`, on a call to a function. In this case `substitute(arg)` merely grabs the actual argument supplied without evaluating it and the `deparse(...)` step turns it back from a language object into a character string equivalent, essentially as it might have been typed.

This is only one use of `substitute` though. In general it may be used to take an expression and do a bit of surgery on it. Any of the *name*s appearing in the first argument may be changed to something else, as defined by a second argument, which has to be a named list. Here is a much simpler example than the one above:

```
> substitute(a+b+c,
          list(a=1, c = as.name("foo")))
1 + b + foo
```

The small detail often overlooked in this kind of demonstration is that the first argument is *quoted*, that is, the surgery is done on the object verbatim. What happens if the thing we want to do surgery upon is a long, complicated expression that we are holding as a language object, such as the case above?

At first sight it may seem that the following should work:

```
> substitute(tAsgn, list(n = "Anemone",
               X = as.name("Anemone")))
tAsgn
```

but you can see the result is a disappointment. The quoting prevents the first argument from being evaluated. So how do we say "evaluate the first argument, dummy"? The solution is to use R to construct a call to `substitute` as we might have typed it and evaluate that. This is what `do.call` does, and here it is essential to use this indirect approach.

```
> do.call("substitute",
       list(tAsgn, list(n = "Anemone",
              X = as.name("Anemone"))))
tList[["Anemone"]] <- try(rpart(Anemone ~
    CARS.Nitrate + CARS.Nitrate.SD + ... +
    SW.K490 + SW.K490.SD + TopographyCode,
    QDPI, method = "class"))
```

The third in the trio of functions is `eval`. As the name suggests this can be used for explicitly submitting a language object to the evaluator for, well, evaluation within the current environment. There are more general and subtle facilities for modifying the environment using a second or third argument, but they need not bother us here. There is no deep mystery, but it is important to know that such a function exists.

Final comments

It is possible to think of simpler ways that might achieve the same result as ours. For example we could put all the `X` variables into one data frame, say `QDPI.X` and all the response variables into `QDPI.Y` and then use a loop like

```
> for(n in namY)
    tList[[n]] <- try(rpart(QDPI.Y[, n] ~ .,
           QDPI.X, method = "class"))
```

and it would work. The problem with this is that the objects we construct have a formula that has, literally `QDPI.Y[, n]` as the dependent variable in the formula. If we want to do anything with the objects afterwards, such as prune them, update them, &c, we need to re-establish what `n` is in this particular case. The original object `n` was the loop variable and that is long gone. This is not difficult, of course, but it is an extra detail we need to carry along that we don't need. Essentially the formula part of the object we generate would not be self-contained and this can cause problems.

The strategy we have adopted has kept all the variables together in one data frame and explicitly encoded the correct response variable by name into the formula of each object as we go. At the end each fitted `rpart` object may be manipulated in the usual way witout this complication involving the now defunct loop variable.

Bibliography

W. N. Venables and B. D. Ripley. *S Programming*. Springer-Verlag, New York, 2000. 24

Bill Venables
CSIRO Marine Labs, Cleveland, Qld, Australia
`Bill.Venables@cmis.csiro.au`

geoRglm: A Package for Generalised Linear Spatial Models

by Ole F. Christensen and Paulo J. Ribeiro Jr

geoRglm is a package for inference in generalised linear spatial models using Markov chain Monte Carlo (MCMC) methods. It has been developed at the Department of Mathematical Sciences, Aalborg University, Denmark and the Department of Mathematics and Statistics, Lancaster University, UK. A web site with further information can be found at `http://www.maths.lancs.ac.uk/~christen/geoRglm`. **geoRglm** is an extension to the **geoR** package (Ribeiro, Jr. and Diggle, 2001). Maximum compatibility between the two packages has been intended and **geoRglm** also uses several of **geoR**'s internal functions.

Generalised linear spatial models

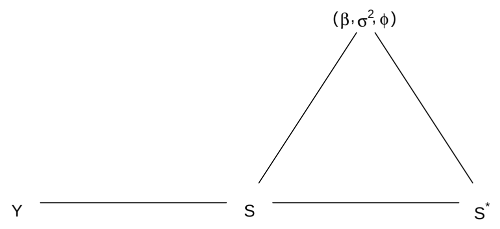
The classical geostatistical model assumes Gaussianity, which may be an unrealistic assumption for some data sets. The *generalised linear spatial model* (GLSM) as presented in Diggle et al. (1998), Zhang (2002) and Christensen and Waagepetersen (2002) provides a natural extension to deal with response variables for which a standard distribution other than the Gaussian more accurately describes the sampling mechanism involved.

The GLSM is a generalised linear mixed model in which the random effects are derived from a spatial process $S(\cdot)$. This leads to the following model specification.

Let $S(\cdot) = \{S(x) : x \in A\}$ be a Gaussian stochastic process with $\mathrm{E}[S(x)] = d(x)^{\mathrm{T}}\beta$, $\mathrm{Var}\{S(x)\} = \sigma^2$ and correlation function $\mathrm{Corr}\{S(x), S(x')\} = \rho(u; \phi)$ where $u = \|x - x'\|$ and ϕ is a parameter. Assume that the responses $Y_1, \ldots, Y_n$ observed at locations $x_1, \ldots, x_n$ in the sampling design, are conditionally independent given $S(\cdot)$, with conditional expectations $\mu_1, \ldots, \mu_n$, where $h(\mu_i) = S(x_i), i = 1, \ldots, n$, for a known link function $h(\cdot)$.

We write $S = (S(x_1), \ldots, S(x_n))^{\mathrm{T}}$ for the unobserved values of the underlying process at $x_1, \ldots, x_n$, and S^* for the values of $S(\cdot)$ at all other locations of interest, typically a fine grid of locations covering the study region.

The conditional independence structure of the GLSM is then indicated by the following graph.



The likelihood for a model of this kind is in general not expressible in closed form, but only as a high-dimensional integral

$$L(\beta, \sigma^2, \phi) = \int \prod_{i=1}^{n} f(y_i; h^{-1}(s_i)) p(s; \beta, \sigma^2, \phi) ds,$$

where $f(y; \mu)$ denotes the density of the error distribution parameterised by the mean μ, and $p(s; \beta, \sigma^2, \phi)$ is the multivariate Gaussian density