**Introduction**

MCHT, an R package that facilitates bootstrap and Monte Carlo hypothesis testing. In this article, I will elaborate on some important technical details about making MCHTest objects, explaining in the process how closures and R environments work.

To recap, I made a basic MCHTest-class object. These are S3-class objects; really they are just functions with a class attribute. All the work is done in the initial function call creating the object. But there’s more to the story.

We want these objects to be self-contained. Specifically, we don’t want changes in the global namespace to change how a MCHTest object behaves. By default, these objects are *not* self-contained and a programmer who isn’t careful can accidentally break these objects. Here I explain how to prefent this from happening.

**Closures and Environments**

A [closure](https://en.wikipedia.org/wiki/Closure_(computer_programming)) is a function created by another function. MCHTest objects are closures, functions created by MCHTest() (then given a class attribute). An environment is an R object where other R objects are effectively defined. For example, there is the global environment where most R objects created by users live.

environment()

##

globalenv()

##

Ever wonder why a variable defined inside a function doesn’t affect anything outside of that function and why it simply disappears? It’s because when a function is called, a new environment is created, and all assignments within the function are done within that new environment. We can see this occuring with some clever use of print().

x <- 2

u <- function() {

x

}

u()

## [1] 2

f <- function() {

x <- 1

function() {

x

}

}

g <- f()

g()

## [1] 1

environment(g)

##

environment(u)

##

parent.env(environment(g))

##

u() is a function and lives in the global environment so it looks for variables in the global environment. g(), however, lives in an environment created by f(). Normally, when a function creates an environment, it disappears the moment the function finishes execution. Closures, however, still use that environment created by the function, so the environment doesn’t disappear when the function finishes execution.

When a function looks for an object, it first looks for that object in its environment. If it doesn’t find the object there, it looks for the object in the parent environment of its environment. It will continue this process until it either finds the object or discovers that none of its environment’s ancestors has the object (prompting an error).

This means that the function is sensitive to changes in its environment or its environment’s ancestors, as we see here:

x <- 3

h <- function() {

function() {

x

}

}

u()

## [1] 3

j <- h()

environment(j)

##

parent.env(environment(j))

##

j()

## [1] 3

**Side Effects and MCHTest objects**

One of R’s attractive features is that it promotes a style of programming that discourages side effects, where changes to one object doesn’t change the behavior of another. But the examples above show how closures can suffer side effects when objects in the global namespace are changed. The closures created above depend on the global environment in surprising ways for those not familiar with how environments in R work.

By default, MCHTest objects can suffer from these side effects, and they can creep in if the functions passed to the parameters of MCHTest() are carelessly defined, as we see below. (The tests being defined are effectively Monte Carlo z-tests; learn about the z-test.)

library(MCHT)

## .------..------..------..------.

## |M.--. ||C.--. ||H.--. ||T.--. |

## | (\/) || :/\: || :/\: || :/\: |

## | :\/: || :\/: || (\_\_) || (\_\_) |

## | '--'M|| '--'C|| '--'H|| '--'T|

## `------'`------'`------'`------' v. 0.1.0

## Type citation("MCHT") for citing this R package in publications

library(doParallel)

registerDoParallel(detectCores())

ts <- function(x, sigma = 1) {

sqrt(length(x)) \* mean(x)/sigma # z-test for mean = 0

}

sg <- function(x, sigma = 1) {

x <- sigma \* x

ts(x, sigma = sigma) # unsafe

}

unsafe.test.1 <- MCHTest(ts, sg, rnorm, seed = 100, N = 100,

fixed\_params = "sigma")

unsafe.test.1(rnorm(10))

##

## Monte Carlo Test

##

## data: rnorm(10)

## S = 1.1972, sigma = 1, p-value = 0.15

ts <- function(x) {

sqrt(length(x)) \* mean(x) # Effective make sigma = 1

}

sg <- function(x) {

ts(x) # again, unsafe

}

unsafe.test.2 <- MCHTest(ts, sg, rnorm, seed = 100, N = 100)

unsafe.test.2(rnorm(10))

##

## Monte Carlo Test

##

## data: rnorm(10)

## S = 0.22926, p-value = 0.46

# ERROR

unsafe.test.1(rnorm(10))

## Error in {: task 1 failed - "unused argument (sigma = sigma)"

What happened? Let’s pick it apart by looking at the stat\_gen parameter of unsafe.test.1().

get\_MCHTest\_settings(unsafe.test.1)$stat\_gen

## function(x, sigma = 1) {

## x <- sigma \* x

## ts(x, sigma = sigma) # unsafe

## }

This function depends on an object called ts(). When the function looks for ts(), it looks *in the global namespace!* This means that changes to ts() in that namespace will change the behavior of the function. The most recent version of ts() does not have a parameter called sigma, prompting an error. *The object is not self-contained!*

**Making MCHTest Objects Self-Contained**

How can we prevent side effects like this? One answer is to define the functions passed to MCHTest() in a way that doesn’t depend on objects defined in the global namespace. For example, we would not call ts() in sg() above but instead rewrite the test statistic as we defined it in ts(). (Using functions and objects defined in packages is okay, though, since these generally don’t change in an R session.)

However, this is not always practical. The test statistic written in ts() could be complicated, and writing that same statistic again would not only be a lot of work but be tempting bugs to invade. Fortunately, MCHTest() supports methods for making MCHTest objects self-contained.

The first step is to set the localize\_functions parameter to TRUE. This changes the environment of the test\_stat stat\_gen, rand\_gen, and pval\_func functions so that they belong to the environment the MCHTest object lives in. Not only does this help make the function self-contained we may even be able to write our inputs in a more idiomatic way, like so:

ts <- function(x, sigma = 1) {

sqrt(length(x)) \* mean(x)/sigma

}

sg <- function(x, sigma = 1) {

x <- sigma \* x

test\_stat(x, sigma = 1) # Would not be able to do this if localize\_functions

# were FALSE

}

safe.test.1 <- MCHTest(ts, sg, function(n) {rnorm(n)}, seed = 100, N = 100,

fixed\_params = "sigma", localize\_functions = TRUE)

safe.test.1(rnorm(10))

##

## Monte Carlo Test

##

## data: rnorm(10)

## S = 2.0277, sigma = 1, p-value = 0.02

ts <- function(x) {

sqrt(length(x)) \* mean(x) # Effective make sigma = 1

}

sg <- function(x) {

ts(x)

}

safe.test.1(rnorm(10)) # Still works

##

## Monte Carlo Test

##

## data: rnorm(10)

## S = 1.0038, sigma = 1, p-value = 0.21

(Notice how rand\_gen was handled; it was wrapped in a function rather than passed directly. In short, this is to prevent the function rnorm from being stripped of its namespace, since it needs functions from that namespace.)

This is the first step to removing side effects. (In fact it makes our functions better written since we can anticipate the existence of test\_stat as a function). However, we could still have variables or functions defined outside of our input functions. We can expose these functions to our localized input functions via the imported\_objects parameter, a list (the doppleganger of R’s environments) containing these objects.

ts <- function(x, sigma = 1) {

sqrt(length(x)) \* mean(x)/sigma

}

sg <- function(x, sigma = 1) {

x <- sigma \* x

ts(x) # We're going to do this safely now

}

safe.test.2 <- MCHTest(ts, sg, function(n) {rnorm(n)}, seed = 100, N = 100,

fixed\_params = "sigma", localize\_functions = TRUE,

imported\_objects = list("ts" = ts))

safe.test.2(rnorm(10))

##

## Monte Carlo Test

##

## data: rnorm(10)

## S = 0.57274, sigma = 1, p-value = 0.39

ts <- function(x) {

sqrt(length(x)) \* mean(x) # Effective make sigma = 1

}

sg <- function(x) {

ts(x)

}

safe.test.2(rnorm(10))

##

## Monte Carlo Test

##

## data: rnorm(10)

## S = 0.24935, sigma = 1, p-value = 0.45

Both safe.test.1() and safe.test.2() are now immune to changes in the global namespace. They are self-contained and thus safe to use.

**Conclusion**

By default, localize\_functions is FALSE. I thought of making it TRUE by default but I feared that those not familiar with the concept of environments would be bewildered by all the errors that would be thrown whenever they tried to use a function they defined. Setting the parameter to TRUE makes using MCHTest() more difficult.

That said, I highly recommend using the parameter in a longer script. It makes the function safer (errors are good when they’re enforcing safety), so become acquainted with it.

**References**

1. H. Wickham, *Advanced R* (2015), CRC Press, Boca Raton