Benchmark Data Manipulation

Shian Su

2021-05-19

Contents

1 Introduction

This vignette contains some examples of common manipulations of data objects in this package. This package is built around the tidy data ideas established by Hadley Wickham's tidyverse, the primary goals are to keep data in an organised manner and enable concise manipulations for achieving a wide range of outcomes.

2 Benchmark Tibble

2.1 Basics

The fundamental object in this package is the benchmark tibble. Tibbles are variant of the `data.frame`, they are used here for their nicer printing properties.

The benchmark tibble is a `data.frame` structure where the `result` column is a special *list-column* which is allowed to contain arbitrary data types compared to a regular column which may only contain atomic data types such as `numeric`, `logical` and `character`. A *list-column* is a list with the same number of elements as there are rows in the data frame, they exist because the typical column is a vector which cannot contain complicated data types. However if the results of a computation are simple atomic values, then the column will be coerced to a regular vector with all the expected behaviours.

IMPORTANT! Because the result column is a list, care must be taken when performing certain operations. Most vectorised operations do not work on lists, and when we will cover how to properly work with these columns in the Operations On Benchmark Tibbles section.

We demonstrate how the benchmark tibble works:

```
library(CellBench)
datasets <- list(
    random_mat1 = matrix(runif(100), 10, 10),
    random_mat2 = matrix(runif(100), 10, 10)
)

cor_method <- list(
    pearson = function(x) { cor(x, method = "pearson") },
    kendall = function(x) { cor(x, method = "kendall") }
)

res <- datasets %>%
    apply_methods(cor_method)
```

As we can see, the table contains the data used, methods applied and the result of the computation. The reason for using tibbles is so that the result column is printed in a summarised form rather than fully expanded as would be the case for non-tibble list-columns.

2.2 Operations On Benchmark Tibbles

```
class(res)
## [1] "benchmark_tbl" "tbl_df"        "tbl"           "data.frame"
```

The benchmark tibble inherits from tibbles which inherit from data.frame, so operations expected to work on the parent classes should be expected to work on the benchmark tibble.

```
res[1:2, ]
## # A tibble: 2 x 3
##   data        cor_method result
##   <fct>       <fct>      <list>
## 1 random_mat1 pearson    <dbl [10 x 10]>
## 2 random_mat1 kendall    <dbl [10 x 10]>
```

By default tibbles only print the first 10 rows, this doesn't change with how many elements you subset. Instead you should use `print(res, n = Inf)` if you wish to print the whole tibble, or a desired number of rows.

We can also make use of the `dplyr` functions along with piping to write concise expressions for manipulating the benchmark tibble.

```
library(dplyr)
res %>%
    filter(cor_method == "pearson")
## # A tibble: 2 x 3
##   data        cor_method result
##   <fct>       <fct>      <list>
## 1 random_mat1 pearson    <dbl [10 x 10]>
## 2 random_mat2 pearson    <dbl [10 x 10]>
```

It is also possible to cbind two benchmark tibbles together, for example if you had added another set of methods

```
cor_method <- list(
    spearman = function(x) cor(x, method = "spearman")
)

res2 <- datasets %>%
    apply_methods(cor_method)

res2
## # A tibble: 2 x 3
##   data        cor_method result
##   <fct>       <fct>      <list>
## 1 random_mat1 spearman   <dbl [10 x 10]>
## 2 random_mat2 spearman   <dbl [10 x 10]>
```

```
rbind(res, res2)
## # A tibble: 6 x 3
##   data        cor_method result
##   <fct>       <fct>      <list>
## 1 random_mat1 pearson    <dbl [10 x 10]>
## 2 random_mat1 kendall    <dbl [10 x 10]>
## 3 random_mat2 pearson    <dbl [10 x 10]>
## 4 random_mat2 kendall    <dbl [10 x 10]>
## 5 random_mat1 spearman   <dbl [10 x 10]>
## 6 random_mat2 spearman   <dbl [10 x 10]>
```

This allows new methods to be added without having to recompute results for old methods.

2.3 Operations On list-columns

We note again that the benchmark column is a list.

```
class(res$result)
## [1] "list"
```

This means some simple vectorised functions will not quite work as expected. For example if we wished to take the exponential of all the matrices using `dplyr::mutate()`. Because `dplyr` feeds entire columns into the functions and expects the entire column to be returned, the result of the following code will attempt to run `exp()` on a `list` which it cannot handle.

```
# this code will fail
res %>%
    mutate(exp_result = exp(result))
```

Instead we must reformulate these to expressions that take in list arguments and return lists or vectors of the same length. This can be done using either `lapply` from the base R library or `map` from the `purrr` package.

```
res %>%
    mutate(exp_result = lapply(result, exp)) %>%
    mutate(sum_of_exp = unlist(lapply(exp_result, sum)))
## # A tibble: 4 x 5
##   data        cor_method result          exp_result      sum_of_exp
##   <fct>       <fct>      <list>          <list>               <dbl>
## 1 random_mat1 pearson    <dbl [10 x 10]> <dbl [10 x 10]>       122.
## 2 random_mat1 kendall    <dbl [10 x 10]> <dbl [10 x 10]>       119.
## 3 random_mat2 pearson    <dbl [10 x 10]> <dbl [10 x 10]>       121.
## 4 random_mat2 kendall    <dbl [10 x 10]> <dbl [10 x 10]>       120.
```

2.4 Unnesting with Lists of data.frames

One of the most useful representations that can be created in the tibble framework is to have data frames with consistent columns as the `result` list-column. This allows the data to be unnested such that the contents of the result data frames are row-contenated and the information in the remaining rows are duplicated accordingly.

```
library(tibble)

df1 <- data.frame(
    little = c(1, 3),
    big = c(5, 7)
)

df1
##   little big
## 1      1   5
## 2      3   7
```

```
df2 <- data.frame(
    little = c(2, 4),
    big = c(6, 8)
)

df2
##   little big
## 1      2   6
## 2      4   8
```

```
tbl <- tibble(
    type = c("odds", "evens"),
    values = list(df1, df2)
)

tbl
## # A tibble: 2 x 2
##   type  values
##   <chr> <list>
## 1 odds  <df [2 x 2]>
## 2 evens <df [2 x 2]>
```

```
tidyr::unnest(tbl)
## Warning: `cols` is now required when using unnest().
## Please use `cols = c(values)`
## # A tibble: 4 x 3
##   type  little   big
##   <chr>  <dbl> <dbl>
## 1 odds       1     5
## 2 odds       3     7
## 3 evens      2     6
## 4 evens      4     8
```

3 Manipulating Functions

3.1 Basics of Functional Programming

The book Advanced R contains an excellent section on Functional Programming. The primary idea we want to make use of is that functions are objects, not too different from numbers or character strings. For example we can think of anonymous functions like raw literal values.

```
# a numeric literal
1
## [1] 1

# a character literal
"a"
## [1] "a"

# a function literal
function(x) { print(x) }
## function(x) { print(x) }
```

We can assign these to variables in the same way

```
# assigning numeric literal
x <- 1

# assigning character literal
x <- "a"

# assigning function literal
f <- function(x) { print(x) }
```

We can also reassign variables to other variables

```
# assigning numeric literal
x <- 1
y <- x # y = 1

# assigning character literal
x <- "a"
y <- x # y = "a"

# assigning function literal
f <- function(x) { print(x) }
g <- f # g = function(x) { print(x) }
```

Being able to accept functions as regular objects is fundamental to making effective use of this package.

3.2 Partial Application

Partial application is a way to manipulate function objects. The idea is that you take a function which accepts multiple arguments, and "partially" apply some arguments to it. The simplest way to perform a partial application is to write a new function that wraps around the original function but with some arguments already filled in.

```r
# function to add two things
plus <- function(x, y) { x + y }

# function that adds 2 to x
plus_two <- function(x) { plus(x, y = 2) }

plus_two(1)
## [1] 3
```

Equivalently, we could use `purrr::partial()` which properly encapsulates this idea into a helper function. This is preferable to writing the wrapping function because it's very explicit in what its purpose is.

Wrapper function can do all sorts of computations beyond just filling in a variable, whereas `purrr::partial()` performs the singular duty of partially filling in arguments, there is no room to sneak in additional work that might complicate the process.

```r
library(purrr)

plus_two <- partial(plus, y = 2)

plus_two(1)
## [1] 3
```

3.3 Sequence of Partial Applications

CellBench offers a function to help construct partially-applied functions with oen or more sequences of arguments.

```r
# define a function that multiplies 3 numbers together
g <- function(x, y, z) {
    x * y * z
}

g(1, 2, 3)
## [1] 6
```

```r
# create a list of functions with the second and third values partially applied
# all combinations of y and z are generates, resulting in a list of 4 functions
g_list <- fn_arg_seq(g, y = c(1, 2), z = c(3, 4))

# apply each of the functions in the list to the value 1
lapply(g_list, function(func) { func(x = 1) })
## $`g(y = 1, z = 3)`
## [1] 3
```

```
##
## $`g(y = 2, z = 3)`
## [1] 6
##
## $`g(y = 1, z = 4)`
## [1] 4
##
## $`g(y = 2, z = 4)`
## [1] 8
```

This can be very useful for testing out a range or grid of parameters with very little code repetition.

3.4 Memoisation

Memoisation is the functional programming techinque of caching the result of computations. When a memoised function is called with arguments it had previously been evaluated with, it will simply recall the return value from the cache rather than redo the computations.

Memoisation is a operation on functions, taking in a regular function and returning a memoised version. CellBench offers memoisation through the `cache_method()` command which wraps around functionality from the `memoise` CRAN package. Memoised functions store their cache on disk, so be careful with functions that return large output objects.

```
# initialise the CellBench cache
cellbench_cache_init()

# dummy simulation of a slow function
f <- function(x) {
    Sys.sleep(2)
    return(x)
}

# create the memoised version of the function
cached_f <- cache_method(f)

# running the first time will be slow
cached_f(1)

# running the second time will be fast
cached_f(1)
```

4 Further Reading

- Introduction to tibbles: https://tibble.tidyverse.org
- Introduction to purrr: https://purrr.tidyverse.org