Stat 243 Problem Set 8

A.J. Torre

November 25, 2018

Comments: I worked with/ got help from Andrea, Narae, Huy, and Malvika on this problem set.

Problems

Question 1: Experimenting with Importance Sampling

a)

So, for this problem, we want to use importance sampling to estimate the mean of a truncated t distribution with df= 3 and cut off such that $X < -4$. Wikipedia, though not a scholarly source, gave me some good information about some characteristics of a truncated distribution, such as that the CDF will be $(F(x) - F(a))/(F(b) - F(a))$, where F is the cdf of our t- distribution and $(a, b]$ is our support. We can observe that this CDF is uniform (again looking at the Wikipedia page for truncated distributions); Andrea helped me understand this part.

```
set.seed(3)
values <- runif(10000, 0, pnorm(-4, mean = -4))
# runif (n,min,max) so this will generate values between 0
# and our cdf value given by pnorm pnorm returns the value of
# the integral from negative infinity to -4 of a normal with
# mean -4 and sd=1 as given
```

So, right now, the values that we have are all probabilities (that correspond to the actual values we want) so to get the actual values less than -4, we have to use qnorm to get the inverse quantile transform, which will give us 10,000 numbers of a normal distribution centered at -4 and are less than -4.

```
q_values <- qnorm(values, -4)
# So, in qnorm, values vector is used as the p argument
# (vector of probabilities) and set the mean =-4 to then get
# 10000 values between negative infinity and -4, sampled from
# a normal distribution with mean=-4
```

To get the weights, use dt to get the t density at each point, do separately for our f and g functions, where f is a t distribution while g is a normal with mean -4, sd =1 and for both, we use the q_values we sampled above.

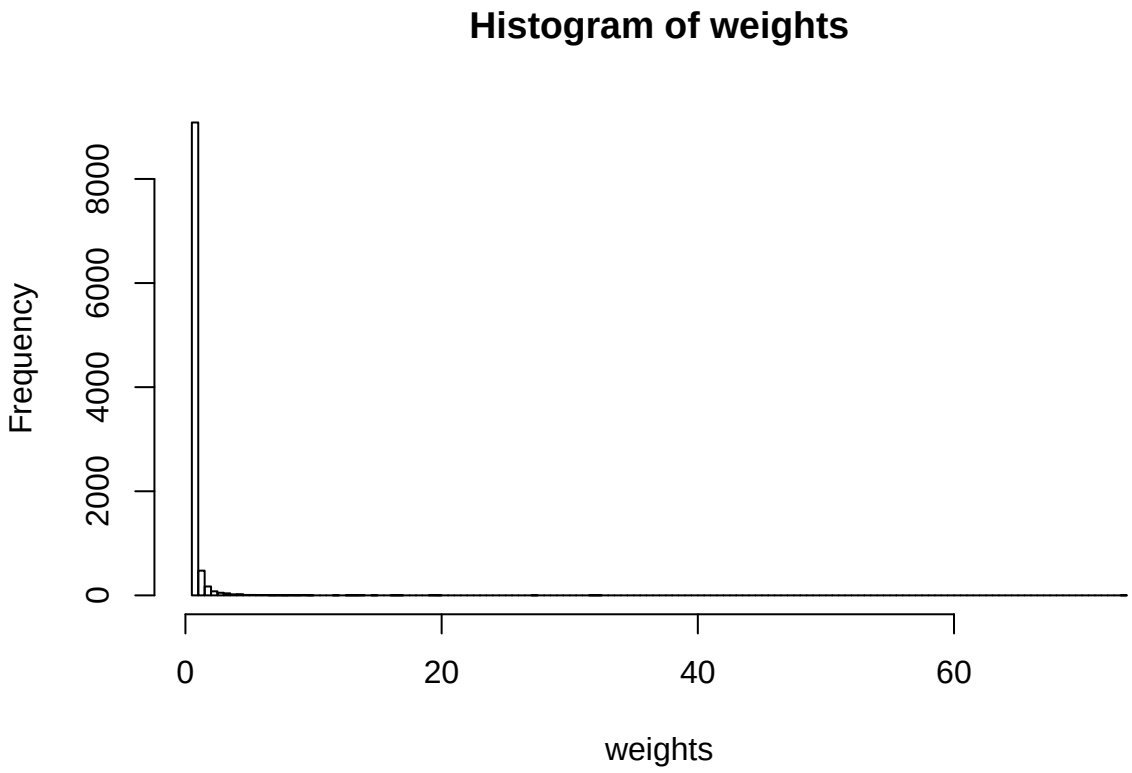
```
density_for_f <- dt(q_values, 3)
density_for_g <- dnorm(q_values, mean = -4, sd = 1)
```

As Narae told me, I should divide these densities by the CDF to normalize it/ make sure it's a valid pdf and will integrate to 1, based on the wikipedia page for truncated distributions.

```
trunc_f <- density_for_f/(pt(q = -4, df = 3))
trunc_g <- density_for_g/(pnorm(-4, mean = -4, sd = 1))
```

So, following from importance sampling, we should do f over g to get our weights. From the histogram, we see that most of our weights are pretty low in number;the majority of the weights are close to 0.

```
weights <- trunc_f/trunc_g
hist(weights, breaks = 200)
```



Lastly, we calculate the mean and variance using formulas in the class notes on mean and variance in importance sampling. We see that our mean is -4.23, which is pretty close to -4, and our variance, found using the variance formula for importance sampling that was given in class, is very small, 0.0073.

```
sample_mean <- mean(weights * q_values)
sample_mean
```

```
## [1] -4.232072
```

```
# following formula from notes on how to get variance of when
# doing importance sampling
sample_var <- var(weights * q_values)/(10000)
sample_var
```

```
## [1] 0.007394466
```

b)

Here, we follow a lot of the same methods and approaches as part a, but we use a t distribution to sample from instead of a normal. So, we will use pt and qt here instead of pnorm and qnorm.

```r
t_values <- runif(10000, 0, pt(-4, ncp = -4, df = 1))
qt_values <- qt(t_values, ncp = -4, df = 1)
# now have 10,000 values sampled from a truncated t
# distribution
```

Then, to get our f and g distributions, we use again a similar approach, but we replace our g function with a t distribution instead, centered at -4 and with 1 degree of freedom.

```r
density_for_ft <- dt(qt_values, df = 3)
density_for_gt <- dt(qt_values, ncp = -4, df = 1)
```

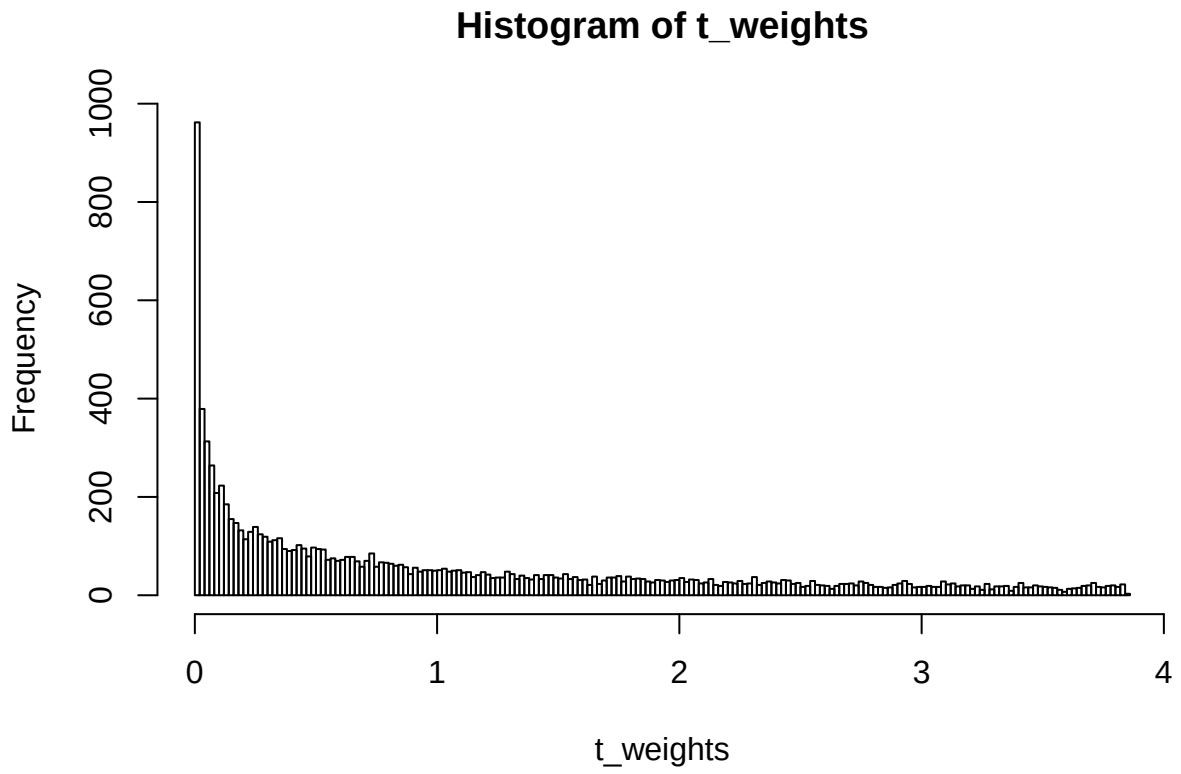
Next, we have to normalize like we did in part a by dividing by the cdf to make sure our pdf is proper and will integrate to 1.

```r
trunc_density_for_ft <- density_for_ft/(pt(q = -4, df = 3))
trunc_density_for_gt <- density_for_gt/(pt(q = -4, ncp = -4,
    df = 1))
```

To get the weights, we do our truncated density for f divded by truncated density for g. We can also use a histogram to compare the weights. Comparing this to part a, we can see that our values of weights are more disbursed than when we sampled from the normal distribution.

```r
t_weights <- trunc_density_for_ft/trunc_density_for_gt

hist(t_weights, breaks = 150)
```



Finally, we calculate the mean and variance in the same way we did in part a. We see our mean is lower here, closer to -6. I discussed this briefly with some classmates (Vaibhav and Narae), who said it's likely because the t function has more values in its lower tail than the normal distribution. We also see a lower variance here.

```
sample_t_mean <- mean(t_weights * qt_values)
# again using formula for finding variance when doing
# importance sampling
sample_t_var <- var(t_weights * qt_values)/10000
sample_t_mean
```

```
## [1] -6.235051
```

```
sample_t_var
```

```
## [1] 0.001779145
```

Question 2: Contour Plotting and Optim

We are given this code to work with for our theta and f function.

```
theta <- function(x1, x2) atan2(x2, x1)/(2 * pi)

f <- function(x) {
    f1 <- 10 * (x[3] - 10 * theta(x[1], x[2]))
    f2 <- 10 * (sqrt(x[1]^2 + x[2]^2) - 1)
    f3 <- x[3]
    return(f1^2 + f2^2 + f3^2)
}
```

To make it easier to hold one variable constant in order to do a 2-D plot, I rewrote the function to take in 3 inputs, but overall, the function will complete the same thing that our original f function does. Instead of taking in a vector of 3 elements, it takes in 3 separate elements.

```
# revised f function to make it easier to hold one variable
# constant while inputting sequences for the other 2
# variables
f_revised <- function(x1, x2, x3) {
    f1 <- 10 * (x3 - 10 * theta(x1, x2))
    f2 <- 10 * (sqrt(x1^2 + x2^2) - 1)
    f3 <- x3
    return(f1^2 + f2^2 + f3^2)
}
```

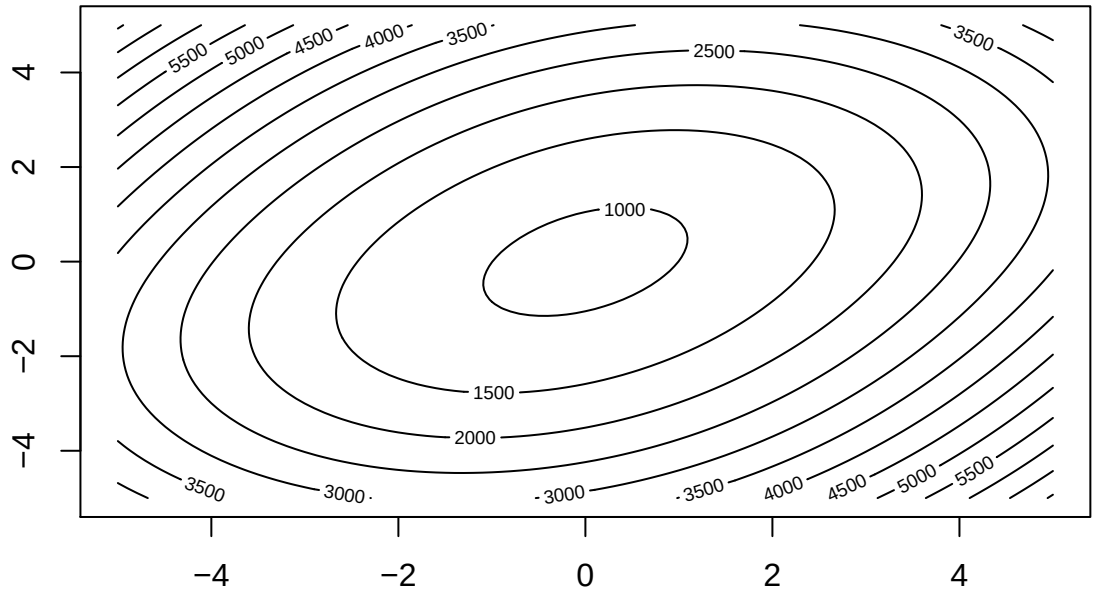
So, first we hold x1 constant and set it some random value, like 4, and have x2 and x3 be some sequence of numbers.

```
# creating sequences to be able to input different values for
# x2 and x3
x2 <- seq(-5, 5, 0.1)
x3 <- seq(-5, 5, 0.1)
# use expand.grid to get every possible outcome
args1 <- expand.grid(x2, x3)
```

Next, we use mapply to apply our function to the list of values created in our previous step while x1 is being held constant, and we plot our contour from this result.

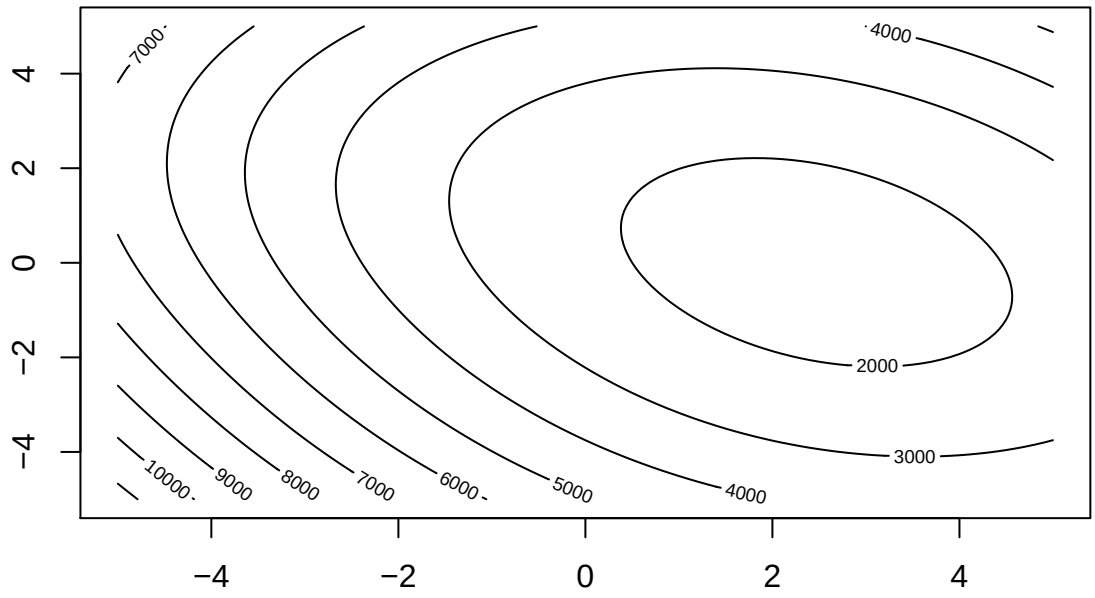
```
# use mapply to actually now plug in values into function
# with x1 constant
f_vals1 <- mapply(f_revised, x2 = args1$Var1, x3 = args1$Var2,
    x1 = 4)

# create a matrix out of outputs since contour needs a matrix
# for input use ncol =101 since length of x2 and x3 is 101
f_vals1_mat <- matrix(unlist(f_vals1), ncol = 101, byrow = TRUE)
contour(x2, x3, f_vals1_mat)
```



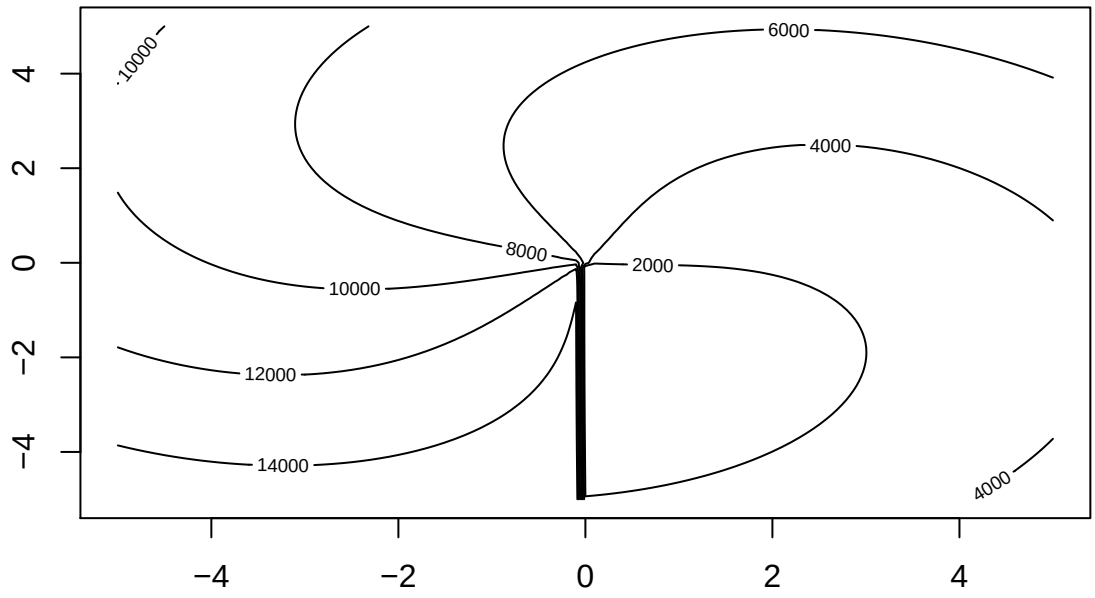
So, we can take this same approach for holding x2 constant and letting x1 and x3 be sequences. We set x2 to some random value, say 5.

```
# creating x1 sequence
x1 <- seq(-5, 5, 0.1)
args2 <- expand.grid(x1, x3)
f_vals2 <- mapply(f_revised, x1 = args1$Var1, x3 = args1$Var2,
    x2 = 5)
# using that length x1 and x3 is 101
f_vals2_mat <- matrix(unlist(f_vals2), ncol = 101, byrow = TRUE)
contour(x1, x3, f_vals2_mat)
```



Again, we can do the same approach and hold x3 constant while x1 and x2 are sequences.

```
args3 <- expand.grid(x1, x2)
f_vals3 <- mapply(f_revised, x1 = args1$Var1, x2 = args1$Var2,
    x3 = 7)
# again using that length x1 and x2 is 101
f_vals3_mat <- matrix(unlist(f_vals3), ncol = 101, byrow = TRUE)
contour(x1, x2, f_vals3_mat)
```



Lastly, we try out using the optim function with different intial parameters and different methods to compare the results. From these results, it seems like there may be some local minima in our function.

```
# trying using the constant x1, x2, and x3 values we used
# above
result1 <- optim(c(4, 5, 7), f, method = "BFGS")

# trying 0,0,0 since it seems pretty central with 2 different
# methods
result2 <- optim(c(0, 0, 0), f, method = "BFGS")
result3 <- optim(c(0, 0, 0), f, method = "Nelder-Mead")

# trying with negative starting values
result4 <- optim(c(-1, -1, -1), f, method = "BFGS")


# to compare 2 diferent methods
result2
result3

result2$value < result3$value
```

So, when we compare the outputs of the optim function using the BFGS method versus the Nelder-Mead, we see that BFGS actually gives us a smaller minimum than Nelder-Mead and our values that minimize the function are slightly different (likely due to some rounding).

Below, we try out using nlm to find our local minima.

```
result5 <- nlm(f, c(4, 5, 7))
result6 <- nlm(f, c(0, 0, 0))
result5
```

```
## $minimum
## [1] 1.701061e-08
##
## $estimate
## [1]  9.999995e-01 -8.223523e-05 -1.300862e-04
##
## $gradient
## [1]  9.740068e-08  1.727805e-07 -1.208802e-07
##
## $code
## [1] 1
##
## $iterations
## [1] 33
```

```
result6
```

```
## $minimum
## [1] 100
##
## $estimate
## [1] 0 0 0
##
## $gradient
## [1] -12500000         0         0
##
## $code
## [1] 3
##
## $iterations
## [1] 1
```

Question 3: EM Algorithm

a)

We want to code an EM Algorithm for the Probit Regression model. The probit model is: Y_i distributed $Ber(p_i)$ for $p_i = P(Y_i = 1) = \Phi(X_i^T\beta)$ where Φ is the standard normal CDF. We can rewrite this model with latent variable z_i such that $y_i = I(z_i > 0)$ and z_i is distributed $N(X_i^T\beta, 1)$. So, I is the indicator that the event $z_i > 0$ occurs and will equal 0 if the event doesn't occur.

For this problem, I asked classmates (Andrea) and used these notes http://sta250.github.io/Stuff/Lecture_13.pdf for help.

So, from the EM Algorithm, our Q function (based on the log likelihood function) is: $Q(\beta|\beta^t) = E(-\frac{1}{2}\sum(z_i - x_i^T\beta)^2|y, \beta^t) = -\frac{1}{2}(E(Z - X\beta)^T(Z - X\beta)|y, \beta^T)$, plus some constant term that we don't have to worry about since Q is a distribution of only β.

Now, our M step is to maximize the result of our E step. We observe that $Z_i|y_i = 0, \beta^t$ is a truncated normal distribution, $N(x_i^T\beta^t, 1)$ from $(-\infty, 0]$ and then $Z_i|y_i = 1, \beta^t$ is another truncated normal distribution,

$N(x_i^T\beta^t, 1)$ from $[0, \infty]$. So, in our maximization, we take the derivative of our Q function, which we simplified/ took the expectation of in our E step.

Then, $\frac{dQ}{d\beta} = E[Z|y, \beta^t]^T X - X^T X\beta$. We say that $Z^(t+1) = E[Z|y, \beta^t]$, and thus we get that $\beta^{(t+1)} = (X^T X)^{-(1)} X^T Z^{(t+1)}$. So, for the probit regression we get that when $y_i = 0$, then $Z_i^{(t+1)} = x_i^T\beta^t - \frac{\phi(x_i^T\beta^t)}{\Phi(-x_i^T\beta^t)}$, and when $y_i = 1$, we get that $Z_i^{(t+1)} = x_i^T\beta^t + \frac{\phi(x_i^T\beta^t)}{1-\Phi(-x_i^T\beta^t)}$.

So, summarizing our EM algorithm: 1) Select a starting value for β and set $t = 0$, 2) E-step, using our above computation for $Z^{(t+1)}$, 3) M-Step, using our above computation for $\beta^{(t+1)}$, and 4) continue this until $\beta^t and \beta^{(t+1)}$ are very close together. In the code below, we write to check whether our z's have converged closer.

I looked at some ways to code EM algorithms in R here: http://gallery.rcpp.org/articles/EM-algorithm-example/ and https://github.com/m-clark/Miscellaneous-R-Code/blob/master/ModelFitting/EM%20Examples/EM%20algorithm%20for%20probit%20example.R#L47 and tried to follow it.

```
em_alg <- function(beta_hat, X, y, epsilon, iters) {
    # beta.hat is a vector of just some betas to start with X,Y
    # matrices epsilon is how close we want our beta to the t to
    # be to our beta to the t+1 max_iterations is how many
    # iterations we're willing to do

    # to start...
    mu <- X %*% beta_hat
    converged <- FALSE
    # bringing in latent variable
    z <- rnorm(length(y))

    while ((!converged) & (iters > 0)) {
        z_prev = z
        # so while we haven't converged yet, we use set previous z to
        # z

        # use above calculation and do ifelse statement for y=1/0
        z = ifelse(y == 1, mu + dnorm(mu)/pnorm(mu), mu - dnorm(mu)/pnorm(-mu))

        # solve for beta in terms of x,x tranpose, & z
        beta_hat_plus_one = solve(t(X) %*% X) %*% t(X) %*% z
        # get our truncated distribution mean by multiplying X and
        # our betas
        mu = X %*% beta_hat

        # log-likelihood function
        ll = sum(y * pnorm(mu, log.p = T) + (1 - y) * pnorm(-mu,
            log.p = T))
        # keep decreasing iters til we fulfill all the ones that the
        # user inputted
        iters = iters - 1
    }

    # define converged based on the closeness of our z's
    converged <- max(abs(z_prev - z)) <= epsilon

    return(list(beta_ests = beta_hat_plus_one, ll = ll))
```

```
}
```

b)

So, when trying to choose betas, I discussed with some classmates how to go about this. We are given in part c) to have $\beta_2 = \beta_3 = 0$. Also, we want that $\frac{\beta_1}{se(\beta_1)} = 2$. So, I propose to choose: $\beta_0 = 1$, $\beta_1 = 2$, $\beta_2 = 0$, and $\beta_3 = 0$. Another valid starting point (if we don't know our data) is (0,0,0,0) as the betas could be positive or negative so we can start in the middle.

c)

So, we want to write an R function that will estimate our first parameters. I looked online on how to code probit regressions in R and found this website https://stats.idre.ucla.edu/r/dae/probit-regression/. So, in this code, we use use glm and family=binomial (probit) after we generate or read in some data in R.

```
n <- 100

# just generating some data for a starting point
example_data <- data.frame(x1 = rnorm(n), x2 = rnorm(n, mean = 0,
    sd = 2), x3 = rnorm(n, mean = 5, sd = 0.5), y = sample(c(0,
    1), n, replace = TRUE))

# below, we follow from the code linked above, we use y~. to
# say treat all the other variables in example_data as
# independent variables
myprobit <- glm(y ~ ., family = binomial(link = "probit"), data = example_data)
myprobit
```

```
##
## Call:  glm(formula = y ~ ., family = binomial(link = "probit"), data = example_data)
##
## Coefficients:
## (Intercept)           x1           x2           x3
##     1.01308      0.04912     -0.04313     -0.16917
##
## Degrees of Freedom: 99 Total (i.e. Null);  96 Residual
## Null Deviance:       136.7
## Residual Deviance: 135.5     AIC: 143.5
```

```
# set first column equal to 1 in order to be able to solve
# for the beta estimates in our next step need X to be 1 by 4
# in order to match up with our beta values
X <- as.matrix(cbind(1, example_data[, 1:3]))

# y is just the y part of our example data
y <- example_data$y

# the starting betas as proposed in part b
betas_init <- c(1, 2, 0, 0)
```

So, below is how we would input the generated data in our em_alg function.

```r
# so try our em alg function
em_alg(betas_init, X, y, 1e-06, iters = 500)
```

```
## $beta_ests
##            [,1]
## 1   0.597477620
## x1  1.035983865
## x2 -0.013688162
## x3 -0.001590806
##
## $ll
## [1] -196.0317
```

d)

Here, we try optim with our log-likelihood function, ll, from our EM Algorithm function. So, using the same starting points as part c), we write a function that just takes the parts of the EM Algorithm function related to the log-likelihood function and then apply optim to that function, fixing X and y optimizing on the beta_hat parameter. But, after talking to some classmates (Huy and Malvika) about this approach, they said I should use the negative of the log-likelihood function as optim is better for solving for the minimum. The output of using optim, with the Nelder-Mead and BFGS methods is below.

```r
# creating a function that just does the log-likelihood part
# of the EM algorithm taking in betas, matrix X, and y for
# input
neg_log_likh <- function(beta_hat, X, y) {
    mu = X %*% beta_hat
    neg_ll = -(sum(y * pnorm(mu, log.p = T) + (1 - y) * pnorm(-mu,
        log.p = T)))
}

# applying optim to the log-likelihood function we just
# created using the same starting Betas as part c and method
# is Nelder-Mead and BFGS; using optim to maximize our
# log-likelihood
ll_max1 <- optim(c(1, 2, 0, 0), neg_log_likh, X = X, y = y, method = "Nelder-Mead")
ll_max2 <- optim(c(1, 2, 0, 0), neg_log_likh, X = X, y = y, method = "BFGS")

ll_max1$value
```

```
## [1] 67.75081
```

```r
ll_max2$value
```

```
## [1] 67.75081
```

```r
ll_max1$value < ll_max2$value
```

```
## [1] FALSE
```

```
ll_max1
```

```
## $par
## [1]  1.01390728  0.04922762 -0.04309506 -0.16930244
##
## $value
## [1] 67.75081
##
## $counts
## function gradient
##      275       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```