> # Jeff Hill, ECON 613 Assignment 2

>

> library(stringr)

> library(dplyr)

Attaching package: ‘dplyr’

The following objects are masked from ‘package:stats’:

filter, lag

The following objects are masked from ‘package:base’:

intersect, setdiff, setequal, union

> library(StatMeasures)

> library(ggplot2)

> library(Hmisc)

Loading required package: lattice

Loading required package: survival

Loading required package: Formula

Attaching package: ‘Hmisc’

The following object is masked from ‘package:StatMeasures’:

contents

The following objects are masked from ‘package:dplyr’:

src, summarize

The following objects are masked from ‘package:base’:

format.pval, units

> library(boot)

Attaching package: ‘boot’

The following object is masked from ‘package:survival’:

aml

The following object is masked from ‘package:lattice’:

melanoma

> 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)

>

> #solving for OLS betas using B = (X'X)^(-1)X'Y

> xpxinv <- solve(t(x)%\*%x)

> betas <- xpxinv%\*%t(x)%\*%y

> # 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

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

> sigmasq <- var(eps)

> varbetas <- sigmasq\*xpxinv

> seOLS <- sqrt(diag(varbetas))

> # the standard error of the betas are:

> seOLS[1]

[1] 0.04094909

> # for intercept: 0.04094909

> seOLS[2]

[1] 0.01739891

> # for beta\_x1: 0.01739891

> seOLS[3]

[1] 0.002882897

> # for beta\_x2: 0.002882897

> seOLS[4]

[1] 0.02200445

> # for beta\_x3: 0.02200445

>

> # 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)

>

> #replications = 49

> # create empty dataframe to put beta values from for loop into

> betadf49 <- data.frame(matrix(NA, nrow = 49, ncol = 4))

> colnames(betadf49) <- c("intercept","beta\_x1","beta\_x2","beta\_x3")

>

> # bootstrap in for loop, sampling y 10000 times with replacement, then calculating betas. store those betas in betadf

> for (i in 1:49) {

+ bootd <- dat[sample(nrow(dat), 10000, replace = T), ]

+ booty <- as.matrix(bootd[,1])

+ bootx <- as.matrix(bootd[,2:5])

+ bootinv <- as.matrix(solve(t(bootx)%\*%bootx))

+ bootbeta <- as.matrix(bootinv%\*%t(bootx)%\*%booty)

+ betadf49[i,] <- t(bootbeta)

+ }

>

> # calculate standard errors of these betas

> se49 <- apply(betadf49, 2, sd)

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

> se49[1]

intercept

0.05008652

> # se for intercept: 0.03701252 \*\*\* Compared to OLS formula SE's \*\*\* for intercept: 0.04094909

> se49[2]

beta\_x1

0.01871654

> # se for beta\_x1: 0.01581407 for beta\_x1: 0.01739891

> se49[3]

beta\_x2

0.003642031

> # se for beta\_x2: 0.002948432 for beta\_x2: 0.002882897

> se49[4]

beta\_x3

0.0252829

> # se for beta\_x3: 0.01596766 for beta\_x3: 0.02200445

>

>

> # Replications = 499

> # create another empty dataframe to put beta values from for loop into

> betadf499 <- data.frame(matrix(NA, nrow = 499, ncol = 4))

> colnames(betadf499) <- c("intercept","beta\_x1","beta\_x2","beta\_x3")

> # bootstrap in for loop, sampling y 10000 times with replacement, then calculating betas. store those betas in betadf

> for (i in 1:499) {

+ bootd <- dat[sample(nrow(dat), 10000, replace = T), ]

+ booty <- as.matrix(bootd[,1])

+ bootx <- as.matrix(bootd[,2:5])

+ bootinv <- as.matrix(solve(t(bootx)%\*%bootx))

+ bootbeta <- as.matrix(bootinv%\*%t(bootx)%\*%booty)

+ betadf499[i,] <- t(bootbeta)

+ }

> # calculate the standard errors of the betas

> se499 <- apply(betadf499, 2, sd)

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

> se499[1]

intercept

0.03980256

> # se for intercept: 0.040901293 \*\*\* Compared to OLS formula SE's \*\*\* for intercept: 0.04094909

> se499[2]

beta\_x1

0.01725393

> # se for beta\_x1: 0.017351702 for beta\_x1: 0.01739891

> se499[3]

beta\_x2

0.003001135

> # se for beta\_x2: 0.002780576 for beta\_x2: 0.002882897

> se499[4]

beta\_x3

0.02132826

> # se for beta\_x3: 0.022487883 for beta\_x3: 0.02200445

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

> # 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) {

+ pro\_x <- as.matrix(dat[,2:5]) #brings in data matrix X

+ pro\_ydum <- as.matrix(dat[,7]) # bring in ydum

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

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

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

+ }

> # test pro\_neg\_ll on some betas

> pro\_neg\_ll(betas)

[1] 2446.618

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

[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]) - pro\_neg\_ll(mat1[,1]))/e # here we define each direction as the difference in likelihood divided by e

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

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

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

> 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]) - pro\_neg\_ll(mat1[,1]))/e

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

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

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

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

+ beta <- beta - alpha\*d

+ liknew <- pro\_neg\_ll(beta)

+ }

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

> # 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.

>

> # write the probit gradient function, based on the F.O.C.s

> pro\_grad <- function (pro\_grad\_betas) {

+ grad\_x <- as.matrix(dat[,2:5]) #brings in data matrix X

+ grad\_ydum <- as.matrix(dat[,7]) # bring in ydum

+ xb <- grad\_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 <- ((grad\_ydum - p) \* d) / (p\*(1-p)) # sum of the first part of the F.O.C.

+ crossprod(E,grad\_x)

+ }

>

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

> pro\_fit <- optim(betas, pro\_neg\_ll,pro\_grad)

> # pro\_fit$par contains the estimated coefficents, which 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 from all 3 models.

>

>

> # logit optimization

> # write logit negative log-likelihood function.

> logit\_neg\_ll <- function (logit\_beta) {

+ logit\_x <- as.matrix(dat[,2:5]) #brings in data matrix X

+ logit\_ydum <- as.matrix(dat[,7]) # bring in ydum

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

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

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

+ }

>

> # write logit gradient function

> logit\_grad <- function (logit\_grad\_betas) {

+ grad\_x <- as.matrix(dat[,2:5]) #brings in data matrix X

+ grad\_ydum <- as.matrix(dat[,7]) # bring in ydum

+ xb <- grad\_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 <- ((grad\_ydum - p) \* d) / (p\*(1-p)) # sum of the first part of the F.O.C.

+ crossprod(E,grad\_x)

+ }

>

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

> logit\_fit <- optim(betas, logit\_neg\_ll,logit\_grad)

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

> # 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.

> lin\_x <- as.matrix(dat[,2:5])

> lin\_ydum <- as.matrix(dat[,7])

> lin\_xpxinv <- solve(t(lin\_x)%\*%lin\_x)

> lin\_betas <- lin\_xpxinv%\*%t(lin\_x)%\*%lin\_ydum

>

> # 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,lin\_betas))

> 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 bootstrap standard errors.

> probit\_betadf499 <- data.frame(matrix(NA, nrow = 499, ncol = 4))

> colnames(probit\_betadf499) <- c("intercept","beta\_x1","beta\_x2","beta\_x3")

> # bootstrap in for-loop, sampling y 10000 times with replacement, then calculating betas. store those betas in probit\_betadf

>

> for (i in 1:499) {

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

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

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

+ # probit NLL function

+ pro\_neg\_ll <- function (pro\_beta) {

+ pro\_x <- boot\_x #brings in data matrix X

+ pro\_ydum <- boot\_ydum # bring in ydum

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

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

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

+ }

+ # probit gradient

+ pro\_grad <- function (pro\_grad\_betas) {

+ grad\_x <- boot\_x #brings in data matrix X

+ grad\_ydum <- boot\_y # bring in ydum

+ xb <- grad\_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 <- ((grad\_ydum - p) \* d) / (p\*(1-p)) # sum of the first part of the F.O.C.

+ crossprod(E,grad\_x)

+ }

+ boot\_pro\_fit <- optim(betas, pro\_neg\_ll,pro\_grad)

+ probit\_betadf499[i,] <- t(boot\_pro\_fit$par) # entire loop takes about a minute and a half.

+ }

> probit\_se499 <- apply(probit\_betadf499, 2, sd)

>

>

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

> # logit bootstrap standard errors.

> logit\_betadf499 <- data.frame(matrix(NA, nrow = 499, ncol = 4))

> colnames(logit\_betadf499) <- c("intercept","beta\_x1","beta\_x2","beta\_x3")

> # bootstrap in for-loop, sampling y 10000 times with replacement, then calculating betas. store those betas in logit\_betadf

>

> for (i in 1:499) {

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

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

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

+ # logit NLL function

+ logit\_neg\_ll <- function (logit\_beta) {

+ logit\_x <- boot\_x #brings in data matrix X

+ logit\_ydum <- boot\_ydum # bring in ydum

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

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

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

+ }

+ # logit gradient

+ logit\_grad <- function (logit\_grad\_betas) {

+ grad\_x <- boot\_x #brings in data matrix X

+ grad\_ydum <- boot\_ydum # bring in ydum

+ xb <- grad\_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 <- ((grad\_ydum - p) \* d) / (p\*(1-p)) # sum of the first part of the F.O.C.

+ crossprod(E,grad\_x)

+ }

+ boot\_logit\_fit <- optim(betas, logit\_neg\_ll,logit\_grad)

+ logit\_betadf499[i,] <- t(boot\_logit\_fit$par) # entire loop takes about a minute and a half.

+ }

> logit\_se499 <- apply(logit\_betadf499, 2, sd)

>

>

>

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

> # linear probability bootstrap standard errors.

> lin\_betadf499 <- data.frame(matrix(NA, nrow = 499, ncol = 4))

> colnames(lin\_betadf499) <- c("intercept","beta\_x1","beta\_x2","beta\_x3")

> # bootstrap in for-loop, sampling y 10000 times with replacement, then calculating betas. store those betas in lin\_betadf

> for (i in 1:499) {

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

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

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

+ boot\_inv <- as.matrix(solve(t(boot\_x)%\*%boot\_x))

+ boot\_beta <- as.matrix(boot\_inv%\*%t(boot\_x)%\*%boot\_ydum)

+ lin\_betadf499[i,] <- t(boot\_beta)

+ }

>

> lin\_se499 <- apply(lin\_betadf499, 2, sd)

>

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

> se\_comp\_df <- as.data.frame(cbind(probit\_se499,logit\_se499,lin\_se499))

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

>

> 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)

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

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

>

> # 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(c(colMeans(x))%\*%probit\_betas)\*probit\_betas

Warning message:

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

Warning message:

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

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

> # 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

> # logit standard deviations produced by delta method: 0.025312935 0.010731260 0.003287703 0.011800993

>

> # standard deviations by bootstrap

> # probit bootstrat

> me\_probit\_df499 <- data.frame(matrix(NA, nrow = 499, ncol = 4))

> colnames(me\_probit\_df499) <- c("intercept","beta\_x1","beta\_x2","beta\_x3")

> for (i in 1:499) {

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

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

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

+ boot\_probit\_glm <-glm(boot\_ydum~boot\_x[,2]+boot\_x[,3]+boot\_x[,4], family = binomial(probit))

+ boot\_probit\_betas <- boot\_probit\_glm$coefficients

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

+ me\_probit\_df499[i,] <- boot\_probit\_me

+ }

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

> me\_probit\_se499 <- apply(me\_probit\_df499, 2, sd)

> # probit standard deviations produced by bootstrap: 0.038586497 0.016186590 0.007057369 0.018398691

>

>

> # logit bootstrap

> me\_logit\_df499 <- data.frame(matrix(NA, nrow = 499, ncol = 4))

> colnames(me\_logit\_df499) <- c("intercept","beta\_x1","beta\_x2","beta\_x3")

> for (i in 1:499) {

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

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

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

+ boot\_logit\_glm <-glm(boot\_ydum~boot\_x[,2]+boot\_x[,3]+boot\_x[,4], family = binomial(logit))

+ boot\_logit\_betas <- boot\_logit\_glm$coefficients

+ boot\_logit\_me <- dlogis(c(colMeans(x))%\*%boot\_logit\_betas)\*boot\_logit\_betas

+ me\_logit\_df499[i,] <- boot\_logit\_me

+ }

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

> me\_logit\_se499 <- apply(me\_logit\_df499, 2, sd)

> # logit standard deviations produced by bootstrap: 0.07499633 0.02984987 0.01308334 0.03177344

>

> # comparison 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: 0.038586497 0.016186590 0.007057369 0.018398691

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

> # logit standard deviations produced by bootstrap: 0.045828773 0.020022542 0.008503187 0.021023087