- Clustering Frankenstein
- RTutor: Improving Interactive Problem Sets by Analyzing Submissions
- Stock Market Data Scenario Set Generation – S&P 100
- Finding duplicates in data frame across columns and replacing them with unique values using R
- The Shiny Developer Series
- Likert Scale Survey: from googleform to #rstats graph
- Upcoming R courses with Jumping Rivers
- Interactive Visualization in R with apexcharter
- RcppCCTZ 0.2.6
- R Markdown Workshop
- Programmatically extract TIOBE Index Ratings
- Machine Learning Training – Draper and Dash Healthcare Predictive Analytics – Summer Discount
- simmer 4.3.0 + JSS publication
- Updating MetaLandSim…
- dime: Deep Interactive Model Explanations

Other sites

- Jobs for R-users
- SAS blogs

Using apply, sapply, lapply in R

December 18, 2012
By Pete

Like 48 Share LinkedIn

Share Tweet

This is an introductory post about using apply, sapply and lapply, best suited for people relatively new to R or unfamiliar with these functions. There is a part 2 coming that will look at density plots with ggplot, but first I thought I would go on a tangent to give some examples of the apply family, as they come up a lot working with R.
I have been comparing three methods on a data set. A sample from the data set was generated, and three different methods were applied to that subset. I wanted to see how their results differed from one another.
I would run my test harness which returned a matrix. The columns values were the metric used for evaluation of each method, and the

rows were the results for a given subset. We have three columns, one for each method, and lets say 30 rows, representing 30 different subsets that the three methods were applied to.
It looked a bit like this

```
        method1   method2    method3
[1,] 0.05517714 0.014054038 0.017260447
[2,] 0.08367678 0.003570883 0.004289079
[3,] 0.05274706 0.028629661 0.071323030
[4,] 0.06769936 0.048446559 0.057432519
[5,] 0.06875188 0.019782518 0.080564474
[6,] 0.04913779 0.100062929 0.102208706
```

We can simulate this data using `rnorm`, to create three sets of observations. The first has mean 0, second mean of 2, third of mean of 5, and with 30 rows.

```
m <- matrix(data=cbind(rnorm(30, 0), rnorm(30, 2), rnorm(30, 5)), nrow=30,
ncol=3)
```

Apply

When do we use apply? When we have some structured blob of data that we wish to perform operations on. Here structured means in some form of matrix. The operations may be informational, or perhaps transforming, subsetting, whatever to the data.

As a commenter pointed out, if you are using a data frame the data types must all be the same otherwise they will be subjected to type conversion. This may or may not be what you want, if the data frame has string/character data as well as numeric data, the numeric data will be converted to strings/characters and numerical operations will probably not give what you expected.

Needless to say such circumstances arise quite frequently when working in R, so spending some time getting familiar with `apply` can be a great boon to our productivity.
Which actual apply function and which specific incantion is required depends on your data, the function you wish to use, and what you want the end result to look like. Hopefully the right choice should be a bit clearer by the end of these examples.
First I want to make sure I created that matrix correctly, three columns each with a mean 0, 2 and 5 respectively. We can use `apply` and the base `mean` function to check this.
We tell `apply` to traverse row wise or column wise by the second argument. In this case we expect to get three numbers at the end, the mean value for each column, so tell `apply` to work along columns by passing 2 as the second argument. But let's do it wrong for the point of illustration:

```
apply(m, 1, mean)
# [1] 2.408150 2.709325 1.718529 0.822519 2.693614 2.259044 1.849530 2.544685
2.957950 2.219874
#[11] 2.582011 2.471938 2.015625 2.101832 2.189781 2.319142 2.504821 2.203066
2.280550 2.401297
#[21] 2.312254 1.833903 1.900122 2.427002 2.426869 1.890895 2.515842 2.363085
3.049760 2.027570
```

Passing a 1 in the second argument, we get 30 values back, giving the mean of each row. Not the three numbers we were expecting, try again.

```
apply(m, 2, mean)
#[1] -0.02664418  1.95812458  4.86857792
```

Great. We can see the mean of each column is roughly 0, 2, and 5 as we expected.

Our own functions

Let's say I see that negative number and realise I wanted to only look at positive values. Let's see how many negative numbers each column has, using apply again:

```
apply(m, 2, function(x) length(x[x<0]))
#[1] 14  1  0
```

So 14 negative values in column one, 1 negative value in column two, and none in column three. More or less what we would expect for three normal distributions with the given means and sd of 1.
Here we have used a simple function we defined in the call to `apply`, rather than some built in function. Note we did not specify a return value for our function. R will magically return the last evaluated value. The actual function is using subsetting to extract all the elements in `x` that are less than 0, and then counting how many are left are using `length`.
The function takes one argument, which I have arbitrarily called `x`. In this case `x` will be a single column of the matrix. Is it a 1 column matrix or a just a vector? Let's have a look:

```
apply(m, 2, function(x) is.matrix(x))
#[1] FALSE FALSE FALSE
```

Not a matrix. Here the function definition is not required, we could instead just pass the `is.matrix` function, as it only takes one argument and has already been wrapped up in a function for us. Let's check they are vectors as we might expect.

```
apply(m, 2, is.vector)
#[1] TRUE TRUE TRUE
```

Why then did we need to wrap up our length function? When we want to define our own handling function for apply, we must at a minimum give a name to the incoming data, so we can use it in our function.

```
apply(m, 2, length(x[x<0]))
#Error in match.fun(FUN) : object 'x' not found
```

We are referring to some value `x` in the function, but R does not know where that is and so gives us an error. There are other forces at play here, but for simplicity just remember to wrap any code up in a function. For example, let's look at the mean value of only the positive values:

```
apply(m, 2, function(x) mean(x[x>0]))
#[1] 0.4466368 2.0415736 4.8685779
```

Using sapply and lapply

These two functions work in a similar way, traversing over a set of data like a list or vector, and calling the specified function for each item.
Sometimes we require traversal of our data in a less than linear way. Say we wanted to compare the current observation with the value 5 periods before it. Use can probably use `rollapply` for this (via quantmod), but a quick and dirty way is to run `sapply` or `lapply` passing a set of index values.
Here we will use `sapply`, which works on a list or vector of data.

```
sapply(1:3, function(x) x^2)
#[1] 1 4 9
```

`lapply` is very similar, however it will return a list rather than a vector:

```
lapply(1:3, function(x) x^2)
#[[1]]
#[1] 1
#
#[[2]]
#[1] 4
#
#[[3]]
#[1] 9
```

Passing `simplify=FALSE` to `sapply` will also give you a list:

```
sapply(1:3, function(x) x^2, simplify=F)
#[[1]]
#[1] 1
#
#[[2]]
#[1] 4
#
#[[3]]
#[1] 9
```

And you can use `unlist` with `lapply` to get a vector.

```
unlist(lapply(1:3, function(x) x^2))
#[1] 1 4 9
```

However the behviour is not as clean when things have names, so best to use `sapply` or `lapply` as makes sense for your data and what you want to receive back. If you want a list returned, use `lapply`. If you want a vector, use `sapply`.

Dirty Deeds

Anyway, a cheap trick is to pass `sapply` a vector of indexes and write your function making some assumptions about the structure of the underlying data. Let's look at our `mean` example again:

```
sapply(1:3, function(x) mean(m[,x]))
[1] -0.02664418  1.95812458  4.86857792
```

We pass the column indexes (1,2,3) to our function, which assumes some variable `m` has our data. Fine for quickies but not very nice, and will likely turn into a maintainability bomb down the line.
We can neaten things up a bit by passing our data in an argument to our function, and using the … special argument which all the apply functions have for passing extra arguments:

```
sapply(1:3, function(x, y) mean(y[,x]), y=m)
#[1] -0.02664418  1.95812458  4.86857792
```

This time, our function has 2 arguments, `x` and `y`. The `x` variable will be as it was before, whatever `sapply` is currently going through. The `y` variable we will pass using the optional arguments to `sapply`.
In this case we have passed in `m`, explicitly naming the `y` argument in the `sapply` call. Not strictly necessary but it makes for easier to read & maintain code. The `y` value will be the same for each call `sapply` makes to our function.
I don't really recommend passing the index arguments like this, it is error prone and can be quite confusing to others reading your code.
I hope you found these examples helpful. Please check out part 2 where we create a density plot of the values in our matrix.

Share Tweet