Simulación Estadística

Contenido del curso

Kevin Pérez - Ing de Sistemas - Estadístico - (E) Maestría en Ciencia de Datos
Departamento de Matemáticas y Estadística - Universidad de Córdoba

Contenido

- ***Capítulo I - Programación***
    - Programación básica en R
    - Programación con Funciones
    - Otras técnicas de programación

Contenido

- ***Capítulo II - Técnicas Numericas***
  - Números aleatorios
  - Generación de variables aleatorias
  - Integración numerica

Contenido

- ***Capítulo III - Técnicas de simulación***
  - Metodo de Monte-carlo
  - Reducción de varianza
  - Bootstrap
  - Procesos estocasticos
  - Otras técnicas de simulación

Motivación

Imagine que ha cargado un archivo de datos, como el que mostramos abajo, que usa `-99` para representar datos faltantes. Usted quiere reemplazar todos los `-99` con `Nas`

```
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
head(df, 3)
```

```
##    a b c   d   e f
## 1  1 6 1   5 -99 1
## 2 10 4 4 -99   9 3
## 3  7 9 5   4   1 4
```

Motivación

Cuando estas empezando a escribir código R, quizás pienses en resolver el problema con un *'copy-paste'*

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

Un problema con el *'copy-paste'* es que es fácil cometer errores con el, Puedes ver los dos en el código?, Estos errores son inconsistencias que surgen debido a que no tienen una descripción clara de la acción deseada (reemplazar - 99 con NA).

Motivación

Cuando empiezas a conocer mejor las buenas practicas de programación, programas una función que arregla los valores perdidos en un vector

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)
```

Esto reduce el alcance de los posibles errores , pero no los elimina : puede escribir -98 en lugar de -99.

Motivación

Podemos aplicar `lappy` a este problema por que los `data.frame` son listas

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df[] <- lapply(df, fix_missing)
```

Este código tiene las siguientes ventajas sobre el *'copy-paste'*

- Es más compacto
- Si el código para un valor perdido cambia, solo necesita ser actualizado en una parte
- Funciona para cualquier número de columnas, no existe forma de de accidentalmente perder omitir una columna
- No existe forma de accidentalmente tratar una columna diferente de otra

Motivación

- Es fácil generalizar esta técnica para un subconjunto de columnas

```
df[1:5] <- lapply(df[1:5], fix_missing)
```

La idea clave es la composición de funciones. Tomar dos funciones simples, una que hace algo para cada columna y una que fija los valores que faltan, y se combinan para fijar los valores faltantes cada columna. La escritura de funciones simples que se pueden entender de forma aislada y lo que hace a la composición una técnica poderosa.

Motivación

¿Qué pasa si diferentes columnas utilizan códigos diferentes para los datos faltan ten? Usted puede verse tentado al *'copy-paste'*:

```
fix_missing_99 <- function(x) {
  x[x == -99] <- NA
  x
}
fix_missing_999 <- function(x) {
  x[x == -999] <- NA
  x
}
fix_missing_9999 <- function(x) {
  x[x == -999] <- NA
  x
}
```

Motivación

Al igual que antes, es fácil para crear errores. En este caso, se podría argumentar que deberíamos añadir otro argumento:

```
fix_missing <- function(x, na.value) {
  x[x == na.value] <- NA
  x
}
```

Algo anda mal!

Indicadores de que algo anda mal!

- `message`: Una notificación genérica producida por la función `message`; la ejecución de la función continua
- `warning`: Un indicador de que algo anda mal pero no necesariamente fatal; la ejecución de la función continua; es generado por la función `warning`.
- `error`: Indicador de que un problema fatal a ocurrido; la ejecución se detiene; es producida por la función `stop`.
- `condition`: Un concepto genérico de que algo inesperado a ocurrido; los programadores pueden crear sus condiciones.

Algo anda mal!

Warning

```
log(-1)
```

```
## Warning in log(-1): Se han producido NaNs
```

```
## [1] NaN
```

Algo anda mal

```
printmessage <- function(x) {
        if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
```

Algo anda mal

```
printmessage <- function(x) {
          if(x > 0)
                print("x is greater than zero")
          else
                print("x is less than or equal to zero")
          invisible(x)
}
printmessage(1)
```

```
## [1] "x is greater than zero"
```

```
printmessage(NA)
```

```
## Error in if (x > 0) print("x is greater than zero") else print("x is less than or equal to
```

Algo anda mal!

```
printmessage2 <- function(x) {
        if(is.na(x))
                print("x is a missing value!")
        else if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
```

Algo anda mal!

```
printmessage2 <- function(x) {
        if(is.na(x))
                print("x is a missing value!")
        else if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
x <- log(-1)
```

```
## Warning in log(-1): Se han producido NaNs
```

```
printmessage2(x)
```

```
## [1] "x is a missing value!"
```

Herramientas de depuración R

Las principales herramientas de depuración en R son:

- `traceback`: Imprime un pantallazo de las llamadas de la función una vez ocurre un error.
- `debug`: Marca una función para el modo “debug” que le permite el ver el paso a través de una ejecución de una función una línea a la vez
- `browser`: Suspende la ejecución de una función mientras es puesta en modo debug
- `trace`: e permite introducir el código de depuración en una función a lugares específicos
- `recover`: Le permite modificar el comportamiento de error para que pueda navegar por las de llamadas de función

Estas son herramientas interactivas diseñadas específicamente para permitirle escoger a través de una función. También existe la técnica más contundente de la inserción de declaraciones de impresión en la función.

traceback

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
>
```

traceback

```
> lm(y ~ x)
Error in eval(expr, envir, enclos) : object 'y' not found
> traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

debug

```
> debug(lm)
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    ...
    if (!qr)
        z$qr <- NULL
    z
}
Browse[2]>
```

debug

```
Browse[2]> n
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action",
    "offset"), names(mf), 0L)
```

recover

```
> options(error = recover)
> read.csv("nosuchfile")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nosuchfile': No such file or directory

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec =
3: file(file, "rt")

Selection:
```

Standar de código para R

1. Siempre utilice archivos de text / editores de texto

Standar de código para R

1. Siempre utilice archivos de text / editores de texto

2. Indente su código

Standar de código para R

1. Siempre utilice archivos de text / editores de texto

2. Indente su código

3. Limite el ancho se su código (80 columnas?)

Indentado

- Indentado mejora la lectura del código
- El ajuste del ancho de la columna previene el anidado y la funciones muy largas
- Sugerencia: Indente mínimo 4 espacios; 8 espacios es lo ideal

Standar de código para R

1. Siempre utilice archivos de text / editores de texto

2. Indente su código

3. Limite el ancho se su código (80 columnas?)

4. Limite el tamaño de sus funciones individuales

Programación funcional - Clousures

Otro tema importante tema en programación es la creación de *Clouseres*, que son funciones escritas por funciones. *Clousures* reciben su nombre debido a que ***encierra*** el entorno de la función padre y puede acceder a todas sus variables. Esto es útil porque nos permite tener dos niveles de parámetros: un nivel de la padre que controla el funcionamiento y un nivel hijo que hace el trabajo.

Programación funcional - Clousures

En el siguiente ejemplo se utiliza esta idea para generar una familia de funciones de potencia en el que una función padre (power()) crea dos funciones secundarias square() and cube()).

```
power <- function(exponent) {
  function(x) {
    x ^ exponent
  }
}
square <- power(2)
cube <- power(3)
```

Programación funcional - Clousures

```
square(4)
```

```
## [1] 16
```

```
cube(4)
```

```
## [1] 64
```

Programación funcional - Clousures

Cuando imprimes un *Clousure* no se ve nada terriblemente útil

```
square
```

```
## function(x) {
##     x ^ exponent
##   }
## <environment: 0x7fbaa3017cd0>
```

```
cube
```

```
## function(x) {
##     x ^ exponent
##   }
## <environment: 0x7fbaa301a6e8>
```

Progrmación funcional - Clousures

Esto se debe a que la función en sí no cambia. La diferencia es el entorno envolvente , el entorno(square). Una forma de ver el contenido del entorno es convertirlo en una lista :

```
as.list(environment(square))
```

```
## $exponent
## [1] 2
```

```
as.list(environment(cube))
```

```
## $exponent
## [1] 3
```

Progrmación funcional - Clousures

Otra forma de ver que esta sucediendo es mediante el uso de la función `pryr::unenclose()` Esta función sustituye las variables definidas en el medio ambiente que envuelve con sus valores:

```
library(pryr)
unenclose(square)
```

```
## function (x)
## {
##     x^2
## }
## <environment: 0x7fd23440ed18>
```

Progrmación funcional - Clousures

El entorno padre de un cierre es el entorno de ejecución de la función que lo creó, como se muestra por este código:

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
```

```
## <environment: 0x7fd23402cbc0>
```

```
environment(zero)
```

```
## <environment: 0x7fd23402cbc0>
```

Progrmación funcional - Clousures

El entorno de ejecución normalmente desaparece después de que la función devuelve un valor. Sin embargo, las funciones capturan sus entornos de cierre. Esto significa que cuando la función ***a*** retorna una función ***b***, la función ***b*** captura y almacena el entorno de ejecución de la función ***a***, y este no desaparece. (Esto tiene consecuencias importantes para el uso de la memoria.)

Progrmación funcional - Mutable state

Tener las variables en dos niveles le permite mantener el estado entre llamados de funciones. Esto es posible porque, mientras que el entorno de ejecución se actualiza cada vez, el entorno envolvente es constante. La clave para manejar las variables en diferentes niveles es el operador de asignación doble flecha (<<-). A diferencia de la asignación habitual (<-) en la que siempre se asigna en el entorno actual, el operador doble flecha doble sigue mirando hacia arriba en la cadena de entornos de los padres hasta que encuentra un nombre coincidente.

En conjunto, un entorno estático padre y (<<-) hacen posible mantener el estado a través de llamadas a funciones. El siguiente ejemplo muestra un contador que registra el número de veces que la función se ha llamado. Cada vez que `new_counter` se ejecuta, crea un entorno, inicializa el contador `i` en este ambiente, y luego crea una nueva función.

Programación funcional - Mutable state

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

La nueva función es un cierre y su entorno de cerramiento es el entorno creado cuando se ejecuta `new_counter()`. Ordinariamente, los entornos de ejecución de la función son temporales, pero un cierre mantiene acceso al entorno en el que se creó. En el siguiente ejemplo, Los cierres `counter_one()` y `counter_two()` cada obtienen sus propios entornos de cierre cuando se ejecuta, por lo que pueden mantener diferentes conteos.

Programación funcional - Mutable state

```
counter_one <- new_counter()
counter_two <- new_counter()

counter_one()
```

```
## [1] 1
```

```
counter_one()
```

```
## [1] 2
```

```
counter_two()
```

```
## [1] 1
```

Programación funcional - Mutable state

¿Qué pasa si no se utiliza un cierre? ¿Qué ocurre si se utiliza <- en lugar de << -? Haga predicciones sobre lo que sucederá si se reemplaza `new_counter()` con las variantes mencionadas, luego ejecute el código y compruebe sus predicciones.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

Programación funcional - Listas de funciones

En R, las funciones se pueden almacenar en listas. Esto hace que sea más fácil trabajar con grupos de funciones relacionadas, de la misma forma que un `data.frame` hace que sea más fácil trabajar con grupos de vectores relacionados.

Vamos a empezar con un ejemplo sencillo de evaluación comparativa. Imagínese que usted está comparando el desempeño de múltiples formas de calcular la media aritmética. Usted puede hacer esto mediante el almacenamiento de cada método (función) en una lista:

Programación funcional - Listas de funciones

```
compute_mean <- list(

  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x),
  manual = function(x) {
    total <- 0
    n <- length(x)
    for (i in seq_along(x)) {
      total <- total + x[i] / n
    }
    total
  }
)
```

Programación funcional - Listas de funciones

Llamar a una función de una lista es sencillo. Se extraen y luego se llaman:

```
set.seed(12345)
x <- runif(1e5)
system.time(compute_mean$base(x))
```

```
##    user  system elapsed
##   0.004   0.000   0.005
```

Programación funcional - Listas de funciones

```
system.time(compute_mean[[2]](x))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(compute_mean[["manual"]](x))
```

```
##    user  system elapsed
##   0.085   0.005   0.091
```

Programación funcional - Listas de funciones

Para llamar a cada función (por ejemplo, para comprobar que todas ellas devuelven el mismo resultado) utilizamos `lappy`. Vamos a necesitar ya sea una función anónima o una nueva función nombrada, ya que no hay una función incorporada para manejar esta situación.

Programación funcional - Listas de funciones

```
lapply(compute_mean, function(f) f(x))
```

```
## $base
## [1] 0.5005245
##
## $sum
## [1] 0.5005245
##
## $manual
## [1] 0.5005245
```

Programación funcional - Listas de funciones

```
call_fun <- function(f, ...) f(...)
lapply(compute_mean, call_fun, x)
```

```
## $base
## [1] 0.5005245
##
## $sum
## [1] 0.5005245
##
## $manual
## [1] 0.5005245
```

Progrmación funcional - Listas de funciones

El tiempo para cada función, para eso podemos combinar `lapply()` y `system.time ()`:

```
lapply(compute_mean, function(f) system.time(f(x)))
```

```
## $base
##    user  system elapsed
##   0.005   0.001   0.006
##
## $sum
##    user  system elapsed
##       0       0       0
##
## $manual
##    user  system elapsed
##   0.072   0.003   0.076
```

Progrmación funcional - Listas de funciones

Otro uso de listas de funciones es resumir un objeto de múltiples maneras. Para hacer eso, podríamos almacenar cada función de resumen en una lista y a continuación correrlas todas ellas con un `lappy`

```
x <- 1:10
funs <- list(
  sum = sum,
  mean = mean,
  median = median
)
lapply(funs, function(f) f(x))
```

```
## $sum
## [1] 55
##
## $mean
## [1] 5.5
##
## $median
## [1] 5.5
```

/

Progrmación funcional - Listas de funciones

Que pasaría si quisiéramos que nuestras funciones de resumen eliminaran automáticamente los valores faltantes?. Un enfoque sería hacer una lista de funciones anónimas que llaman a funciones de resumen con los argumentos adecuados:

```
funs2 <- list(
  sum = function(x, ...) sum(x, ..., na.rm = TRUE),
  mean = function(x, ...) mean(x, ..., na.rm = TRUE),
  median = function(x, ...) median(x, ..., na.rm = TRUE)
)
lapply(funs2, function(f) f(x))
```

```
## $sum
## [1] 55
##
## $mean
## [1] 5.5
##
## $median
## [1] 5.5
```

Programación funcional - Listas de funciones

Esto, sin embargo, conduce a una gran cantidad de duplicación. Aparte de un nombre de función diferente, cada función es casi idéntica. Un mejor enfoque sería modificar nuestra `lapply()` para incluir un argumento adicional:

```
lapply(funs, function(f) f(x, na.rm = TRUE))
```

```
## $sum
## [1] 55
##
## $mean
## [1] 5.5
##
## $median
## [1] 5.5
```