Lab 5: Introduction to Maximum Likelihood

EDP 380C.28: Simulation in R

Fall 2022 – Unique: 12104

Brian T. Keller, Ph.D.
bk@utexas.edu
University of Texas at Austin

October 10, 2022

Objectives

- Learn about maximum likelihood estimation by implementing it for multiple regression.
- Learn about methods used in numerical optimization problems.
- Practice setting up Monte Carlo simulations to evaluate code implementations.

Maximum likelihood estimation is a method that estimates the population parameters of a probability distribution given some observed data. To achieve this goal, we first specify a probabilistic model for the data we have observed and then find the solution to maximize the ***likelihood*** of the data we observed. In this lab, we will investigate using numerical optimization algorithms to find the solutions to maximum likelihood problems.

1 Constructing a Likelihood Function

Let us begin by first constructing the likelihood function for a single variable, Y, normally distributed with a mean and standard deviation. We can write this model using a distributional notation as follows.

$$y \sim \mathcal{N}\left(\,\mu, \sigma\,\right) \tag{1}$$

As a reminder, the '$\sim$' can be read as "distributed as" and $\mathcal{N}\left(\mu, \sigma\right)$ represents a normal distribution with mean $= \mu$ and standard deviation $= \sigma$. To write out the likelihood function for the model in Equation (1), we can leverage the normal distribution's probability density function (PDF). The PDF represents the height of a normal distribution for a given observation with a known mean and variance.

For the single observation, the normal distribution's PDF is given in Equation (2).

$$f\left(\,y \mid \mu, \sigma\,\right) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right) \exp\left\{-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2\right\} \tag{2}$$

Notice how succinctly this falls into the functional notation we have used throughout the course. In addition, we have moved from using an arrow to represent "imply" (i.e., $\Rightarrow$) to an equal sign. This illustrates the link between the functional notation in Equation (2) and the distributional notation in Equation (1). In essence, the functional notation refers to the functional form of the probability density function.

With the functional notation specified for a single observation, we can construct the likelihood function by multiplying across each observation to obtain a one number summary.

$$\mathcal{L}\left(\mu, \sigma \mid y\right) = \prod_{i=1}^{N} f\left(y_i \mid \mu, \sigma\right) \tag{3}$$

As already discussed, the likelihood function conditions or "fixes" the observed data (i.e., appears to the right of the mid bar), and the parameters vary. Thus, Equation (3) provides the relative evidence or support for a chosen parameter value, given the data we observed (y). Finally, as the sample size increases, the likelihood value approaches 0 because we continuously multiply decimal numbers and will run into rounding errors with computers. Therefore, generally, we prefer to work with the logarithm of the likelihood function, known as the log-likelihood function.

$$\begin{aligned}\ell\left(\mu, \sigma \mid y\right) = \ln\left\{\mathcal{L}\left(\mu, \sigma \mid y\right)\right\} &= \ln\left\{\prod_{i=1}^{N} f\left(y_i \mid \mu, \sigma\right)\right\} \\ &= \sum_{i=1}^{N} \ln\left\{f\left(y_i \mid \mu, \sigma\right)\right\}\end{aligned} \tag{4}$$

As Equation (4) illustrates, we can distribute the logarithm and compute the summation of each observation log-likelihood value instead of the product of the likelihood values.

1.1 Creating a Function to Compute the Log-Likelihood in R

The first task is to translate Equation (4) into R. While this task appears complex, we will break down the computation using **functional programming** techniques. Looking at Equation (4), we can decompose the task up into three manageable independent steps:

$$\begin{array}{lcll} (1) & \sum & \rightarrow & \text{Summation} \\ (2) & \ln & \rightarrow & \text{Natural logarithm} \\ (3) & f\left(\ldots\right) & \rightarrow & \text{Computing a density with inputs.} \end{array}$$

After decomposing the equation into its three main parts, we work backward to implement each part via a function. Said differently, first, we must evaluate the density with the inputs, then take the natural logarithm, and finally, sum across all observations.

1.a In R:

Create a function a general function in R to compute the log likelihood of a set of observations given a density function. The function should take on the following form:

```
logL <- function(y, dens)
```

where `y` is the outcome and `dens` will be the general function used to compute the density on `y`.

The next step is to implement the density function to evaluate the normal PDF based on a given mean and standard deviation. In R, the `dnorm()` function will accomplish this for us.

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
```

The default inputs are given above, where '`x`' is a vector of quantities to be evaluated (i.e., $y_1, \ldots, y_N$), '`mean`' is the mean of the distribution (i.e., μ) or vector of means (i.e., for each observation having a different predicted score: $\hat{y}_1, \ldots, \hat{y}_N$), '`sd`' is the standard deviation of the distribution (i.e., σ), and `log` is a logical value. If `log = TRUE` the function returns the natural logarithm of the result. Although there are numerical advantages to using `dnorm` to compute the natural logarithm, we will avoid it to illustrate each step concretely.

1.b In R:

Create a function factory (see Lab 4), that creates a function for evaluating the normal distribution's PDF at a given mean and standard deviation. The function should have the following form:

```
normal <- function(mu, sigma)
```

where `mu` is the mean (μ) and `sigma` is the standard deviation (σ) of the distribution. This factory ought to return a function that will evaluate a value (or vector of values) at the normal distribution specified by μ and σ.

1.2 Brute Force Solving the Mean and Variance

Now that we have the general setup for constructing and computing log-likelihoods, we can use it to find the maximum likelihood solution for the two parameters (μ and σ). As a reminder, what we have done so far is construct a function that characterizes the "fit" of a normal distribution to the data we have observed. The idea is to find parameter values that maximize this function. Currently, our likelihood function can be represented by a three-dimensional space: one dimension is the mean, one dimension is the standard deviation, and one dimension is the actual likelihood value.

Conceptually, we could take two approaches to accomplish this task. One method to solve this problem is to brute force computing the likelihood values across a wide range of potential parameter values to find a solution. With two dimensions, this is still feasible. Still, one could imagine how increasing the number of parameters would significantly increase the complexity (e.g., having ten parameters requires creating all combinations of values across ten dimensions). The second method would be to use Monte Carlo methods we discussed in the previous lab, where we randomly draw values (typically with a uniform distribution) across the ranges and evaluate the potential likelihood value. This approach is more scalable as we increase the dimensions, but it still requires many points to obtain an accurate estimate.

One of the advantages of the normal distribution is that the mean and standard deviation parameters are independent of each other. Intuitively, this makes sense because if we change the distribution's scale, the mean is untouched, and moving the location of the distribution leaves the standard deviation unchanged. Because of this property, we can fix the variance to an arbitrary starting value and solve for the mean first. Once we obtain the solution for the mean, we can then fix the mean to the solution and solve for the mean.

1.c In R:

(1) Set the seed to `12345` and generate 100 observations for Y based on the following distribution.

$$Y \sim \mathcal{N}\,(\,50, 25\,)$$

(2) Using the functions you have already set up, obtain the solutions for the mean and variance by using both the grid search method and the Monte Carlo method. Use a range from 0 to 100 for both the mean and the standard deviation ($10,000$ points each). Make sure to leverage the fact that the mean and standard deviation can be checked separately.

Hint: If you have a vector of all the likelihood computations, use the `which.max()` function to obtain the index of the element that is the maximum.

Answer:

Construct a simulation comparing the grid search method and the Monte Carlo method's estimation of the maximum likelihood estimate. This simulation should consist of 100 replications evaluating the two methods' percent and standardized bias for the mean and standard deviation. Compute the jackknife Monte Carlo standard error for the two evaluation criteria. Present these results in a nicely formatted table in the comments (i.e., parameters as rows; percent bias, percent bias SE, standardized bias, and standardized bias SE as columns).

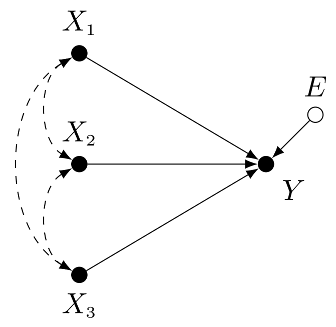

Figure 1: Graph of Data Generating Process for Three Predictors

2 Maximum Likelihood for Linear Regression

Next, we will work on creating optimization algorithms to obtain maximum likelihood estimates for linear regression. The ordinary least squares estimates discussed in Lab 2 are equivalent to the maximum likelihood solution; however, we will obtain these results through a different approach. Consider the data generating model in Figure 1. We will use this data generating model to first create data based on Method 1 from Lab 3. We can write the outcome from Figure 1, using the following functional notation.

$$f\left(\,Y \mid X_1, X_2, X_3\,\right) \;\Rightarrow\; y \sim \mathcal{N}\,(\,\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3, \sigma_e\,) \tag{5}$$

2.a In R:

(1) Look in the directory labeled `scripts` for the files labeled `rmvnorm.R`, `generate_reg.R`, and `ols_regression.R`. In these files, add the `rmvnorm()` function you implemented in Lab 2, the `generate_reg()` function you implemented in Lab 3 (i.e., Method 1), and the `ols_regression` function you implemented in Lab 3, respectively.

Make sure to fill out the documentation based on the skeleton provided in the comments.

(2) Set the seed to `29837`.

(3) Generate data based on Figure 1 with $n = 1000$, with all predictors correlated 0.2 and standard deviations from 1 to 3 and means 3, 5, 12. Set the regression slopes to $\beta_1 = 1$, $\beta_2 = -1$, $\beta_3 = 1$, the $R^2 = 0.35$, and $\mu_y = 100$.

Answer:

Use the `ols_regression()` function to estimate the regression model. Include the resulting output as a comment in the code.

By specifying the analysis model with Equation (5), we can easily map the linear regression model onto the two-parameter mean and variance model from the previous section. That is, the mean is equal to the predicted score, and the standard deviation is equal to the residual standard deviation; therefore, we get the following function by substituting those expressions in Equation (3).

$$f\left(y \mid x_1, x_2, x_3, \boldsymbol{\beta}, \sigma_e\right) = \left(\frac{1}{\sigma_e\sqrt{2\pi}}\right)\exp\left\{-\frac{1}{2}\left[\frac{y - (\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3)}{\sigma_e}\right]^2\right\} \tag{6}$$

While the equation may seem more complicated with the extra terms, it maps conceptually onto everything we have already covered. To reiterate, we are simply substituting the mean (i.e., μ) with the predicted score and the standard deviation (i.e., σ) with the residual standard deviation of the regression model. Just like before, we can construct the likelihood function by multiplying across each observation:

$$\mathcal{L}\left(\boldsymbol{\beta}, \sigma_e \mid y, x_1, x_2, x_3\right) = \prod_{i=1}^{N} f\left(y \mid x_1, x_2, x_3, \boldsymbol{\beta}, \sigma_e\right) \tag{7}$$

and the log likelihood by taking the natural logarithm.

$$\begin{aligned}\ell\left(\boldsymbol{\beta}, \sigma_e \mid y, x_1, x_2, x_3\right) &= \ln\left\{\mathcal{L}\left(\boldsymbol{\beta}, \sigma_e \mid y, x_1, x_2, x_3\right)\right\} \\ &= \sum_{i=1}^{N} \ln\left\{f\left(y \mid x_1, x_2, x_3, \boldsymbol{\beta}, \sigma_e\right)\right\}\end{aligned} \tag{8}$$

2.1 Gradient Ascent

Gradient ascent starts with some initial guess for the parameters and then takes small steps to slowly "climb" the likelihood function until it reaches the maximal point (i.e., the point that maximizes the likelihood). The iterative nature of gradient ascent leads to a straightforward implementation. At each iteration, we take the previous iteration's value and compute the new parameter value based on a step size. This step size depends on the slope of a tangent line (i.e., first derivative; $\frac{\partial \ell}{\partial \theta}$) and some adjustment factor (i.e., the learning rate; γ). More formally, we can write each update step as follows,

$$\theta^{(t)} = \theta^{(t-1)} + \gamma \frac{\partial \ell}{\partial \theta} \tag{9}$$

where θ is a vector of parameters in the model, γ is the learning rate, and $\frac{\partial \ell}{\partial \theta}$ is a vector of the partial first derivatives (i.e., the gradient or score vector). Returning to the linear regression in Equation (5), we have five parameters: $\theta = \{\beta_0, \beta_1, \beta_2, \beta_3, \sigma_e\}$; therefore, the gradient will be a 5×1 column vector. Without going into the details of the derivation, we can compute the gradient using two matrix equations. One equation is for the vector of regression coefficients (i.e., $\boldsymbol{\beta}$) and one for the residual variance.

$$\begin{aligned} \frac{\partial \ell}{\partial \boldsymbol{\beta}} &= -\sigma_e^{-2} \left(-\mathbf{X}^{\intercal}\mathbf{y} + \mathbf{X}^{\intercal}\mathbf{X}\boldsymbol{\beta}\right) \\ \frac{\partial \ell}{\partial \sigma_e^2} &= -0.5N\sigma_e^{-2} + 0.5\sigma_e^{-4} \left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right)^{\intercal} \left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right) \end{aligned} \tag{10}$$

It is important to note that the $\mathbf{X}$ matrix is the design matrix discussed in Lab 3 and includes a column of ones to represent the intercept in the model. With the gradients computed, we perform one iteration of Equation (9) to make the next update of the parameters and then recompute the log-likelihood value based on the new parameters.

2.b In R:

(1) Create a function that implements Equation (10). The function should take on the following form.

```
gradient <- function(p, Y, X)
```

The `gradient()` function requires three inputs: `p` is the list of parameters, `Y` a matrix of outcomes, and `X` a design matrix of predictors and a vector of ones.

(2) Create a function that implements Equation (9). The function should take on the following form.

```
gradient_ascent_update <- function(p, Y, X, rate)
```

The `gradient_ascent_update()` function requires four inputs: `p` is the list of parameters, `Y` a matrix of outcomes, `X` a design matrix of predictors and a vector of ones, and `rate` is the learning rate used to perform a single update.

The algorithm then continues until some convergence criteria (i.e., a stopping rule) is reached. In general, optimization algorithms generally choose a stopping rule based on two things: (1) has the parameter or log likelihood changed between iterations or (2) has the algorithm reached a max iteration count. Note, the second criteria usually implies that the algorithm failed to converge.

2.c In R:

(1) Construct a function that implements gradient ascent until convergence based on the previously created functions. The function should take on the following form.

```
function(starts, Y, X, rate, max_iter = 1000, tol = 1e-06)
```

The function requires six inputs: `starts` is the list of starting parameters, `Y` a matrix of outcomes, `X` a design matrix of predictors and a vector of ones, `rate` learning rate, `max_iter` is the maximum number of iterations, and `tol` is the tolerance for convergence. The output of the function should be the final parameter estimates as a list and the number of iterations it took to converge.

(2) Using `lapply` test five different learning rates: 1, 0.5, 0.1, 0.001, 0.0001. Compare the results and find the top two performing learning rates (i.e., which had the highest log-likelihoods).

Hint: Seeing the an error `missing value where TRUE/FALSE needed` implies non-convergence because it obtained a negative variance.

Answer:

Based on the top two learning rates, perform a 100 replication simulation evaluating which learning rate performs better upon repeated sampling. Each replication ought to generate a new data set and estimate the model with gradient ascent for each learning rate, and using the `ols_regression` function from previous Labs. Evaluate the resulting parameter estimates using percent bias, standardized bias, and mean square error. Compute the jackknife Monte Carlo standard errors of the evaluation criteria. Include in the code a comment of the table with labeled parameters on the row and labeled evaluation criteria and their standard errors as the columns.

2.2 Newton-Raphson

The Newton-Raphson algorithm for numeric optimization follows a similar form as gradient ascent. We start with some initial starting values and perform iterative updates to compute each new parameter. The Netwon-Raphson method works by using a second order Taylor series to approximate the log likelihood function. Conceptually, the algorithm evaluates this approximation at the current parameter values plus some increment, which we will call u.

$$\ell(\theta + u) \approx \ell(\theta) + \left[\frac{\partial \ell(\theta)\, \ell(\theta)}{\partial \theta}\right] u + 0.5 \left[\frac{\partial^2 \ell(\theta)}{\partial \theta^2}\right] u^2 \tag{11}$$

In essence, this is an equation for the parabola we are using to approximate the curve. To find the maximum of the parabola, we set the derivative of Equation (11) with respect to u to 0.

$$\begin{aligned} 0 &= \frac{\partial}{\partial u}\left(\ell(\theta) + \left[\frac{\partial \ell(\theta)}{\partial \theta}\right] u + 0.5 \left[\frac{\partial^2 \ell(\theta)}{\partial \theta^2}\right] u^2\right) \\ &= \left[\frac{\partial \ell(\theta)}{\partial \theta}\right] + \left[\frac{\partial^2 \ell(\theta)}{\partial \theta^2}\right] u \end{aligned} \tag{12}$$

We then solve for u to obtain what parameter value is at the maximum of the fitted parabola in Equation (11). Note, I drop the (θ) from $\ell(\theta)$ for simplicity.

$$u = -\left(\frac{\partial^2 \ell}{\partial \theta^2}\right)^{-1} \frac{\partial \ell}{\partial \theta} \tag{13}$$

While completely understanding the above derivation isn't necessary for this class, the higher-level take away is that Equation (11) is the equation for a fitted parabola that approximates the curve we are maximizing, and Equation (13) is how much we must increase/decrease the parameter to move to the next maximum of the parabola. Therefore, the update step for one iteration is as follows.

$$\theta^{(t)} = \theta^{(t-1)} - \left(\frac{\partial^2 \ell}{\partial \theta^2}\right)^{-1} \frac{\partial \ell}{\partial \theta} \tag{14}$$

We can write the update step in terms of the Hessian matrix (i.e., matrix of second order partial derivatives) and the score vector (i.e., gradient / vector of first order partial derivatives):

$$\theta^{(t)} = \theta^{(t-1)} - \left[\mathbf{H}(\theta)\right]^{-1} s(\theta) \tag{15}$$

where $\mathbf{H}(\theta)$ is the Hessian matrix and $s(\theta)$ is the score vector. The score vector can simply be computed using the function described in gradient ascent. The Hessian matrix is computed as follows:

$$\mathbf{H}(\boldsymbol{\beta}, \sigma_e) = \begin{bmatrix} \frac{\partial^2 \ell}{\partial \boldsymbol{\beta}^2} & \frac{\partial^2 \ell}{\partial \boldsymbol{\beta} \sigma_e^2} \\ \frac{\partial^2 \ell}{\partial \boldsymbol{\beta} \sigma_e} & \frac{\partial^2 \ell}{\partial \sigma_e^4} \end{bmatrix} \tag{16}$$

where

$$\begin{aligned} \frac{\partial^2 \ell}{\partial \boldsymbol{\beta}^2} &= -\sigma_e^{-2} \mathbf{X}^\intercal \mathbf{X} \\ \frac{\partial^2 \ell}{\partial \sigma_e^4} &= 0.5 N \sigma_e^{-4} - \sigma_e^{-6} \left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right)^\intercal \left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right) \\ \frac{\partial^2 \ell}{\partial \boldsymbol{\beta} \sigma_e^2} &= -\sigma_e^{-4} \left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right)^\intercal \mathbf{X} \end{aligned} \tag{17}$$

2.d In R:

(1) Create a function that implements Equation (16). The function should take on the following form.

```
hessian <- function(p, Y, X)
```

The `hessian()` function requires three inputs: `p` is the list of parameters, `Y` a matrix of outcomes, and `X` a design matrix of predictors and a vector of ones. The function ought to output a square matrix with rows and columns equal to the number of parameters (e.g., 5×5 for our example).

(2) Construct a function that implements the Newton-Raphson method until convergence. The function should take on the following form.

```
function(starts, Y, X, max_iter = 1000, tol = 1e-06)
```

The function requires five inputs: `starts` is the list of starting parameters, `Y` a matrix of outcomes, `X` a design matrix of predictors and a vector of ones, `max_iter` is the maximum number of iterations, and `tol` is the tolerance for convergence. The output of the function should be the final parameter estimates as a list and the number of iterations it took to converge.

Answer:

Perform a 100 replication simulation evaluating the Newton-Raphson implementation. Each replication ought to generate a new data set and estimate the model with the Newton-Raphson method and the `ols_regression` from the previous Lab. Evaluate the resulting parameter estimates using percent bias, standardized bias, and mean square error. Compute the jackknife Monte Carlo standard errors of the evaluation criteria. Include in the code a comment of the table with labeled parameters on the row and labeled evaluation criteria and their standard errors as the columns.

2.3 Fisher Scoring

The final numerical optimization algorithm we will implement is Fisher Scoring. Fisher Scoring is a straightforward extension of the Newton-Raphson algorithm, but using the expected information matrix instead of the Hessian matrix.

$$\theta^{(t)} = \theta^{(t-1)} + \left[\mathcal{I}\left(\theta\right)\right]^{-1} s\left(\theta\right) \tag{18}$$

To compute the expected information for the regression model, we need to multiply the Hessian matrix in equations (16) and (17) by negative one, and then take the expectation. Looking at the individual elements in Equation (17) we get the following after taking the expectation.

$$\begin{aligned}
\mathrm{E}\left(-\frac{\partial^2 \ell}{\partial \boldsymbol{\beta}^2}\right) &= \mathrm{E}\left(\sigma_e^{-2}\mathbf{X}^\intercal\mathbf{X}\right) = \sigma_e^{-2}\mathbf{X}^\intercal\mathbf{X} \\
\mathrm{E}\left(-\frac{\partial^2 \ell}{\partial \sigma_e^4}\right) &= \mathrm{E}\left(0.5N\sigma_e^{-4} + \sigma_e^{-6}\left(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\right)^\intercal\left(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\right)\right) = 0.5N\sigma_e^{-4} \\
\mathrm{E}\left(-\frac{\partial^2 \ell}{\partial \boldsymbol{\beta}\sigma_e^2}\right) &= \mathrm{E}\left(\sigma_e^{-4}\left(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\right)^\intercal\mathbf{X}\right) = 0
\end{aligned} \tag{19}$$

Thus, we can obtain the following information matrix.

$$\mathcal{I}\left(\boldsymbol{\beta}, \sigma_e\right) = \begin{bmatrix} \sigma_e^{-2}\mathbf{X}^{\intercal}\mathbf{X} & 0 \\ 0 & 0.5N\sigma_e^{-4} \end{bmatrix} \tag{20}$$

2.e In R:

(1) Create a function that implements Equation (19). The function should take on the following form.

```
information <- function(p, Y, X)
```

The `information()` function requires three inputs: `p` is the list of parameters, `Y` a matrix of outcomes, and `X` a design matrix of predictors and a vector of ones. The function ought to output a square matrix with rows and columns equal to the number of parameters (e.g., 5×5 for our example).

(2) Construct a function that implements the Fisher Scoring method until convergence. The function should take on the following form.

```
function(starts, Y, X, max_iter = 1000, tol = 1e-06)
```

The function requires five inputs: `starts` is the list of starting parameters, `Y` a matrix of outcomes, `X` a design matrix of predictors and a vector of ones, `max_iter` is the maximum number of iterations, and `tol` is the tolerance for convergence. The output of the function should be the final parameter estimates as a list and the number of iterations it took to converge.

Answer:

Perform a 100 replication simulation evaluating the Fisher Scoring implementation. Each replication ought to generate a new data set and estimate the model with the Fisher Scoring method and the `ols_regression` from the previous Lab. Evaluate the resulting parameter estimates using percent bias, standardized bias, and mean square error. Compute the jackknife Monte Carlo standard errors of the evaluation criteria. Include in the code a comment of the table with labeled parameters on the row and labeled evaluation criteria and their standard errors as the columns.

2.4 Computing Observed Standard Errors

After obtaining maximum likelihood estimates, we can compute the standard errors. One advantage of using the Newton-Raphson algorithm is that the standard errors are straightforward to compute because the update step requires the Hessian matrix. We can calculate the variance/covariance matrix of the parameters by taking the inverse of the Hessian matrix multiplied by -1.

$$\text{Var}\left(\,\theta\,\right) = \left[-\mathbf{H}\left(\theta\right)\right]^{-1} \tag{21}$$

The standard errors are equal to the square root of the diagonal elements (i.e., square root of the parameter variance matrix).

2.f In R:

Create a function that implements Equation (21). The function should take on the following form.

```
comptue_se <- function(p, Y, X)
```

The `comptue_se()` function requires three inputs: `p` is the list of parameters, `Y` a matrix of outcomes, and `X` a design matrix of predictors and a vector of ones. The function ought to output a vector with elements equal to the number of parameters (e.g., 5 for this example).

Answer:

Compute the standard errors for the original data set you generated (i.e., 1.a.3) based on the gradient ascent, Newton-Raphson, and Fisher scoring methods. Compare these standard errors (i.e., compute raw bias) to the standard errors obtained from OLS linear regression. Include the raw bias in a nicely presented table (i.e., labeled parameters as rows, and method's raw bias as columns).