Quantitative Economics_HW3

March 8, 2022

In the name of God

1 Regression: Behind the Scene

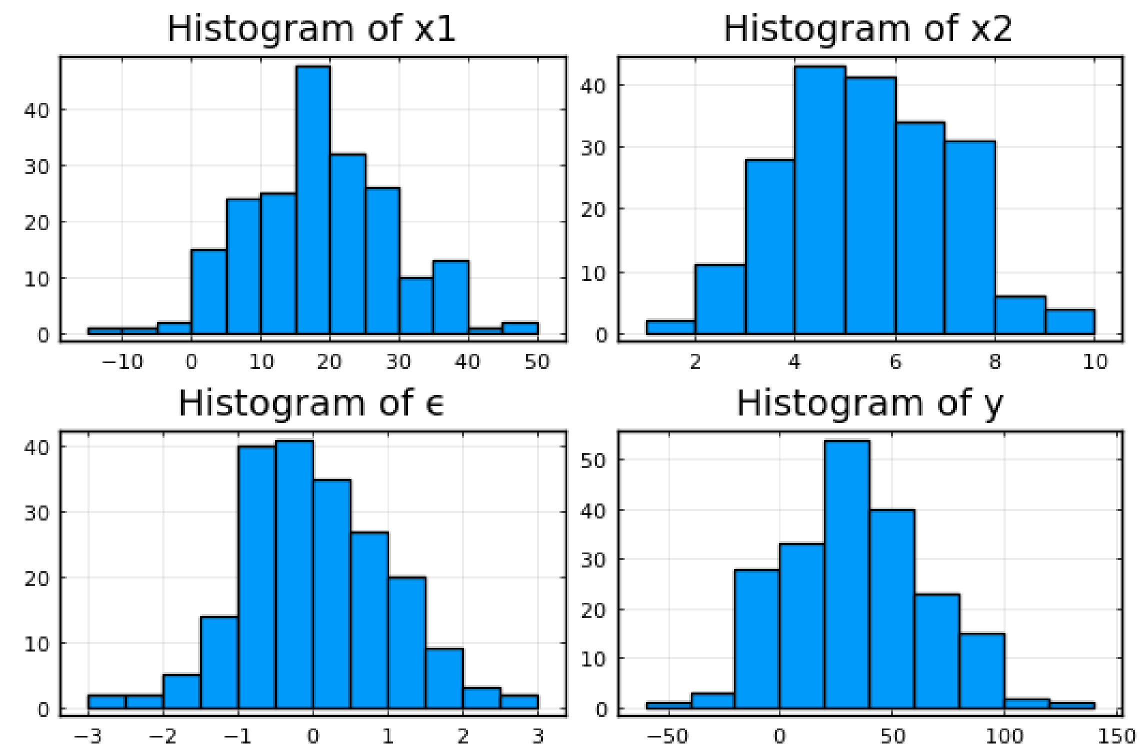
1.1 Data Generating Process

[1]:
```
using Random, Distributions, Plots, LinearAlgebra, StatsPlots, GLM, Optim, DataFrames, KernelDensity, HypothesisTests
pyplot();
```

[2]:
```
plots = []
Random.seed!(1395)
N = 200
x1 = rand(Normal(20,10), N)
x2 = rand(Binomial(10,0.5), N)
  = rand(Normal(0,1), N)
y = 3x1 - 5x2 +   .+ 2
print("mean of y: $(mean(y))")
p1 = histogram(x1, title = "Histogram of x1")
push!(plots,p1)
p2 = histogram(x2, title = "Histogram of x2")
push!(plots,p2)
p3 = histogram( , title = "Histogram of  ")
push!(plots,p3)
p4 = histogram(y, title = "Histogram of y")
push!(plots,p4)
plot(plots..., legend=false, framestyle = :box)
```

```
mean of y: 33.581650759707166
```

[2]:




```
 Warning: `vendor()` is deprecated, use `BLAS.get_config()` and inspect the
output instead
   caller = npyinitialize() at numpy.jl:67
 @ PyCall C:\Users\ASUS\.julia\packages\PyCall\L0fLP\src\numpy.jl:67
```

1.2 Ordinary Least Squares (OLS)

$$y = X\beta + u$$

In the multiple regression context, in order to obtain the parameter estimates, $ _1, _2,... _k,$ the RSS would be minimised with respect to all the elements of $ $. Now the residuals are expressed in a vector:

$$\hat{u} = \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ . \\ . \\ . \\ \hat{u}_T \end{bmatrix}$$

The RSS is still the relevant loss function, and would be given in a matrix notation:

$$L = \hat{u}'\hat{u} = \begin{bmatrix} \hat{u}_1 & \hat{u}_2 & . & . & . & \hat{u}_T \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ . \\ . \\ . \\ \hat{u}_T \end{bmatrix} = \sum_{t=1}^{T} \hat{u}_t$$

Denoting the vector of estimated parameters as $\hat{\beta}$, it is also possible to write:

$$L = \hat{u}'\hat{u} = (y - X\hat{\beta})'(y - X\hat{\beta}) = yy' - \hat{\beta}'X'y - y'X\hat{\beta} + \hat{\beta}'X'X\hat{\beta} = y'y - 2\hat{\beta}'X'y + \hat{\beta}'X'X\hat{\beta}$$

The first order condition:

$$\frac{\partial L}{\partial \hat{\beta}} = 0 \Rightarrow -2X'y + 2X'X\hat{\beta} = 0 \Rightarrow X'y = X'X\hat{\beta}$$

Pre-multiplying both sides by the $(X'X)^{-1}$

$$\hat{\beta} = (X'X)^{-1}X'y = \begin{bmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \\ . \\ . \\ . \\ \hat{\beta}_K \end{bmatrix}$$

Dimension of y: $T \times 1$

Dimension of X: $T \times K$

Dimension of β: $K \times 1$

1.3 Maximum Likelihood (ML)

1.4 1.3 Maximum Likelihood (ML)

1.4.1 1 & 2.

Let

$$X_1, X_2, ..., X_n$$

be a random sample from the population distribution

$$f(x;\theta)$$

Because of the random sampling assumption, the joint distribution of

$$X_1, X_2, ..., X_n$$

is simply the product of the densities:

$$f(x_1;\theta)f(x_2;\theta)...f(x_n;\theta)$$

Now, we can define the likelihood function as

$$L(\theta; X_1, X_2, ..., X_n) = f(x_1;\theta)f(x_2;\theta)...f(x_n;\theta)$$

so the log-likelihood function is:

$$\mathscr{L}(\theta) = log[L(\theta; X_1, X_2, ..., X_n)]$$

thus the log likelihood for the whole samples is:

$$\mathscr{L}(\theta) = \sum_{i=1}^{n} log[f(x_i;\theta)] = \sum_{i=1}^{n} (\theta; X_i)$$

and obviously the log-likelihood contribution of a simple observation i is:

$$log(f(x_i;\theta)) = (\theta; x_i)$$

1.4.2 3

as we know, we considered a model:

$$Y = X\beta + \epsilon$$

where random noise variables (epsilon) are i.i.d. and N(0, sigma^2).

we can rewrite the model in non-matrix form as follow:

$$Y_i = \beta_1 X_{i1} + ... + \beta_n X_{in} + \epsilon$$

we know that the P.D.F of Y is:

$$f_i(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2\sigma^2}(x - \beta_1 X_{i1} - ... - \beta_n X_{in})^2)$$

thus, the likelihood function is:

$$\prod_{i=1}^{n} f_i(Y_i) = (\frac{1}{\sqrt{2\pi}\sigma})^n \exp(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(Y_i - \beta_1 X_{i1} - ... - \beta_n X_{in})^2)$$

$$= (\frac{1}{\sqrt{2\pi}\sigma})^n \exp(-\frac{1}{2\sigma^2}|Y - X\beta|^2)$$

To maximize the likelihood function, first, we need to minimize |Y − X(beta)|^2. If we rewrite the norm squared using scalar product:

$$|Y - X\beta|^2 = (Y - \sum_{i=1}^{n}\beta_i X_i, Y - \sum_{i=1}^{n}\beta_i X_i)$$

$$= (Y, Y) - 2\sum_{i=1}^{n}\beta_i (Y, X_i) + \sum_{i,j=1}^{n}\beta_i\beta_j (X_i, X_j)$$

then setting the derivatives in each (beta_i) equal to zero

$$-2(Y, X_i) + 2\sum_{j=1}^{n}\beta_j (X_i, X_j) = 0$$

we get

$$(Y, X_i) = \sum_{i=1}^{n}\beta_j (X_i, X_j)$$

for all

$$i \leq n$$

In matrix notations this can be written as

$$X'Y = X'X\beta$$

So we can solve for beta to get the MLE

$$\hat{\beta}_{MLE} = (X'X)^{-1}X'Y$$

therefore we show that:

$$\hat{\beta}_{MLE} = \hat{\beta}_{OLS}$$

1.4.3 4

as we see above we optimize K+1 variables. there is no constraint!! there is always a beta vector if [x'x]^(-1) is available.

1.4.4 5

It is now easy to minimize over sigma to get

$$\hat{\sigma}^2 = \frac{1}{n}|Y - X\hat{\beta}|^2 = \frac{1}{n}|Y - X(X'X)^{-1}X'Y|^2$$

or we can write down:

1.5 Estimation

[3]:
```
Random.seed!(1395)
N = 10000
n = [10 20 50 100 1000 10000]
x1 = rand(Normal(20,10), N)
x2 = rand(Binomial(10,0.5), N)
  = rand(Normal(0,1), N)
y = 3x1 - 5x2 +   .+ 2;
```

1.5.1 OLS

[4]:
```
function OLS(DF, Formula::AbstractTerm)
    ols = lm(Formula, DF)
    betas = coeftable(ols).cols[1]
    return betas
end;
```

[5]:
```
 _py = DataFrame()
 _py."n" = Int64[]
 _py."intercept" = Float64[]
 _py." 1" = Float64[]
 _py." 2" = Float64[]
for i in n
    df = DataFrame(X1=x1[1:i], X2=x2[1:i], Y=y[1:i])
    formula = @formula(Y ~ X1 + X2)
    nn = [i]
    beta = OLS(df, formula)
    row = vcat(nn, beta)
    push!( _py, row)
end
println( _py)
```

```
6×4 DataFrame
 Row   n      intercept   1        2


       Int64  Float64    Float64

Float64

   1      10    3.34921  2.93925  -5.07858
   2      20    3.18787  2.95672  -5.0499
   3      50    2.90102  2.99593  -5.12861
   4     100    2.37409  2.99295  -5.01694
   5    1000    1.88928  2.9998   -4.97407
   6   10000    1.94001  3.00192  -4.99611
```

1.5.2 Algebraic

[6]:
```julia
_algebra = DataFrame()
_algebra."n" = Int64[]
_algebra."intercept" = Float64[]
_algebra." 1" = Float64[]
_algebra." 2" = Float64[]
for i in n
    X = zeros((i,3))
    X[:,1] = ones(i, 1)
    X[:,2] = x1[1:i]
    X[:,3] = x2[1:i]
    Y = y[1:i]
    nn = [i]
    beta = ((X'X)^(-1))X'Y
    row = vcat(nn, beta)
    push!( _algebra, row)
end
println( _algebra)
```


```
6×4 DataFrame
 Row   n      intercept   1        2


       Int64  Float64    Float64

Float64

   1      10    3.34921  2.93925  -5.07858
   2      20    3.18787  2.95672  -5.0499
   3      50    2.90102  2.99593  -5.12861
   4     100    2.37409  2.99295  -5.01694
   5    1000    1.88928  2.9998   -4.97407
   6   10000    1.94001  3.00192  -4.99611
```

1.5.3 SSR

[64]:
```
_optim = DataFrame()
_optim."n" = Int64[]
_optim."intercept" = Float64[]
_optim." 1" = Float64[]
_optim." 2" = Float64[]
for i in n
    X0 = ones(i, 1)
    X1 = x1[1:i]
    X2 = x2[1:i]
    Y = y[1:i]
    SSR( ) = sum((Y .- ( [2]X1 .+  [3]X2 .+  [1]X0)).^2)
    opt = optimize(SSR, [0.0, 0.0, 0.0])
    nn = [i]
    beta = opt.minimizer
    row = vcat(nn, beta)
    push!( _optim, row)
end
println( _optim)
```

```
6×4 DataFrame
 Row   n      intercept   1        2


      Int64  Float64    Float64

Float64


   1      10    3.34911  2.93925  -5.07856
   2      20    3.18792  2.95672  -5.04991
   3      50    2.90106  2.99592  -5.12862
   4     100    2.37411  2.99295  -5.01694
   5    1000    1.88928  2.9998   -4.97407
   6   10000    1.94001  3.00192  -4.99611
```

1.5.4 ML

[44]:
```
function MLE(x_1,x_2,y,n)
     =1
    X=[ones(size(x_1)) x_1 x_2]
    f(b) = -(sum(-1/(2*  ^2)*(y .- b[1]ones(size(x_1)) .- b[2]x_1 .- b[3]x_2).
 ↪^2) - n/2*(log(2 )+2log( )));
     _opt = optimize(f, [0.0, 0.0, 0.0])
    return  _opt.minimizer
end
```

[44]: `MLE (generic function with 1 method)`

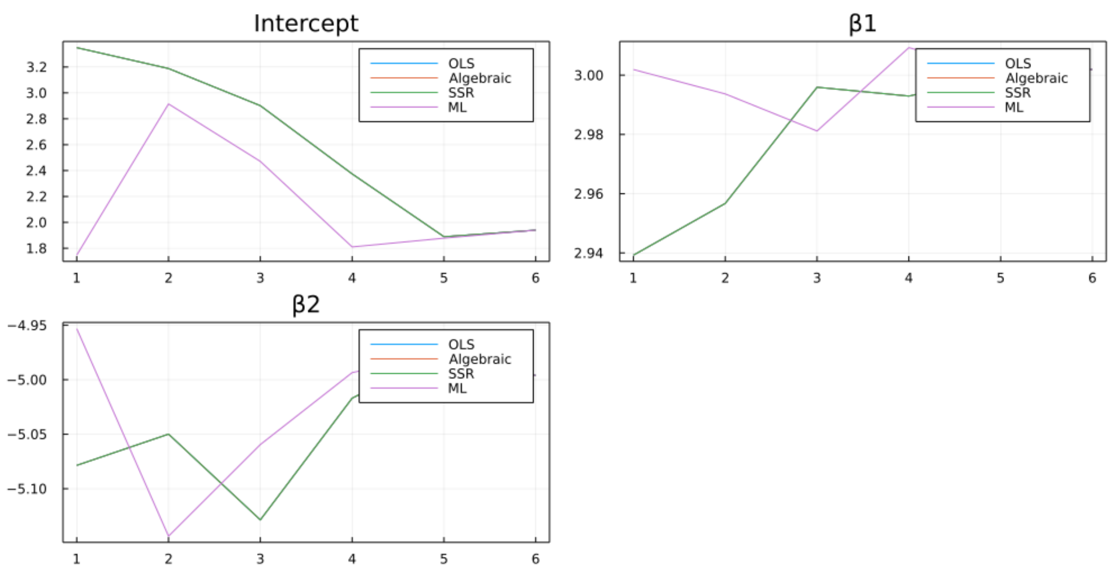
[45]:
```
 _ML = DataFrame()
 _ML."n" = Int64[]
 _ML."intercept" = Float64[]
 _ML." 1" = Float64[]
 _ML." 2" = Float64[]

for i in n
    Random.seed!(1395)
    x1 = rand(Normal(20,10), i)
    x2 = rand(Binomial(10,0.5), i)
      = rand(Normal(0,1), i)
    y = 3x1 .- 5x2 .+   .+ 2;
    temp = MLE(x1,x2,y,i)
    append!( _ML."n",i)
    append!( _ML."intercept",temp[1])
    append!( _ML." 1",temp[2])
    append!( _ML." 2",temp[3])
end
println( _ML)
```

```
6×4 DataFrame
 Row   n      intercept   1        2


       Int64  Float64    Float64

Float64

   1      10    1.74671  3.0019   -4.95315
   2      20    2.9146   2.99367  -5.14354
   3      50    2.47061  2.98116  -5.05948
   4     100    1.81032  3.00929  -4.99348
   5    1000    1.87795  3.00092  -4.97479
   6   10000    1.94001  3.00192  -4.99611
```

[71]:
```
plots = []
gr(fmt = :png, size = (1000, 500))
p1 = plot( _py[:,2], title= " Intercept ", label= "OLS")
plot!(p1, _algebra[:,2], label= "Algebraic" )
plot!(p1, _optim[:,2], label= "SSR")
plot!(p1, _ML[:,2], label= "ML")
push!(plots,p1)
p2 = plot( _py[:,3], title= "  1 ", label= "OLS")
plot!(p2, _algebra[:,3], label= "Algebraic" )
plot!(p2, _optim[:,3], label= "SSR")
plot!(p2, _ML[:,3], label= "ML")
push!(plots,p2)
p3 = plot( _py[:,4], title= "  2 ", label= "OLS")
```

```
plot!(p3, _algebra[:,4], label= "Algebraic" )
plot!(p3, _optim[:,4], label= "SSR")
plot!(p3, _ML[:,4], label= "ML")
push!(plots,p3)
plot(plots..., framestyle = :box)
```

[71]:



[ ]:

```

```

2 Monte-Carlo Simulation

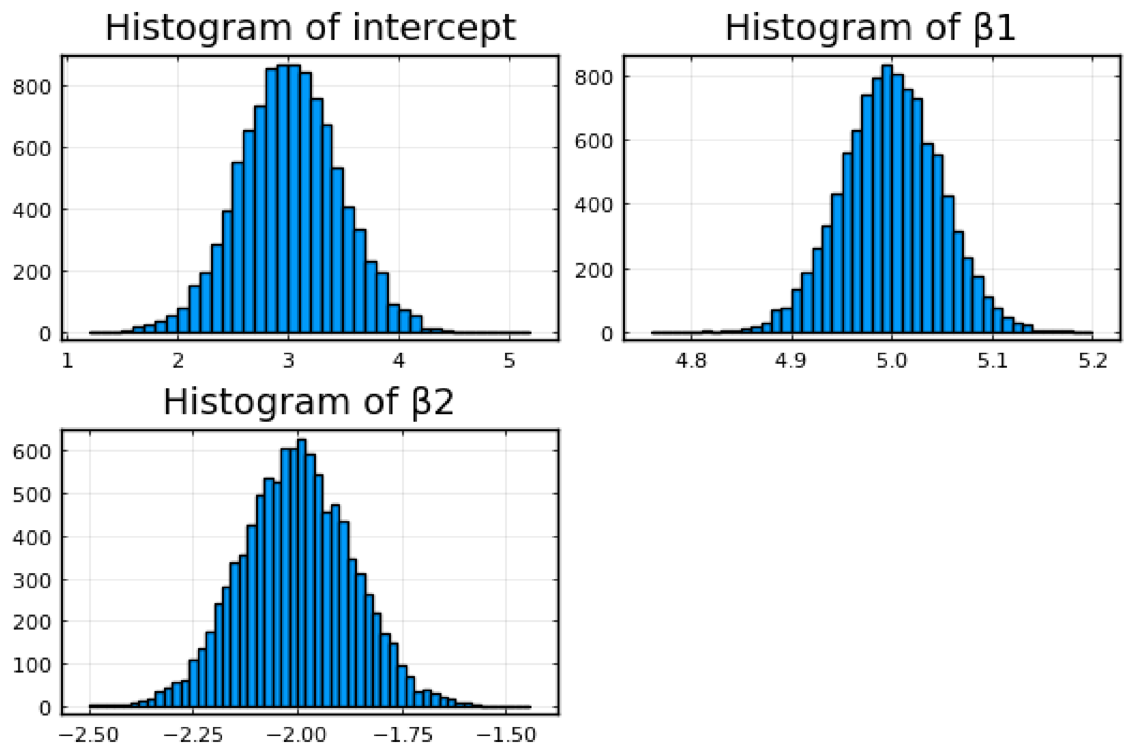
2.1 Small-Sample Properties

[8]:

```
  = DataFrame()
 ."intercept" = Float64[]
 ." 1" = Float64[]
 ." 2" = Float64[]
R = 10000
N = 50
plots = []
for r in 1:R
    x1 = rand(Normal(7,3), N)
    x2 = rand(Binomial(5,0.4), N)
      = rand(Normal(0,1), N)
    y = 5x1 - 2x2 +   .+ 3
    df = DataFrame(X1=x1, X2=x2, Y=y)
    formula = @formula(Y ~ X1 + X2)
    row = OLS(df, formula)
    push!( , row)
```

```
end
for i in names( )
    p = histogram( [:,i], title = "Histogram of $(i)")
    push!(plots,p)
end
plot(plots..., legend=false, framestyle = :box)
```

[8]:



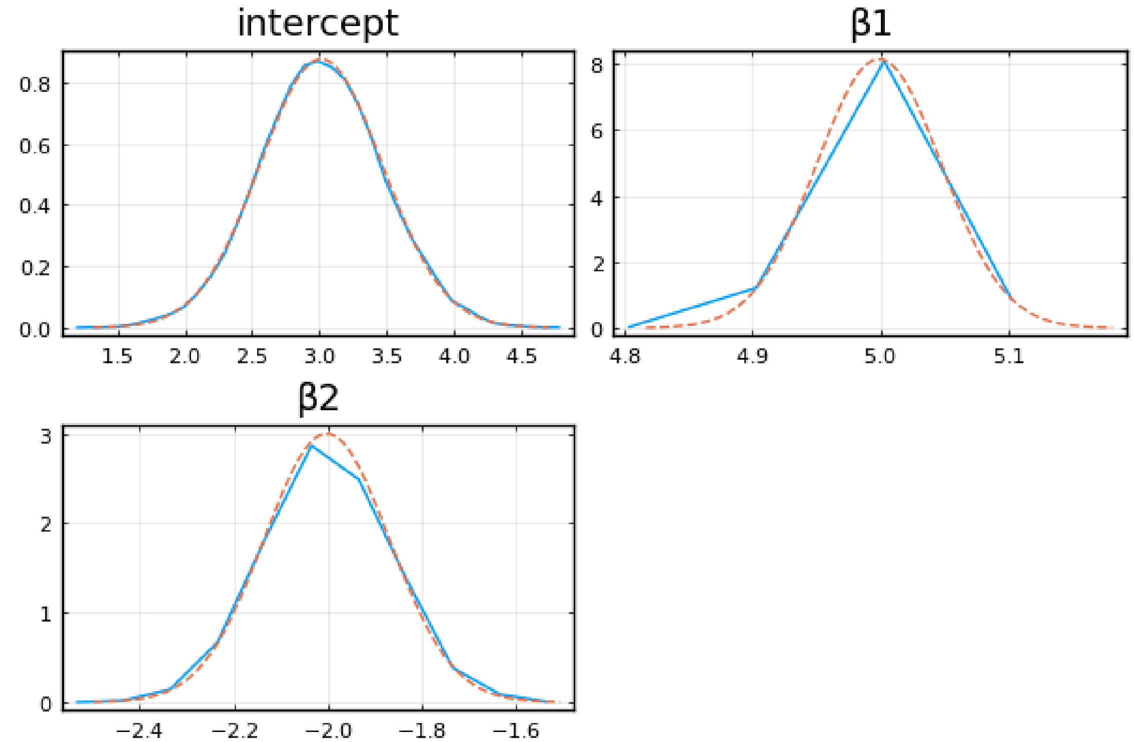
[9]:

```
plots = []
for i in names( )
      = mean( [:,i])
      = std( [:,i])
    x =  -4 :0.1: +4
    p = plot(x, pdf(kde( [:,i]), x), title="$(i)")
    plot!(p, Normal(mean( [:,i]),std( [:,i])), linestyle = :dash)
    push!(plots,p)
    ks = ExactOneSampleKSTest( [:,i], Normal( , ))
    pv = pvalue(ks)
    println("P-value of KSTest for $(i) distribution: $(pv)")
end
plot(plots..., legend=false, framestyle = :box)
```

```
P-value of KSTest for intercept distribution: 0.8625735041988462
P-value of KSTest for  1 distribution: 0.7936005325008184
```

```
P-value of KSTest for  2 distribution: 0.6011779828772568
```

[9]:



2.2 Asymptotic versus Small Sample

[10]:

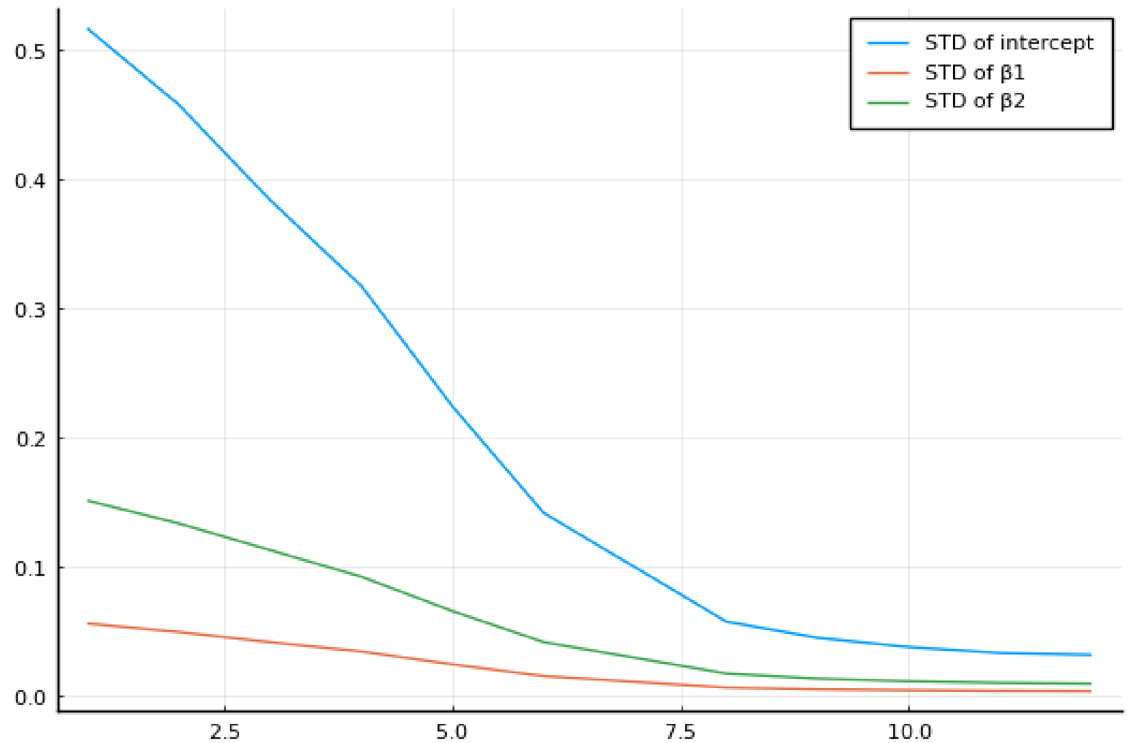
```
function Q1(N)
    R = 10000
    intcep = zeros(R,1)
    b1 = zeros(R,1)
    b2 = zeros(R,1)
    for r in 1:R
        x1 = rand(Normal(7,3), N)
        x2 = rand(Binomial(5,0.4), N)
          = rand(Normal(0,1), N)
        y = 5x1 - 2x2 +   .+ 3
        df = DataFrame(X1=x1, X2=x2, Y=y)
        formula = @formula(Y ~ X1 + X2)
        intcep[r] = OLS(df, formula)[1]
        b1[r] = OLS(df, formula)[2]
        b2[r] = OLS(df, formula)[3]
    end
    return intcep, b1, b2
end
```

[10]: Q1 (generic function with 1 method)

[11]:
```
it = [40 50 70 100 200 500 1000 3000 5000 7000 9000 10000]
l = length(it)
std0 = zeros(l,1)
std1 = zeros(l,1)
std2 = zeros(l,1)
j=1
for i in it
    B = Q1(i)
    std0[j] = std(B[1])
    std1[j] = std(B[2])
    std2[j] = std(B[3])
    j += 1
end
```

[12]:
```
p = plot(std0, label="STD of intercept")
plot!(p, std1, label="STD of  1" )
plot!(p, std2, label="STD of  2" )
```

[12]:



As we increase the sample size, distribution of estimated parameters goes to a Normal distribution and its variance decreases.

2.3 True Size of Test

[14]:
```
R = 10000
N = 50
  = 0
for r in 1:R
    x1 = rand(Normal(7,3), N)
    x2 = rand(Binomial(5,0.4), N)
       = rand(Normal(0,1), N)
    y = 5x1 - 2x2 +   .+ 3
    df = DataFrame(X1=x1, X2=x2, Y=y)
    formula = @formula(Y ~ X1 + X2)
    ols = lm(formula, df)
    STD = coeftable(ols).cols[2][2]
     1 = coeftable(ols).cols[1][2]
    t = abs(( 1-5)/STD)
    if (t>2.01174)
           += 1
    end
end
println("True size of test is $( /R)")
```

```
True size of test is 0.0489
```

2.4 Number of Replications

Variance of this binomial trial:

Mean of Binomial distribution $= np$

Variance of Binomial distribution $= np(1-p)$, so:

$$np = \alpha \Rightarrow p = \frac{\alpha}{n} \Rightarrow \sigma^2 = n(\frac{\alpha}{n})(1-\frac{\alpha}{n}) = \alpha(1-\frac{\alpha}{n})$$

Confidence interval:

$$CI = \hat{\alpha} \pm t_{n,0.95}\frac{S}{\sqrt{n}}$$

[15]:
```
R = 100
N = 50
Sum = zeros(1,2000)
for j in 1:2000
       = 0
    for r in 1:R
        x1 = rand(Normal(7,3), N)
```

```
        x2 = rand(Binomial(5,0.4), N)
         = rand(Normal(0,1), N)
        y = 5x1 - 2x2 +  .+ 3
        df = DataFrame(X1=x1, X2=x2, Y=y)
        formula = @formula(Y ~ X1 + X2)
        ols = lm(formula, df)
        STD = coeftable(ols).cols[2][2]
         1 = coeftable(ols).cols[1][2]
        t = abs(( 1-5)/STD)
        if (t>2.01174)
             += 1
        end
    end
    Sum[j] =  /R
end
low = round(mean(Sum)-1.96*std(Sum), digits=3)
up = round(mean(Sum)+1.96*std(Sum), digits=3)
println("The 95% confidence interval for  =0.05 is: ($(low), $(up))")
```

```
The 95% confidence interval for  =0.05 is: (0.008, 0.093)
```

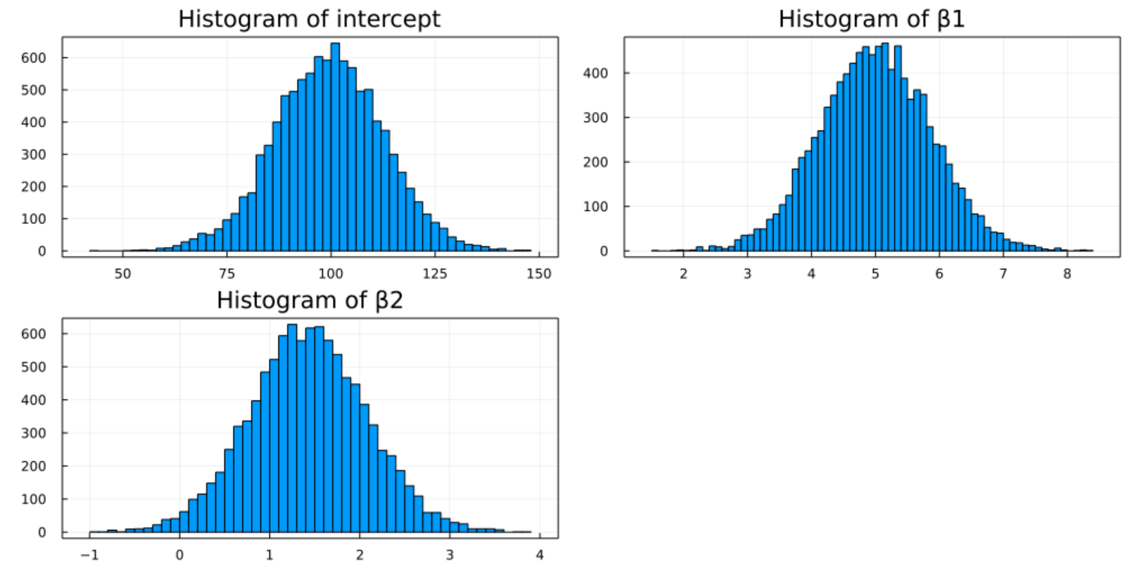
2.5 Endogeneity

2.5.1 Repetition of part 1

[79]:
```
 _end = DataFrame()
 _end."intercept" = Float64[]
 _end." 1" = Float64[]
 _end." 2" = Float64[]
R = 10000
N = 50
plots = []
for r in 1:R
    x1 = rand(Normal(7,3), N)
    z = rand(Binomial(20,0.7), N)
    x2 = zeros(N,1)
    for ii in 1:N
        x2[ii] = rand(Binomial(3*z[ii],0.4), 1)[1]
    end
    x2 = vec(x2)
     2 = rand(Normal(0,1), N)
      = 11z +  2
    y = 5x1 - 2x2 +  .+ 3
    df = DataFrame(X1=x1, X2=x2, Y=y)
    formula = @formula(Y ~ X1 + X2)
    row = OLS(df, formula)
    push!( _end, row)
end
```

```
for i in names( _end)
    p = histogram( _end[:,i], title = "Histogram of $(i)")
    push!(plots,p)
end
plot(plots..., legend=false, framestyle = :box)
```

[79]:

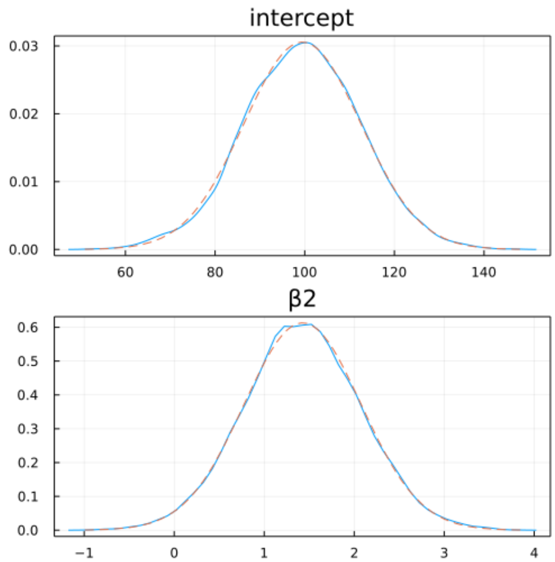


[80]:

```
plots = []
for i in names( _end)
      = mean( _end[:,i])
      = std( _end[:,i])
    x =  -4 :0.1: +4
    p = plot(x, pdf(kde( _end[:,i]), x), title="$(i)")
    plot!(p, Normal(mean( _end[:,i]),std( _end[:,i])), linestyle = :dash)
    push!(plots,p)
    ks = ExactOneSampleKSTest( _end[:,i], Normal( , ))
    pv = pvalue(ks)
    println("P-value of KSTest for $(i) distribution: $(pv)")
end
plot(plots..., legend=false, framestyle = :box)
```

```
P-value of KSTest for intercept distribution: 0.5569972035066583
P-value of KSTest for  1 distribution: 0.9877173865115912
P-value of KSTest for  2 distribution: 0.4309288939961642
```

[80]:



2.5.2 Repetition of part 2

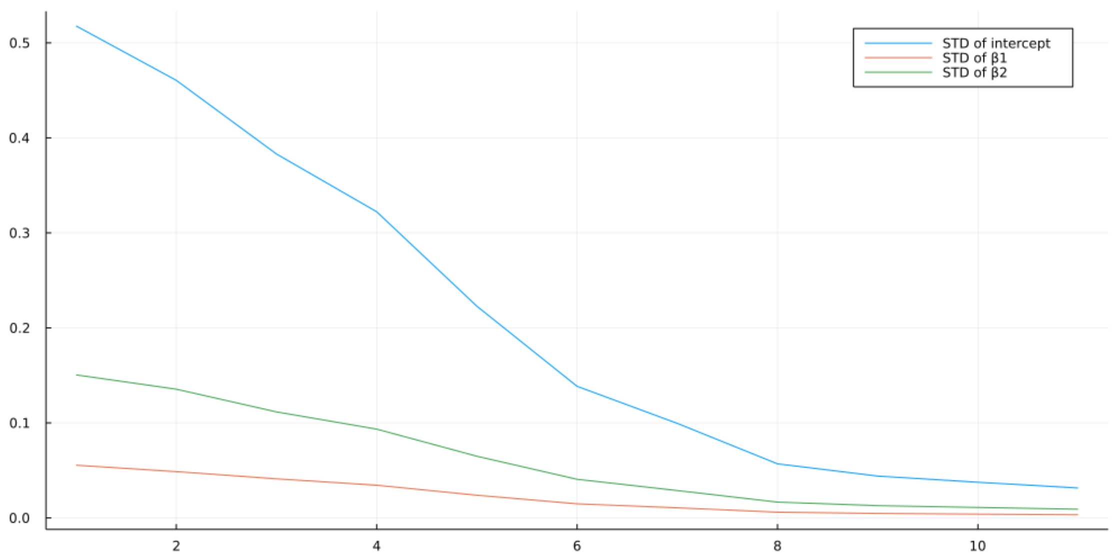
[81]:
```
function Q5(N)
    R = 10000
    intcep = zeros(R,1)
    b1 = zeros(R,1)
    b2 = zeros(R,1)
    for r in 1:R
        x1 = rand(Normal(7,3), N)
        z = rand(Binomial(20,0.7), N)
        x2 = zeros(N,1)
        for ii in 1:N
            x2[ii] = rand(Binomial(3*z[ii],0.4), 1)[1]
        end
        x2 = vec(x2)
         2 = rand(Normal(0,1), N)
          = 11z +  2
        y = 5x1 - 2x2 +   .+ 3
        df = DataFrame(X1=x1, X2=x2, Y=y)
        formula = @formula(Y ~ X1 + X2)
        intcep[r] = OLS(df, formula)[1]
        b1[r] = OLS(df, formula)[2]
        b2[r] = OLS(df, formula)[3]
    end
    return intcep, b1, b2
end
```

[81]:
```
Q5 (generic function with 1 method)
```

[82]:
```
it = [40 50 70 100 200 500 1000 3000 5000 7000  10000]
l = length(it)
std0 = zeros(l,1)
std1 = zeros(l,1)
std2 = zeros(l,1)
j=1
for i in it
    B = Q1(i)
    std0[j] = std(B[1])
    std1[j] = std(B[2])
    std2[j] = std(B[3])
    j += 1
end
```

[83]:
```
p = plot(std0, label="STD of intercept")
plot!(p, std1, label="STD of  1" )
plot!(p, std2, label="STD of  2" )
```

[83]:



2.5.3 Repetition of part 3

[92]:
```
R = 10000
N = 50
  = 0
for r in 1:R
    x1 = rand(Normal(7,3), N)
    z = rand(Binomial(20,0.7), N)
    x2 = zeros(N,1)
    for ii in 1:N
```

```
        x2[ii] = rand(Binomial(3*z[ii],0.4), 1)[1]
    end
    x2 = vec(x2)
     2 = rand(Normal(0,1), N)
      = 11z +  2
    y = 5x1 - 2x2 +   .+ 3
    df = DataFrame(X1=x1, X2=x2, Y=y)
    formula = @formula(Y ~ X1 + X2)
    ols = lm(formula, df)
    STD = coeftable(ols).cols[2][2]
     1 = coeftable(ols).cols[1][2]
    t = abs(( 1-5)/STD)
    if (t>2.01174)
         += 1
    end
end
println("True size of test is $( /R)")
```

```
True size of test is 0.0507
```

OLS estimator splits the outcome into the “explained” part and “residual” part. we assume that these two-part are orthogonal. in other words, we assume these two-part are uncorrelated. but this is a strong assumption and it’s very hard to prove. there is 3 type of endogeneity: 1- omitted variable bias, 2- reverse causality, and 3- measurement error.

the important point is that this problem doesn’t get solved by increasing sample size and it’s a part of this estimation method!

3 Simulator Class

[152]:
```
abstract type Simulator end
struct Monte_Carlo <: Simulator
    h::Function
    g
end

function Simulation(Type::Monte_Carlo)
      = 0
    for i in 1:10^6
        samples = rand(Type.g)[1]
          += Type.h(samples)
    end
    return  /10^6
end
```

[152]:
```
Simulation (generic function with 1 method)
```

[153]:
```
f(x) = x^2
test1 = Monte_Carlo(f,Normal(0,1))
Simulation(test1)
```

[153]: 0.9994844978121924

4 Frequency Simulator Class

[181]:
```
struct Frequency_ <: Simulator
    h::Function
    g
    a::Float64
    b::Float64
end

function Simulation(Type::Frequency_)
    count=0
    for i in 1:10^6
        u,v = rand(Type.g(Type.a, Type.b),2)
        if Type.h(u,v)<0
            count+=1
        end
    end
    return count/10^6
end
```

[181]: Simulation (generic function with 2 methods)

[188]:
```
f2 = circle(x,y) = (x-0.5)^2 + (y-0.5)^2 - 0.5^2
test2 = Frequency_(f2,Normal(0,1),0,1)
```

[188]: Frequency_(circle, Normal{Float64}(=0.0, =1.0), 0.0, 1.0)

5 Important Sampling Simulator Class

[182]:
```
struct Important <: Simulator
    h::Function
    g
    p
end

function Simulation(Type::Important)
      = 0
    for i in 1:10^6
        w = Type.h .* Type.g ./ Type.p
```

```
        samples = rand(Type.p)[1]
          += w(samples)
    end
    return  /10^6
end
```

[182]: Simulation (generic function with 3 methods)

[192]:
```
f3(x) = x^2
f4(x) = x
test3 = Important(f,Normal(0,1),f4)
```

[192]: Important(f, Normal{Float64}(=0.0, =1.0), f4)