Homework 3

Claire Kelling, Statistics 540

April 5, 2018

Problem 1

Find the MLE for a logistic regression model. For $i = 1, ..., n$

$$Y_i \sim Ber(p_i(\theta)); \text{ where } \theta = (\beta_1, \beta_2)$$

$$p_i(\theta) = exp(\beta_1 X_{1i} + \beta_2 X_{2i})/(1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i}))$$

where each Y_i is a binary response corresponding to predictors X_{1i}, X_{2i}. Write a Newton-Raphson algorithm to find the MLE $(\hat{\beta}_1, \hat{\beta}_2)$. The data for this problem are available online.

If $Y_i \sim Ber(p_i(\theta))$, then the likelihood function $f(y_i, p_i(\theta)) = p_i(\theta)^{y_i}(1 - p_i(\theta))^{1-y_i}$.
Therefore, the log-likelihood function for the full data is as follows:

$$\begin{aligned}
\ell &= log(\prod_{i=1}^{n} p_i(\theta)^{y_i}(1 - p_i(\theta))^{1-y_i}) \\
&= \sum_{i=1}^{n} y_i log(p_i(\theta)) + (1 - y_i) log(1 - p_i(\theta)) \\
&= \sum_{i=1}^{n} log(1 - p_i(\theta)) + y_i log(\frac{p_i(\theta)}{1 - p_i(\theta)})
\end{aligned} \tag{1}$$

We also know that

$$\begin{aligned}
log(\frac{p_i(\theta)}{1 - p_i(\theta)}) &= \beta_1 X_{1i} + \beta_2 X_{2i} \\
p_i(\theta) &= \frac{exp(\beta_1 X_{1i} + \beta_2 X_{2i})}{1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i})} \\
1 - p_i(\theta) &= \frac{1}{1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i})}
\end{aligned} \tag{2}$$

So,

$$\begin{aligned}
\ell &= \sum_{i=1}^{n} log(1 - p_i(\theta)) + y_i log(\frac{p_i(\theta)}{1 - p_i(\theta)}) \\
&= \sum_{i=1}^{n} log(\frac{1}{1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i})}) + y_i(\beta_1 X_{1i} + \beta_2 X_{2i})
\end{aligned} \tag{3}$$

This will be useful for obtaining the MLE's. Now, we would like to find the score function.

$$\nabla \ell(\theta) = \begin{bmatrix} \frac{\partial \ell(\beta)}{\partial \beta_1} \\ \frac{\partial \ell(\beta)}{\partial \beta_2} \end{bmatrix} \tag{4}$$

$$\begin{aligned}
\frac{\partial \ell}{\partial \beta_j} &= \sum_{i=1}^{n} y_i \frac{\partial}{\partial \beta_j}(\beta_1 X_{1i} + \beta_2 X_{2i}) - \frac{\partial}{\partial \beta_j} log(1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i})) \\
\frac{\partial}{\partial \beta_j}(\beta_1 X_{1i} + \beta_2 X_{2i}) &= x_{ij} \\
\frac{\partial}{\partial \beta_j} log(1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i})) &= \frac{\frac{\partial}{\partial \beta_j} exp(\beta_1 X_{1i} + \beta_2 X_{2i})}{1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i})} \\
&= \frac{exp(\beta_1 X_{1i} + \beta_2 X_{2i})}{1 + exp(\beta_1 X_{1i} + \beta_2 X_{2i})} \frac{\partial}{\partial \beta_j}(\beta_1 X_{1i} + \beta_2 X_{2i}) \\
&= p_i(\theta) x_{ij} \\
\Rightarrow \frac{\partial \ell}{\partial \beta_j} &= \sum_{i=1}^{n} (y_i x_{ij} - p_i(\theta) x_{ij}) \\
&= \sum_{i=1}^{n} x_{ij}(y_i - p_i(\theta))
\end{aligned} \tag{5}$$

Now, we would like the second partial derivatives.

$$\begin{aligned}
\frac{\partial^2 \ell}{\partial \beta_j \partial \beta_k} &= \frac{\partial}{\partial \beta_k} \frac{\partial \ell}{\partial \beta_j} \\
&= -\sum_{i=1}^{n} x_{ij} \frac{\partial p_i(\theta)}{\partial \beta_k} \\
(\frac{\partial p_i(\theta)}{\partial \beta_k} &= x_{ik} p_i(\theta)(1 - p_i(\theta))) \\
\Rightarrow \frac{\partial^2 \ell}{\partial \beta_j \partial \beta_k} &= -\sum_{i=1}^{n} x_{ij} x_{ik} p_i(\theta)(1 - p_i(\theta))
\end{aligned} \tag{6}$$

(a) Provide details about how you set up the algorithm, including any analytical quantities you derived. You should also include the stopping criteria, how you obtained initial values, how you tuned s_n (stepsize/learning rate) and other relevant information, e.g. did you run the algorithm using multiple initial values?

The analytical quantities that I derived for this problem, including the score function and second derivative, are included above. Specifically, we have the following equations for the log-likelihood, the score function, and the second derivative.

$$\ell = \sum_{i=1}^{n} y_i log(p_i(\theta)) + (1 - y_i) log(1 - p_i(\theta))$$

$$\frac{\partial \ell}{\partial \beta_j} = \sum_{i=1}^{n} x_{ij}(y_i - p_i(\theta))$$

$$\frac{\partial^2 \ell}{\partial \beta_j \partial \beta_k} = -\sum_{i=1}^{n} x_{ij} x_{ik} p_i(\theta)(1 - p_i(\theta)).$$

We use these three equations in our algorithm, outlined in the pseudo-code in part (b), below.

Stopping Criteria: I include three stopping criteria for my algorithm, and I set a maximum number of iterations that if it does not converge by this point, I need to adjust some of the initial values and/or step.

1. $\{\theta^{(n)}\}$ sequence: I set a cutoff, ϵ_1, for the convergence of $\theta^{(n)}$. I tracked the difference between successive estimates of $\theta^{(n)}$, and if it is less than this ϵ_1, and the other stopping criteria are satisfied, then I stop the algorithm. This is because we would like the value of $|\theta^{(n+1)} - \theta^{(n)}|$ to be close to 0 for convergence.
2. $\{f(\theta^{(n)})\}$ sequence: I set a cutoff, ϵ_2, for the convergence of $f(\theta^{(n)})$. I tracked the difference between successive estimates of $f(\theta^{(n)})$, and if it is less than this ϵ_2, and the other stopping criteria are satisfied, then I stop the algorithm. This is because we would like the value of $|f(\theta^{(n+1)}) - f(\theta^{(n)})|$ to be close to 0 for convergence.
3. $\{f'(\theta^{(n)})\}$ sequence: I set a cutoff, ϵ_3, for the convergence of $f'(\theta^{(n)})$. I tracked the value of successive estimates of $f(\theta^{(n)})$, and if it is less than this ϵ_3, and the other stopping criteria are satisfied, then I stop the algorithm. This is because we would like the value of $|f(\theta^{(n)})|$ to be close to 0 for convergence.

4. maxit = Maximum number of iterations: I set a maximum number of iterations for my algorithm to run, and if the other stopping criterion are not satisfied by this point, then I will change the initial values and/or the step value.

For this algorithm, my values of $(\epsilon_1, \epsilon_2, \epsilon_3, maxit)$ are as follows:

$$(\epsilon_1, \epsilon_2, \epsilon_3, maxit) = (1e-6, 1e-6, 1e-6, 50).$$

Initial Values: I try a couple of different techniques for choosing initial values. I find that the algorithm is pretty flexible to the choice of initial values, and takes a short amount of time to converge regardless of initial values. These are all with a step-size of 1.

- First, I try both values being 0.
- I also try the values being chosen from the glm fitted model.
- Lastly, I also try both values being widely off, at (2,2).

We see in Figure 1 that, regardless of the initial value, my estimate of the coefficients seems to be similar. In the case that I used the glm fitted model starting values, I actually had to lower the ϵ_i values to 1e-20 so that it did not converge in just one iteration. Even in the case where the starting values was (2,2), and the estimates jumped around, it still converged in less than 10 iterations.

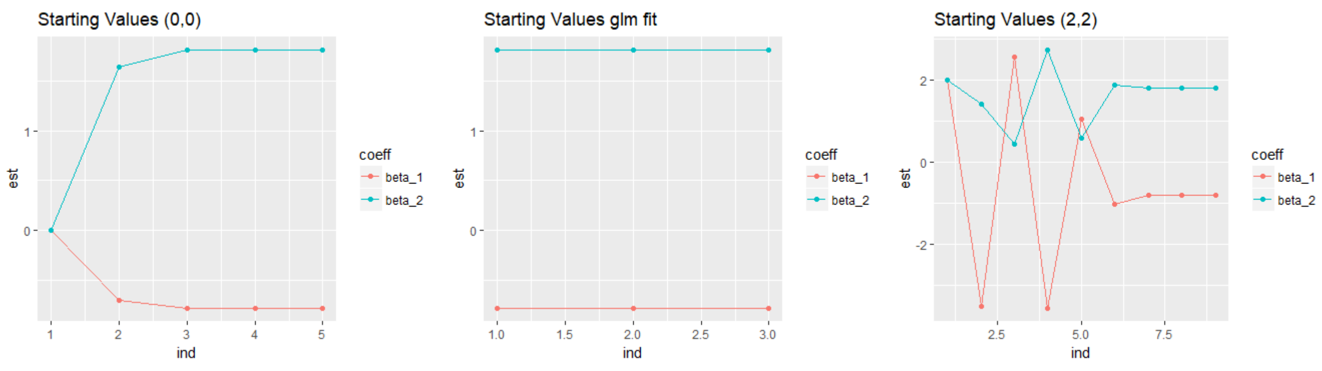

Figure 1: Testing Initial Values for Newton Raphson

I will add the caveat that if our estimates are very high in absolute value, such as (100,100), then we have problems with the evaluation of the log-likelihood. However, if we start close to 0 (as would be expected with a binary response) then we see that the algorithm is not very sensitive to initial values.

Tuning of s_n: I try a couple of different values for tuning of s_n. These values are all with starting values for the coefficients as (0,0).

- First, I try both values being 1.
- I also try both of the values being 0.5.
- Lastly, I vary the values, at (1.5,0.5).

We see in Figure 2 that, despite the value of the step size, the algorithm still converges to the same values. However, the step size does have an impact on the convergence speed. We see that the algorithm converges the fastest at step sizes of (1,1), which is 1 for both coefficients. If we decrease or increase the step size, it will take longer. If we decrease the step size to 0.5, the algorithm still converges rather smoothly, but it takes longer. If we increase the step to 1.5, then the algorithm jumps around a bit more, so the convergence is not as smooth and it once again takes longer. Intuitively, both of these results make sense.

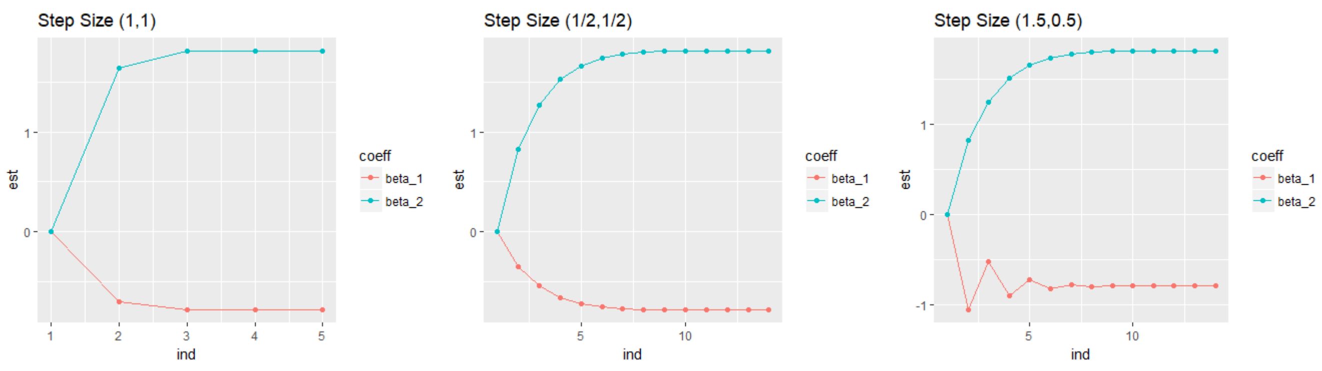

Figure 2: Testing Initial Values for Newton Raphson

(b) Provide your pseudocode for the main algorithm. You can use notation defined in class (and specified above) in order to make the pseudocode succinct and easy to read, e.g. $d\ell_{12}(\beta_1^{(n)}, \beta_2^{(n)})$.

Generally, we have $\theta^{(n+1)} = \theta^{(n)} + sI(\theta)^{-1}S(\theta)$ where

- s = step size
- $I(\theta) = -d^2\ell(\theta)$ = observed fisher information
- $S(\theta) = \nabla\ell(\theta)$ = score function
- We could also replace $I(\theta)$ with $J(\theta)$, the expected Fisher information, where $J(\theta) = -E(\frac{\partial^2}{\partial\theta^2}\ell(\theta))$

I present the pseudo-code for my specific algorithm below.
Pseudocode:

```
1. Propose an initial value of $(\beta_1, \beta_2)$
2. While convergence is not satisfied, by the 3 criteria specified above, and the maximum number of iterations
   has not been reached:
    (a) Calculate the score function, $S(\theta)$
    (b) Calculate the expected fisher information matrix, $I(\theta) = -d^2\ell(\theta)$, as shown in part a)
    (c) Calculate the next estimate for $\theta$, $\theta^{(n+1)} = \theta^{(n)} + sI(\theta)^{-1}S(\theta)$ where s is the step size
    (d) Calculate all of our criteria for convergence: namely, we need to evaluate the log-likelihood and the
        score function at the new estimated value
    (e) Store the history in a data frame
```

(c) Provide the MLE as well as estimate of the standard error, along with asymptotic 95% confidence intervals.

For this problem, I will include my estimates for the MLE and other measures from the iteration that included initial values of (0,0) and step sizes of (1,1) but I will note that the estimates do not change much at all for this algorithm if we change the initial values or step sizes. We calculate the asymptotic 95% Confidence interval with estimate ± 1.96*se(estimate)/$\sqrt{n}$.

	estimate	standard error	asymptotic 95% CI
β_1	-0.79112	0.38380	(-0.844312, -0.737928)
β_2	1.82252	0.45617	(1.759298, 1.885742)

Table 1: Newton-Raphson Estimates

(d) Report the total computing time taken by your algorithm (e.g. you could use system.time). If you have 2-3 different versions of the algorithm, report timings for each version.

Below, in Table 4, we see the times for all of the different versions of the algorithm that I have referenced above. This includes both varying the starting values as well as varying the step-size. We see that this is a very short time for all of the iterations. However, when we alter the step size away from (1,1), it takes longer. Also, the starting values do make a difference, but hardly at all.

starting values	step size	computing time
(0,0)	(1,1)	0.02
glm fit	(1,1)	<0.01
(2,2)	(1,1)	0.04
(0,0)	(0.5,0.5)	0.07
(0,0)	(1.5,0.5)	0.08

Table 2: Computing Time

(e) Provide the computational cost of the algorithm (flops) per iteration as a function of n. Also provide the number of iterations it took before the stopping criteria was met. Of course, the number of iterations will vary by initial conditions so you should provide at least 2-3 different counts depending on where you started.

In Table 3, we see the flop counts for all of the different operations involved in the algorithm for Newton-Raphson. We see that it is of order n.

Operation	Details	Flop Count
p.2 = $\frac{exp(X\%*\%beta)}{1+exp(X\%*\%beta)}$	(nx2)*(2x1) matrix, + 4 (division and subtraction for both coeff)	4n + 4
score.2 = t(X) %*% (y - p.2)	(2xn)*(nx1) + n (subtraction)	8n + n
v.2 = diag(as.vector(p.2 * (1 - p.2)))	2 + 2 (subtraction and multiplication)	4
increm = solve(t(X) %*% v.2 %*% X, score.2)	dense$_{2\times n}$ $*$ diagonal$_{n\times n}$ + dense$_{2\times n}$ $*$ dense$_{n\times 2}$ + solve($dense_{2\times 2}$) + dense$_{2\times 2}$ $*$ dense$_{2\times 1}$	2n + 4n + 8/3 + 4
beta.1 = beta.2 + step * increm	2+2 (addition and multiplication)	4
Total		19n + 20.67 (= O(n))

Table 3: Flop count for Newton-Raphson

To note, there are a couple of other calculations when calculating the difference in the estimates and other convergence (stopping) criteria, but this does not impact the order of the flops, which is still of the order n.

In Table 4, we see the number of iterations until convergence for all of the different versions of the Newton-Raphson algorithm. We see it is quite small for all of the versions of the algorithm.

starting values	step size	Iterations until convergence
(0, 0)	(1, 1)	5
glm fit	(1, 1)	3
(2, 2)	(1, 1)	9
(0, 0)	(0.5, 0.5)	14
(0, 0)	(1.5, 0.5)	14

Table 4: Iterations until Convergence

(f) Summarize in 1-2 sentences the sensitivity of the algorithm to the initial value, that is, does the algorithm work well regardless of where you start or does it need to be in a neighborhood of a certain value? (This is not required, but because this is a simple 2-D optimization problem, you could also provide the 2-D log-likelihood surface to obtain insights.)

In summary, we see that this algorithm is not too sensitive to the initial values or the step sizes. No matter what we have tried, as long as the initial values are reasonably close to 0, the algorithm converges quickly. In conclusion, yes, the algorithm does work well regardless of where we start and the step size.

Problem 2

Implement a gradient descent algorithm (work with the negative log likelihood). Answer (a)-(f) for this algorithm.

(a) Provide details about how you set up the algorithm, including any analytical quantities you derived. You should also include the stopping criteria, how you obtained initial values, how you tuned s_n (stepsize/learning rate) and other relevant information, e.g. did you run the algorithm using multiple initial values?

The analytical quantities that I derived for this problem, including the score function, are included above. Specifically, we have the following equations for the negative log-likelihood and the score function (or the gradient).

$$f(\theta) = -\ell = -\sum_{i=1}^{n} y_i log(p_i(\theta)) + (1 - y_i) log(1 - p_i(\theta))$$

$$\nabla f(\theta) = -\frac{\partial \ell}{\partial \beta_j} = -\sum_{i=1}^{n} x_{ij}(y_i - p_i(\theta))$$

We use these two equations in our algorithm, outlined in the pseudo-code in part (b), below.

Stopping Criteria: I include three stopping criteria for my algorithm, and I set a maximum number of iterations that if it does not converge by this point, I need to adjust some of the initial values and/or step. **These are the same stopping criterion that I used in Problem 1, in order to compare most effectively.** For this algorithm, my values of $(\epsilon_1, \epsilon_2, \epsilon_3, maxit)$ are as follows:

$$(\epsilon_1, \epsilon_2, \epsilon_3, maxit) = (1e-6, 1e-6, 1e-6, 10,000).$$

Initial Values: I try a couple of different techniques for choosing initial values. I find that the algorithm is pretty flexible to the choice of initial values, although it takes a longer amount of time to converge if the initial values are not close.

- First, I try both values being 0.
- I also try the values being 2.
- Lastly, I also try the values being close to their estimated MLE (-1,2).

We see in Figure 3 that, regardless of the initial value, my estimate of the coefficients seems to be similar. However, the estimates seem to take a while to converge, regardless of the initial value. This is with the same stopping criteria as with Newton-Raphson. For example, even when the starting values are quite close to the actual values, it still takes 32 iterations to converge, compared to 43 when it is at (0,0). Therefore, I would say this algorithm is fairly inflexible to the starting values, but takes a longer to converge than Newton Raphson.

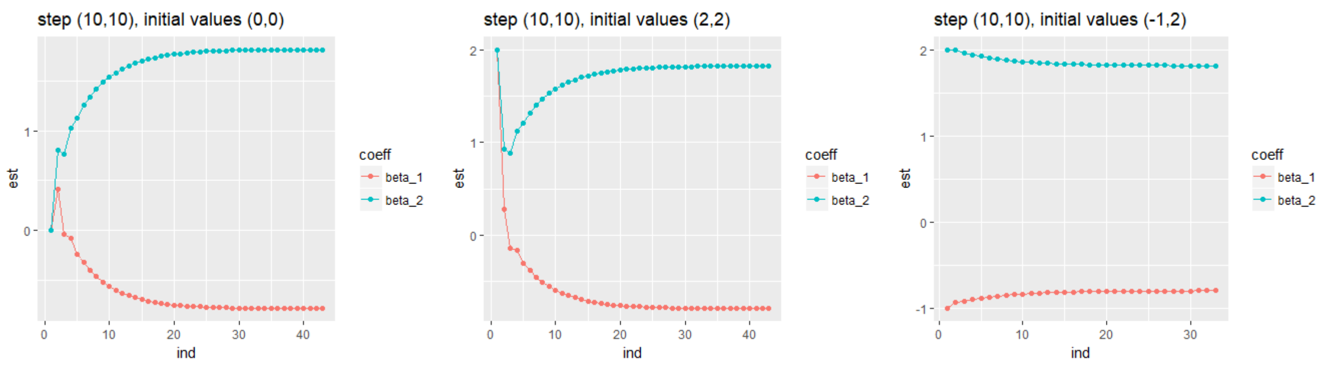

Figure 3: Testing Initial Values for Gradient Descent

Tuning of s_n: I try a couple of different values for tuning of s_n. These values are all with starting values for the coefficients as (0,0).

- First, I try both values being 10.
- I also try both of the values being 15.
- Lastly, I vary the values, at (1,1).

We see in Figure 4 that, despite the value of the step size, the algorithm still converges to the same values. However, the step size does have an impact on the convergence speed. We see that the algorithm converges the fastest at step sizes of (10,10). If we have the step sizes at (1,1), as we did with Newton-Raphson, the algorithm takes quite a while to converge at over 300 iterations. Also, when the step size is (15,15), we see that the estimates jump around a fair bit so it also takes quite a long time to converge.

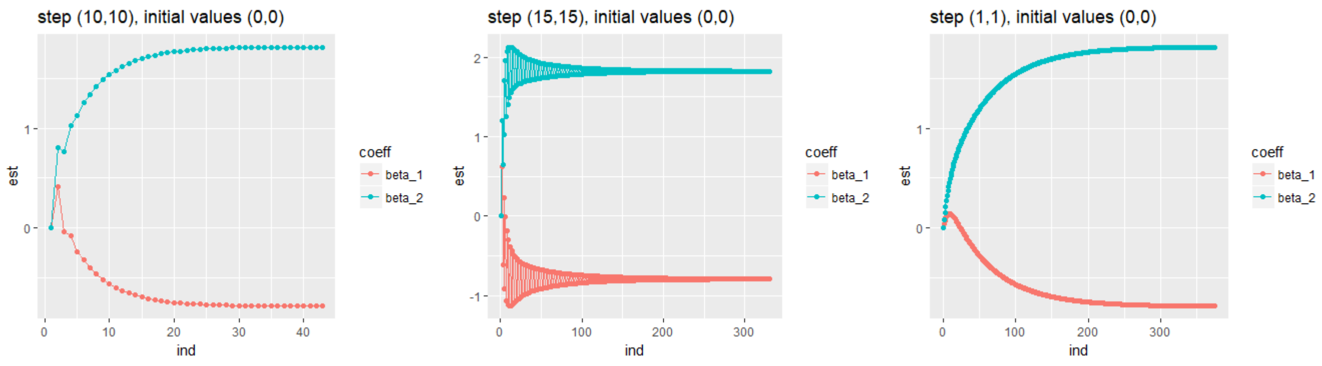

Figure 4: Testing Step Sizes for Gradient Descent

Due to the patterns seen in Figure 4, we will use a step size of 10 for the estimates.

(b) Provide your pseudocode for the main algorithm. You can use notation defined in class (and specified above) in order to make the pseudocode succinct and easy to read, e.g. $d\ell_{12}(\beta_1^{(n)}, \beta_2^{(n)})$.

Goal: Minimize $f(\theta)$, where the idea is that f decreases fastest in the direction of the negative gradient. Generally, We have $\theta^{(n+1)} = \theta^{(n)} - s_n \nabla f(\theta^{(n)}) = \theta^{(n)} - s_n(\sum_{i=1}^m \nabla f_i(\theta))/m$ where

- s_n is the step size/learning rate
- m = number of data points

I present the pseudo-code for my specific algorithm below.
Pseudocode:

```
1. Propose an initial value of $(\beta_1, \beta_2)$
2. While convergence is not satisfied, by the 3 criteria specified above, and the maximum number of iterations
   has not been reached:
    (a) Calculate the score function, $-S(\theta) = \nabla f_i(\theta))$
    (b) Calculate the next estimate for $\theta$, $\theta^{(n+1)} = \theta^{(n)} - s_n(\sum_{i=1}^m \nabla f_i(\theta))/m$ where s is the step size
    (c) Calculate all of our criteria for convergence: namely, we need to evaluate the log-likelihood and the
        score function at the new estimated value
    (d) Store the history in a data frame
```

(c) Provide the MLE as well as estimate of the standard error, along with asymptotic 95% confidence intervals.

For this problem, I will include my estimates for the MLE and other measures from the iteration that included initial values of (-1,2) and step sizes of (10,10) but I note that, once again, the estimates do not change much at all for this algorithm if we change the initial values or step sizes. We calculate the asymptotic 95% Confidence interval with estimate ± 1.96*se(estimate)/$\sqrt{n}$.

	estimate	standard error	asymptotic 95% CI
β_1	-0.79184	0.38382	(-0.8450347, -0.7386453)
β_2	1.82342	0.45622	(1.760191, 1.886649)

Table 5: Gradient Descent Estimates

(d) Report the total computing time taken by your algorithm (e.g. you could use system.time). If you have 2-3 different versions of the algorithm, report timings for each version.

Below, in Table 8, we see the times for all of the different versions of the algorithm that I have referenced above. This includes both varying the starting values as well as varying the step-size. We see that the time is still relatively short for all of the iterations, but the step size definitely makes a difference. If the step size is too short or too long, the computing time is much longer. Therefore, we suggest using a step size of 10 and then the initial values do not matter that much, as long as they are reasonably close to 0.

starting values	step size	computing time
(0,0)	(10, 10)	0.06
glm fit	(10,10)	0.02
(-1,2)	(10,10)	0.06
(2,2)	(10,10)	0.06
(0,0)	(15,15)	0.55
(0,0)	(1,1)	0.61

Table 6: Computing Time

(e) Provide the computational cost of the algorithm (flops) per iteration as a function of n. Also provide the number of iterations it took before the stopping criteria was met. Of course, the number of iterations will vary by initial conditions so you should provide at least 2-3 different counts depending on where you started.

In Table 11, we see the flop counts for all of the different operations involved in the algorithm for Gradient Descent. We see that it is of order n.

Operation	Details	Flop Count
p.2 = $\frac{exp(X\%*\%beta)}{1+exp(X\%*\%beta)}$	(nx2)*(2x1) matrix, + 4 (division and subtraction for both coeff)	4n + 4
score.2 = t(X) %*% (y - p.2)	(2xn)*(nx1) + n (subtraction)	8n + n = 9n
increm <- -score.2/length(y)	2+2 (negation and division)	4
beta.1 = beta.2 + step * increm	2+2 (addition and multiplication)	4
Total		13n + 12 (= O(n))

Table 7: Flop count for Gradient Descent

To note, there are a couple of other calculations when finding the difference in the estimates and other convergence (stopping) criteria, but this does not impact the order of the flops, which is still of the order n.

In Table 8, we see the number of iterations until convergence for all of the different versions of the Gradient Descent algorithm. We see it is quite small for all of the versions of the algorithm. We see that the step size definitely plays a role here, where the initial value does not as much.

starting values	step size	Iterations until convergence
(0, 0)	(10, 10)	43
glm fit	(10, 10)	7
(-1,2)	(10,10)	32
(2, 2)	(10, 10)	42
(0, 0)	(15, 15)	329
(0, 0)	(1, 1)	374

Table 8: Iterations until Convergence

(f) Summarize in 1-2 sentences the sensitivity of the algorithm to the initial value, that is, does the algorithm work well regardless of where you start or does it need to be in a neighborhood of a certain value? (This is not required, but because this is a simple 2-D optimization problem, you could also provide the 2-D log-likelihood surface to obtain insights.)

Through this analysis, we see that this Gradient Descent algorithm is largely unaffected by the initial values. It will still converge to the same values, regardless of the starting values- as long as they are reasonably close to 0. However, we see that this algorithm is sensitive to the step size- if the step size is too small or too large, it takes many iterations to converge. Even using the same step size as the Newton Raphson results in more than 300 iterations until convergence.

Problem 3

Now implement a stochastic gradient descent algorithm and repeat (a)-(f).

(a) Provide details about how you set up the algorithm, including any analytical quantities you derived. You should also include the stopping criteria, how you obtained initial values, how you tuned s_n (stepsize/learning rate) and other relevant information, e.g. did you run the algorithm using multiple initial values?

Similar to Gradient Descent, the analytical quantities that I derived for this problem, including the score function, are included above. Specifically, we have the following equations for the negative log-likelihood and the score function (or the gradient).

$$f(\theta) = -\ell = -\sum_{i=1}^{n} y_i log(p_i(\theta)) + (1 - y_i) log(1 - p_i(\theta))$$

$$\nabla f(\theta) = -\frac{\partial \ell}{\partial \beta_j} = -\sum_{i=1}^{n} x_{ij}(y_i - p_i(\theta))$$

We use these two equations in our algorithm, outlined in the pseudo-code in part (b), below.

Stopping Criteria: I include three stopping criteria for my algorithm, and I set a maximum number of iterations that if it does not converge by this point, I need to adjust some of the initial values and/or step. **These are the same stopping criterion that I used in Problem 1, in order to compare most effectively.** For this algorithm, my values of $(\epsilon_1, \epsilon_2, \epsilon_3, maxit)$ are as follows:

$$(\epsilon_1, \epsilon_2, \epsilon_3, maxit) = (1e-6, 1e-6, 1e-6, 10{,}000).$$

Initial Values: I try a couple of different techniques for choosing initial values. I find that the algorithm is not flexible to the choice of initial values. Each is with a step size of (0.05, 0.05).

- First, I also try the values being (0,0).
- First, I try the values being pretty close to the final values found by the other algorithms (-1, 2).
- Lastly, I also try the values being pretty far away, at (10,10).

We see in Figure 5 that the estimates are not very sensitive to our choice of initial value. Regardless of our initial values, our algorithm converges to around the same final values. However, we see that if we are far away from the initial values, it does increase the number of iterations required to converge.

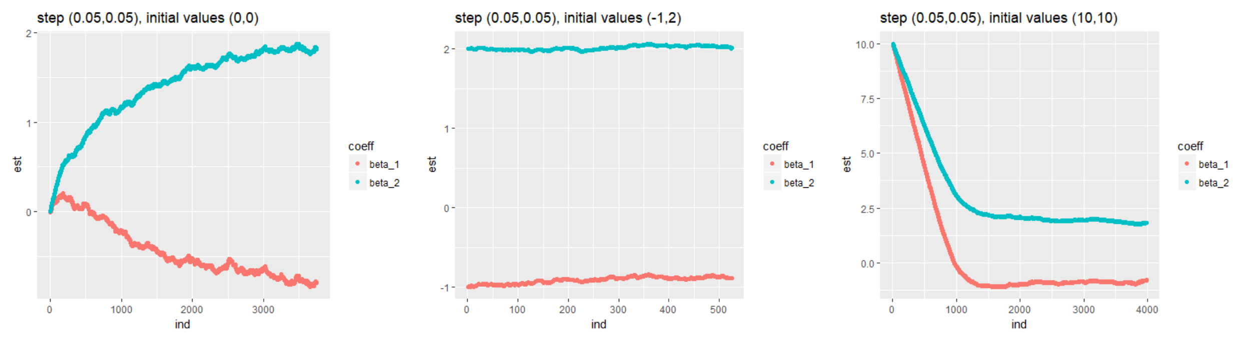

Figure 5: Testing Initial Values for Stochastic Gradient Descent

Tuning of s_n: I try a couple of different values for tuning of s_n. These values are all with starting values for the coefficients as (0,0).

- First, I try both values being 0.05.
- I also try both values being 0.07.

We see in Figure 6 that, with these starting values, the algorithm converges to the same value, but the number of iterations until convergence varies a bit. We recognize that these values of s_n are quite close, but if we try a step size much out of this range, the algorithm does not converge or does not converge quickly. Therefore, we believe that this is an effective step size. However, in future iterations of this algorithm, we would like to consider having a variable step size that decreases as we get closer to the optimum.

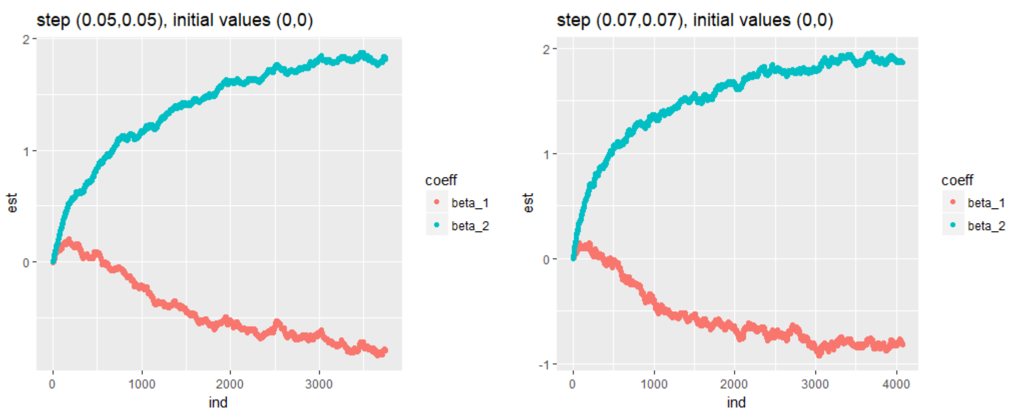

Figure 6: Testing Step Sizes for Stochastic Gradient Descent

Due to the patterns seen in Figure 6, we will use a step size of 0.5 for the estimates and the initial values of (-0.75,1.75).

Tuning of batch size: Lastly, I also try a couple of different values for tuning of the batch size. This is the number of randomly chosen data points that I evaluate the gradient with. If I have just one randomly chosen point, the algorithm is pretty unpredictable and does not converge.

- First, I try the batch size being 10.
- I also try the batch size being 50.

We see in Figure 7 that the algorithm converges a lot more smoothly as we increase the batch size, when we compare to Figure 5. This is because, as we increase the batch size, we are approaching the Gradient Descent, which is discussed in Problem 2. We find that the algorithm is too unpredictable with a small batch size of 1, but we would like to keep the batch size small to create a smaller cost per iteration. Therefore, we proceed with a batch size of 10.

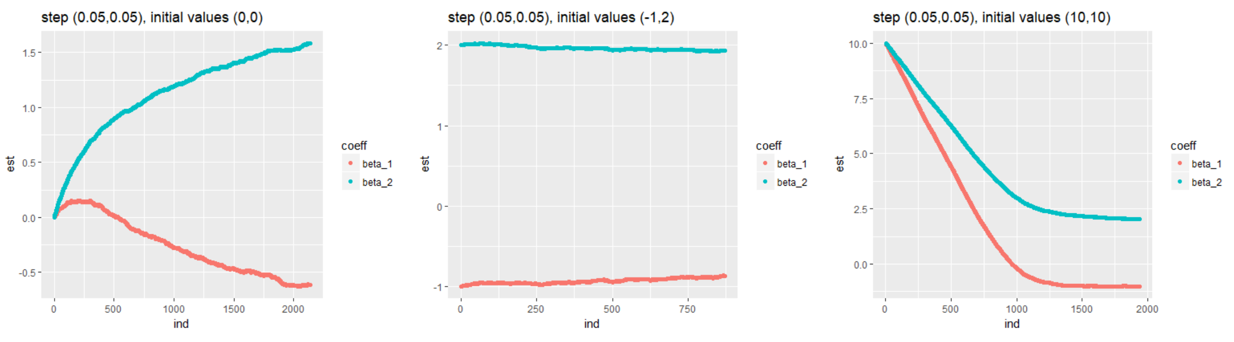

Figure 7: Testing Step Sizes for Stochastic Gradient Descent

(b) Provide your pseudocode for the main algorithm. You can use notation defined in class (and specified above) in order to make the pseudocode succinct and easy to read, e.g. $d\ell_{12}(\beta_1^{(n)}, \beta_2^{(n)})$.

Generally, we have $\theta^{(n+1)} = \theta^{(n)} - s_n \nabla f_{i_m^*}(\theta)$ where

- s_n is the step size/learning rate
- m = number of data points
- i_m^* is randomly sampled from (1,...,m), and perhaps consists of a "batch" or set of points

I present the pseudo-code for my specific algorithm below.

Pseudocode:

```
1. Propose an initial value of $(\beta_1, \beta_2)$
2. While convergence is not satisfied, by the 3 criteria specified above, and the maximum number of iterations
   has not been reached:
   (a) Chose a random set of indices, $i_m^*$ from (1,...,m), of the size of the batch (b), so that you have
       $i_m^* = (i_{m_1}^*, ..., i_{m_b}^*)$
   (b) Calculate the score function at this index, $-S(\theta) = \nabla f_{i_m^*}(\theta)$
   (c) Calculate the next estimate for $\theta$, $\theta^{(n+1)} = \theta^{(n)} - s_n \nabla f_{i_m^*}(\theta)$ where $s_n$ is the step size
```

(d) Calculate all of our criteria for convergence: namely, we need to evaluate the log-likelihood and the score function at the new estimated value

(e) Store the history in a data frame

(c) Provide the MLE as well as estimate of the standard error, along with asymptotic 95% confidence intervals.

For this problem, I will include my estimates for the MLE and other measures from the iteration that included initial values of (0,0) and step sizes of (0.05,0.05) and a batch size of 10. I have included these estimates in Table 9. We calculate the asymptotic 95% Confidence interval with estimate ± 1.96*se(estimate)/$\sqrt{n}$.

	estimate	standard error	asymptotic 95% CI
β_1	-0.73528	0.38182	(-0.7881976, -0.6823624)
β_2	1.74785	0.45213	(1.685188, 1.810512)

Table 9: Stochastic Gradient Descent Estimates

(d) Report the total computing time taken by your algorithm (e.g. you could use system.time). If you have 2-3 different versions of the algorithm, report timings for each version.

We see in Table 10 that the algorithm takes a bit longer when the starting values are not close to the MLE's found by the other algorithms, but not more. WE see that when we increase the batch size to 50 instead of 10, this decreases the algorithm's computing time.

batch size	step size	starting values	computing time
10	(0.05, 0.05)	(-1, 2)	0.48
10	(0.05, 0.05)	(0, 0)	3.75
10	(0.05, 0.05)	(10, 10)	3.19
10	(0.07, 0.07)	(0, 0)	3.17
50	(0.05, 0.05)	(-1, 2)	0.52
50	(0.05, 0.05)	(0, 0)	2.00
50	(0.05, 0.05)	(10, 10)	1.31
50	(0.07, 0.07)	(0, 0)	3.05

Table 10: Computing Time

(e) Provide the computational cost of the algorithm (flops) per iteration as a function of n. Also provide the number of iterations it took before the stopping criteria was met. Of course, the number of iterations will vary by initial conditions so you should provide at least 2-3 different counts depending on where you started.

In Table 11, we see the flop counts for all of the different operations involved in the algorithm for Stochastic Gradient Descent. We see that it is of order b, the batch size, which is chosen to be smaller than n.

Operation	Details	Flop Count
p.2 = $\frac{exp(X[j,]\%*\%beta)}{1+exp(X[j,]\%*\%beta)}$	(1x2)*(2x1) matrix, + 4 (division and subtraction for both coeff)	2+4=6
score.2 <- t(X[j,]) * (y[j] - p.2[j])	b+b (subtraction and multiplication)	2b
increm <- -score.2/length(y)	2+2 (negation and division)	4
beta.1 = beta.2 + step * increm	2+2 (addition and multiplication)	4
Total		b+14 (= O(b))

Table 11: Flop count for Stochastic Gradient Descent

To note, there are a couple of other calculations when finding the difference in the estimates and other convergence (stopping) criteria, but this does not impact the order of the flops, which is still of the order b.

In Table 12, we see the number of iterations until convergence for all of the different versions of the Stochastic Gradient Descent algorithm. We see that the starting value does make a difference, as does the batch size. The batch size, in general, seems to have a negative effect on the number of iterations until convergence.

batch size	starting values	step size	Iterations until convergence
10	(0.05, 0.05)	(-1, 2)	524
10	(0.05, 0.05)	(10, 10)	3988
10	(0.05, 0.05)	(0, 0)	3751
10	(0.07, 0.07)	(0, 0)	4072
50	(0.05, 0.05)	(-1, 2)	870
50	(0.05, 0.05)	(0, 0)	2137
50	(0.05, 0.05)	(10, 10)	1938
50	(0.07, 0.07)	(0, 0)	3368

Table 12: Iterations until Convergence

(f) Summarize in 1-2 sentences the sensitivity of the algorithm to the initial value, that is, does the algorithm work well regardless of where you start or does it need to be in a neighborhood of a certain value? (This is not required, but because this is a simple 2-D optimization problem, you could also provide the 2-D log-likelihood surface to obtain insights.)

For Stochastic Gradient Descent, we see that this is algorithm is not very flexible to either the initial starting values are the step size. The estimates can vary pretty widely depending on the value for the step size, though it tends to converge to the same value if we have a good step size. Therefore, out of the three algorithms we have tested, we believe that this is the least flexible algorithm to step sizes. We also see that this is quite a long computing time and the number of iterations is quite large for all combinations.

Problem 4

Compare the three algorithms above: Newton-Raphson, gradient descent, and stochastic gradient descent. Provide a 1-2 sentence summary of which algorithm you would recommend and why. Then provide a more detailed comparison, for example how stable the algorithms are with respect to initial values, how sensitive they are to choice of s_n, a comparison of the total computational cost, and the computational cost per iteration.

Summary: In summary, we see that the Newton-Raphson algorithm is the most flexible to both the initial values and the step size. Regardless of the values we choose, we see that the algorithm converges to the same values and relatively quickly. Next, the Gradient Descent Algorithm is somewhat flexible, as the estimates do not change much despite the initial values or step size, but it takes much longer if the step size is too large or too small. Lastly, the Stochastic Gradient Descent algorithm is the least flexible to the step size, and it will converge with a range of initial values but it takes quite a long time to converge.

I have included more details below to go into a deeper analysis of my preference for these algorithms:

- **Initial Values:** We see that the Newton-Raphson method converges very quickly, no matter what the starting value is. For example, even if the values were (0,0), the algorithm converged in 5 iterations. We notice that the Gradient Descent algorithm always converges to the same values, but not as quickly as the Newton-Raphson method. No matter the initial value, the Gradient Descent algorithm seems to take around the same number of iterations, even if it is relatively close. Lastly, the Stochastic Gradient Descent is not somewhat flexible to the initial values, but it takes many many iterations to converge.
- **Step Size:** This is quite similar to the analysis of the flexibility to the initial values. We see that the Newton-Raphson method converges very quickly, no matter what the step size is. We notice that the Gradient Descent algorithm always converges to the same values, but not as quickly as the Newton-Raphson method. However, the Gradient Descent algorithm seems take longer to converge if the step sizes are not in a certain range. Lastly, the Stochastic Gradient Descent is not flexible to the step sizes, and it takes much longer to converge if the step size is not in a certain range, and sometimes does not converge if the step size is not in an acceptable range.
- **Computational Cost:** When we measure computational cost in terms of system/computation time, we see that the Newton Raphson method converges very quickly, but this is mostly due to the very low number of iterations until convergence. We also see that the Gradient Descent method does not take a long time to converge, no matter the initial values. Even the longer versions still take less than 1 second. Lastly, the Stochastic Gradient Descent takes quite a while depending on the initial values and step sizes.
- **Computational Cost per iteration:** In terms of flops, this is the only metric in which the order of preference is reversed. Stochastic Gradient Descent has the smallest number of flops, with order b chosen to be less than n, because we are choosing a random set of data points. Gradient Descent and Newton Raphson are both of order n, though Gradient Descent has less flops than Newton Raphson. Therefore, the cost per iteration is the lowest with Stochastic Gradient Descent, but this is not leading to effective or consistent estimations.

Note: I recognize that implementing a back-tracking method may have improved my speed of convergence for the Gradient Descent and Stochastic Gradient Descent method. However, I found that for the purposes of this exercise, my algorithms proved the point that Stochastic Gradient Descent is less stable than Gradient Descent and Newton Raphson converges the fastest and is the most flexible to initial values/step sizes.

Problem 5

E-M algorithm: Return to a lightbulb lifetimes problem similar to the problem in a previous homework. Assume lightbulbs made by a particular company are independent and gamma distributed with parameters α, β (parameterization is such that expected value is $\alpha\beta$). Suppose in an experiment m bulbs are switched on at the same time, but are only completely observed up to time τ. Let the lifetimes of these bulbs be $A_1, A_2, ..., A_M$. However, since the bulbs are only observed till time τ, not all these lifetimes will be observed. Now suppose that at time τ, the experimenter observes the number of lightbulbs, W, still working at time τ, and the lifetimes of all lightbulbs that stopped working by τ. For convenience, denote these bulbs that stopped working by time as $A_1, A_2, ..., A_{m-W}$. Hence, the missing information consists of the lifetimes of the lightbulbs still working at time τ, $A_{m-W+1}, ..., A_m$. For a particular experiment, let τ be 200 days and m = 300. The data on the lightbulb lifetimes for the bulbs that stopped working by are here: http://personal.psu.edu/muh10/540/data/bulbsHW3.dat Assume that the remaining bulbs were still working at time τ. Find the MLE for (α, β) using the E-M algorithm. Repeat parts (a)-(f) from above for this algorithm as well.

(a) Provide details about how you set up the algorithm, including any analytical quantities you derived. You should also include the stopping criteria, how you obtained initial values, how you tuned s_n (stepsize/learning rate) and other relevant information, e.g. did you run the algorithm using multiple initial values?

We know that the lightbulb lifetimes are independent and have a Gamma(α, β) distribution, such that $\alpha\beta$ is the mean. If $X \sim Gamma(\alpha, \beta)$, then

$$f_\theta(x) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} exp(\frac{x}{\beta}).$$

$$\text{and } \ell = log(\prod_{i=1}^{n} f_\theta(x_i)) = -n\alpha log(\beta) - nlog(\Gamma(\alpha)) + (\alpha - 1)\sum_{i=1}^{n} log(x_i) - \sum_{i=1}^{n} \frac{x_i}{\beta}$$

First, we have to derive the simple complete data likelihood. The log-likelihood, when accounting for both the W lightbulbs that were unobserved and the m-W lightbulbs that were observed separately with a total of m lightbulbs, is as follows:

$$\begin{aligned}\ell(\theta) = & -m\alpha log(\beta) - mlog(\Gamma(\alpha)) + (\alpha - 1)[\sum_{i=1}^{m-W} log(x_i) + log(W(E[logX|X > \tau, X \sim Gamma(\alpha^k, \beta^k)]))] \\ & -\frac{1}{\beta}[\sum_{i=1}^{m-W} x_i + W(E[X|X > \tau, X \sim Gamma(\alpha^k, \beta^k)])]\end{aligned} \tag{7}$$

This is very similar to the last problem on Homework 2, but instead of being exponentially distributed, the times are Gamma distributed. Also, there is no prior distribution for τ. So, we have that the y_i that are observed have a truncated Gamma distribution, the z_i that are not observed have a shifted gamma distribution. Lastly, we know that W, the total number of lightbulbs that have survived up to time τ, is 101. We know that this is a binomial variable. For this binomial variable, n is 300 (the total number of bulbs), and p is the probability of a lightbulb surviving up to time τ. We know that the probability of surviving up until time τ for a Gamma distribution is simply $p(\tau) = 1 - \frac{\Gamma_x(\alpha)}{\Gamma(\alpha)}$, where $\Gamma_x(\alpha)$ is the incomplete gamma function. So, for the binomial pmf, we have

$$f(W|\theta) \propto p^W(1-p)^{m-W} \text{ where p} = 1 - \frac{\Gamma_x(\alpha)}{\Gamma(\alpha)}.$$

$$\propto (1 - \frac{\Gamma_x(\alpha)}{\Gamma(\alpha)})^W (\frac{\Gamma_x(\alpha)}{\Gamma(\alpha)})^{m-W}$$

I have attempted to implement this version of the EM algorithm, where we have a truncated Gamma, a shifted Gamma, and a Binomial pdf multiplied as our likelihood and we use the same Q function that we use below. However, we decide that this method is not as intuitive and is not as theoretically strong as the argument presented below.

So, for this problem, I have been closely following the structure of the paper *Likelihood inference based on left truncated and right censored data from a gamma distribution* (Balakrishnan and Mitra, 2013). They create an EM algorithm for left-truncated and right-censored data, from a Gamma distribution. We do not have left-truncated data, as we know the lifetimes from time 0. Therefore, our problem seems to be a general form of their invitation. I do not re-derive their proof of the Q function, but rather restate it here.

The authors prove that if the lifetimes, A_i, are iid Gamma(α, β), then we have the following log-likelihood and Q function, which is similar to the complete data log-likelihood presented above.

$$\begin{aligned} Q(\theta|\theta^k) &= -m\alpha log(\beta) - mlog(\Gamma(\alpha)) + (\alpha-1)[\sum_{i=1}^{m-W} log(A_i) + WC_1] - (1/\beta)[\sum_{i=1}^{m-W} A_i + WC_2] \\ &= -m\alpha log(\beta) - mlog(\Gamma(\alpha)) + (\alpha-1)\sum_{i=1}^{m-W} log(A_i) - (1/\beta)\sum_{i=1}^{m-W} A_i + W(\alpha-1)C_1 - (W/\beta)C_2 \end{aligned} \tag{8}$$

where

$$\begin{aligned} C_1 &= E[logX|X>\tau, X \sim Gamma(\alpha^k, \beta^k)] = log(\beta^k) + \frac{1}{\Gamma(\alpha^k, \tau/\beta^k)}[\frac{d}{d\alpha}\Gamma(\alpha, \tau/\beta^k)]_{\alpha=\alpha^k} \\ C_2 &= E[X|X>\tau, X \sim Gamma(\alpha^k, \beta^k)] = \frac{\beta^k\Gamma(\alpha^k+1, \tau/\beta^k)}{\Gamma(\alpha^k, \tau/\beta^k)} \end{aligned} \tag{9}$$

We have that $\Gamma(.,.)$ is the upper incomplete Gamma function defined as $\Gamma(\alpha, x) = \int_x^\infty u^{\alpha-1}e^{-u}du$ for any $\alpha, x > 0$.

We see that C_1 is difficult to compute with the derivative of the incomplete Gamma function. Instead of trying to compute this derivative directly, we do a numerical approximation of this derivative through finite difference:

$$[\frac{d}{da}\Gamma(\alpha, \tau/\beta^k)]_{\alpha=\alpha^k} \approx (1/\Delta)[\Gamma(\alpha+\Delta, \tau/\beta^k) - \Gamma(\alpha, \tau/\beta^k)]$$

Stopping Criteria: My stopping criterion is if the successive estimates of θ are less than a certain threshold, say ϵ. Therefore, we stop the algorithm if $|\theta^{(n+1)} - \theta^{(n)}| < \epsilon$. For our problem, we choose ϵ to be 1e-6.

Initial Values: We choose several different initial values, listed below

I try a couple of different values for tuning of the initial values.

- First, we choose (2,2).
- Next, we choose (100,100).
- Finally, we choose (160,10).

We see in Figure 8 that the algorithm converges to the same values, regardless of the initial values. We also see that it takes about the same time, no matter what my initial values are. In fact, even if I start at 10,10, which is relatively close to the final values, it still takes about 97 iterations to satisfy the stopping criteria.

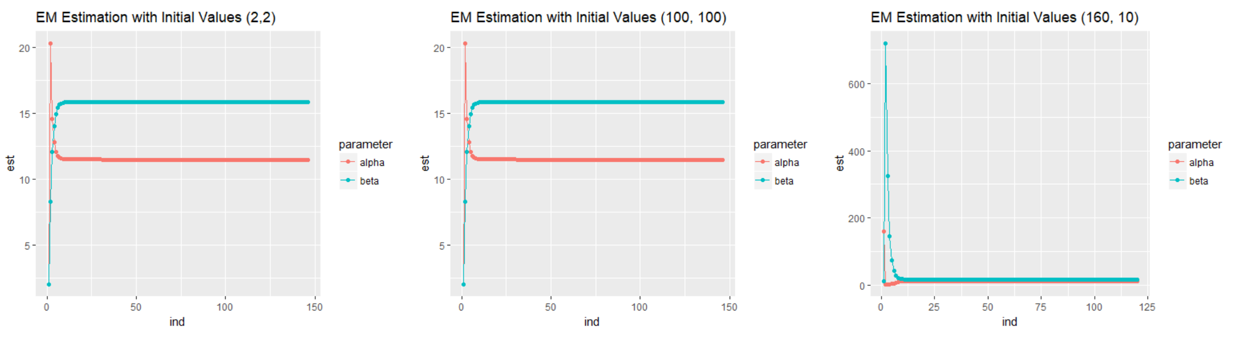

Figure 8: Testing Initial Values for EM

Value for Finite Difference: We also tested what value of Δ to use for the finite difference derivative approximation. We found that the derivative calculation got more and more accurate as we decreased the size of Δ, but there was a point where the finite precision numbers of R limited the accuracy of approximating the derivative. We can see in Figure 9 that with two different starting points of α and β, the convergence of the approximation of the derivative stops being accurate at around the same point- below 1e-8. Therefore, for our problem we choose Δ to be 1e-7 to make sure that our approximation is accurate.

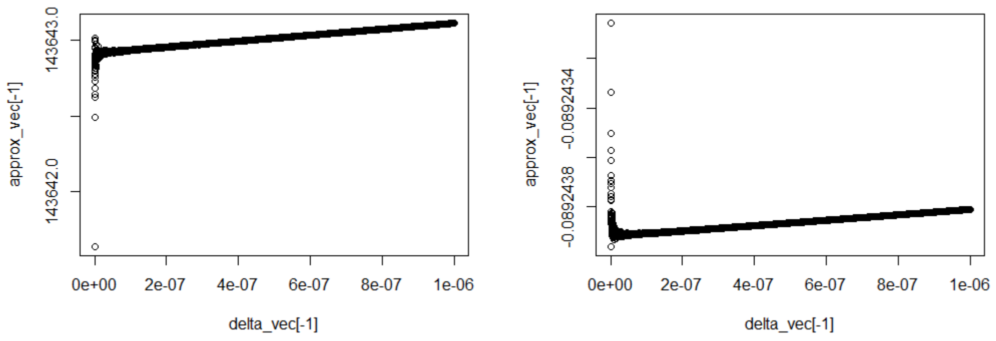

Figure 9: Testing Initial Values for EM

(b) Provide your pseudocode for the main algorithm. You can use notation defined in class (and specified above) in order to make the pseudocode succinct and easy to read, e.g. $d\ell_{12}(\beta_1^{(n)}, \beta_2^{(n)})$.

I present the pseudo-code for my specific algorithm below.
Pseudocode:

```
1. Propose an initial value of $(\alpha, \beta)$, call it $(\alpha^0, \beta^0)$
2. While convergence is not satisfied, by the criterion specified above, and the maximum number of iterations
   has not been reached:
   (a) Calculate $C_1$ and $C_2$ using $\alpha^k$ and $\beta^k$
   (b) Find $\theta^{(k+1)} = argmax_\theta Q(\theta|\theta^{(k)})$
   (c) Store the value in a dataframe and assess the stopping criterion
```

(c) Provide the MLE as well as estimate of the standard error, along with asymptotic 95% confidence intervals.

We provide the estimates for the MLE based on several initial values in Table 13. We see that, regardless of the initial values, the algorithm converges to pretty much the exact same values across all of them, at this number of digits. The estimate of the MLE is extremely flexible to the starting values in this case.

For this part, I will follow the paper by Balakrishnan and Mitra to find the Standard Error of the MLE estimates. They use $I_T(\lambda)$, $I_Y(\lambda)$, and $I_{C|Y}(\lambda)$ to denote the complete information matrix, observed information matrix, and the missing information matrix. They show that $I_Y(\lambda) = I_T(\lambda) - I_{C|Y}(\lambda)$ and that the inverse of $I_Y(\hat{\lambda})$ is the estimated asymptotic variance-covariance matrix of the MLE's. For my purpose, I am only interested in the variance of the MLE's.

So, I compute the following:

$$\begin{aligned} I_T(\lambda) &= -E[\frac{\partial^2}{\partial\lambda^2} logL_c(t;\lambda)] \\ \text{and } -E[\frac{\partial^2}{\partial\beta^2} logL_c(t;\lambda)] &= \frac{m\alpha}{\beta^2} \\ \text{and } -E[\frac{\partial^2}{\partial\alpha^2} logL_c(t;\lambda)] &= m\Psi'(\alpha) \end{aligned} \tag{10}$$

where $\Psi'(\alpha)$ is the trigamma function, which is easily calculated in R.

Next, we have

$$\begin{aligned}
I_{C|Y}(\lambda) &= \sum_{i:\delta_i=0} I^{(i)}_{C|Y}(\lambda) \\
I_{C|Y}(\lambda) &= -E[\frac{\partial^2}{\partial\lambda^2} log f_{C_i}(c_i|C_i > y_i, \lambda)] \\
\text{and } -\frac{\partial^2}{\partial\beta^2} log f_{C_i|Y_i} &= \frac{2c_i}{\beta^3} - \frac{\alpha}{\beta^2} + \frac{e^{-\tau/\beta}\tau^{\alpha}}{\Gamma(\alpha,\tau/\beta)\beta^{\alpha+2}} \times (\frac{\tau}{\beta} - \alpha - 1) - (\frac{e^{-\tau/\beta}\frac{\tau^{\alpha}}{\beta^{\alpha+1}}}{\Gamma(\alpha,\tau/\beta)})^2 \qquad (11) \\
\text{and } -\frac{\partial^2}{\partial\alpha^2} log f_{C_i|Y_i} &= \frac{\frac{\partial^2}{\partial\alpha^2}\Gamma(\alpha,\tau/\beta)}{\Gamma(\alpha,\tau/\beta)} - (\frac{\frac{\partial}{\partial\alpha}\Gamma(\alpha,\tau/\beta)}{\Gamma(\alpha,\tau/\beta)})^2 \\
\text{and } E(C_i|C_i > \tau) &= \frac{\beta\Gamma(\alpha+1,\tau/\beta)}{\Gamma(\alpha,\tau/\beta)}
\end{aligned}$$

We use the values presented in the equation to calculate the standard errors in R. We use finite difference as shown in part a for the approximation of the first derivative of the incomplete gamma function. We use a second order finite difference approximation for the second derivative. This calculation is shown below, and we choose Δ to be 5e-7 after assessing for numerical stability.

$$\frac{\partial^2}{\partial\alpha^2}\Gamma(\alpha,\tau/\beta) \approx \frac{\Gamma(\alpha-\Delta,\tau/\beta) - 2\Gamma(\alpha,\tau/\beta) + \Gamma(\alpha+\Delta,\tau/\beta)}{\Delta^2}$$

The values for the standard error and therefore the asymptotic 95% Confidence interval are approximately the same across all of the starting values, as our estimates for α and β are very similar.

Initial Values	Parameter	estimate	standard error	asymptotic 95% CI
(2,2)	α	11.50831	0.03676996	(11.50413, 11.51245)
	β	15.87788	0.07313391	(15.86964, 15.88619)
(100,100)	α	11.50831	"	(",")
	β	15.87788 "	(",")	
(160,10)	α	11.50823 "	(",")	
	β	15.878 "	(",")	

Table 13: Stochastic Gradient Descent Estimates

(d) Report the total computing time taken by your algorithm (e.g. you could use system.time). If you have 2-3 different versions of the algorithm, report timings for each version.

We provide the starting values and the computing time in Table 16. We see that the algorithm takes a few seconds to converge, regardless of the starting value. In fact, when the starting values are pretty far away at (100,100), the algorithm consistently finds the MLE's faster than when starting at (2,2), even though they do this in the same number of iterations.

starting values	computing time
(2,2)	8.48
(100,100)	1.61
(160,10)	7.36

Table 14: Computing Time

(e) Provide the computational cost of the algorithm (flops) per iteration as a function of n. Also provide the number of iterations it took before the stopping criteria was met. Of course, the number of iterations will vary by initial conditions so you should provide at least 2-3 different counts depending on where you started.

In Table 15, we include all of the flop counts for the various parts of this algorithm. We note that this flop count is of order O(n), similar to the algorithms presented in the first part of the homework (Gradient Descent and NR). We do not know the exact number of flops in the incomplete gamma function, but it is less than order n.

Description	Operation	Flop Count
Calculating C2		
	num = beta*incgam(alpha+1, tau/beta)	3
	denom = incgam(alpha, tau/beta)	
	e_2 = num/denom	1
Calculating C1		
	approx = 1/delta * (incgam(alpha + delta, tau/beta)- incgam(alpha, tau/beta))	6
	e_1 = log(beta) + (1/(incgam(alpha, tau/beta)))*(approx)	4
Q function		
	q_eval = -m*alpha*log(beta) - m*(lgamma(alpha))+	5
	(alpha-1)*(sum(log(data)))- (1/beta)*sum(data)-	2n + 6
	W*(alpha-1)*C1(alpha.k, beta.k, tau) -	4
	(W/beta)*C2(alpha.k,beta.k,tau)	2
Total		2n + 31 + O(1)

Table 15: Flop count for EM Algorithm

To note, there are a couple of other calculations when finding the difference in the estimates and other convergence (stopping) criteria, but this does not impact the order of the flops, which is still of the order n.

In Table 16, we see that the algorithm takes approximately the same number of iterations until convergence, regardless of the initial values. Initial values do not play a large role in determining the number of iterations, and the number of iterations is pretty low.

starting values	Iterations until convergence
(2,2)	145
(100,100)	145
(160,10)	119

Table 16: Iterations until Convergence

(f) Summarize in 1-2 sentences the sensitivity of the algorithm to the initial value, that is, does the algorithm work well regardless of where you start or does it need to be in a neighborhood of a certain value? (This is not required, but because this is a simple 2-D optimization problem, you could also provide the 2-D log-likelihood surface to obtain insights.)

In conclusion, we note that this algorithm is quite flexible to the initial values. No matter what the initial values were, the algorithm converged the same estimates of α and β, even when the initial values were quite large/different from the MLE's. We also notice that the number of iterations is relatively small and consistent across all starting values.

References

Narayanaswamy Balakrishnan and Debanjan Mitra. 2013. Likelihood inference based on left truncated and right censored data from a gamma distribution. IEEE Transactions on reliability 62, 3 (2013), 679–688.