Lab 3: Optimization

TA: Isabel Laterzo

2/3/2021

Some of today's code has been adapted from Simon Hoellerbauer and Roger D. Peng.

Today we'll tackle optimization. Take a function $f(x)$ - we want to know what value of x will maximize or minimize that function. In other words, let us say that the local maximum (or minimum) of a functtion is called x^*. Then we could say, for maximization, that:

$$x^* = argmax f(x)$$

For minimization, we would say:

$$x^* = argmin f(x)$$

Today, we will learn how to do that with the function `optim()`, particularly as it applies to fitting Normal linear models.

#Newton Raphson for MLE One method of optimization is to employ derivative-based methods. These can be solved analytically or computed numerically. We'll predominantly take the second the approach. A concept that is important, however, is the Newton Raphson algorithm, which allows us to search for any differentiable function's root. We'll apply this in particular to the derivative of functions, in which case the root is the optima.

Assumptions: our function of interest is smooth and has a single minimum.

The basic format for N-R, in its general form, is as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

But since we are dealing with maximum likelihood estimation, we define $f(x_n) = l'(\theta_n)$, where $l(\theta_n)$ is the appropriate log likelihood function. Since we are interested in optimizing, the derivatives in this case are very important. The first derivative maximizes the likelihood. The second derivative helps us adjsut the amount of change needed to reach the root. For more information on this, see these helpful notes. So, we have:

$$\theta_{n+1} = \theta_n - \frac{l'(\theta_n)}{l''(\theta_n)}$$

Where $l''(\theta_n)$ is the Hessian of the log-likelihood function.

I will show this with a simple Poisson model. First, we observe data $x_1, x_2, ..., x_n \sim Poisson(\mu)$ and we want to estimate μ via maximum likliehood. The log-likelihood of the Poisson model is:

$$l(\mu) \sum_{i=1}^{n} x_i log\mu - \mu = n\bar{x} log\mu - n\mu$$

The first derivative of the Poisson log-likelihood (called the score function) is:

$$l^{'}(\mu) = \frac{n\bar{x}}{\mu} - n$$

The second derivative is:

$$l^{''}(\mu) = -\frac{n\bar{x}}{u^2}$$

Thus the Newton iteration becomes:

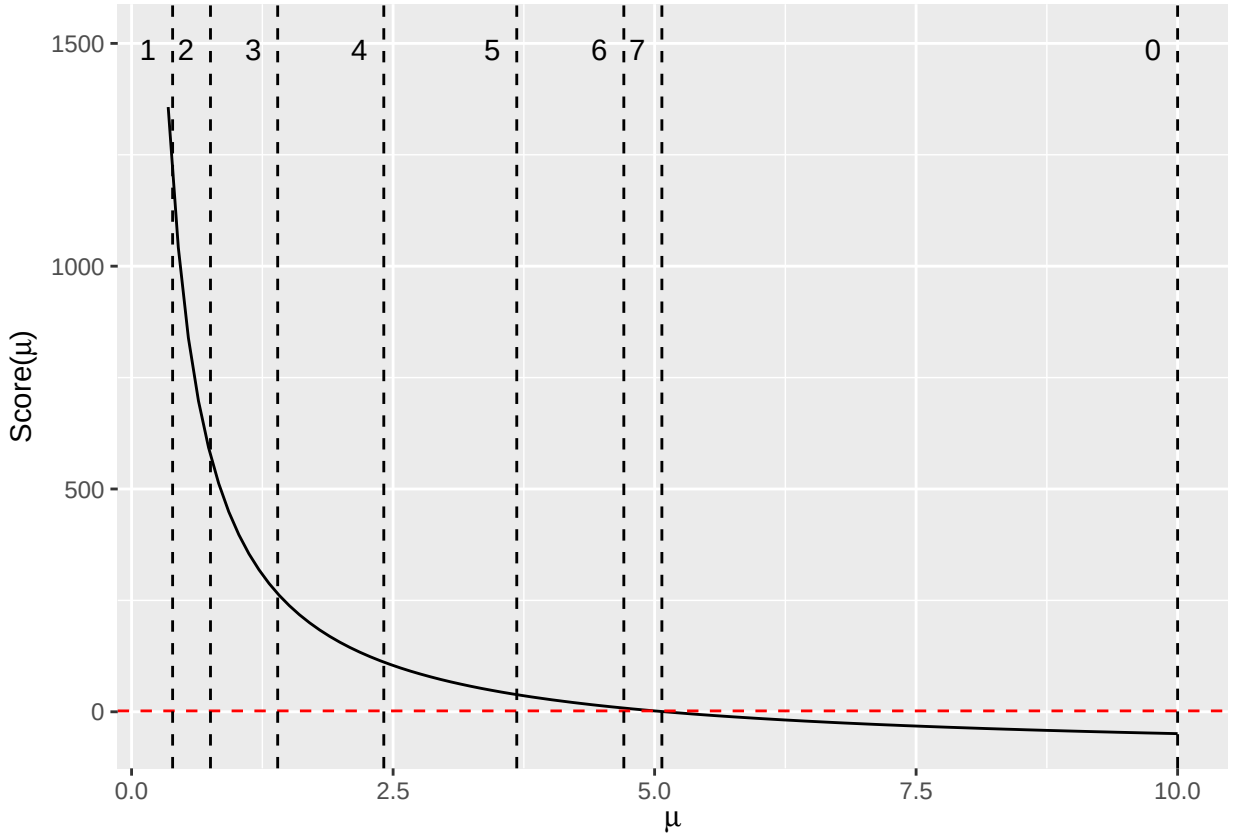
$$\mu_{n+1} = \mu_n - [-\frac{n\bar{x}}{\mu_n^2}]^{-1}(\frac{n\bar{x}}{\mu_n} - n)$$
$$= 2\mu_n - \frac{\mu_n^2}{\bar{x}}$$

You would iterate Newton's method multiple times until it converges and finds the root. Let's visualize this via a function in R that executes the functional iteration of the NR method for however many times we specify the algorithm to run.

Here, we are going to set a starting point for the NR method of $\mu_0 = 10$. We will also set our iterations to 7, and then plot the score function with the values for each of these iterations. Don't worry so much about this code, but look at the plot.

```
#first generate our data
set.seed(2017-08-09)
x <- rpois(100, 5)
xbar <- mean(x)
n <- length(x)

#our score function (first derivative of the log likelihood)
score <- function(mu) {
        n * xbar / mu - n
}


Funcall <- function(f, ...) f(...)
Iterate <- function(f, n = 1) {
        function(x) {
                Reduce(Funcall, rep.int(list(f), n), x, right = TRUE)
        }
}

single_iteration <- function(x) {
        2 * x - x^2 / xbar #our second derivative here
}
g <- function(x0, n) {
        giter <- Iterate(single_iteration, n)
        giter(x0)
}

g <- Vectorize(g, "n") #vectorize our iterator with respect to n
iterates <- g(10, 1:7)

ggplot(data.frame(x=c(seq(.35, 10))), aes(x)) + stat_function(fun = score) +
  ylab(expression(paste("Score(", mu, ")"))) + xlab(expression(paste(mu))) +
```

```
  geom_hline(yintercept = 2, color = "red", linetype = "dashed") +
  annotate(geom = "vline", x = c(10, iterates),
           xintercept = c(10, iterates), linetype = "dashed") +
  annotate(geom = "text", label = c(as.character(c(0:7))),
           x = c(10, iterates), y = c(rep(1510, 8)),
           angle = 0, vjust = 1, hjust = 2)
```



Here we can see that across the 7 iterations, we get closer and closer to the root (which is 5.1). The algorithm continues (for as many iterations as you specify) until the parameter values stabilize, or θ_{n+1} and θ_n become approximately equal, and the root is approximated. If this doesn't make sense to you, watch this video. We can see that by the 7th NR iteration, we are quite close to the root (5.1).This method is good for finding local optima as well, if you have a multimodal case for example. In those cases, you can specify intervals and find the local optimum (minimum or maximum).

The point here is that you understand the intuition, don't worry so much about performing this yourself. Because, as we will see, there's very easy ways to optimze using certain functions in R, namely `optim()`!

#Using `optim()` to maximize the log likelihood

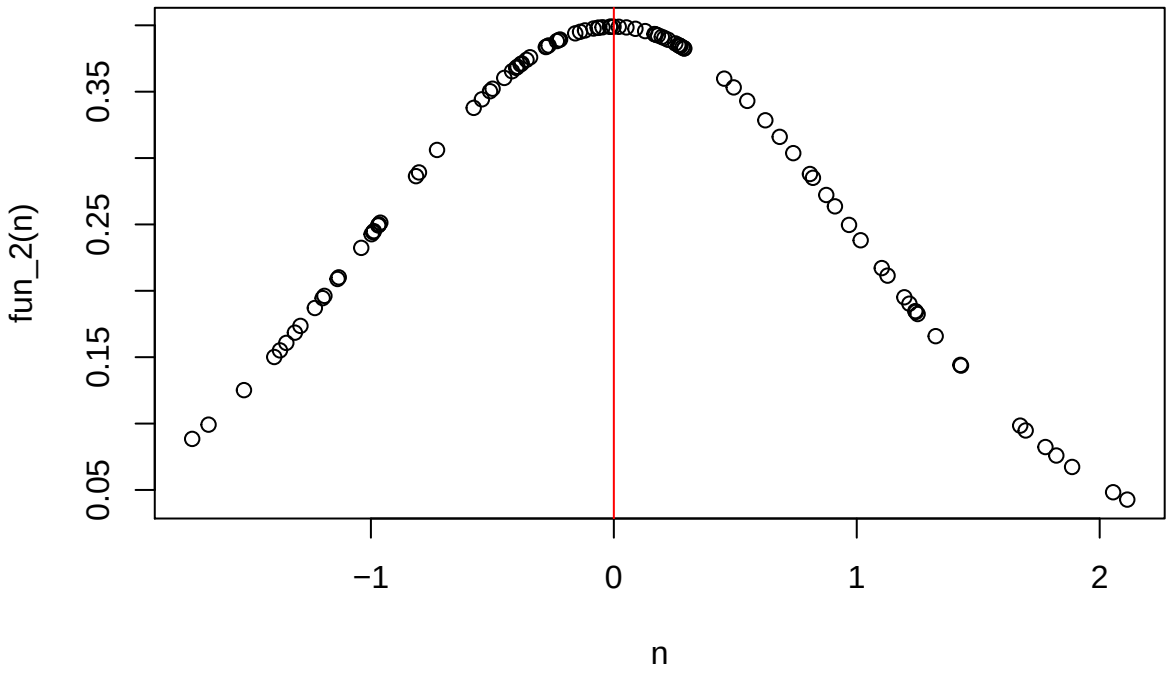
Although it is useful to learn Newton Raphson, today we will mostly be focusing on learning how to use `R` to optimize for us. Specifically, we'll be using the function `optim()`. To see how `optim()` works, let's use it to optimize the mean of a normally distributed random variable. Note that since we are just optimizing a single parameter here, we'll use the cousin function `optimize()`, but the intuition is the same.

```
set.seed(2345) #first set our seed
n <-rnorm(100) #generate a normal random variable

fun_2 <- function(x) { #create a function
  y <- dnorm(x) #use the density of a normal distribution
```

```
}

plot(n, fun_2(n)) #plot now our "x" against our "y"


#now lets optimize to find the mean of this function!
max <- optimize(fun_2, #call our function
                lower = -4, upper = 4, #set our bounds
                maximum = TRUE) #say we want to maximize

max$maximum #look at that
```

```
## [1] 3.330669e-16
```

```
#plot out
plot(n, fun_2(n)) + abline(v = max$maximum, col = "red")
```




```
## integer(0)
```

Using this with a normal linear model is not that much more complicated. You can manually optimize your linear regressions in R using the `optim()` function. The `lm()` command engages in some optimization itself, but knowing how to use the `optim()` command itself is useful (and generates some slightly different estimates than `lm()`!). Let's walk through it:

First let's generate some data again:

```
set.seed(2345)
X <- cbind(1, rnorm(100)) #what are we doing here? Binding together with a constant

#What are the parameters of the normal distribution?
theta_true <- c(1,1,1)
#first element is Beta0, the second element is Beta1, and the last element is sigma2

#just another way to simulate data
y <- rnorm(100, mean = X%*%theta_true[1:2], sd = theta_true[3])
```

Now, we will write out the function we want to optimize. In this case, we want to optimize our log likelihood function. That is, we want to find the values of our parameters that maximize this log likelihood function.

```
#write our log likelihood function
ols.lf <- function(theta, d, X){
  n <- nrow(X)
  k <- ncol(X)
  beta <- theta[1:k] #first k elements of theta are our betas
  sigma <- exp(theta[k+1]) #K+1 element is our sigma 2, exp makes sure
  #this is positive, we can't have a negative standard dev
  e <- d - X%*%beta #residuals

  logl <- sum(dnorm(e, mean = 0, sigma, log = TRUE)) #using dnorm again
                                                      #but note the differences

  return(-logl) #we do negative because the optim fn automatically minimizes.
}
```

Now we will pass this function into `optim()`. If you look at the `optim()` help file, you'll see there a couple of main arguments:

First, `par` which are initial values for the parameters to be optimzed over. These initial values are actually super important, and setting them close to your optimal values can be helpful. Here we'll generate parameters from the normal distribution, as that is how our data is distributed.

Second, `fn` which is where our function is called.

Third, `hessian` which tells it whether or not we want the Hessian matrix to be returned. We do, this is needed for our standard errors.

Fourth, `method` which tells the function which "gradient" to use. By default, it does not use a gradient, but you can specify quasi-Newton methods such as "BFGS".

```
set.seed(123118)

init_par <- rnorm(3) #values for our parameters, drawn from the normal
                      #just like our data

p <- optim(par = init_par, #our starting parameters

           ols.lf, #the function we want it to optimize

           hessian = TRUE, #return Hessian matrix (need for se)

           method = "BFGS", #calls a quasi-Newton method

           #control - read about this in the help file

           d = y, #passing our above data into the function
           X = X)

#we need hessian = T for the se
coef <- p$par #contains estimated arg mins
coef
```

```
## [1]  0.83995840  1.08003343 -0.07399211
```

```
#the final element is the log of sigma2
rse <- exp(coef[3]) #standard deviation of conditional distribution
rse
```

```
## [1] 0.928679
```

```
se2 <- sqrt(diag(solve(p$hessian))) #se
se2
```

```
## [1] 0.09291987 0.10093209 0.07071178
```

```
vals <- p$value #log likelihood min
vals
```

```
## [1] 134.493
```

```
#did this converge in the min number of iterations?
converge <- p$convergence
converge #yes
```

```
## [1] 0
```

Now that we have our estimates from optimization, let's see how they compare to the same model but called with `lm()`.

```
#checking our answer
m1 <- lm(y ~ X[,2])
m1$coef
```

```
## (Intercept)      X[, 2]
##   0.8399586   1.0800330
```

```
library(broom)
tidy(m1)
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    0.840    0.0939      8.95 2.31e-14
## 2 X[, 2]         1.08     0.102      10.6  6.27e-18
```

```
#how does this compare?
```

In addition, we could do the above process all in one function. Try filling in the function below and optimizing.

```
#Could do this all in one function
lm.optim <- function(par, y, X, fn = ols.lf){

  est <- optim(par = par, fn = fn, hessian = TRUE, d = y, X = X)

  return(data.frame(coefficients = est$par, se = sqrt(diag(solve(est$hessian)))))

}

lm.optim(init_par, y = y, X = X)
```

```
##   coefficients         se
## 1   0.83981399 0.09292916
## 2   1.08007596 0.10094218
## 3  -0.07389217 0.07071885
```

#Practice on your own: Now, on your own, try doing this with `mtcars` data, instead of generated data. Let's say we want to set our y value as `mpg`, $x1$ as `wt`, and $x2$ as `gear`.

```
y2 <- ______
X2 <-cbind(_________________)


#in practice, you can use other ways to select your starting values
#such as a grid search, but we will stick with selecting from a normal distrib
set.seed(2030)
params <- rnorm(_)


#now for the optim() part!
p2 <- optim(par = ____, #our starting parameters

            _____, #the function we want it to optimize

            ______, #return Hessian matrix (need for se)

            _______, #calls a quasi-Newton method


            ______, #passing our above data into the function
            ______)


#we need hessian = T for the se
coef2 <- ____
-----

#the final element is the log of sigma2
rse2 <- ____ #standard deviation of conditional distribution
----

se2_2 <- _____) #se
----

vals <- _____ #log likelihood min
----

#did this converge in the min number of iterations?
converge <- ______
---------


#compare to lm() model
----------
----------
```