Restrict functions to a smaller domain with `restrict_fun()` in the `doBy` package

Søren Højsgaard

doBy version 4.6.13 as of 2022-05-02

Contents

1 Introduction

The **doBy** package contains a variety of utility functions. This working document describes some of these functions. The package originally grew out of a need to calculate groupwise summary statistics (much in the spirit of `PROC SUMMARY` of the SAS system), but today the package contains many different utilities.

2 Restrict a functions domain: `restrict_fun()`

The `restrict_fun` function can restrict the domain of a function. For example, if $f(x, y) = x + y$ then $g(x) = f(x, 10)$ is a restriction of f to be a function of x alone.

There are two approaches: 1) Store the restricted arguments in an auxillary environment and 2) substitute the restricted arguments into the function.

2.1 Using an auxillary environment

```
> f1  <- function(a, b, c=4, d=9){
      a + b + c + d
  }
> f1_ <- restrict_fun(f1, list(b=7, d=10))
> class(f1_)

## [1] "scaffold"
```

We see the new function is a function of a and c with c being given a default value, but what the function does is not clear. However, it does evaluate correctly:

```
> f1_

## function (a, c = 4)
## {
##     args <- arg_getter()
##     do.call(fun, args)
## }
## <environment: 0x560dd7ef6b68>

> f1_(100)

## [1] 121
```

The restricted values are stored in an extra environment in the `scaffold` object and the original function is stored in the scaffold functions environment:

```
> get_restrictions(f1_)

## $b
## [1] 7
##
## $d
## [1] 10

> ## attr(f1_, "arg_env")$args ## Same result
> get_fun(f1_)

## function(a, b, c=4, d=9){
##     a + b + c + d
## }

> ## environment(f1_)$fun ## Same result
```

Similarly

```
> rnorm5 <- restrict_fun(rnorm, list(n=5))
> rnorm5()

## [1]  1.06144  0.07263  0.46731 -1.24649 -0.41485
```

2.2 Substitute restricted values into function

With substitution, it is clear what is happening:

```
> f1s_ <- restrict_fun_sub(f1, list(b=7, d=10))
> f1s_

## function (a, c = 4)
## {
##     a + 7 + c + 10
## }

> f1s_(100)

## [1] 121
```

However, absurdities can arise:

```
> f2  <- function(a) {
      a <- a + 1
      a
  }
> ## Notice that the following is absurd
> f2s_ <- restrict_fun_sub(f2, list(a = 10))
> f2s_

## function ()
## {
##     10 <- 10 + 1
##     10
## }

> # do not run: f2s_()
> try(f2s_())

## Error in 10 <- 10 + 1 : invalid (do_set) left-hand side to assignment

> ## Using the environment approch, the result makes sense
> f2_ <- restrict_fun(f2, list(a = 10))
> f2_

## function ()
## {
##     args <- arg_getter()
##     do.call(fun, args)
## }
## <environment: 0x560dda65cdf0>

> f2_()

## [1] 11
```

3 Example: Benchmarking

Consider a simple task: Creating and inverting Toeplitz matrices for increasing dimensions:

```
> n <- 4
> toeplitz(1:n)

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    1    2    3
## [3,]    3    2    1    2
## [4,]    4    3    2    1
```

A naive implementation is

```
> inv_toeplitz <- function(n) {
      solve(toeplitz(1:n))
  }
> inv_toeplitz(4)

##      [,1] [,2] [,3] [,4]
## [1,] -0.4  0.5  0.0  0.1
## [2,]  0.5 -1.0  0.5  0.0
## [3,]  0.0  0.5 -1.0  0.5
```

```
## [4,]  0.1  0.0  0.5 -0.4
```

We can benchmark timing for different values of n as

```
> library(microbenchmark)
> microbenchmark(
      inv_toeplitz(4), inv_toeplitz(8), inv_toeplitz(16),
      inv_toeplitz(32), inv_toeplitz(64),
      times=5
  )

## Unit: microseconds
##              expr    min     lq   mean median     uq     max neval cld
##   inv_toeplitz(4)  17.46  17.72  18.75  17.77  20.34   20.45     5   a
##   inv_toeplitz(8)  18.95  19.93  21.41  20.13  23.89   24.18     5   a
##  inv_toeplitz(16)  26.60  27.69  33.24  27.90  36.12   47.88     5   a
##  inv_toeplitz(32)  54.64  55.40 331.65  61.13  64.74 1422.35     5   a
##  inv_toeplitz(64) 168.01 168.85 174.14 171.51 172.44  189.89     5   a
```

However, it is tedious (and hence error prone) to write these function calls.

A programmatic approach using **`restrict_fun`** is as follows: First create list of scaffold objects:

```
> n.vec  <- c(4, 8, 16, 32, 64)
> scaf.list <- lapply(n.vec,
                    function(ni){
                        restrict_fun(inv_toeplitz, list(n=ni))}
                    )
```

Each element is a function (a scaffold object, to be precise) and we can evaluate each / all functions as:

```
> scaf.list[[1]]

## function ()
## {
##     args <- arg_getter()
##     do.call(fun, args)
## }
## <environment: 0x560dd796ed38>

> scaf.list[[1]]()

##      [,1] [,2] [,3] [,4]
## [1,] -0.4  0.5  0.0  0.1
## [2,]  0.5 -1.0  0.5  0.0
## [3,]  0.0  0.5 -1.0  0.5
## [4,]  0.1  0.0  0.5 -0.4
```

To use the list of functions in connection with microbenchmark we bquote all functions using

```
> bquote_list <- function(fnlist){
      lapply(fnlist, function(g) {
          bquote(.(g)())
      }
      )
  }
```

We get:

```
> bq.list <- bquote_list(scaf.list)
> bq.list[[1]]

## (function ()
## {
##     args <- arg_getter()
##     do.call(fun, args)
## })()

> ## Evaluate one:
> eval(bq.list[[1]])

##      [,1] [,2] [,3] [,4]
## [1,] -0.4  0.5  0.0  0.1
## [2,]  0.5 -1.0  0.5  0.0
## [3,]  0.0  0.5 -1.0  0.5
## [4,]  0.1  0.0  0.5 -0.4

> ## Evaluate all:
> ## sapply(bq.list, eval)
```

To use microbenchmark we must name the elements of the list:

```
> names(bq.list) <- n.vec
> microbenchmark(
    list  = bq.list,
    times = 5
  )

## Unit: microseconds
##  expr    min     lq   mean median     uq    max neval cld
##     4  20.69  22.04  23.83  22.63  23.14  30.64     5  a
##     8  24.30  25.09  26.52  26.64  27.62  28.98     5  a
##    16  30.72  31.42  34.55  33.03  36.65  40.91     5  a
##    32  57.66  60.46 122.16  60.93  65.75 365.99     5  ab
##    64 171.53 172.02 182.16 177.90 178.33 211.01     5   b
```

To summarize: to experiment with many difference values of n we can do

```
> n.vec <- seq(50, 700, by=50)
> scaf.list <- lapply(n.vec,
                      function(ni){
                          restrict_fun(inv_toeplitz, list(n=ni))}
                      )
> bq.list <- bquote_list(scaf.list)
> names(bq.list) <- n.vec
> mb <- microbenchmark(
    list  = bq.list,
    times = 5
  )
> doBy::mb_summary(mb)  %>% head(4)

##   expr    min     lq   mean median     uq    max neval         unit
## 1   50  248.6  256.5  267.2  262.6  279.2  289.1     5 microseconds
## 2  100  838.9  883.8  913.3  915.7  917.7 1010.3     5 microseconds
## 3  150 2367.3 2428.4 2442.5 2459.9 2465.8 2490.9     5 microseconds
## 4  200 5013.9 5163.2 5171.4 5190.9 5234.1 5254.9     5 microseconds
```

Notice: Above, `doBy::mb_summary` is a faster version of the `summary` method for `microbenchmark` objects than the method provided by the `microbenchmark` package.

```
> par(mfrow=c(1,2))
> y <- mb_summary(mb)$mean
> plot(n.vec, y)
> plot(log(n.vec), log(y))
> mm <- lm(log(y) ~ log(n.vec))
> broom::tidy(mm)

## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    -5.02    0.395      -12.7 2.51e- 8
## 2 log(n.vec)      2.60    0.0685      38.0 7.14e-14

> abline(mm)
```