Biostatistics 615 - Statistical Computing

Special Topic
Rcpp and R packages

Hyun Min Kang

Topic Overview

- Evaluating computational efficiency with `microbenchmark`
- Efficient implementation of a function using `Rcpp`
- Writing R packages

Topic Overview

- Evaluating computational efficiency with `microbenchmark`
- Efficient implementation of a function using `Rcpp`
- Writing R packages

Microbenchmarking

Accurate Timing Functions

- A microbenchmark is a measurement of the performance of a very small piece of code, something that might take microseconds (µs) or nanoseconds (ns) to run
- It provides very precise timings, making it possible to compare operations that only take a tiny amount of time. For example, the following code compares the speed of three ways of computing a square root.

Hands-on Session

Visit https://bit.ly/615top99r

- Evaluating computational efficiency with `microbenchmark`
- Example - Comparing for loop, sapply, and apply
- Example - Comparing R and Rcpp

Topic Overview

- Evaluating computational efficiency with `microbenchmark`
- Efficient implementation of a function using `Rcpp`
- Writing R packages

High Performance Functions With Rcpp

This magic comes by way of the Rcpp package, a fantastic tool makes it very simple to connect C++ to R
Typical bottlenecks that C++ can address include:

- Loops that can not be easily vectorised because subsequent iterations depend on previous ones.
- Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than that in R
- Problems that require advanced data structures and algorithms that R does not provide. Through the standard template library (STL), C++ has efficient implementations of many important data structures, from ordered maps to double-ended queues.

Prerequistes

All examples in this lecture need version 0.10.1 or above of the `Rcpp` package. This version includes `cppFunction()` and `sourceCpp()`, which makes it very easy to connect `C++` to `R`. Install the latest version of `Rcpp` from CRAN with `install.packages("Rcpp")`. You willl also need a working `C++` compiler. To get it:

- On Windows, install Rtools `https://cran.r-project.org/bin/windows/Rtools/`.
- On Mac, install Xcode from the app store.
- On Linux, sudo apt-get install r-base-dev or similar.

Hands-on Session

Visit https://bit.ly/615top99r

- Using `cppFunction`

How cppFunction works

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function. We will summarize the basics by translating simple R functions to their C++ equivalents. We start simple with a function that has no inputs and a scalar output, and then it gets progressively more complicated.

Hands-on Session

Visit https://bit.ly/615top99r

- `Rcpp` example - no input, scalar output

Difference between C++ and R functions

This small function illustrates a number of important differences between `R` and `C++`:

- The syntax to create a function looks like the syntax to call a function; you do not use assignment to create functions as you do in `R`.
- You must declare the type of output the function returns. This function returns an int (a scalar integer). The classes for the most common types of `R` vectors are: `NumericVector`, `IntegerVector`, `CharacterVector`, and `LogicalVector`.
- Scalars and vectors are different. The scalar equivalents of `numeric`, `integer`, `character`, and `logical` vectors are: `double`, `int`, `String`, and `bool`.
- You must use an explicit `return` statement to return a value from a function.
- Every statement is terminated by a `;`.

Hands-on Session

Visit https://bit.ly/615top99r

- Rcpp example - scalar input, scalar output
- Rcpp example - vector input, scalar output
- Rcpp example - vector input, vector output
- Rcpp example - Matrix input, vector output

Using sourceCpp()

Inline C++ with `cppFunction()`. This makes presentation simpler, but for real problems, it is usually easier to use stand-alone C++ files and then source them into R using `sourceCpp()`.

This lets you take advantage of text editor support for C++ files (e.g., syntax highlighting) as well as making it easier to identify the line numbers in compilation errors.

Your stand-alone C++ file should have extension `.cpp`, and needs to start with:

```
1  #include <Rcpp.h>
2  using namespace Rcpp;
```

Each C++ function that will be used in R needs to be defined with:

```
1  // [[Rcpp::export]]
```

Hands-on Session

Visit https://bit.ly/615top99r

- `Rcpp` example - Using `sourceCpp()`
- `Rcpp` example - Attributes and other classes

Lists and data frames

- `Rcpp` also provides classes `List` and `DataFrame`, but they are more useful for output than input.
- This is because lists and data frames can contain arbitrary classes but `C++` needs to know their classes in advance.
- If the list has known structure, you can extract the components and manually convert them to their `C++` equivalents with `as()`.
- For example, the object created by `lm()`, the function that fits a linear model, is a list whose components are always of the same type.

Hands-on Session

Visit https://bit.ly/615top99r

- `Rcpp` example - Lists and DataFrames
- `Rcpp` example - Function

Topic Overview

- Evaluating computational efficiency with `microbenchmark`
- Efficient implementation of a function using `Rcpp`
- Writing R packages

Writing R package - Introduction

- The R packaging system has been one of the key factors of the overall success of the R project
- Packages allow for easy, transparent and cross-platform extension of the R base system.
- R packages are (after a short learning phase) a comfortable way to maintain collections of R functions and data sets.

Step-by-step Instructions to Write an R package

Below is a step-by-step instruction on how to create an R package

1. Install `devtools`, `roxygen2`, `usethis` packages if not yet installed.
2. Write your R function(s) into R file(s)
3. Document your R function in `roxygen2` style
4. Run `package.skeleton()` function to create a package.
5. Modify `DESCRIPTION` and `man/pkgName-package.Rd` to document the package.
6. Run a specific series of commands to build the package
   - `usethis::use_gpl3_license("Type Your Name")` (or other license)
   - Delete `NAMESPACE`, `man/fastSimpleLinearRegression.Rd`, and `Read-and-delete-me` files
   - `devtools::document()`
   - `devtools::check()`
   - `devtools::build()`
7. Install the package and test it.

2. Write your R function(s) into R file(s)

For example, in `simplelm.R`, we have the following code for fast simple linear regression

```
1  fastSimpleLinearRegression = function(y, x)  {
2    y = y - mean(y)
3    x = x - mean(x)
4    n = length(y)
5    stopifnot(length(x) == n) # for error handling
6    s2y = sum( y * y ) / ( n - 1 ) # \sigma_y^2
7    s2x = sum( x * x ) / ( n - 1 ) # \sigma_x^2
8    sxy = sum( x * y ) / ( n - 1 ) # \sigma_xy
9    rxy = sxy/sqrt( s2y * s2x ) # \rho_xy
10   b = rxy * sqrt( s2y / s2x )
11   se.b = sqrt( s2y * ( 1 - rxy * rxy ) / (n-2) / s2x )
12   tstat = rxy * sqrt( ( n - 2 ) / ( 1 - rxy * rxy ) )
13   p = pt( abs(tstat) , n - 2 , lower.tail=FALSE )*2
14   return(list( beta = b , se.beta = se.b , t.stat = tstat, p.value = p ))
15 }
```

3. Document your R function in roxygen2 style

roxygen2 is a package that automatically creates R package documentations for R and Rcpp source codes.

The comment lines starts with # (for R) or (for Rcpp) and generates automated documentation for each exported function.

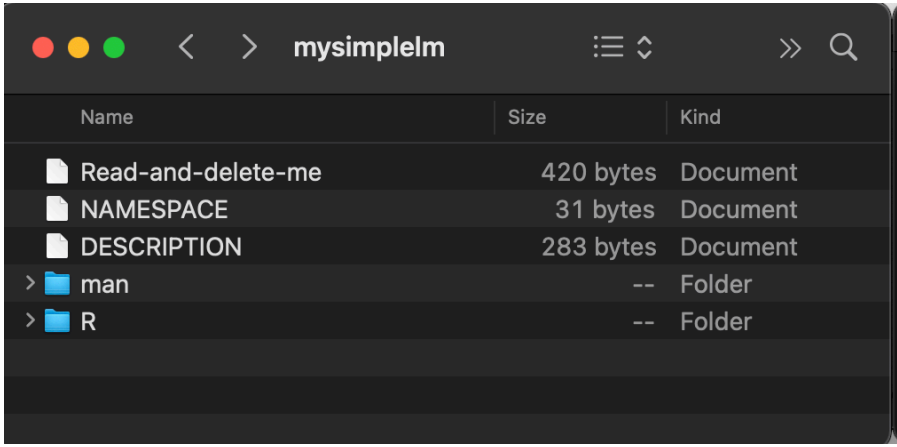
```
1  #' A fast implementation of simple linear regression
2  #' @param x A size n vector
3  #' @param y A size m vector
4  #' @return A list containing beta, se.beta, t.stat, p.value
5  #' @examples
6  #' y = rnorm(100)
7  #' x = rnorm(100)
8  #' fit = fastSimpleLinearRegression(y,x)
9  #' @importFrom stats pt
10 #' @export
11 fastSimpleLinearRegression = function(y, x)  {
12 ...
```

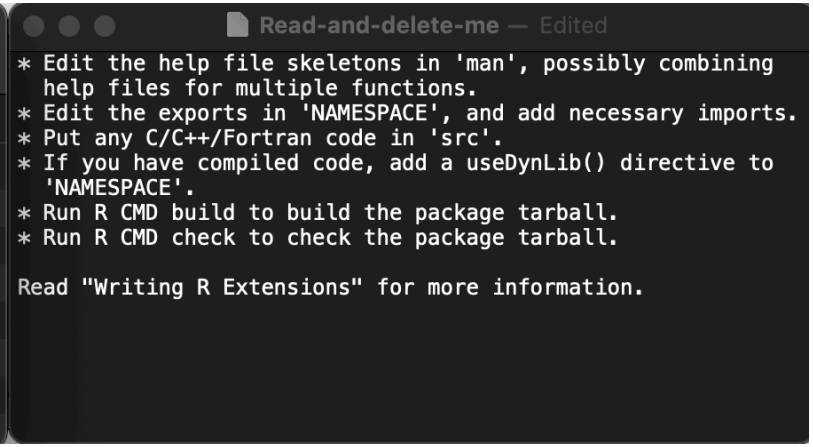
4. Run package.skeletion()

In a clean R session, and run package.skeleton()

```
1  package.skeleton("mysimplelm",code_files="simplelm.R")
```

```
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Copying code files ...
Making help files ...
Done.
Further steps are described in './mysimplelm/Read-and-delete-me'.
```





5a : Modify DESCRIPTION file

DESCRIPTION

```
1  Package: mysimplelm
2  Type: Package
3  Title: A fast simple linear model
4  Version: 1.0
5  Date: 2021-11-04
6  Author: Hyun Min Kang
7  Maintainer: Hyun Min Kang <hmkang@umich.edu>
8  Description: This package provides a fast simple linear regression fitting.
9  License: GPL-3
10 Encoding: UTF-8
```

5b : Modify the package documentation file

man/mysimplelm-package.Rd

```
\name{mysimplelm-package}
\alias{mysimplelm-package}
\alias{mysimplelm}
\docType{package}
\title{
\packageTitle{mysimplelm}
}
\description{
\packageDescription{mysimplelm}
}
\details{
The DESCRIPTION file:
\packageDESCRIPTION{mysimplelm}
\packageIndices{mysimplelm}
This package provides a fast simple linear regression fitting`
}
\author{
\packageAuthor{mysimplelm}
Maintainer: \packageMaintainer{mysimplelm}
}
\examples{
x = rnorm(100)
y = rnorm(100)
fit = fastSimpleLinearRegression(y,x)
}
```

Step 6a: Run a series of commands

```
1  setwd("mysimplelm")
2  usethis::use_gpl3_license()
```

```
Setting active project to '/Users/hmkang/Downloads/mysimplelm'
Setting License field in DESCRIPTION to 'GPL-3'
Writing 'LICENSE.md'
Adding '^LICENSE\\.md\$' to '.Rbuildignore'
```

```
Delete NAMESPACE and man/fastSimpleLinearRegression.Rd if exist.
```

```
1  devtools::document()
```

```
Updating mysimplelm documentation
First time using roxygen2. Upgrading automatically...
Loading mysimplelm
Writing NAMESPACE
Writing NAMESPACE
Writing fastSimpleLinearRegression.Rd
```

Step 6b: Check and build

```
devtools::check()
```

```
Updating mysimplelm documentation
Loading mysimplelm
Writing NAMESPACE
Writing NAMESPACE
...
Duration: 7.5s
0 errors | 0 warnings | 0 notes
```

```
devtools::build()
```

```
checking for file '/Users/hmkang/Downloads/mysimplelm/'DESCRIPTION...
...
- building 'mysimplelm_1.0.tar.'gz
[1] "/Users/hmkang/Downloads/mysimplelm_1.0.tar.gz"
```

Step 7: Install and test

```
1  setwd("..")
2  install.packages("mysimplelm_1.0.tar.gz",repos=NULL)
```

```
* installing *source* package ‘’mysimplelm ...
** R
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (mysimplelm)
```

```
1  fastSimpleLinearRegression(rnorm(100),rnorm(100))
```

```
$beta
[1] -0.02182883
$se.beta
[1] 0.1016057
$t.stat
[1] -0.2148386
$p.value
[1] 0.8303396
```

How about C++ code?

Using the `Rcpp` package, we can improve the efficiency of `R` code

In ``cppfastLM.cpp", we have the following code:

```
#include <Rcpp.h>
#include <cmath>
using namespace Rcpp;
//' A fast implementation of simple linear regression
//' @param x A size n vector
//' @param y A size m vector
//' @return A list containing beta, se.beta, t.stat, p.value
//' @importFrom stats pt
// [[Rcpp::export]]
List cppfastLM(NumericVector y, NumericVector x){
  y = y - mean(y);
  x = x - mean(x);
  int  n = y.size();
  double s2y = sum( y * y );
  s2y /= n - 1.0;
  double s2x = sum( x * x );
  s2x /= n - 1.0;
  double sxy = sum( x * y );
  sxy /= n - 1.0;
  double rxy = sxy;
  rxy /= sqrt( s2y * s2x );
  double b = rxy * sqrt( s2y / s2x );
  double se_b = sqrt(s2y * ( 1.0 - rxy * rxy ) / (n-2.0) / s2x);
  NumericVector tstat;
  tstat.push_back(rxy * sqrt( ( n - 2 ) / ( 1 - rxy * rxy ) ));
  NumericVector p = pt(abs(tstat), n - 2 ,0, 0)*2;
  return Rcpp::List::create(Rcpp::Named("beta") = b, Rcpp::Named("se.beta") = se_b,
                    Rcpp::Named("t.stat") = tstat[0], Rcpp::Named("p.value") = p[0]);
}
```

Step-by-step Instructions to Write an Rcpp package

Below is a step-by-step instruction on how to create an R package with Rcpp code

1. Install `Rcpp`, `devtools`, `roxygen2`, `usethis` packages if needed.
2. Write your R/Rcpp function(s) into R/C++ file(s)
3. Document your R/Rcpp functions in `roxygen2` style
4. Run `Rcpp.package.skeleton(...)` function.
5. Modify `DESCRIPTION` and `man/pkgName-package.Rd` to document the package.
6. Run a specific series of commands to build the package
   - `usethis::use_gpl3_license()`
   - Run `compileAttributes(verbose=TRUE)`
   - Run `devtools::load_all()`
   - Delete `NAMESPACE` and `Read-and-delete-me` files
   - `devtools::document()`
   - `devtools::check()`
   - `devtools::build()`
7. Install the package and test it.

Using Rcpp.packages.skeleton()

In **Step 4**, now we use `Rcpp.packages.skeleton()`

```
1  Rcpp.package.skeleton("myfastsimplelm",code_files = "simplelm.R",
2                        cpp_files="cppFastLM.cpp", example_code=FALSE)
```

```
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Copying code files ...
Making help files ...
Done.
Further steps are described in './myfastsimplelm/Read-and-delete-me'.

Adding Rcpp settings
 >> added Imports: Rcpp
 >> added LinkingTo: Rcpp
 >> added useDynLib directive to NAMESPACE
 >> added importFrom(Rcpp, evalCpp) directive to NAMESPACE
 >> copied cppFastLM.cpp to src directory
 >> added example src file using Rcpp attributes
 >> compiled Rcpp attributes
```

Performance Comparison

```
1  library(microbenchmark)
2  library(myfastsimplelm)
3  n = 1000000
4  x = rnorm(n)
5  y = 2*x+rnorm(n)
6  microbenchmark(lm(y~0+x),
7                 fastSimpleLinearRegression(y,x),
8                 cppfastLM(y,x),
9                 times = 10L)
```

```
Unit: milliseconds
                              expr        min
                     lm(y ~ 0 + x) 630.289271
 fastSimpleLinearRegression(y, x)  18.127215
                   cppfastLM(y, x)   7.176658
         lq      mean     median         uq
 686.148319 810.42767 756.022109 802.849028
  31.825939  96.65432  61.273852 190.640336
   7.350376   7.72509   7.672616   8.228796
         max neval
 1225.938566    10
  228.754991    10
    8.321194    10
```