R use Apply, Sapply and dplyr Mutate to Evaluate one Function Across Rows of a Matrix

Go back to fan's R4Econ Repository or Intro Stats with R Repository.

Issue and Goal

- r apply matrix to function row by row
- r evaluate function on grid
- Apply a function to every row of a matrix or a data frame
- r apply
- r sapply
- sapply over matrix row by row
- apply dplyr vectorize
- function as parameters using formulas
- do

We want evaluate linear function f(x__i, y__i, ar__x, ar__y, c, d), where c and d are constants, and ar__x and ar__y are arrays, both fixed. x__i and y__i vary over each row of matrix. More specifically, we have a functions, this function takes inputs that are individual specific. We would like to evaluate this function concurrently across N individuals.

The function is such that across the N individuals, some of the function parameter inputs are the same, but others are different. If we are looking at demand for a particular product, the prices of all products enter the demand equation for each product, but the product's own price enters also in a different way.

The objective is either to just evaluate this function across N individuals, or this is a part of a nonlinear solution system.

Set Up

```r
rm(list = ls(all.names = TRUE))
options(knitr.duplicate.label = 'allow')
```

```r
library(tidyverse)
library(knitr)
library(kableExtra)
# file name
st_file_name = 'fs_applysapplymutate'
# Generate R File
purl(paste0(st_file_name, ".Rmd"), output=paste0(st_file_name, ".R"), documentation = 2)
# Generate PDF and HTML
# rmarkdown::render("C:/Users/fan/R4Econ/support/function/fs_funceval.Rmd", "pdf_document")
# rmarkdown::render("C:/Users/fan/R4Econ/support/function/fs_funceval.Rmd", "html_document")
```

Set up Input Arrays

There is a function that takes $M = Q + P$ inputs, we want to evaluate this function N times. Each time, there are M inputs, where all but Q of the M inputs, meaning P of the M inputs, are the same. In particular,

$P = Q * N$.

$$M = Q + P = Q + Q * N$$

```
# it_child_count = N, the number of children
it_N_child_cnt = 5
# it_heter_param = Q, number of parameters that are heterogeneous across children
it_Q_hetpa_cnt = 2

# P fixed parameters, nN is N dimensional, nP is P dimensional
ar_nN_A = seq(-2, 2, length.out = it_N_child_cnt)
ar_nN_alpha = seq(0.1, 0.9, length.out = it_N_child_cnt)
ar_nP_A_alpha = c(ar_nN_A, ar_nN_alpha)

# N by Q varying parameters
mt_nN_by_nQ_A_alpha = cbind(ar_nN_A, ar_nN_alpha)

# display
kable(mt_nN_by_nQ_A_alpha) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "responsive"))
```

ar_nN_A	ar_nN_alpha
-2	0.1
-1	0.3
0	0.5
1	0.7
2	0.9

Using apply

First we use the apply function, we have to hard-code the arrays that are fixed for each of the N individuals. Then apply allows us to loop over the matrix that is N by Q, each row one at a time, from 1 to N.

```
# Define Implicit Function
ffi_linear_hardcode <- function(ar_A_alpha){
  # ar_A_alpha[1] is A
  # ar_A_alpha[2] is alpha

  fl_out = sum(ar_A_alpha[1]*ar_nN_A + 1/(ar_A_alpha[2] + 1/ar_nN_alpha))

  return(fl_out)
}

# Evaluate function row by row
ar_func_apply = apply(mt_nN_by_nQ_A_alpha, 1, ffi_linear_hardcode)
```

Using sapply

- r convert matrix to list
- Convert a matrix to a list of vectors in R

Sapply allows us to not have tohard code in the A and alpha arrays. But Sapply works over List or Vector, not Matrix. So we have to convert the N by Q matrix to a N element list Now update the function with sapply.

```r
ls_ar_nN_by_nQ_A_alpha = as.list(data.frame(t(mt_nN_by_nQ_A_alpha)))

# Define Implicit Function
ffi_linear_sapply <- function(ar_A_alpha, ar_A, ar_alpha){
  # ar_A_alpha[1] is A
  # ar_A_alpha[2] is alpha

  fl_out = sum(ar_A_alpha[1]*ar_nN_A + 1/(ar_A_alpha[2] + 1/ar_nN_alpha))

  return(fl_out)
}

# Evaluate function row by row
ar_func_sapply = sapply(ls_ar_nN_by_nQ_A_alpha, ffi_linear_sapply,
                        ar_A=ar_nN_A, ar_alpha=ar_nN_alpha)
```

Using dplyr mutate

- dplyr mutate own function
- dplyr all row function
- dplyr do function
- apply function each row dplyr
- applying a function to every row of a table using dplyr
- dplyr rowwise

```r
# Convert Matrix to Tibble
ar_st_col_names = c('fl_A', 'fl_alpha')
tb_nN_by_nQ_A_alpha <- as_tibble(mt_nN_by_nQ_A_alpha) %>% rename_all(~c(ar_st_col_names))
# Show
kable(tb_nN_by_nQ_A_alpha) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "responsive"))
```

fl_A	fl_alpha
-2	0.1
-1	0.3
0	0.5
1	0.7
2	0.9

```r
# Define Implicit Function
ffi_linear_dplyrdo <- function(fl_A, fl_alpha, ar_nN_A, ar_nN_alpha){
  # ar_A_alpha[1] is A
  # ar_A_alpha[2] is alpha

  print(paste0('cur row, fl_A=', fl_A, ', fl_alpha=', fl_alpha))
  fl_out = sum(fl_A*ar_nN_A + 1/(fl_alpha + 1/ar_nN_alpha))

  return(fl_out)
}

# Evaluate function row by row of tibble
# fl_A, fl_alpha are from columns of tb_nN_by_nQ_A_alpha
tb_nN_by_nQ_A_alpha_show <- tb_nN_by_nQ_A_alpha %>% rowwise() %>%
                    mutate(dplyr_eval = ffi_linear_dplyrdo(fl_A, fl_alpha, ar_nN_A, ar_nN_alpha))
```

```
## [1] "cur row, fl_A=-2, fl_alpha=0.1"
## [1] "cur row, fl_A=-1, fl_alpha=0.3"
## [1] "cur row, fl_A=0, fl_alpha=0.5"
## [1] "cur row, fl_A=1, fl_alpha=0.7"
## [1] "cur row, fl_A=2, fl_alpha=0.9"
```

```
# Show
kable(tb_nN_by_nQ_A_alpha_show) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "responsive"))
```

fl_A	fl_alpha	dplyr_eval
-2	0.1	2.346356
-1	0.3	2.094273
0	0.5	1.895316
1	0.7	1.733708
2	0.9	1.599477

Using Dplyr Mutate with Function and Parameters as Parameters

We want to allow the function itself to be a parameter, and the parameters of the function to also be parameters.

- dplyr mutate pass function
- r function quosure string multiple
- r function multiple parameters as one string

First, hard code arrays that will not be changing across iterations in, reduces two parameters

```
# Define function, fixed inputs are not parameters, but defined earlier as a part of the function
# ar_nN_A, ar_nN_alpha are fixed, not parameters
ffi_linear_dplyrdo_func <- function(fl_A, fl_alpha){
  fl_out <- sum(fl_A*ar_nN_A + 1/(fl_alpha + 1/ar_nN_alpha))
  return(fl_out)
}

# Evaluate function row by row of tibble
tbfunc_A_nN_by_nQ_A_alpha = tb_nN_by_nQ_A_alpha %>% rowwise() %>%
                    mutate(dplyr_eval = ffi_linear_dplyrdo_func(fl_A, fl_alpha))
```

DPLYR Apply Do Row by Row

We can use *do* to apply anonymous functions to each row to generate output that has the same number of rows as dataframe.

```
tbfunc_B_nN_by_nQ_A_alpha = tb_nN_by_nQ_A_alpha %>% rowwise() %>%
                    do(abc = .$fl_A + .$fl_alpha) %>%
                    unnest(abc)
tbfunc_B_nN_by_nQ_A_alpha = tb_nN_by_nQ_A_alpha %>% rowwise() %>%
                    do(abc = sum(.$fl_A*ar_nN_A + .$fl_alpha + ar_nN_alpha)) %>%
                    unnest(abc)
# Show
print(tbfunc_B_nN_by_nQ_A_alpha )
```

```
## # A tibble: 5 x 1
##     abc
##   <dbl>
## 1    3
## 2    4
## 3    5
## 4    6.
## 5    7.
```

DPLYR Do With Flexible Function with Varying, Fixed and Row-specific Inputs

Now, we have three types of parameters, for something like a bisection type calculation. We will supply the program with a function with some hard-coded value inside, and as parameters, we will have one parameter which is a row in the current matrix, and another parameter which is a sclar values. The three types of parameters are dealt with sparately:

1. parameters that are fixed for all bisection iterations, but differ for each row
   - these are hard-coded into the function
2. parameters that are fixed for all bisection iterations, but are shared across rows
   - these are the first parameter of the function, a list
3. parameters that differ for each iteration, but differ acoss iterations
   - second scalar value parameter for the function

- dplyr mutate function applow to each row dot notation
- note rowwise might be bad according to Hadley, should use pmap?

```r
ffi_linear_dplyrdo_fdot <- function(ls_row, fl_param){
  # Type 1 Param = ar_nN_A, ar_nN_alpha
  # Type 2 Param = ls_row$fl_A, ls_row$fl_alpha
  # Type 3 Param = fl_param

  fl_out <- (sum(ls_row$fl_A*ar_nN_A + 1/(ls_row$fl_alpha + 1/ar_nN_alpha))) + fl_param
  return(fl_out)
}

cur_func <- ffi_linear_dplyrdo_fdot
fl_param <- 0
dplyr_eval_flex <- tb_nN_by_nQ_A_alpha %>% rowwise() %>%
                              do(dplyr_eval_flex = cur_func(., fl_param)) %>%
                              unnest(dplyr_eval_flex)
tbfunc_B_nN_by_nQ_A_alpha <- tb_nN_by_nQ_A_alpha %>% add_column(dplyr_eval_flex)
# Show
kable(tbfunc_B_nN_by_nQ_A_alpha) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "responsive"))
```

fl_A	fl_alpha	dplyr_eval_flex
-2	0.1	2.346356
-1	0.3	2.094273
0	0.5	1.895316
1	0.7	1.733708
2	0.9	1.599477

Compare Results

```r
# Show overall Results
mt_results <- cbind(ar_func_apply, ar_func_sapply,
                    tb_nN_by_nQ_A_alpha_show['dplyr_eval'],
                    tbfunc_B_nN_by_nQ_A_alpha['dplyr_eval_flex'],
                    mt_nN_by_nQ_A_alpha)
colnames(mt_results) <- c('eval_lin_apply', 'eval_lin_sapply',
                          'eval_dplyr_mutate',
                          'eval_dplyr_mutate_flex',
                          'A_child', 'alpha_child')
kable(mt_results) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "responsive"))
```

	eval_lin_apply	eval_lin_sapply	eval_dplyr_mutate	eval_dplyr_mutate_flex	A_child	alpha_child
X1	2.346356	2.346356	2.346356	2.346356	-2	0.1
X2	2.094273	2.094273	2.094273	2.094273	-1	0.3
X3	1.895316	1.895316	1.895316	1.895316	0	0.5
X4	1.733708	1.733708	1.733708	1.733708	1	0.7
X5	1.599477	1.599477	1.599477	1.599477	2	0.9