Stat 405/705
Class 11
Statistical computing with R

Richard P. Waterman

Wharton

Table of contents I

Today's module

Topics to be covered in this module:

- Last time
- Review case study: level of effort
- The R-ecosystem
- Extending R with packages
- Functions used in today's class
- Next time

Last time

- Simulation modeling
- Case study: level of effort

The R-ecosystem

- There is both a large community of individuals active in R
- And a large code base to draw from
- Pretty much every standard thing you want to do, someone else has already done
- See what is already available before reinventing the wheel

Useful resources from the community

- Comprehensive R Archive Network (CRAN): https://cran.r-project.org/
- The R-help mailing list. Ask questions and stay up to date with what's new: https://stat.ethz.ch/mailman/listinfo/r-help
- stackoverflow. Ask and answer questions: http://stackoverflow.com/tags/r/info
- Documentation browser: http://www.rdocumentation.org/

Add-on packages

- When you have enough functions that you have written and think they may be useful to someone else, then you can bundle them as a package and submit to CRAN
- There are rules to be followed and conventions to be recognized when creating a package
- CRAN has 16,000 (as of 10/8/2018) different packages available
- The steps in *using* a package are to
  1. Download it
  2. Make it available within R with the `library command`
- In RStudio, use the Tools→Install Packages ... to do this
- In R, use the Packages→Install Packages to do this

Add-on packages

- Most R users use the terms `package` and `library` interchangeably
- Formally, a package holds a set of commands that extend R
- The directory into which the package is dropped is called the library

Some useful R packages

Useful ultimately depends on what you do. But here are some suggestions:

Data manipulation	
`tidyverse`	Data science related packages
`dplyr`	Splitting and merging data, part of the `tidyverse`
Graphics	
`RColorBrewer`	Color palettes
`ggplot2`	Higher level attractive graphics. part of the `tidyverse`
`animation`	Facilitate making animations
Reporting	
`xtable`	Automatic table generation HTML/LaTeX
`knitr`	Live reports, interleaf R within a document

Some useful R packages

Useful ultimately depends on what you do. But here are some suggestions:

Database and connectivity	
`RODBC`	SQL database connectivity
`parallel`	Big chunk parallelization
Analytics	
`lme4`	Linear mixed effects models
`randomForest`	Goto predictive analytics methodology
`ranger`	Fast random forests
`pROC`	Summarizing predictive analytics
`glmnet`	GLMs for data mining
`caret`	Tuning parameter tools for data mining
General	
`MASS`	Contains many useful workhorse functions
`boot`	Bootstrapping

Using packages to help improve the simulation example

- We will do two things:
  1. Parallelize the simulation
  2. Create graphics and build them into an animation

The parallel package

- Now part of the R core.
- Spread the computational workload over a set of different computers, or computing cores on the same machine
- It is suitable for parallelizing large computational tasks (**not** fine grained linear algebra activities such as matrix inversion etc.)
- Simulation modeling and *bootstrap* type activities are an ideal application of this package
- The key feature for success: the computational chunks can be executed independently of one another (no inter-process communication) and all that is ultimately required are the outputs from each, which then need to be aggregated.

The parallel package

The basic idea:

- Create a set of clones (nodes) of the current R (master) process, appropriately initialized (libraries loaded etc).
- Send those clones work to do and collect the results.
- Close down the clones.
- The clones can be on the same machine, or on different machines.
- We will work with the same machine, assuming multiple computing cores.

Extending the apply class of functions

The main idea:

- We are very familiar with the `apply` concept.
- The `parallel` package will provide analogues of the `apply` function, but now as the list/matrix is iterated over the work will be sent to different cores/machines.

Parallelizing the simulation

The work plan:

1. Get the simulation model working properly on a single machine.
2. Wrap the entire simulation in a function, where the arguments to the function are the values you want to iterate over (the levels of effort) and any constants/inputs you may want to change (population size).
3. Have this function return the features of the simulation you want to keep (the number of remaining infringers at the end of each week).
4. Deploy the simulation.
5. Summarize the results as appropriate.

Timer functions

For reference here they are again.

```
timer.start <- function(name = "Timer 1"){ # A single argument
  a <- Sys.time() # Save the current time into "a"
  return(list(NAME = name, START = a))  # Return the list
}

timer.stop <- function(st){ # "st" is the output from "timer.start"
  b <- Sys.time() # Save the current time
  elapsed.time <- b - st[["START"]] # Work out the elapsed time
  print (paste(st[["NAME"]], "took:", # A human readable message
               elapsed.time, attributes(elapsed.time)$units))
  invisible(elapsed.time) # Return the timer difference invisibly
}
```

Review the wrapped simulation

The function `effort.sim` contains the wrapped simulation. It's up to you what you want to keep as a constant in the function and what you want to pass in as an argument:

```
# Read in the function and check out its arguments
source('C:/Users/richardw/Dropbox (Penn)/Teaching/705s2019/Data/class_10_si
args(effort.sim) # Print out the arguments

## function (X = 10, n.infringers = 1000, n.weeks = 52, het.choice = 1)
## NULL
```

The argument **X** will contain the effort level and matches what the parallelized `lappy` function expects as its argument.

Review the wrapped simulation

```
### Create variables holding the directory and file name
### of the simulation function
my.location <- 'C:/Users/richardw/Dropbox (Penn)/Teaching/705s2019/Data/'
my.file.name <- "class_10_sim_as_function.r"

#Read in the function itself
source(paste(my.location, my.file.name,sep=""))

#### Check that the effort simulation function is working
res1 <- effort.sim(X = 10, n.infringers = 1000,
                   n.weeks = 52, het.choice=1)

## Number of infringers on iteration 11 is 987
## Number of infringers on iteration 21 is 977
## Number of infringers on iteration 31 is 966
## Number of infringers on iteration 41 is 950
## Number of infringers on iteration 51 is 940
## [1] "The number of infringers at one year is  940 with effort 10"
```

Review the wrapped simulation

```
# Load the "parallel" package
library(parallel) # The parallelization package

# Detect the number of cores on a machine (for curiosity)
num.cores <- detectCores()
print(num.cores) # How many do we have?

## [1] 8

# Get the timer started
t1 <- timer.start("Parallel computation timer") # Start timing
```

Review the wrapped simulation

```
# Fire up the cluster, making use of 4 cores
my.cluster <- makeCluster(spec = 4)
# Set the random number generator for the cluster.
# This is subtle as you would not want each node to
# use the same seed.
clusterSetRNGStream(cl = my.cluster, iseed = 19390909)

#### I am not using either of these commands in this instance,
#### but often do.
#### Export objects that the individual nodes need to know about
# clusterExport(cl = my.cluster, c("objects.to.be.exported"))

####  Code to be executed when each node of the cluster
####  fires up (initialization)
####  "require" is an alternative to the "library" function
####  If the package doesn't exist, it returns a warning, not an error.
# clusterEvalQ(cl = my.cluster, expr = require(lme4))
####  Now the "lme4" package is  loaded on each node
```

Review the wrapped simulation

```
#### The key command for the parallel implementation.
#### Apply the function "fun" on each node of the cluster,
#### with an argument, X, that varies with the level
#### of effort
#### par(allel)L(ist)applyL(oad)B(alance)

results <- parLapplyLB(cl = my.cluster,
                       X = seq(100,1000,100),
                       fun = effort.sim,
                       n.infringers = 10000,
                       het.choice=1)
# Shut down the cluster
stopCluster(cl = my.cluster)
```

Review the wrapped simulation

```
#### Make sure everything returned without any errors by
#### checking we have numeric answers
success <- sapply(results, is.numeric)
print(success)

##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

#### Combine the individual columns in the result matrices
#### with the "cbind" command
#### "do.call" creates a function call from "what" and "args"
big.result <- do.call(what = cbind, args = results[success])

### Stop timing
timer.stop(t1)

## [1] "Parallel computation timer took: 15.2248089313507 secs"
```

Review the wrapped simulation

Review the `big.result` matrix:

```
head(big.result,4)

##        Effort_100 Effort_200 Effort_300 Effort_400
## Week_1      10000      10000      10000      10000
## Week_2       9994       9988       9969       9962
## Week_3       9985       9966       9937       9919
## Week_4       9969       9947       9886       9882
##        Effort_500 Effort_600 Effort_700 Effort_800
## Week_1      10000      10000      10000      10000
## Week_2       9957       9940       9927       9915
## Week_3       9908       9881       9850       9841
## Week_4       9853       9815       9770       9758
##        Effort_900 Effort_1000
## Week_1      10000       10000
## Week_2       9898        9893
## Week_3       9794        9779
## Week_4       9693        9637
```

Parallelization discussion

- We obtained a significant speed-up compared to a single core
- Some package functions have already been parallelized and more will undoubtedly become parallelized over time
- If you really need to use many cores use an Amazon Web Services account!

Animating the simulation results

We will use a package called `animation` to help with this.

```
library(animation)

## Warning:  package 'animation' was built under R version 3.5.2

ani.record(reset = TRUE) # clear history before recording
#### Create the plots:
for (i in 1:ncol(big.result)) { # Iterate over the columns
      plot(big.result[,i],      # Make a plot for each column
      main= paste("Effort level",colnames(big.result)[i]),
      xlab = "Week",
      ylab = "Infingers",
      ylim = c(0, 10000),
          )
      ani.record()              # Record the current frame
    }
```

Animating the results

Replay and save the animations to a web page.

```
oopts = ani.options(interval = 0.25)
# Have a look at the animation  in R
# ani.replay()

# Create a webpage to display the animation
saveHTML(ani.replay(), img.name = "record_plot")

## animation option 'nmax' changed:  50 --> 10
## animation option 'nmax' changed:  10 --> 50
## HTML file created at:  index.html

# The animation package has many other options
# for saving the output, animated GIFs etc.
```

Module summary

Topics covered today include:

- The R eco-system
- Resources
- Packages

Next time

- More on graphics
- `ggplot2`

Today's function list

Do you know what each of these functions does?

```
args
clusterSetRNGStream
detectCores()
do.call
library
makeCluster()
parLapplyLB
require
sapply
source
stopCluster
```