EPIB-613 - Programming and Functions

Today we will cover:

1. The basic idea of creating your own functions in R.
2. Seeing several useful examples of functions in R.

Writing your Own Functions

It is very easy to write your own functions in R. Once these are written, they become permanent functions in your version of R, available just like any other function. Here we will see several examples, first a very trivial example, then several more complex (and more useful).

Of course, it is impossible to teach programming to any level of depth in just a single lecture, and we will not even try. However, we will see how functions are constructed, and a few examples, of increasing complexity.

Adding two numbers

First the easy example, creating a function to add two numbers. Of course, this is a silly example, since R already does this easily, but its simplicity will allow us to focus on how functions are constructed in general, without worrying about more difficult programming lines.

```
> add.2.numbers <- function(a,b)
{
sum = a+b
return(sum)
}
> add.2.numbers(3,4)
[1] 7
> pi
[1] 3.141593
> exp(1)
[1] 2.718282
> add.2.numbers(pi, exp(1))
[1] 5.859874
```

That is all one needs to do to make a function.

Calculating a Bayesian posterior density for a mean

Now for something more complex, we will create a function to carry out Bayesian inferences for a single normal mean, when the variance is assumed known.

The theory is the following (in case it was not covered in this year's version of 607):

Suppose that your data set, $X = (x_1, x_2, \ldots, x_n)$, with sample size n is known to arise from a Normal distribution, $x_i \sim N(\mu, \sigma^2)$, $i = 1, 2, \ldots, n$, i.e., the likelihood function for the data is normal. Suppose also that the variance of the data, σ^2 is a known constant, and so the only unknown parameter to be estimated is the mean. Suppose that the prior distribution for the unknown mean μ is $N(\theta, \tau^2)$.

If we combine the information in the prior distribution with the information in the data set via Bayes Theorem, we derive the posterior distribution, which summarizes current information about the mean μ. It happens that the posterior is also normal, and the exact formula is:

$$\mu_{posterior} \sim N\left(\frac{\frac{\theta}{\tau^2} + \frac{n\overline{x}}{\sigma^2}}{\frac{1}{\tau^2} + \frac{n}{\sigma^2}}, \left[\frac{1}{\tau^2} + \frac{n}{\sigma^2}\right]^{-1}\right) ,$$

where $\overline{x}$ is the mean of the observed data, and $[\cdot]^{-1}$ indicates matrix inversion.

A function that implements this formula in R is:

```
> post.normal.mean <- function(x, prior.mean, prior.var, data.var)
{
####################################################################
#  R function for Bayesian analysis of normal mean, variance known #
#  Parameters included are:                                        #
#                                                                  #
#  Inputs:                                                         #
#                                                                  #
#   x = vector of data                                             #
#   prior.mean = prior mean                                        #
#   prior.var  = prior variance                                    #
#   data.var   = assumed known variance of data                    #
#                                                                  #
#  Outputs:                                                        #
#                                                                  #
#   post.mean = posterior mean                                     #
#   post.var  = posterior variance                                 #
#                                                                  #
####################################################################

n<- length(x)
x.bar <- mean(x)
post.mean.numerator <- prior.mean/prior.var + n*x.bar/data.var
post.mean.denominator <- 1/prior.var + n/data.var
post.mean <-  post.mean.numerator/post.mean.denominator
post.var <- (1/(1/prior.var + n/data.var))

a <- "Post mean = "
b <- "Post Var = "

cat(a, post.mean, ",", b, post.var, "\n")
}
> post.normal.mean(1:10, 0, 1000, 1)
Post mean =  5.49945 , Post Var =  0.09999
```

Another useful way to create output in functions is illustrated below. This format is especially useful if you want to use the program's results later on in your R session.

```
> post.normal.mean <- function(x, prior.mean, prior.var, data.var)
{
####################################################################
#  R function for Bayesian analysis of normal mean, variance known #
#  Parameters included are:                                        #
#                                                                  #
#  Inputs:                                                         #
#                                                                  #
#   x = vector of data                                             #
#   prior.mean = prior mean                                        #
#   prior.var  = prior variance                                    #
#   data.var   = assumed known variance of data                    #
#                                                                  #
#  Outputs:                                                        #
#                                                                  #
#   post.mean = posterior mean                                     #
#   post.var  = posterior variance                                 #
#                                                                  #
####################################################################

n<- length(x)
x.bar <- mean(x)
post.mean.numerator <- prior.mean/prior.var + n*x.bar/data.var
post.mean.denominator <- 1/prior.var + n/data.var
post.mean <-  post.mean.numerator/post.mean.denominator
post.var <- (1/(1/prior.var + n/data.var))

posterior.parameters <- list(post.mean= post.mean, post.var = post.var)
return(posterior.parameters)
}
> post.normal.mean(1:10, 0, 1000, 1)
$post.mean
[1] 5.49945
```

```
$post.var
[1] 0.09999
```

A useful function relating to functions is args, which reminds you of the required arguments of the function. For example:

```
> args(post.normal.mean)
function (x, prior.mean, prior.var, data.var)
```

Adding CIs to linear and logistic regression output

By default, R provides coefficients and p-values in its regression output, but not confidence intervals. However, it is easy to add these in, since R does provide all of the information needed to obtain these confidence intervals.

Unlike simple linear regression of a single variable, we cannot know in advance how many independent (X) variables we will want to use in our function. Therefore, at the beginning, we will discover the number of beta coefficients we need to estimate, and automatically get confidence intervals for all of them.

Here is the function:

```
multiple.regression.with.ci <- function(regress.out, level=0.95)
{
################################################################
#                                                              #
#  This function takes the output from an lm                   #
#  (linear model) command in R and provides not                #
#  only the usual output from the summary command, but         #
#  adds confidence intervals for intercept and slope.          #
#                                                              #
#  This version accommodates multiple regression parameters    #
#                                                              #
################################################################
usual.output <- summary(regress.out)
t.quantile <- qt(1-(1-level)/2, df=regress.out$df)
number.vars <- length(regress.out$coefficients)
temp.store.result <- matrix(rep(NA, number.vars*2), nrow=number.vars)
for(i in 1:number.vars)
{
     temp.store.result[i,] <- summary(regress.out)$coefficients[i] +
      c(-1, 1) * t.quantile * summary(regress.out)$coefficients[i+number.vars]
}
  intercept.ci <- temp.store.result[1,]
  slopes.ci <- temp.store.result[-1,]
  output <- list(regression.table = usual.output, intercept.ci = intercept.ci,
              slopes.ci = slopes.ci)
return(output)
}
```

Now test the function on some data relating flouride concentration to DMF teeth:

```
> dmf = c( 236, 246, 252, 258, 281, 303, 323, 343, 412, 444, 556, 652,
673, 703, 706, 722, 733, 772, 810, 823, 1027)
> flor =  c( 1.9, 2.6, 1.8, 1.2, 1.2, 1.2, 1.3, 0.9, 0.6, 0.5, 0.4, 0.3,
0.0, 0.2, 0.1, 0.0, 0.2, 0.1, 0.0, 0.1, 0.1)
> flor2 <- flor^2
> dmf.data <- data.frame(dmf, flor, flor2)

> attach(dmf.data)  #  For convenience, to make the dmf and flor variables
>                   #  available outside the data.frame
> regression.out <- lm(dmf ~ flor + flor2

> multiple.regression.with.ci(regression.out)
$regression.table

Call:
lm(formula = dmf ~ flor + flor2)

Residuals:
     Min       1Q   Median       3Q      Max
-138.510  -34.566   -1.510   31.240  277.030

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   811.51      31.55  25.721 1.20e-15 ***
flor         -631.85      79.91  -7.907 2.90e-07 ***
flor2         164.48      35.19   4.675 0.000189 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 87.91 on 18 degrees of freedom
```

```
Multiple R-Squared: 0.8816,     Adjusted R-squared: 0.8684
F-statistic:    67 on 2 and 18 DF,  p-value: 4.578e-09

$intercept.ci
[1] 745.2254 877.7952

$slopes.ci
           [,1]      [,2]
[1,] -799.73982 -463.9532
[2,]   90.55367  238.3980
```

You may wish to modify the function to create “prettier” output, but it works.

Here is a similar function for logistic regression data:

```
logistic.regression.or.ci <- function(regress.out, level=0.95)
{
################################################################
#                                                              #
#  This function takes the output from a glm                   #
#  (logistic model) command in R and provides not              #
#  only the usual output from the summary command, but         #
#  adds confidence intervals for all coefficients and OR's.    #
#                                                              #
#  This version accommodates multiple regression parameters    #
#                                                              #
################################################################
usual.output <- summary(regress.out)
z.quantile <- qnorm(1-(1-level)/2)
number.vars <- length(regress.out$coefficients)
OR <- exp(regress.out$coefficients[-1])
temp.store.result <- matrix(rep(NA, number.vars*2), nrow=number.vars)
for(i in 1:number.vars)
{
      temp.store.result[i,] <- summary(regress.out)$coefficients[i] +
      c(-1, 1) * z.quantile * summary(regress.out)$coefficients[i+number.vars]
}
  intercept.ci <- temp.store.result[1,]
  slopes.ci <- temp.store.result[-1,]
  OR.ci <- exp(slopes.ci)
  output <- list(regression.table = usual.output, intercept.ci = intercept.ci,
             slopes.ci = slopes.ci, OR=OR, OR.ci = OR.ci)
return(output)
}
```

Maybe just as easy to use the confint command, but this again shows how functions can work.

Exact Confidence intervals for binomial data

You can calculate binomial parameter confidence intervals using the prop.test command, but this depends on a normal approximation. For small sample sizes or for proportions near zero or one, the approximation may not be very accurate. Here is a (rather complex) function that is exact regardless of sample sizes:

```
ci.prop <- function(x, n, level = 0.95)
{
        # Confidence interval for a single probability
        #
        # Returns the exact level% confidence interval for a single binomial
        # proportion when x successes are observed, out of n trials.
        # Formula was found in Johnson & Kotz, Discrete distributions, pp. 58-60.
        # keywords: confidence_interval binomial proportion
        if(length(n) == 1 && length(x) > 1) n <- rep(n, length(x))
        if(length(level) == 1 && length(x) > 1)
                level <- rep(level, length(x))
        bounds <- matrix(NA, length(x), 2)
        xs <- x
        ns <- n
        levelss <- level
        which.0 <- seq(along = x)[x == 0]
        which.n <- seq(along = x)[x == n]
        which.to.remove <- c(which.0, which.n)
        if(length(which.to.remove) > 0) {
                which.others <- seq(along = x)[ - which.to.remove]
        }
        else {
                which.others <- seq(along = x)
        }
        if(length(which.0) > 0) {
                level.0 <- level[which.0]
                n.0 <- n[which.0]
                bounds[which.0, 1] <- 0
                bounds[which.0, 2] <- 1 - (1 - level.0)^(1/n.0)
        }
        if(length(which.n) > 0) {
                level.n <- level[which.n]
                n.n <- n[which.n]
                bounds[which.n, 2] <- 1
                bounds[which.n, 1] <- (1 - level.n)^(1/n.n)
        }
        if(length(which.others) > 0) {
                x <- x[which.others]
                n <- n[which.others]
                level <- level[which.others]
                v1 <- c(2 * x, 2 * (x + 1))
                v2 <- c(2 * (n - x + 1), 2 * (n - x))
                F.quantiles <- qf(c((1 - level)/2, (1 + level)/2), v1, v2)
                other.bounds <- matrix((v1 * F.quantiles)/(v2 + v1 *
                        F.quantiles), ncol = 2)
                bounds[which.others, 1] <- other.bounds[, 1]
                bounds[which.others, 2] <- other.bounds[, 2]
        }
        results <- matrix(c(xs, ns, xs/ns, levelss, c(bounds)), nrow = length(
                xs))
        dimnames(results) <- list(NULL, c("x", "n", "x/n", "level", "p_L",
                "p_U"))
        results
}
```

Some examples of its use:

```
> prop.test(0,10)

        1-sample proportions test with continuity correction

data:  0 out of 10
95 percent confidence interval:
 0.0000000 0.3445372
sample estimates:
p
0

> ci.prop(0,10)
     x  n x/n level p_L       p_U
[1,] 0 10   0  0.95   0 0.2588656

> prop.test(20, 100)
```

```
        1-sample proportions test with continuity correction

data:  20 out of 100, null probability 0.5
95 percent confidence interval:
 0.1292482 0.2943230
sample estimates:
  p
0.2

> ci.prop(20,100)
      x   n x/n level       p_L       p_U
[1,] 20 100 0.2  0.95 0.1266556 0.2918427
```

So, similar answers for large sample sizes, but quite different for small sample sizes.

This function contains rather complex features, taking care of cases with invalid data, for example, if x is larger than n, which is impossible. At this level, I do not expect you to be able to understand or follow all lines, but include this as a useful function to cut and paste into your version of R, and as an illustration of what one can do.

Conclusion

R is a very powerful package, we have just scratched the surface of what is available. Its developmental environment, a bit of what we have seen today, makes it the first choice of many, if not the majority, of statisticians today.