We want to fit a linear model where the names of the data columns carrying the outcome to predict (y), the explanatory variables (x1, x2), and per-example row weights (wt) are given to us as strings.

Lets start with our example data and parameters. The point is: we are assuming the data and parameters come to us as arguments and are not known at the time of writing the script or program.

# our inputs

d <- data.frame(

x1 = c(1, 2, 3, 4),

x2 = c(0, 0, 1, 1),

y = c(3, 3, 4, -5),

wt = c(1, 2, 1, 1))

knitr::kable(d)

| **x1** | **x2** | **y** | **wt** |
| --- | --- | --- | --- |
| 1 | 0 | 3 | 1 |
| 2 | 0 | 3 | 2 |
| 3 | 1 | 4 | 1 |
| 4 | 1 | -5 | 1 |

outcome\_name <- "y"

explanatory\_vars <- c("x1", "x2")

name\_for\_weight\_column <- "wt"

For everything except the weights this is easy, as the linear regression function lm() is willing to take strings in its first argument “formula” (and also, there are tools for building up formula objects).

# start our generic solution

formula\_str <- paste(

outcome\_name,

"~",

paste(explanatory\_vars, collapse = " + "))

print(formula\_str)

## [1] "y ~ x1 + x2"

model <- lm(formula\_str,

data = d)

print(model)

##

## Call:

## lm(formula = formula\_str, data = d)

##

## Coefficients:

## (Intercept) x1 x2

## 9.75 -4.50 5.50

format(model$terms)

## [1] "y ~ x1 + x2"

However, once we try to add weights we have problems.

lm(formula\_str,

data = d,

weights = name\_for\_weight\_column)

## Error in model.frame.default(formula = formula\_str, data = d, weights = name\_for\_weight\_column, : variable lengths differ (found for '(weights)')

This is a bit disappointing, as much of the point of working in R is being able to write parameterized scripts and programs over the R functions. So we really want to be able to take names of columns from an external source.

The reason is the following (taken from help(lm)):

All of weights, subset and offset are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

This means the weights argument is not treated as a value, but instead the name typed in is captured through “non standard evaluation” (NSE). The data frame environment, and formula environment are searched for a column or value named “name\_for\_weight\_column”, and not for one named “wt”.

Non-Standard Evaluation:

This is designed to clarify the various evaluation rules for function arguments in R and

to make some suggestions for new code. The descriptions are based on R 1.5.1.

1 Standard evaluation model

R passes arguments by value: the arguments are evaluated in the calling environment and their

values are passed to the function. If arguments are not specified then defaults are used and these

are evaluated in the environment inside the function, so that local variables are found first, and

then variables visible in the environment where the function was defined.

The evaluation of defaults in the environment inside the function is important, but can be abused.

In my opinion we should discourage

function (formula, data = parent.frame(), ..., subset, ylab = varnames[response],

ask = TRUE)

where the expression for ylab refers entirely to variables internal to the function.

[In fact, arguments are passed as promises to compute values rather than the values themselves.

The only relevance of this point is in the detailed implementation of nonstandard evaluation rules.]

2 Nonstandard models

Many modelling and graphical functions have a formula argument and a data argument. If variables in the formula were required to be in the data argument life would be a lot simpler, but

this requirement was not made when formulas were introduced. Authors of modelling and graphics

functions are thus required to implement a limited form of dynamic scope, which they have not

done in an entirely consistent way.

1

The two most common cases are handled in a uniform way across all the R and S-PLUS functions

I am aware of

• All the variables in the formula are present in the data object, and there are no vector

arguments other than the formula and data object

• All the variables in the formula are present in the data object or in the global environment,

the function is called from the global environment, and the formula is specified explicitly

(rather than as a variable), and there are no vector arguments other than the formula and

data object.

When other vector arguments are given (eg weights, pch), or the function is not called from the

global environment, or the formula was specified as a variable there may be differences between

functions and between S dialects. Some of these differences are clearly deliberate, some result from

insufficient paranoia on the part of the authors (myself included).

2.1 Most modelling functions

In the call

lm(y~x, data=df, weights=w)

the variables x, y, and w are looked up in df (which can be a list, data frame, or environment)

and then in the environment of the formula y~x. The environment of the formula is by default the

environment it was created in. Most commonly this will be the environment where lm was called

and in this case R and S-PLUS are compatible. Functions that work this way include lm, aov, glm,

the survival functions, loglm(MASS), and gam(mgcv). [though gam is incorrectly documented to

use parent.frame].

The nonstandard evaluation is usually accomplished by some variation on the following standard

idiom

mf <- match.call()

mf[[1]] <- as.name("model.frame")

mf$singular.ok <- mf$method <- mf$some.other.arg <- NULL

mf <- eval(mf,parent.frame())

The first line gets a copy of the current call. The second replaces the name of the function to

be called with model.frame. The third line removes arguments that should not be passed to

model.frame and so have the standard evaluation rules. Finally the constructed call to model.frame

is evaluated in the calling environment.

2

One point of variation is whether a specified list of arguments is removed from the call (as above)

or whether all but a specified list are removed. In the case above a ... argument would be passed

to model.frame, but in many functions the ... argument is actually passed to a control function

(eg glm.control, coxph.control).

A further infelicity is that the default na.action argument specified is a modelling function is not

actually used. A side effect of the match.call()/eval() procedure is that the default specified by

model.frame overrides the default specified by, say, glm(). One possible fix is to add the following

line before the eval step above

mf$na.action <- substitute(na.action)

2.2 Mixed models

The lme() function puts all its variables in a call to model.frame whose data argument is either a

specified data frame or the calling environment. However, the formula argument to model.frame is

constructed inside various nlme utility functions and does not have a useful environment attached

to it.

The effect is that the data argument is specified, variable lookup is done in that data frame and then

in the environment inside asOneFormula (for most purposes equivalent to the global environment).

I’m not sure exactly what happens in nlme, but the same principle seems to hold as for lme: either

all variables should be in the supplied data frame or all variables should be in the calling frame and

no data argument should be used. The documentation can be read to say this, but I don’t think

it’s clear if you don’t already know.

2.3 Base graphics

Formula methods for graphics use a similar but not identical scheme; in the call

plot(y~x, data=df, col=z)

x and y are looked up in df and then the environment of the formula, but the point colours argument

col=z is looked up first in df and then in the calling environment.

In this case only the formula is passed to model.frame. The additional graphical arguments are

evaluated in the data=df argument enclosed in the calling environment parent.frame. The reason

for this more complicated scheme is that model.frame requires all the variables to be vectors

of the same length and graphical parameters may be scalars or vectors of varying lengths. The

inconsistency in enclosing environment is still undesirable.

3

2.4 Lattice graphics

Lattice uses a slightly different system again. The formula arguments are looked up in the specified

data argument and then in the environment of latticeParseFormula. Other arguments (eg

groups and subset are evaluated in the data argument and then in the calling environment.

Presumably the use of the environment of latticeParseFormula is a minor bug. It causes problems

only when a lattice function is called inside another function and one of the arguments is a local

variable and a data= argument is provided. An artificial example is

data(trees)

h <- function(df){

x <- 1:10

y <- 1:10

xyplot(y~x, data=df)

}

h(trees)

3 Macro-like functions

Functions quote, substitute, evalq, with and expression all take unquoted expressions as

arguments. These necessarily use different evaluation rules, but can be fitted in conceptually by

thinking of them as macros. I may well add a couple more of these, capture.output (a temporary

version of sink) and bq, a version of the Lisp backquote operator.

4 Unquoted character strings

Two functions take either quoted or unquoted character strings: help and library.

5 Functions of models

Functions such as summary, residuals and so on generally operate as if the model object contained

all the necessary data (which in many cases it does). Difficulties arise with functions that refit

models.

The update() function refits the model by constructing a function call and evaluating it in the

calling environment. In some cases this is clearly what users expect, as in this code snippet from

MASS

ph.fun <- function(data, i) {

4

d <- data

d$calls <- d$fitted + d$res[i]

coef(update(fit, data=d))

}

but in other cases users want to use the original data frame (local or not) and just update the

formula (PR#1861).

The step() and stepAIC functions look up the data in the environment of the model formula, and

so (typically) performs the model search using the data from the original fit.

The phrase model<-update(model) can be used to refit a model to data in the local environment,

even changing the environment associated with the model formula.

6 Variable capture with with()

The with() function allows an expression to be evaluated with variable lookup in a specified data

frame, and then the calling environment. The plot example above can be written

with(df, plot(y~x,col=z))

For interactive use at the command line this is very effective. For programming some care is needed

to ensure that variables in the data frame do not accidentally override local variables.

7 Recycling, subsetting and NA removal

A further difficulty in handling the nonstandard evaluation mechanisms is the removal of missing

values and the use of the subset argument. The modelling functions accept a na.action argument

specifying how to handle missing values. If rows of the model frame containing missing values are

removed (as is the default), it is not clear whether the same rows of other arguments should be

removed. Similarly, the subset argument applied to the variables defined in the formula may

or may not be applied to the other nonstandardly evaluated arguments. Differences of opinion

exist within R-core on the correct behaviour, and each possibility makes some things hard. For

interactive use it is possible to get around the difficulties using with, but this is harder when

programming because of the possibilities of unintended variable capture.

The current implementation is that functions apply the subset argument to all these arguments.

The na.action argument is not needed in base graphics functions (as NAs are not plotted); in the

modelling functions, all rows with missing values in either the formula variables or other variables

such as weights are removed. I think this is the wrong behaviour for graphics functions, but the

right behaviour for modelling functions if the extra arguments are things like weights and strata

5

that are conceptually part of the model frame. On the other hand, having different behaviour for

the two classes of functions is difficult.

If subsetting and NA removal are done then a further decision is needed about the recycling rule;

should it be applied before or after the final subset is taken? In the base graphics functions it

happens afterwards, which I think is wrong:

x <- 1:10

y <- 1:10

z <- 1:2

plot(x~y, col=z)

plot(x~y, col=z, subset=2\*(1:5))

df <- data.frame(x,y,z)

plot(x~y, col=z, data=df, subset=2\*(1:5))

The first two plot both red and black points, the last plots only red points.

8 Proposals

The ambiguity in evaluation rules arises because some arguments need to be evaluated according

to formula/data rules and some don’t. One possible solution for new code is to pass formulas or

quoted expressions when the standard variable lookup is not to be used.

That is, a new modelling function

xyzlm(y~x, data=df, foo=z)

would look up z in the calling environment. If the xyzlm function wants to look up z in df it should

specify one of the following

xyzlm(y~x, data=df, foo=~z)

xyzlm(y~x, data=df, foo=quote(z))

xyzlm(y~x, data=df, foo=expression(z))

This allows more flexibility than the current system and is not ambiguous as the evaluation rules

in the function call are standard. A possible refinement would be to say that a formula argument

takes part in subsetting and NA removal but an expression argument does not.

People should be encouraged to

1. Thorougly document nonstandard evaluation if it can’t be avoided

6

2. Use the environment of a suitable formula as the enclosing environment when evaluating in

a data frame.

3. Use standard patterns (like the model.frame/eval one) where possible (to keep the insanity

localised)

4. For new code, where possible, pass formulas or quoted expressions when the standard variable

lookup is not desired. It would be useful to have a single function like model.frame that does

the necessary evaluation.

At a minimum, lattice and base graphics should use the same evaluation rules, and it probably

makes sense for them to use the environment of the formula as the enclosing environment for

compatibility with model.frame.

This is a big hindrance to using lm() programmatically. The NSE notation style can be convenient when working interactively, but is often a burden when programming.

However, R has some ways out of the problem.

The first solution is bquote() which is part of R itself (in the base package).

eval(bquote(

lm(

formula\_str,

data = d,

weights = .([as.name](http://as.name)(name\_for\_weight\_column)))

))

##

## Call:

## lm(formula = formula\_str, data = d, weights = wt)

##

## Coefficients:

## (Intercept) x1 x2

## 9.429 -3.857 3.571

In the above, the .() notation indicates to replace the .()-expression with its evaluated value before evaluating the rest of the expression. This is a substitution principle based on escaping notation (also called quasiquoting).

Another solution is the let() function from the wrapr package (a user extension package, not part of R itself).

wrapr::let(

c(NAME\_FOR\_WEIGHT\_COLUMN = name\_for\_weight\_column),

lm(

formula\_str,

data = d,

weights = NAME\_FOR\_WEIGHT\_COLUMN)

)

##

## Call:

## lm(formula = formula\_str, data = d, weights = wt)

##

## Coefficients:

## (Intercept) x1 x2

## 9.429 -3.857 3.571

In the above, the left-hand sides of the named vector are symbols to be replaced and the right had sides refer to values to replace them with. The specification c(NAME\_FOR\_WEIGHT\_COLUMN = name\_for\_weight\_column) means to replace NAME\_FOR\_WEIGHT\_COLUMN with wt (the value referred to by name\_for\_weight\_column). This is a substitution principle based on named substitution targets.

Another solution is the !! notation from the rlang package (a user extension package, not part of R itself).

rlang::eval\_tidy(rlang::quo(

lm(

formula\_str,

data = d,

weights = !![as.name](http://as.name)(name\_for\_weight\_column))

))

##

## Call:

## lm(formula = formula\_str, data = d, weights = wt)

##

## Coefficients:

## (Intercept) x1 x2

## 9.429 -3.857 3.571

In the above, the !! notation indicates to replace the !!-expression with its evaluated value before evaluating the rest of the expression. This is a substitution principle based on escaping notation (also called quasiquoting).

Note the argument types and/or the internals of lm() do not currently appear to allow the use of the newer rlang double curly brace notation (a notation that can replace !!rlang::enquo(), and possibly other forms).

rlang::eval\_tidy(rlang::quo(

lm(

formula\_str,

data = d,

weights = {{name\_for\_weight\_column}})

))

## Error in model.frame.default(formula = formula\_str, data = d, weights = ~"wt", : invalid type (language) for variable '(weights)'

We only mention the “{{}}” notation as rlang familiar readers are likely to wonder about using it.

These are tools that let programs treat programs as data. As such they are very powerful. In fact they are big hammers for such a simple problem as specifying a column name that is already stored as data. However, the above tools should not be blamed for the awkwardness of having a need for them.