Functions

Jim Harner

10/4/2020

1.6 Functions

To understand computations in R, two slogans are helpful:
"Everything that exists is an object. Everything that happens is a function call."
— John Chambers

Material in this section is based partially on Hadley's Wickham's Advanced R Functions chapter.

1.6.1 Function Definition

Functions have inputs and produce an output based on the code inside a function. The following code illustrates how a function is defined:

```
a <- 10
f <- function(x) x + a
```

In the function definition of `f`, `x` is a bound variable (to the argument x) whereas `a` is a free variable.

Functions have three elements:

- the body of the function, i.e., the code;
- the list of arguments;
- a map specifying the location of the variables.

```
body(f)
```

```
## x + a
```

```
formals(f)
```

```
## $x
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

In this case `a` is found in the Global environment, which is where the function is defined.

1.6.2 Lexical Scoping

Scoping refers to how R binds values to variables. R supports two types of scoping:

- lexical
- dynamic

By far the most common type is lexical scoping.

Lexical scoping finds the values of variables based on where functions were defined, not on where they are called. We continue the above example to illustrate this.

For the function `f` above, what is the value of `f(2)`?

```
f(2)
```

```
## [1] 12
```

As expected `f(2) = 12`. `f` was defined in the Global environment and looks for free variables there.

Now define the function `g`:

```
g <- function(a) f(2)
```

What is the value of `g(20)`? `g` is called with the variable `a`, so you might think `g(20) = 22`. This is called *dynamic scoping*, i.e., `f` fines the values of the free variable (`a`) in the environment in which it is called. On the other hand, if `f` looks up `a` in the environment in which it is defined, then `g(20) = 12`. This is lexical scoping.

```
g(20)
```

```
## [1] 12
```

As can be seen, `g(20) = 12`, i.e., R uses lexical scoping. Generally, dynamic scoping is used in interactive environments.

The rules for scoping can become complicated as the this simple example illustrates. To understand scoping, it is important to understand environments in R. Hadley Wickham has an in depth discussion in the Advanced R Environments chapter.

1.6.3 Functional Programming

"An object is data with functions. A closure is a function with data."
— John D. Cook

R is a functional programming language, but not in a pure sense. Basically, functions are first class citizens and can be manipulated in much the same way as data structures, i.e., you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

Consider:

```
integrate(function(x) x + a, 0, 2)
```

```
## 22 with absolute error < 2.4e-13
```

Here `function(x) x + a` is an anonymous function since it is not assigned to a variable. `a` is found in the global environment, i.e., we are integrating `x + 10` from 0 to 2.

Anonymous functions are also used to form function closures, functions created by functions. Closures enclose the environment of the parent function and can access its variables. Thus, we have two levels of parameters: a parent level that controls operation and a child level that does the work.

Consider.

```
intercept <- function(a) {
  function(x) x + a
}
```

In this case `intercept` returns a function. We have created a family of intercept functions—one for each `a`.

```
ten <- intercept(10)
ten(2)
```

```
## [1] 12
```

```
twelve <- intercept(12)
twelve(2)
```

```
## [1] 14
```

The complement to a closure is a *functional*, a function that takes a function as an input and returns a vector as output. A *function operator* is a function that takes one (or more) functions as input and returns a function as output. The primary difference is that functionals extract common patterns of loop use, whereas function operators extract common patterns of anonymous function use.

Hadley Wickham has an in depth discussion of closures, functionals, and function operators in the Advanced R Functional Programming chapter. The following example illustrates many of the concepts in functional programming.

1.6.4 Finding the Root of a Differentiable Function

Newton's method for finding a root of a differentiable function f takes a guess x and computes hopefully an improved guess as:

$$x - \frac{f(x)}{Df(x)}$$

where Df denotes the derivative of f.

Create a function called `newton_search` with four arguments: `f`, `df`, `guess`, `conv` (the convergence criterion), where `f` is function of interest and `df` is its derivative.

```
newton_search <- function(f, df, guess, conv=0.001) {
# Note: If f does not have a root, we could be in an infinite loop.
  improve <- function(guess, f, df) {
    guess - f(guess) / df(guess)
  }
  while(abs(f(guess)) > conv) {
    guess <- improve(guess, f, df)
  }
  guess
}
```

We pass `f` and `df` as arguments to `newton_serach`. We define a local functions (or helper function) within `newton_search` to compute the improvement and then test for convergence.

Use this function to find the root of $sin(x)$ near 3 using the actual symbolic derivative. The exact answer is π.

```
newton_search(f = sin, df = cos, guess = 3)
```

```
## [1] 3.142547
```

```
newton_search(f = sin, df = cos, guess = 3, conv = 0.000001)
```

```
## [1] 3.141593
```

For reference: $\pi = 3.1415927$

In general you may not be able to compute the derivative exactly. Use the symmetric difference quotient to

approximate the derivative of f at x numerically by the definition:

$$Df \approx \frac{f(x+h) - f(x-h)}{2h}$$

for small h.

Define a function `make_derivative` with arguments `f` and `h`. The result returned should be a function closure that remembers both `f` and `h`.

```
make_derivative <- function(f, h) {
  function(x) (f(x + h) - f(x - h)) / (2*h)
}
```

Find the root of $sin(x)$ near 3 using numerical derivatives.

```
Dsin <- make_derivative(f = sin, h = .001)
newton_search(f = sin, df = Dsin, guess = 3)
```

```
## [1] 3.142547
```

```
newton_search(f = sin, df = Dsin, guess = 3, conv = 0.000001)
```

```
## [1] 3.141593
```

1.6.5 Finding the Maximum Likelihood Estimator of a Parameter

The log-likelihood of the gamma distribution with scale parameter 1 can be written as:

$$(\alpha - 1)s - n \log \Gamma(\alpha)$$

where α is the shape parameter and $s = \sum \log X_i$ is the sufficient statistic.

Randomly draw a sample of $n = 30$ with a shape parameter of $\alpha = 4.5$. Using `newton_search` and `make_derivative`, find the maximum likelihood estimate of α.

We use the function `rgamma` to draw a random sample from the canonical gamma distribution with shape parameter $\alpha = 3.5$.

```
x <- rgamma(n=30, shape=4.5)
```

We use the moment estimator of α, i.e., $\bar{X}$ as the initial guess. The log-likelihood function is an R functional.

```
gllik <- function() {
  s <- sum(log(x))
  n <- length(x)
  function(a) {
    (a - 1) * s - n * lgamma(a)
  }
}
```

Notice that the functional `gllik` encapsulates `s` and `n` as data.

You must apply `newton_search` to the first and second derivatives (derived numerically using `make_derivative`) of the log-likelihood. The answer should be near 4.5.

```
# The first and second derivatives of the likelihood:
dgllik <- make_derivative(gllik(), 0.001)
ddgllik <- make_derivative(dgllik, 0.001)

# The mle
newton_search(f = dgllik, df = ddgllik, guess = mean(x))
```

```
## [1] 4.690699
```

Note: The moment estimator is 4.5924906