# Lab 5: Optimization: Answers

Eric Parajon

At the end of the day, maximum likelihood estimation is an optimization problem. We're looking for the set of inputs that maximizes that likelihood function. (Alternatively, you can swap the sign and treat it as a minimization problem, because the actual numbers are meaningless - only the relationship between them matters.)

Let's say we have 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 $x^*$ (because it's the star on top of the plot of the function). Then we could say, for maximization, that:

$$x^* = argmaxf(x)$$

For minimization, we would say:

$$x^* = argminf(x)$$

Today, we're going to learn how to do that with the function `optim()`, which we will apply to fitting normal linear models. You may be thinking that we already have a function, `lm()`, that does that without requiring any mental effort from you. This is true, but understanding optimization will both a) help you better understand the concepts behind OLS and MLE and b) make you appreciate the R devs who wrote convenient modeling functions so that you no longer have to do this every time you want to fit a model.

## Newton-Raphson for MLE

One method of optimization is to employ derivative-based methods. These can be solved analytically or computed numerically, i.e. iteratively. We'll predominantly take the second approach - computers are very bad at calculating derivatives analytically since this involves manipulation of symbols rather than numbers.

Recall that the root of a function is the values of the inputs when $f(.) = 0$. Also, remember that the maximum of a function is when its derivative is equal to zero. By approximating the root of the first derivative, we can find the maximum of the likelihood function. But why approximate it if we can solve it analytically? In addition to computers struggling to solve derivatives numerically, likelihood functions are not always particularly tractable - the derivative may be very complex.

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

We'll be using the Newton-Raphson algorithm to do so (yes, Isaac Newton). The basic format of the algorithm is:

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

Let's think about what that means. We're stating that given an unknown $x_{n+1}$ but a known $f(x_n)$, we can compute the unknown value using the output of the function for the known value and the first derivative. Why? Think about what that fraction is doing - you are dividing the output of the function at $x_n$ by the derivative of the function at $x_n$. What does the derivative represent?

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. (We use log likelihoods because they are mathematically easier to work with.)

Since we are interested in optimizing, the derivatives in this case are very important. The first derivative tells us the slope of the function at any given point. If you're trying to maximize a function, you're looking for the point at which that derivative becomes zero. Think of climbing a hill - as you approach the top, the slope will remain positive but gradually decrease towards zero near the peak. The same thing applies for the derivative - you're trying to find the inflection point at which the first derivative is zero.

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 role of the first derivative should be simple to understand given the above explanation. The role of the second derivative is a bit different.

Let's suppose we're "hiking" up that mathemetical hill trying to find the maximum, and using the first derivative to decide how big of a "step" we take from our current location. However, the first derivative only gives us information on the steepness of that hill. If all we're paying attention to is gaining elevation, we could take a very circular path up to the top. That's where the second derivative comes in.

The second derivative can be thought of as "the rate of change in the rate of change".Remember that as we approach the maximum, the first derivative should be decreasing. Likewise, the second derivative should be as well. (Remember - what's the second order condition for maximization?) We can use the first derivative to determine how big of a step to take, and the second derivative to tell us what direction is the most direct route to the top. (The second derivative encodes "curvature" information for the function.)

See these helpful notes for more detail. So, we have:

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

Where $l''(\theta_n)$ is known as 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 $\mu$ via maximum likliehood. The log-likelihood of the Poisson model is:

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

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:

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

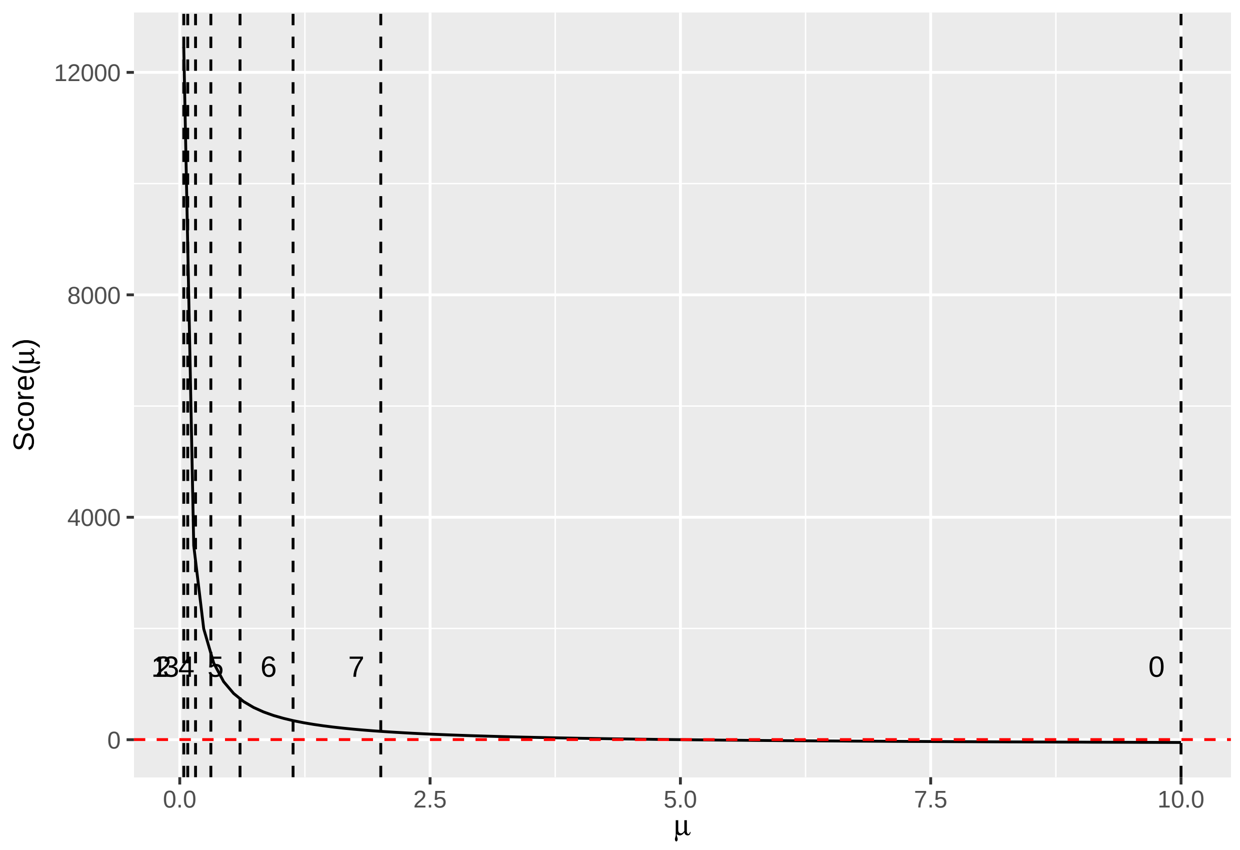
Thus the Newton iteration becomes:

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

You would iterate Newton's method multiple times until it converges and finds the root. In theory, the algorithm would converge perfectly when we find that root - however, we'll never actually reach the root exactly. Why? (Think of Zeno's Paradox.) Thus, we need to either (or both) cap the number of interations or set some convergence criteria. For simplicity's sake, we'll just cap the maximum number of iterations here.

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.

```r
#first generate our data (remember to set the seed)
set.seed(123)
x <- rpois(100, 5) #sample from a poisson
xbar <- mean(x) #mean of the poisson samples
n <- length(x) #n obs

#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 $\theta_{n+1}$ and $\theta_n$ 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

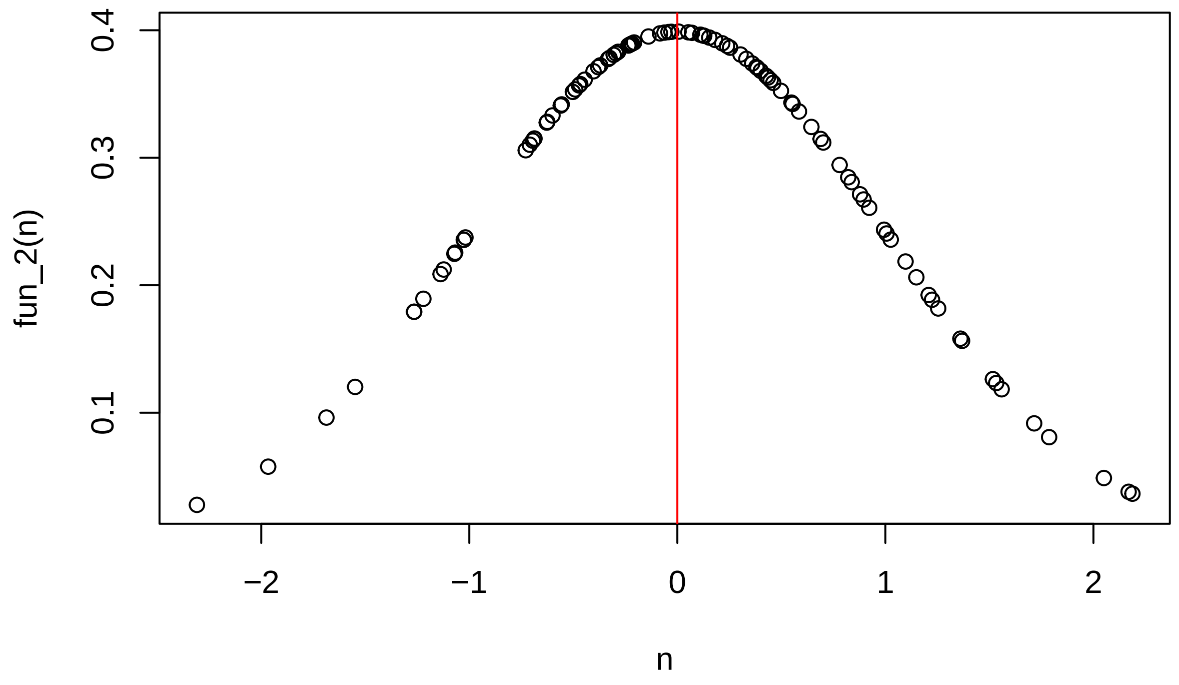
Although it is useful to learn Newton Raphson, today we will mostly be focusing on learning how to use `R` to optimize for us. Specifically, we'll be using the function `optim()`. To see how `optim()` works, let's use it to optimize the mean of a normally distributed random variable. Note that since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

```r
set.seed(123) #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(123)
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) #number of observations
  k <- ncol(X) #number of beta parameters to be estimated
  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
  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 optimized over. You can specify these parameters using, for example, the means of the values, but here we'll just use arbitrary values of '$(1, 1, 1)$'.

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 optimization method to use - by default, it uses Nelder Mead. But, we specify "BFGS" which is a quasi-Newton method, similar to the Newton-Raphson algorithm we use above. Unlike the Newton-Raphson, it uses an approximation to the Hessian - the Hessian is often very difficult to compute and invert.

```
set.seed(123118)

init_par <- rnorm(3) #values for our paramters

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
print(paste('Coefficients:', coef))
```

```
## [1] "Coefficients: 0.89719920308978"    "Coefficients: 0.947524559056595"
## [3] "Coefficients: -0.0398126668528593"
```

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

```
## [1] 0.9609694
```

```
se2 <- sqrt(diag(solve(p$hessian))) #se
print(paste('Standard errors:', se2))
```

```
## [1] "Standard errors: 0.0965718418468904" "Standard errors: 0.105805634167356"
## [3] "Standard errors: 0.0707114341918784"
```

```
vals <- p$value #log likelihood min
print(paste('Value of ll at minimum', vals))
```

```
## [1] "Value of ll at minimum 137.911450739686"
```

```
#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]
##   0.8971969   0.9475284
```

```
library(broom) #sweeps things into dataframes
tidy(m1)
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    0.897    0.0976      9.20 6.69e-15
## 2 X[, 2]         0.948    0.107       8.87 3.50e-14
```

```
#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 = init_par, #our starting parameters
           ols.lf,
           hessian = TRUE,
           method = "BFGS",
           d = y, #passing our above data into the function
           X = X)

  pars <- est$par
  ses <- sqrt(diag(solve(est$hessian)))
  return(data.frame(params = pars, se = ses)) #put the optim vs lm estimates into a dataframe

}

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

```
##        params         se
## 1  0.89719920 0.09657184
## 2  0.94752456 0.10580563
## 3 -0.03981267 0.07071143
```

```
# notice how this doesn't clog up your workspace with objects?
# that's because the function creates an "enclosing environment" that discards
# the temporary objects formed during function execution
# the function returns either the output of its last line OR whatever output
# is specified using return() - using return() explicitly is good practice
```

## 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 <- mtcars$mpg
X2 <-cbind(1, mtcars$wt, mtcars$gear)

#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(4)


p2 <- optim(par = params, #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


           d = y2, #passing our above data into the function
           X = X2)


#we need hessian = T for the se
coef2 <- p2$par #contains estimated arg mins
coef2

#the final element is the log of sigma2
rse2 <- exp(coef2[4]) #standard deviation of conditional distribution
rse2

se2_2 <- sqrt(diag(solve(p2$hessian))) #se
```

```r
se2_2

vals2 <- p2$value #log likelihood min
vals2

#did this converge in the min number of iterations?
converge2 <- p2$convergence
converge2 #yes


#compare to lm() model
m2 <- lm(mpg ~ wt + gear, data = mtcars)
summary(m2)
```