> # Jeff Hill, ECON 613 Assignment 2

>

> library(stringr)

> library(dplyr)

> library(StatMeasures)

> library(ggplot2)

> library(Hmisc)

> library(boot)

> library(numDeriv)

> #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

> # Exercise 1 Data Creation

> set.seed(613)

>

> # generate variables pulling from different distributions

> # uniform [1:3]

> x1 <- runif(10000, min = 1, max = 3)

> # gamma shape:3 scale: 2

> x2 <- rgamma(10000, 3, scale = 2)

> # binomial

> x3 <- rbinom(10000, 1, 0.3)

> # normal mean 2

> eps <- rnorm(10000, mean = 2, sd = 1)

>

> # creating y and ydum

> y <- 0.5 + 1.2\*x1 -0.9\*(x2) + 0.1\*x3 +eps

> ydum <- as.numeric(y > mean(y))

>

>

> #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

> # Exercise 2 OLS

> # correlation between y and X1

> rcorr(y, x1)

x y

x 1.00 0.22

y 0.22 1.00

n= 10000

P

x y

x 0

y 0

> # the correlation between y and x1 is 0.22, which is different from 1.2 by 0.98. since correlation

> # is bound between -1 and 1, it would be problematic if we got correlation close to 1.2 (above 1)

>

> # Regression of Y on x = [1,x1,x2,x3]

> # creating X matrix

> intercept <- rep(1, 10000)

> x <- matrix(c(intercept,x1,x2,x3), nrow = 10000, ncol = 4, byrow = FALSE)

> colnames(x) <- c('intercept',"x1","x2","x3")

>

> # creating a function for solving for OLS betas that takes in y,x and outputs betas.

> ols\_coef <- function(y,x) {

+ solve(t(x)%\*%x)%\*%t(x)%\*%y

+ }

> # using this function to solve for our ols betas

> betas <- ols\_coef(y,x)

> # the betas are:

> betas[1]

[1] 2.466498

> # intercept: 2.466498

> betas[2]

[1] 1.234452

> # beta for x1: 1.234452

> betas[3]

[1] -0.9058537

> # beta for x2: -.9058537

> betas[4]

[1] 0.1240774

> # beta for x3: 0.1240774

> # these values make sense as they match quite closely to the expected values for the betas from the formula for y.

> # the only notable difference is the intercept term which is 2 larger than expected (2.5 vs. 0.5). This is due to the error term,

> # eps, having a mean of 2, which is captured by the constant term.

>

> # calculating standard errors

> # create standard error function following OLS formula: variance(Betas) = (X'X)^(-1)\*sigma^2 where sigma^2 is the variance of eps

> ols\_se <- function (eps,x) {

+ sqrt(diag(var(eps)\*solve(t(x)%\*%x)))

+ }

>

> ols\_se(eps,x)

intercept x1 x2 x3

0.040949088 0.017398910 0.002882897 0.022004454

> # the standard error of the betas are:

> # for intercept: 0.040949088

> # for beta\_x1: 0.017398910

> # for beta\_x2: 0.002882897

> # for beta\_x3: 0.022004454

>

> # Now using bootstrap with 49 and 499 replications respectively.

> # create data frame suitable for bootstrapping

> dat <- data.frame(y,intercept,x1,x2,x3,eps,ydum)

>

> # create bootstrap function that has inputs data and number of repetitions, creates a matrix to store the bootstrap betas in,

> # stores those betas and then calculates the standard errors.

> ols\_se\_bootstrap <- function(data,reps) {

+ betadf <- data.frame(matrix(NA, nrow = reps, ncol = 4)) # here we create the empty matrix to temporarily store betas in

+ for (i in 1:reps) {

+ boot\_sample <- data[sample(nrow(data), 10000, replace = T), ]

+ boot\_y <- as.matrix(boot\_sample[,1])

+ boot\_x <- as.matrix(boot\_sample[,2:5])

+ betadf[i,] <- t(ols\_coef(boot\_y,boot\_x))

+ }

+ apply(betadf, 2, sd)

+ }

> # return the standard errors produced by bootstrap with 49 replications:

> ols\_se\_bootstrap(dat,49)

X1 X2 X3 X4

0.050086521 0.018716541 0.003642031 0.025282905

> # the standard error of the betas from a trial using bootstrap 49 replications are:

> # se for intercept: 0.050086521 \*\*\* Compared to OLS formula SE's \*\*\* for intercept: 0.040949088

> # se for beta\_x1: 0.018716541 for beta\_x1: 0.017398910

> # se for beta\_x2: 0.003642031 for beta\_x2: 0.002882897

> # se for beta\_x3: 0.025282905 for beta\_x3: 0.022004454

>

>

> # now for bootstrap standard errors with 499 replications:

> ols\_se\_bootstrap(dat,499)

X1 X2 X3 X4

0.039802565 0.017253930 0.003001135 0.021328256

> # the standard error of the betas using bootstrap 499 replications are:

> # se for intercept: 0.039802565 \*\*\* Compared to OLS formula SE's \*\*\* for intercept: 0.040949088

> # se for beta\_x1: 0.017253930 for beta\_x1: 0.017398910

> # se for beta\_x2: 0.003001135 for beta\_x2: 0.002882897

> # se for beta\_x3: 0.021328256 for beta\_x3: 0.022004454

>

>

> #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

> # Exercise 3 Numerical Optimization

> # consider the probit estimation of ydum on X

> # we begin by writing a function that returns the likelihood of the probit we want to be sure to minimize

> # the negative log-likelihood, instead of maximizing the log-likelihood this function will take inputs betas

> # (named beta), since those are what we want to optimize ultimately. input beta will be a vector of length 4

> pro\_neg\_ll <- function (pro\_beta,x,y\_dum) {

+ xb <- x%\*%pro\_beta # create XB

+ p <- pnorm(xb) # apply F() to XB, where F() is cumulative standard normal distribution function

+ -sum( y\_dum\*log(p) + (1 - y\_dum)\*log(1 - p) ) # sum the log of each individual likelihood, then make negative

+ }

> # test pro\_neg\_ll on some betas

> pro\_neg\_ll(betas, x, ydum)

[1] 2446.618

> pro\_neg\_ll(c(2.4,1.23,-0.9,.1), x, ydum)

[1] 2502.171

>

> # now impliment steepest ascent optimization to maximize this function

> # begin by setting intial values for beta vector, alpha, old and new likelihood.

> beta <- c(0,0,0,0)

> e <- 10^-10 # the amount we are incrementing our 4 betas by

> e\_mat <- matrix(c(e,0,0,0,0,e,0,0,0,0,e,0,0,0,0,e),nrow = 4, ncol = 4) # the matrix we add to mat1 in the while loop

> # that increments the betas

> alpha = .00001

> likold = 1

> liknew = 0

>

> while(abs(liknew - likold) > 10^-6) { # this while loop continues to iterate as long as liknew is different enough from

+ # lik old.

+ likold <- liknew # store the old likelihood from the last iteration for use in calculation the partial derivatives

+ mat1 <- matrix(c(beta,beta,beta,beta), nrow = 4, ncol = 4, byrow = F) # create a 4x4 matrix of betas 1 through 4

+ mat2 <- mat1 + e\_mat # create a second matrix that is equal to mat1 with "e" added to positions 11, 22, 33, and 44

+ d1 <- (pro\_neg\_ll(mat2[,1],x,ydum) - pro\_neg\_ll(mat1[,1],x,ydum))/e # here we define each direction as the diff. in likelihood over e

+ d2 <- (pro\_neg\_ll(mat2[,2],x,ydum) - pro\_neg\_ll(mat1[,2],x,ydum))/e

+ d3 <- (pro\_neg\_ll(mat2[,3],x,ydum) - pro\_neg\_ll(mat1[,3],x,ydum))/e

+ d4 <- (pro\_neg\_ll(mat2[,4],x,ydum) - pro\_neg\_ll(mat1[,4],x,ydum))/e

+ d <- c(d1,d2,d3,d4) # collect all the directions into a vector

+ beta <- beta - alpha\*d # construct our new beta = the old beta minus alpha times our direction vector

+ liknew <- pro\_neg\_ll(beta,x,ydum) # compute the likelihood of the new beta, which will be compared to the old likelihood at the

+ # beginning of the next iteration.

+ }

> print(beta) # optimal beta values resulting from this optimization are:

[1] 3.02055601 1.16047017 -0.88794791 0.04261497

> # intercept: 3.02055601 beta\_x1: 1.16047017 beta\_x2: -0.88794791 beta\_x3: 0.04261497

>

> # this while loop was tested at intial beta = c(0,0,0,0), c(-1,-1,-1,-1), c(1,1,1,1), c(2,2,2,2)

> # and it yielded the same optimal betas to within 1\*10^-5

>

> # testing again at alpha = .000001 NOTE: this loop takes 8 minutes, considering alpha is 1/10 the size of the previous alpha

> beta <- c(0,0,0,0)

> e <- 10^-10 # the amount we are incrementing our 4 betas by

> e\_mat <- matrix(c(e,0,0,0,0,e,0,0,0,0,e,0,0,0,0,e),nrow = 4, ncol = 4)

> alpha = .000001

> likold = 1

> liknew = 0

> while(abs(liknew - likold) > 10^-6) {

+ likold <- liknew # store the old likelihood from the last iteration for use in calculation the partial derivatives

+ mat1 <- matrix(c(beta,beta,beta,beta), nrow = 4, ncol = 4, byrow = F) # create a 4x4 matrix of betas 1 through 4

+ mat2 <- mat1 + e\_mat # create a second matrix that is equal to mat1 with "e" added to positions 11, 22, 33, and 44

+ d1 <- (pro\_neg\_ll(mat2[,1],x,ydum) - pro\_neg\_ll(mat1[,1],x,ydum))/e

+ d2 <- (pro\_neg\_ll(mat2[,2],x,ydum) - pro\_neg\_ll(mat1[,2],x,ydum))/e

+ d3 <- (pro\_neg\_ll(mat2[,3],x,ydum) - pro\_neg\_ll(mat1[,3],x,ydum))/e

+ d4 <- (pro\_neg\_ll(mat2[,4],x,ydum) - pro\_neg\_ll(mat1[,4],x,ydum))/e

+ d <- c(d1,d2,d3,d4)

+ beta <- beta - alpha\*d

+ liknew <- pro\_neg\_ll(beta,x,ydum)

+ }

> beta # optimal beta values resulting from this optimization at alpha = .000001 are:

[1] 3.01362382 1.16158053 -0.88719639 0.04308758

> # intercept: 3.01362382 beta\_x1: 1.16158053 beta\_x2: -0.88719639 beta\_x3: 0.04308758

> # which as you can see, are very close to the betas gathered when alpha = .00001. However

> # as expected, this while loop took SIGNIFICANTLY longer since we were moving in much shorter

> # increments.

>

> # the coefficients for exercise 3 are quite different from the coefficients in exercise 2. Exercise 2 ran an OLS of Y on X

> # where in exercise 3 we run a probit of ydum on X. We would not expect the coefficients to be identical.

>

>

> #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

> # exercise 4 discrete choice

>

> # using optim for probit. need to create gradient function, and optim will take in intital values for betas,

> # the probit function we created before, and then the gradient function for probit, and output the optimal

> # results.

>

> # we already have the probit negative log-likelihood function, so now just write the probit gradient function,

> # based on the F.O.C.s

> pro\_grad <- function (pro\_grad\_betas,x,y\_dum) {

+ xb <- x%\*%pro\_grad\_betas # create XB

+ p <- pnorm(xb) # apply F() to XB, where F() is cumulative standard normal distribution function

+ d <- dnorm(xb) # apply F'() to XB, where F'() is the probability standard normal distribution function

+ E <- ((y\_dum - p) \* d) / (p\*(1-p)) # sum of the first part of the F.O.C.

+ crossprod(E,x)

+ }

>

> # compose the optim command calling on intial betas, probit NLL function, and gradient.

> pro\_fit <- optim(betas, pro\_neg\_ll,x,ydum,gr = pro\_grad)

> # pro\_fit$par contains the estimated coefficents

> pro\_fit$par

[,1]

intercept 3.02402514

x1 1.15991748

x2 -0.88832316

x3 0.04234817

> # intercept 3.02402514

> # x1 1.15991748

> # x2 -0.88832316

> # x3 0.04234817

> # These match the betas resulting from our manual MLE optimization

> # in exercise 3. This is a great victory.

> # Intreptation of coefficients will be at the end once we have the coefficients and significance from all 3 models.

>

>

> # logit optimization

> # write logit negative log-likelihood function.

> logit\_neg\_ll <- function (logit\_beta,x,y\_dum) {

+ xb <- x%\*%logit\_beta # create XB

+ p <- exp(xb)/(1+exp(xb)) # apply F() to XB, where F() is the appropriate logit function

+ -sum( y\_dum\*log(p) + (1 - y\_dum)\*log(1 - p) ) # sum the log of each individual likelihood, then make negative

+ }

>

> # write logit gradient function

> logit\_grad <- function (logit\_grad\_betas,x,y\_dum) {

+ xb <- x%\*%logit\_grad\_betas # create XB

+ p <- exp(xb)/(1+exp(xb)) # apply F() to XB, where F() is the appropriate logit function

+ d <- exp(xb)/((1+exp(xb))^2) # apply F'() to XB, where F'() is derivate of the logit function

+ E <- ((y\_dum - p) \* d) / (p\*(1-p)) # sum of the first part of the F.O.C.

+ crossprod(E,x)

+ }

>

> # compose the optim command calling on intial betas, logit NLL function, and gradient.

> logit\_fit <- optim(betas, logit\_neg\_ll,x,ydum,gr = logit\_grad)

> # logit\_fit$par contains the estimated coefficents.

> logit\_fit$par

[,1]

intercept 5.4177581

x1 2.0739752

x2 -1.5903537

x3 0.0709954

> # intercept 5.4177581

> # x1 2.0739752

> # x2 -1.5903537

> # x3 0.0709954

> # Intreptation of coefficients will be at the end once we have the coefficients from all 3 models.

>

>

> # linear probability optimization.

> # here no need for optim, just "manually" calcuate betas for OLS regression on ydum using our function from exercise 2

> ols\_coef(ydum,x)

[,1]

intercept 0.896525083

x1 0.143025669

x2 -0.103295216

x3 0.009431036

> # linear probability coefficients

> # intercept 0.896525083

> # x1 0.143025669

> # x2 -0.103295216

> # x3 0.009431036

>

> # create a small dataframe containing coefficients from these 3 optimizations for convenient comparison.

> comp\_df <- as.data.frame(cbind(pro\_fit$par,logit\_fit$par,ols\_coef(ydum,x)))

> colnames(comp\_df) <- c('probit','logit','linear prob')

>

> # now we have our coefficents for each model. to interpret we will need significance, so lets use bootstrap to get

> # the standard errors we need.

>

> #>>>>>>>>>>>>>>>>>>>

> # probit and logit bootstrap standard errors.

> # bootstrap function for MLE model standard errors. Takes 4 inputs, data, number of bootstrap repetitions, the model type,

> # and the gradient type. Including the model and gradient allows us to consolidate the standard error bootstrapping for both

> # logit and probit to one function, instead of writing 2.

>

> mle\_se\_bootstrap <- function(data, reps, lik\_func, grad\_func) {

+ betadf <- data.frame(matrix(NA, nrow = reps, ncol = 4)) # here we create the empty matrix to temporarily store betas in

+ for (i in 1:reps) {

+ boot\_sample <- data[sample(nrow(data), 10000, replace = T), ]

+ boot\_ydum <- as.matrix(boot\_sample[,7])

+ boot\_x <- as.matrix(boot\_sample[,2:5])

+ boot\_fit <- optim(betas, lik\_func, boot\_x, boot\_ydum, gr = grad\_func)

+ betadf[i,] <- t(boot\_fit$par)

+ }

+ apply(betadf, 2, sd)

+ }

> # mel\_se\_bootstrap will be run for both probit and logit models below when we compare coefficients and significance.

>

> #>>>>>>>>>>>>>

> # linear probability bootstrap standard errors.

> # create a linear probability standard error function very similar to the ols standard error function in exercise 2.

> # the difference here is that we calculate variance based on y - yhat here, where above we calculated it using epsilon

> # (which in that case was given by the way we constructed y)

> lin\_prob\_se <- function (y, x, beta) {

+ var <- sum(((y - x%\*%beta)^2)/(nrow(x)-ncol(x)))

+ sqrt(diag(var\*solve(t(x)%\*%x)))

+ }

> # the linear probability standard errors are:

> lin\_prob\_se(ydum,x,ols\_coef(ydum,x))

intercept x1 x2 x3

0.0134967894 0.0057346681 0.0009502008 0.0072526520

>

>

> # now create similar comparison dataframe for the standard errors of the betas of probit, logit, and lin prob models

> # this command takes about 3 minutes as the mle\_se\_bootstrap for logit and probit takes a little while for each.

> se\_comp\_df <- as.data.frame(cbind(mle\_se\_bootstrap(dat,499,pro\_neg\_ll,pro\_grad), # here is where we run the probit se bootstrap

+ mle\_se\_bootstrap(dat,499,logit\_neg\_ll,logit\_grad), # here is where we run the logit se bootstrap

+ lin\_prob\_se(ydum,x,ols\_coef(ydum,x))))

> colnames(se\_comp\_df) <- c('probit','logit','linear prob') # rename the columns for clarity

>

> t\_stat\_df <- comp\_df /se\_comp\_df #using the matrix of prob/logit/lin betas (namely comp\_df) and the matrix of their standard errors,

> # create matrix of t\_statistics.

>

> p\_val\_df <- data.frame(matrix(NA, nrow = 4, ncol = 3)) #create matrix of p\_values of t\_statistics using pt('beta', df = 10000)

> p\_val\_df[,1] <- pt(abs(t\_stat\_df[,1]),10000) # fill the matrix

> p\_val\_df[,2] <- pt(abs(t\_stat\_df[,2]),10000)

> p\_val\_df[,3] <- pt(abs(t\_stat\_df[,3]),10000)

> p\_val\_df

X1 X2 X3

1 1.0000000 1.0000000 1.0000000

2 1.0000000 1.0000000 1.0000000

3 1.0000000 1.0000000 1.0000000

4 0.8237383 0.7946288 0.9032457

> # p\_val\_df tells us the significance of the betas found in comp\_df for probit, logit, and lin prob. models. for logit and probit,

> # beta\_x1 and beta\_x2 are both highly significant. Thus they can be interpreted only as far as their sign. The x1 and x2betas for

> # logit and probit are all positive, thus we can say they have a positive effect on the probability of ydum = 1 that is statistically

> # significant. the betas for x1 for logit and probit are not statistically significant at a 5% level, so we cannot say their effect

> # on ydum is statistically significantly different from zero.

>

> # interpretation for lin prob. beta\_x3 is not statistically significant at even a 10% level, thus its effect on ydum is not significantly

> # diffferent from zero. beta\_x1 is significant at a 1% level, so we say for a 1 unit increase in x1, the estimated increase of ydum

> # is .143. Beta\_x2 is significant at a 1% level, so we say for a 1 unit increase in x2, the estimated increase of ydum is -0.103.

> # note for the linear probability model, fitted values may exceed 1 or be lower than 0. since fitted values can be interpreted as

> # probability of y = 1, probability > 1 or < 0 do not make sense.

>

>

> #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

> # exercise 5 Marginal Effects

> # fetch betas and variance/covariance matrices from the glm probit and logit regressions

> logit\_glm <- glm(ydum~x1+x2+x3, family = binomial(logit))

Warning message:

glm.fit: fitted probabilities numerically 0 or 1 occurred

> probit\_glm <-glm(ydum~x1+x2+x3, family = binomial(probit))

Warning message:

glm.fit: fitted probabilities numerically 0 or 1 occurred

> # store betas

> logit\_betas <- logit\_glm$coefficients

> probit\_betas <- probit\_glm$coefficients

> # store vcov matrices

> logit\_vcov <- vcov(logit\_glm)

> probit\_vcov <- vcov(probit\_glm)

>

> #calculate marginal effects: NOTE: these marginal effects are for the representative individual, xbar

> #probit marginal effects

> probit\_me <- dnorm(colMeans(x)%\*%probit\_betas)\*probit\_betas

Warning message:

In dnorm(colMeans(x) %\*% probit\_betas) \* probit\_betas :

Recycling array of length 1 in array-vector arithmetic is deprecated.

Use c() or as.vector() instead.

> #logit marginal effects

> logit\_me <- exp(colMeans(x)%\*%logit\_betas)/((1+exp(colMeans(x)%\*%logit\_betas))^2)\*logit\_betas

Warning message:

In exp(colMeans(x) %\*% logit\_betas)/((1 + exp(colMeans(x) %\*% logit\_betas))^2) \* :

Recycling array of length 1 in array-vector arithmetic is deprecated.

Use c() or as.vector() instead.

>

>

>

> # now compute standard deviations of marginal effects via delta method.

> # here we will use those vcov matrices.

> # probit

> # using the jacobian function from numDeriv package, we can feed it a probit marginal effect function and the probit coefficients

> # and it will return the jacobian matrix. then multiply t(jacobian)\*vcov(probit)\*jacobian to yield var covariance matrix of probit

> # Marginal effects. same process is repeated for logit.

>

> probit\_me\_fun <- function(probit\_b) { #function that takes probit betas and returns marginal effects

+ dnorm(c(colMeans(x))%\*%probit\_b)\*probit\_b

+ }

>

> jac\_prob <-jacobian(probit\_me\_fun,probit\_betas) # probit jacobian

There were 33 warnings (use warnings() to see them)

> delt\_p <-t(jac\_prob)%\*%probit\_vcov%\*%jac\_prob

> jac\_prob\_sd <-sqrt(diag(delt\_p)) #taking the square root of diagonal terms

> jac\_prob\_sd

[1] 0.040020902 0.017199120 0.006647109 0.018844260

> # probit standard deviations produced by delta method: 0.040020902 0.017199120 0.006647109 0.018844260

>

> #logit delta method

> logit\_me\_fun <- function(logit\_b) { #function that takes logit betas and returns marginal effects of average individual

+ exp(c(colMeans(x))%\*%logit\_b)/((1+exp(c(colMeans(x))%\*%logit\_b))^2)\*logit\_b

+ }

>

> jac\_logit <-jacobian(logit\_me\_fun,logit\_betas) # logit jacobian

There were 33 warnings (use warnings() to see them)

> delt\_l <-t(jac\_logit)%\*%logit\_vcov%\*%jac\_logit

> jac\_logit\_sd <-sqrt(diag(delt\_l)) #taking the square root of diagonal terms

> jac\_logit\_sd

[1] 0.047285764 0.019974584 0.006438529 0.021140521

> # logit standard deviations produced by delta method: 0.047285764 0.019974584 0.006438529 0.021140521

>

> # standard deviations by bootstrap

> # probit and logit marginal effects bootstrap again combined into one function. The function glm\_se\_bootstrap takes 3 inputs,

> # data, number of repetitions, and then type, either "logit" or "probit". It calculates marginal effects in bootstrap, and

> # takes the standard error of them at the end.

>

> glm\_se\_bootstrap <- function(data, reps, type) {

+ betadf <- data.frame(matrix(NA, nrow = reps, ncol = 4)) # matrix to store the marginal effects in.

+ for (i in 1:reps) {

+ boot\_sample <- data[sample(nrow(data), 10000, replace = T), ]

+ boot\_ydum <- as.matrix(boot\_sample[,7])

+ boot\_x <- as.matrix(boot\_sample[,2:5])

+ boot\_fit <- glm(boot\_ydum~boot\_x[,2]+boot\_x[,3]+boot\_x[,4], family = binomial(type)) #glm, either logit or probit

+ boot\_betas <- boot\_fit$coefficients

+ if (type == "logit") {

+ boot\_me <- dlogis(c(colMeans(boot\_x))%\*%boot\_betas)\*boot\_betas # if else conditions for logit and probit. if type is neither,

+ } else if (type == "probit") { # then we return an error message.

+ boot\_me <- dnorm(c(colMeans(boot\_x))%\*%boot\_betas)\*boot\_betas

+ } else {

+ print("type not equal to logit or probit")

+ }

+ betadf[i,] <- t(boot\_me)

+ }

+ apply(betadf, 2, sd)

+ }

> glm\_se\_bootstrap(dat,499,"probit")

X1 X2 X3 X4

0.040885557 0.016950872 0.007174902 0.018471647

There were 50 or more warnings (use warnings() to see the first 50)

> glm\_se\_bootstrap(dat,499,"logit")

X1 X2 X3 X4

0.044770028 0.018828455 0.008308763 0.021731524

There were 50 or more warnings (use warnings() to see the first 50)

> # probit standard deviations produced by bootstrap: 0.040885557 0.016950872 0.007174902 0.018471647

> # logit standard deviations produced by bootstrap: 0.044770028 0.018828455 0.008308763 0.021731524

>

> # comparison ME\_intercept ME\_beta\_x1 ME\_beta\_x2 ME\_beta\_x3

> # probit standard deviations produced by delta method: 0.040020902 0.017199120 0.006647109 0.018844260

> # probit standard deviations produced by bootstrap: 0.040885557 0.016950872 0.007174902 0.018471647

>

> # logit standard deviations produced by delta method: 0.047285764 0.019974584 0.006438529 0.021140521

> # logit standard deviations produced by bootstrap: 0.044770028 0.018828455 0.008308763 0.021731524