ECON 613 Assignment 2

Nond Prueksiri

February 12, 2019

Excercise 1 Data Creation

Reset the work environment

```
rm(list = ls())
```

Set seed to 100 for reproduction purposes

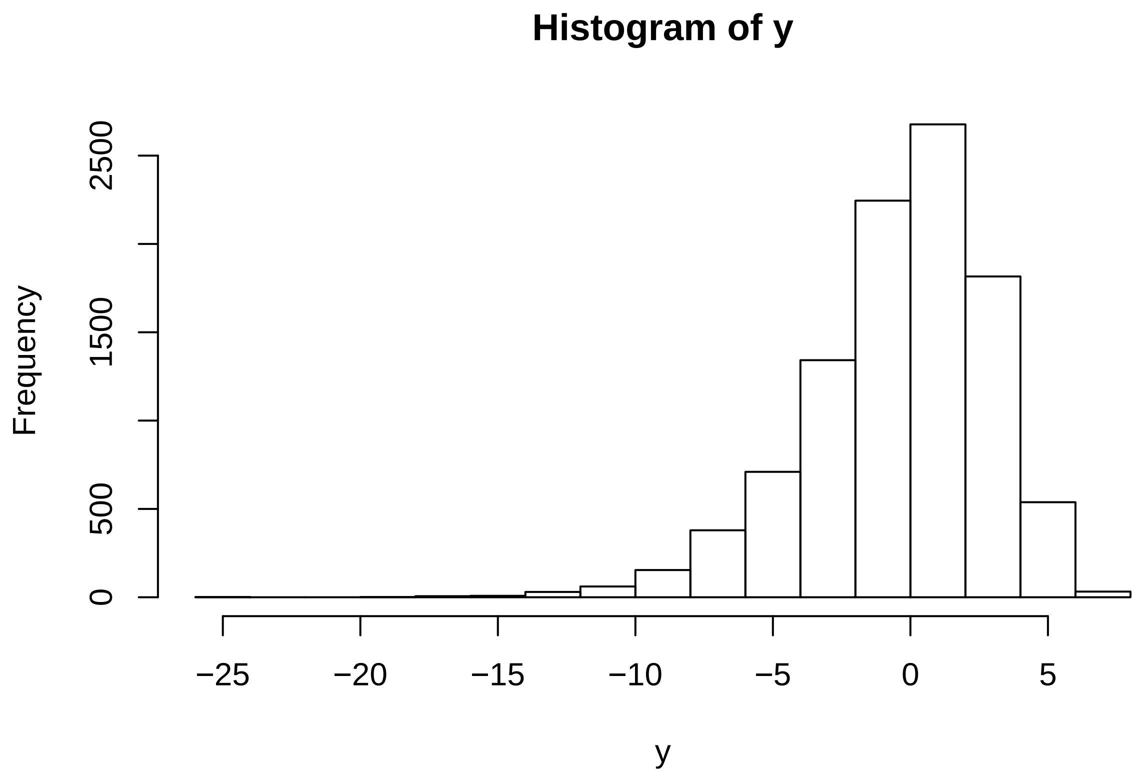
```
set.seed(100)
```

Create data as directed

```
x1  <- runif(10000, min = 1, max = 3)
x2  <- rgamma(10000, shape = 3, scale = 2)
x3  <- rbinom(10000, size = 1, prob = 0.3)
eps <- rnorm(10000, 2, 1)
```

Create the variable y

```
y <- 0.5 + 1.2*x1 - 0.9*x2 + 0.1*x3 + eps
hist(y)
```



Create dummy variable

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

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.0000  1.0000  0.5594  1.0000  1.0000
```

Create dataframe and matrices for further uses

```
# Dataframe for discrete choices
dat <- cbind(ydum, x1, x2, x3)
dat <- as.data.frame(dat)

# X Matrix for OLS
mx <- as.matrix(cbind(1, x1, x2, x3))
X <- as.matrix(mx)

# Y Matrix for OLS
my <- as.matrix(y)

# YDUM Matrix for discrete choices
mydum <- as.matrix(ydum)
Y <- as.matrix(mydum)
```

Excercise 2 OLS

Correlation between Y and X1

```r
cor(y, x1)
```

```
## [1] 0.2162074
```

Note that cor(y,x1) is not 1.2 because the range of correlation is between -1 and 1, moreover, the fact that (from excercise 1) Y = 0.5 + 1.2X1 + … indicates that cov(Y,X1)/var(X) = 1.2 not the correlation.

Estimate the regression of Y on X where X = (1,X1,X2,X3)

Using the OLS, let beta = vector of coefficients, mx = matrix of (1, X1, X2, X3), my = vector of Y The calculation is beta = inv(X'X)(X'Y)

```r
betaols <- solve(t(mx)%*%mx)%*%t(mx)%*%my

row.names(betaols) <- c("intercept", "x1", "x2", "x3")

colnames(betaols) <- c("y")

print(t(betaols))
```

```
##   intercept     x1         x2        x3
## y  2.456103 1.2158 -0.8984434 0.1018762
```

Estimate standard errors using OLS

Compute variance-covariance matrix by OLS formulas; sigma^2 times inverse(X'X), diagonal elements of the matrix are the standard errors

```r
# Create res = residual vector
res <- as.matrix(y-betaols[1]-betaols[2]*mx[,2]-betaols[3]*mx[,3]-betaols[4]*mx[,4])

# Assign value of n and k
n <- nrow(my)
k <- ncol(mx)

# Compute the Variance-covariance matrix VCV
VCV <- 1/(n-k) * as.numeric(t(res)%*%res) * solve(t(mx)%*%mx)

# Obtain SE from diagonal elements
se <- sqrt(diag(VCV))

# Report the output
ols_output <- cbind(betaols, se)
colnames(ols_output) <- c("Coefficient", "Standard Error")
row.names(ols_output) <- c("intercept", "x1", "x2", "x3")
print(ols_output)
```

```
##           Coefficient Standard Error
## intercept   2.4561034    0.040982313
## x1          1.2158000    0.017491090
## x2         -0.8984434    0.002952839
## x3          0.1018762    0.022040052
```

Estimate standard errors using bootstrap

Create ols and bootse function for estimating SE by bootstrap

```r
# Create OLS estimation function
ols <- function(my,mx) {
       beta <- solve(t(mx)%*%mx)%*%t(mx)%*%my
       }

# Create bootse function
boot <- as.numeric()

# Function bootse(xmatrix,ymatrix, number of replications)
bootse <- function(mx, my, rep) {

            # Loop of replication to rep times
            for (i in 1:rep) {

            ## draw sample with replacement (n = 10,000)
            boot_x <- mx[sample(nrow(mx), replace = TRUE), ]
            boot_y <- my[sample(nrow(my), replace = TRUE), ]

            ## estimate OLS coeff. for each samples drew, bind to the matrix
            boot   <- cbind(boot, ols(boot_y, boot_x))
            }

          # Matrix of OLS results ('rep' rows), calculate SE for each coefficient
          boot <- t(boot)
          boot_se <- apply(boot, 2, sd)

          # Report the result
          names(boot_se) <- c("intercept", "x1", "x2", "x3")
          print(boot_se)
        }
```

Calculate SE using 49 replication

```r
bootse(mx, my, 49)
```

```
##  intercept         x1         x2         x3
## 0.12582986 0.04727813 0.01095722 0.07426039
```

Calculate SE using 499 replication

```r
bootse(mx, my, 499)
```

```
##   intercept          x1          x2          x3
## 0.131785437 0.056087691 0.009551975 0.071049481
```

Excercise 3 Numerical Optimization

Write down the likelihood funtion of probit

Create dataframe to use for likelihood function

```r
dat <- as.data.frame(ydum)
dat <- cbind(dat, 1, x1, x2, x3)
```

Create log-likelihood funtion of probit

```r
probit_ll <- function (beta, df = dat) {

    # Calculate the vector xb = (X)(beta)
    xb <- as.matrix(dat[ ,2:5]) %*% beta
```

```
    # Fit xb into CDF of normal distribution
    p <- pnorm(xb)

    # Get the log-likelihood function
    logl <- sum((1 - dat[,1]) * log(1 - p) + dat[,1] * log(p))
    return(logl)
  }
```

Implement the steepest ascent optimization algorithm

Create function that returns first approximation of gradient for likelihood function

```
probit_grd <- function(beta, df = dat , l = probit_ll, d = 0.001, start = 0) {
    if(old_ll == 0) {old_ll  <- l(beta)}

    # Create default gradient vector
    grd <- matrix(nrow = 4, ncol = 1)

    # Calculate first approximation of each x
    for (i in 1:4) {
      beta_grd <- beta

      # Calculate f(x+d)
      beta_grd[i,1] <- beta[i,1]+d
      pos_ll <- l(beta_grd)

      # Calculate f(x-d)
      beta_grd[i,1] <- beta[i,1]-d
      neg_ll <- l(beta_grd)

      # Calculate the midpoint (f(x+d) - f(X-d))/2d
      grd[i,1] <- (pos_ll - neg_ll)/ 2*d
      }
    return(grd)
  }
```

Write algorithm for the steepest ascent optimization

```
# Set initial value for searching
beta_new <- as.matrix(c(0,0,0,0))

# Calculate the log-likelihood for the initial value
new_ll <- probit_ll(beta_new)

# Set level of optimization
tolerance <- 1      # Level of tolerance difference in ll between iteration
alpha     <- 0.5    # Scaling parameter
maxiter   <- 200    # Maximum of iterations regardless of tolerance

# Iteration
for (i in 1:maxiter) {
    beta_old <- beta_new      # Assign initial value of beta in iteration
    old_ll   <- new_ll        # Assign initial value of ll in iteration
    beta_new <- beta_old + alpha * probit_grd(beta_old , start = old_ll)
                                # Ascent to increase ll
    new_ll   <- probit_ll(beta_new) # Calculate for new ll

  if(abs(new_ll - old_ll) <= tolerance) { # Break loop once reaches tol
    break
  }
  iteration <- i    # report number of iterations
}
```

Report the coefficients (beta) from optimization and compare to the true value. Note that the optimization is set at tolerance = 1, alpha = 0.1 and cap the maximum iterations at 200 times

```
compare <- c(0.5, 1.2, -0.9, 0.1)
compare <- cbind(compare, beta_new)
colnames(compare) <- c("true value", "optimization")
row.names(compare) <- c("intercept", "x1", "x2", "x3")
print(compare)
```

```
##           true value optimization
## intercept        0.5   0.18027757
## x1               1.2   0.40262939
## x2              -0.9  -0.18794445
## x3               0.1   0.05337354
```

Note that the results are different from the face that the matrix Y used in estimation of the probit model is discrete choice not continuous variable.

Exercise 4 Discrete Choice

Optimize probit

Write down the negative log-likelihood function for probit, then use non-linear minimization pre-programmed package

```
# Set initial value for optimization
beta <- c(0,0,0,0)

# Create negative log-liklihood function for probit
  ## Follows the same logic as in ex 3 but in this case the negative ll
probit_nll <- function (beta, X = mx, Y = mydum) {
  xb <- X %*% beta
  p <- pnorm(xb)
  -sum((1 - Y) * log(1 - p) + Y * log(p))
  }

# Use pre-programmed optimization package "nlm"
probit_result <- nlm(probit_nll, beta)

# Report result
print(probit_result)
```

```
## $minimum
## [1] 2186.347
##
## $estimate
## [1]  2.81678631  1.23906388 -0.89214737  0.04803931
##
## $gradient
## [1] -1.804921e-04  2.789267e-05 -9.308678e-04  6.057235e-04
##
## $code
## [1] 1
##
## $iterations
## [1] 36
```

Optimize logit

Write down the negative log-likelihood function for logit, then use non-linear minimization pre-programmed package

```
# Create negative log-liklihood function for logit
logit_nll <- function(beta, X = mx, Y = mydum) {
```

```r
  # predictor for xb (at initial value)
  xb <- X %*% beta

  # calculate logistic CDF
  p <- plogis(xb)

  # derive negative log-likelihood function
  -sum((1 - Y) * log(1 - p) + Y * log(p))
}

# Use pre-programmed optimization package "nlm"
logit_result <- nlm(logit_nll, beta)

# Report result
print(logit_result)
```

```
## $minimum
## [1] 2190.592
##
## $estimate
## [1]  5.06604955  2.23095551 -1.60596229  0.08672426
##
## $gradient
## [1] -1.106786e-04 -1.416655e-04 -5.787829e-04 -2.319211e-05
##
## $code
## [1] 1
##
## $iterations
## [1] 38
```

Linear Probability Model

Calculate linear probability model using OLS

```r
# Using OLS formula
lpm_result <- solve(t(mx)%*%mx)%*%t(mx)%*%mydum

# Report result
print(lpm_result)
```

```
##           [,1]
##     0.87952298
## x1  0.15208902
## x2 -0.10554274
## x3  0.01055706
```

Compare the estimates among the three model

In this sub-task, function glm and lm is utilized to obtain accurate standard errors.

```r
# Estimate three models above using "lm" and "glm"
probit <- glm(ydum ~ x1+x2+x3, family = binomial(link='probit'))
logit  <- glm(ydum ~ x1+x2+x3, family = binomial(link='logit'))
lpm    <- lm(ydum ~ x1+x2+x3)
```

Report table

```r
all_result <- probit$coefficients

all_result <- rbind(all_result, sqrt(diag(vcov(probit))), logit$coefficients,
                    sqrt(diag(vcov(logit))), lpm$coefficients, sqrt(diag(vcov(lpm))))
```

```
row.names(all_result) <- c("Coef.-Probit", "Se.-Probit", "Coef.-Logit",
                           "Se.-Logit" ,"Coef.-LPM", "Se.-LPM")

print(all_result)
```

```
##              (Intercept)          x1            x2          x3
## Coef.-Probit  2.81677032 1.239054070 -0.8921408040 0.048036235
## Se.-Probit    0.09725690 0.044139974  0.0180388105 0.046861550
## Coef.-Logit   5.06601765 2.230938743 -1.6059503674 0.086720681
## Se.-Logit     0.18221378 0.082515643  0.0361166401 0.084252829
## Coef.-LPM     0.87952298 0.152089020 -0.1055427425 0.010557063
## Se.-LPM       0.01345961 0.005744507  0.0009697853 0.007238499
```

Note that for the coefficients, they have the same sign for all of the models, that is, negative for the x2, otherwise, positive. At this level of analysis, we cannot directly conclude the megitude of the relationship. All we can say is that, for the independent variables that have positive corresponding coefficient, an increase in x1 or x3 associates with an increase in probability that ydum = 1. On the other hand, for x2, a decrease in x2 associates with an increase in probability that ydum = 1. Considering their standard errors, the coefficients are all statistically significant except for x3 for all of the models, as seen from relatively high point estimates comparing to their corresponding standard errors.

Exercise 5 Marginal Effects

Marginal Effect for probit

Using probit result in ex 4 to estimate marginal effects

```
probit_coeff <- as.matrix(probit$coefficients)

# Calculate the mean of f'(xb)
probit_fprime <- mean(dnorm(mx %*% probit_coeff))

# Calculate marginal effects b*mean(f'(xb))
probit_mfx <- probit_coeff*probit_fprime

# Report result
colnames(probit_mfx) <- c("Marginal Effects for Probit Model")
print(probit_mfx)
```

```
##             Marginal Effects for Probit Model
## (Intercept)                       0.342005585
## x1                                0.150443012
## x2                               -0.108321625
## x3                                0.005832446
```

Marginal Effect for logit

Using logit result in ex 4 to estimate marginal effects

```
# Calculate the mean of f'(xb)
logit_coeff <- as.matrix(logit$coefficients)

# Calculate marginal effects b*mean(f'(xb))
logit_fprime <- mean(exp(-(mx %*% logit_coeff))/((exp(-(mx %*% logit_coeff)) + 1)^2))

# Calculate marginal effects b*mean(f'(xb))
logit_mfx <- logit_coeff*logit_fprime

# Report result
colnames(logit_mfx) <- c("Marginal Effects for Logit Model")
print(logit_mfx)
```

```
##              Marginal Effects for Logit Model
## (Intercept)                      0.340761837
## x1                               0.150062404
## x2                              -0.108023034
## x3                               0.005833201
```

Compute the standard deviation

Delta Method

Writing function computing Jacobian and Variance-covariance matrix

```
# Compute matrix inverse(X'X)
inv_xx <- solve(t(X) %*% X)

# Derive pdf of each model (first-order derivate)
probit_pdf <- t(as.matrix(dnorm(mx %*% probit$coefficients)))
logit_pdf  <- t(as.matrix(exp(-(mx %*% logit$coefficients)/
              ((exp(-(mx %*% logit$coefficients)) + 1)^2))))

# Write mfxse function to calculate asymptotic variance using delta method

mfxse <- function(pdf, X, model) {

  # Compute Jacobian matrix
  jac <- (1/nrow(X)) * (pdf %*% X)

  # Obtain model variance covariance matrix
  varcov <- vcov(model)

  # delta method asy.var = JVJ'
  avar <- jac %*% varcov %*% t(jac)

  # extract variance for each marginal effect
  se <- c(avar*inv_xx[1,1], avar*inv_xx[2,2], avar*inv_xx[3,3], avar*inv_xx[4,4])
  se <- sqrt(se)
  return(se)
}
```

SE for marginal effects of probit from delta method

```
probit_se_delta <- mfxse(probit_pdf, mx, probit)

print(probit_se_delta)
```

```
## [1] 1.058806e-04 4.518942e-05 7.628859e-06 5.694197e-05
```

SE for marginal effects of logit from delta method

```
logit_se_delta <- mfxse(logit_pdf, mx, logit)

print(logit_se_delta)
```

```
## [1] 0.0021999159 0.0009389154 0.0001585073 0.0011831020
```

Bootstrap Method

```
# Create default bootstrap matrix
boot_mfxmat <- matrix(nrow = 1, ncol = 4)

dat <- data.frame(ydum, x1 , x2, x3)
```

Bootstrap for standard error of mfx in probit

```r
# Estimate mfx for 49 times
for (i in 1:49) {
  boot_sample <- dat[sample(nrow(dat), replace = TRUE), ]
  boot_est    <- glm(ydum ~ ., data = boot_sample, family = binomial(link='probit'))
  boot_coeff  <- as.matrix(coef(summary(boot_est))[, "Estimate"])
  boot_pdf    <- mean(dnorm(mx %*% boot_coeff))
  boot_mfx    <- boot_coeff * boot_pdf
  boot_mfxmat <- rbind(boot_mfxmat, t(boot_mfx))
}

  boot_mfxmat <- boot_mfxmat[-1, ]
  boot_mfxmatse <- c(sd(boot_mfxmat[ , 1]), sd(boot_mfxmat[ , 2]),
                     sd(boot_mfxmat[ , 3]), sd(boot_mfxmat[ , 4]))


  print(boot_mfxmatse)
```

```
## [1] 0.0090561735 0.0040715342 0.0003182283 0.0052852822
```

Bootstrap for standard error of mfx in logit

```r
# Estimate mfx for 49 times
for (i in 1:49) {
  boot_sample <- dat[sample(nrow(dat), replace = TRUE), ]
  boot_est    <- glm(ydum ~ ., data = boot_sample, family = binomial(link='logit'))
  boot_coeff  <- as.matrix(coef(summary(boot_est))[, "Estimate"])
  boot_pdf    <- mean(dlogis(mx %*% boot_coeff))
  boot_mfx    <- boot_coeff * boot_pdf
  boot_mfxmat <- rbind(boot_mfxmat, t(boot_mfx))
}
  boot_mfxmat <- boot_mfxmat[-1, ]
  boot_mfxmatse <- c(sd(boot_mfxmat[ , 1]), sd(boot_mfxmat[ , 2]),
                     sd(boot_mfxmat[ , 3]), sd(boot_mfxmat[ , 4]))


  print(boot_mfxmatse)
```

```
## [1] 0.0101374468 0.0044747426 0.0003682493 0.0065287183
```