John Higgins
Programming Bootcamp
13 July, 2022

Note: All code and plots can be found here!

Problem 1.

Solution:

a) The regression table and R code to produce it is included below:

Table 1

	Dependent variable:
	logw
coll	0.515^{***}
	(0.001)
exper	0.040^{***}
	(0.0002)
exp2	-0.001^{***}
	(0.00000)
Constant	2.175^{***}
	(0.002)
Observations	1,864,558
R^2	0.153
Adjusted R^2	0.153
Residual Std. Error	0.585 (df = 1864554)
F Statistic	$112{,}542.800^{***}$ (df = 3; 1864554)
Note:	*p<0.1; **p<0.05; ***p<0.01

```
> colnames(lwage) <- c("logw", "coll", "exper")
> lwage$exp2 <- lwage$exper**2
> model <- lm(logw ~ coll + exper + exp2, data=lwage)
> summary(model)
> stargazer(model)
```

b) Let $\{X_1, \ldots, X_n\}$ be a random sample with $X_i \sim F(X \mid \theta)$, where θ is a parameter vector for the distribution F. Let $f(X \mid \theta)$ indicate the pdf of $F(X \mid \theta)$. The likelihood of the sample given parameter vector θ is given by

$$L(\theta) = \prod_{i=1}^{n} f(X_i \mid \theta)$$

The log-likelihood is thus

$$\ell(\theta) = \log(L(\theta)) = \sum_{i=1}^{n} \log(f(X_i \mid \theta))$$

Given that we assume $Y_i = \beta_0 + \beta_1 c_i + \beta_2 x_i + \beta_3 x_i^2 + \varepsilon_i$, where $\varepsilon \sim N(0, \sigma^2)$ is exogenous. It follows that $Y_i \sim N(X_i'\theta, \sigma^2)$, where $X_i = (1, c_i, x_i, x_i^2)$ is the observation of the exogenous variables and $\theta = (\beta_0, \beta_1, \beta_2, \beta_3, s^2)$ is a parameter vector. The log-likelihood of the random sample $\{Y_1, \ldots, Y_n\}$ given parameter vector guess $\theta = (b_0, b_1, b_2, b_3)$ and observed collection of exogenous variables $X = \{X_1, X_2, \ldots, X_n\}$ is therefore given by

$$\begin{aligned}
\ell(\theta) &= \sum_{i=1}^{n} \log\left[\frac{1}{s\sqrt{2\pi}} \exp\left(-\frac{1}{2s^2}\left(Y_i - X_i'\theta\right)\right)\right] \\
&= -n\log(s) - n\log(\sqrt{2\pi}) - \frac{1}{2s^2}\sum_{i=1}^{n}(Y_i - X_i'\theta)^2
\end{aligned}$$

c) After estimating the model via log likelihood, we obtain the following coefficient estimates and their coefficients. Note that these are virtually identical to the previous coefficients. Code is included at conclusion of problem.

Coef	Value	t-stat
β_0	2.175	342.05
β_1	0.515	175.66
β_2	0.040	70.60
β_3	-0.001	-54.81

Table 2: log-likelihood opt coefficients

d) The standard errors using the bootstrap method are 0.0065 for β_0, 0.0038 for β_1, 0.0006 for β_2, and 0.00001 for β_3. I did not get my SSCC account request approved in time, so I just ran the code on my desktop. I used 5 workers for the parallelized part of the code. Instead of taking bootstrap samples of size $n/2$, I took bootstrap samples of size $n/10$ (note: in keeping with standard procedure for the bootstrap, I sampled with replacement from the data). I ran into significant memory issues which made using samples of size $n/2$ basically stall my computer. When run with samples of size $n/10$, the code ran much faster. The single-core bootstrap procedure took 787 seconds to complete, whereas the parallelized version took 314 seconds.

```
using Distributed
addprocs(4)
@everywhere begin
cd(dirname(@__FILE__()))
using CSV
using DataFrames


using Optim, NLSolversBase, Random, SharedArrays
using LinearAlgebra: diag
Random.seed!(0);


function log_like(Xa, Ya, beta, log_sigma)
    n = length(Ya)
    sig = exp(log_sigma)
    llike = -n/2*log(2\pi) - n/2* log(sig^2) - (sum((Ya - Xa * beta)
   .^2) / (2*sig^2))
    llike = -llike
end

nvar=4

end
```

```
@everywhere function estimator(X_d2::Array{Float64}, Y_d2::Vector{Float64})
    n = length(Y_d2)
    nvar = 4

    func = TwiceDifferentiable(vars -> log_like(X_d2, Y_d2, vars[1:nvar], vars[nvar + 1]),
                                ones(nvar+1); autodiff=:forward);

    opt = optimize(func, ones(nvar+1))

    parameters = Optim.minimizer(opt)

    parameters[nvar+1] = exp(parameters[nvar+1])

    \beta = parameters[1:nvar]
    \beta
end

#bootstrap function (not parallelized)
@everywhere function bootstrapper(X_d::Array{Float64}, Y_d::Array{Float64}, sims::Int64)
    n = length(Y_d)
    #take smaller sample of data because I'm running into memory problems on my desktop
    n2 = Int(floor(n/10))
    #assign empty arrays for coefficient estimates
    b0, b1, b2, b3 = zeros(sims), zeros(sims), zeros(sims), zeros(sims)
    #run bootstrap samples
    for j=1:sims
        #new random seed
        Random.seed!(j);
```

```
        #choose random selection with replacement from data index range
        selected = rand(1:n, n2)
        #create empty arrays for bootstrap sample
        X_b, Y_b = zeros(n2,4), zeros(n2)
        #loop over bootstrap indices
        for i=1:n2
            #find the i'th element from the random sample
            ind = selected[i]
            #construct bootstrap samples
            X_b[i,:] = X_d[ind,:]
            Y_b[i,:] = Y_d[ind,:]
        end
        #estimate model using bootstrap sample
        \beta_b = estimator(X_b, Y_b)
        #progress tracking
        println(j/sims*100, "% finished!")
        #assign coefficients to array
        b0[j], b1[j], b2[j], b3[j] = \beta_b[1], \beta_b[2], \beta_b[3], \beta_b[4]
    end
    b0, b1, b2, b3
end

#parallelized bootstrap function
@everywhere function bootstrapper_par(X_d::Array{Float64}, Y_d::Array{Float64}, sims::Int64)
    n = length(Y_d)
    #take smaller sample of data because I'm running into memory problems on my desktop
    n2 = Int(floor(n/10))
    #assign empty shared arrays for coefficient estimates
    b0, b1, b2, b3 = SharedArray{Float64}(sims), SharedArray{Float64
```

```
}(sims), SharedArray{Float64}(sims), SharedArray{Float64}(sims)
    #run all bootstrap samples
    @sync @distributed for j=1:sims
        #new random seed
        Random.seed!(j);
        println(j)
        #choose random selection with replacement from data index range
        selected = rand(1:n, n2)
        #create empty arrays for bootstrap samples
        X_b, Y_b = zeros(n2,4), zeros(n2)
        #loop over indices of the bootstrap sample
        for i=1:n2
            #find the i'th element from the random sample
            ind = selected[i]
            #construct bootstrap samples
            X_b[i,:] = X_d[ind,:]
            Y_b[i,:] = Y_d[ind,:]
        end
        #estimate model with bootstrap sample
        \beta_b = estimator(X_b, Y_b)
        #assign coefficients
        b0[j], b1[j], b2[j], b3[j] = \beta_b[1], \beta_b[2], \beta_b[3], \beta_b[4]
    end
    b0, b1, b2, b3
end

#model estimator using log-likelihood
function est_model(X_d2::Array{Float64}, Y_d2::Vector{Float64})
    n = length(Y_d2)
    nvar = 4

```

```
    func = TwiceDifferentiable(vars -> log_like(X_d2, Y_d2, vars[1:nvar], vars[nvar + 1]),
                               ones(nvar+1); autodiff=:forward);

    opt = optimize(func, ones(nvar+1))

    parameters = Optim.minimizer(opt)

    parameters[nvar+1] = exp(parameters[nvar+1])

    numerical_hessian = hessian!(func,parameters)

    var_cov_matrix = inv(numerical_hessian)

    \beta = parameters[1:nvar]

    temp = diag(var_cov_matrix)
    temp1 = temp[1:nvar]

    t_stats = \beta./sqrt.(temp1)

    \beta, t_stats
end

#import data and make necessary transformations
lwage_pre = DataFrame(CSV.File("lwage.csv", header=0, types=Float64))
lwage_mat = Matrix(lwage_pre)
exp2 = lwage_mat[:,3].^2
lwage = hcat(lwage_mat, exp2)

#define X and Y matrices
@everywhere using ParallelDataTransfer
n = length(lwage[:,1]); sendto(workers(), n=n)
```

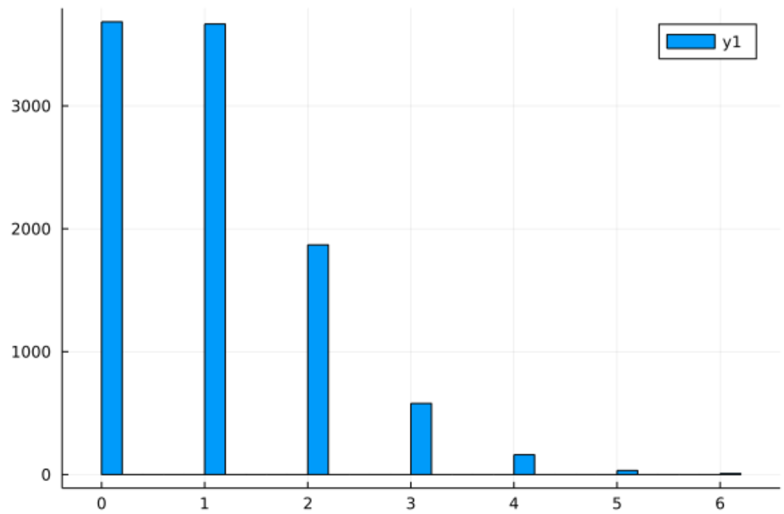
```
X = hcat(ones(n), lwage[1:end, 1:end .!= 1]); sendto(workers(), X=X)
Y = lwage[:, 1]; sendto(workers(), Y=Y)
nvar = 4

#estimate full model
coef, t_stat = est_model(X,Y)

#run bootstrap procedure
@elapsed b_0, b_1, b_2, b_3 = bootstrapper(X,Y, 100)
@elapsed b_0p, b_1p, b_2p, b_3p = bootstrapper_par(X,Y, 100)

#find standard errors
using Statistics
s0 = std(b_0)
s1 = std(b_1)
s2= std(b_2)
s3 = std(b_3)
s0p = std(b_0p)
s1p = std(b_1p)
s2p= std(b_2p)
s3p = std(b_3p)
```

$\square$

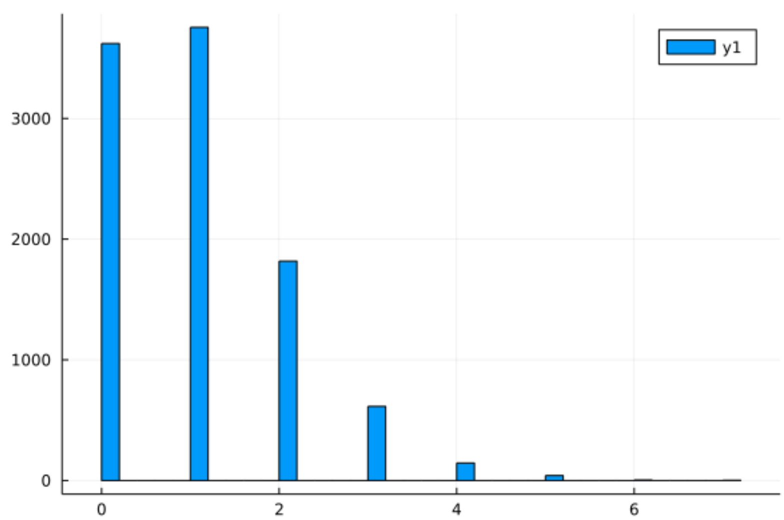
Problem 2.

Solution:

a) For $n = 10$ and 10,000 iterations, the average number of slips matched is 0.9998 and the histogram of the number of slips matched is included below:



b) For $n = 20$ and 10,000 iterations, the average number of slips matched is 1.0056 and the histogram of the number of slips matched is included below:



We can see that it is virtually identical to the distribution in part a). Code for this problem is included below:

```
using Plots
using Random

function draw_sim(peeps::Vector{Int64})
    #find the total number of slips to generate
```

```
    n = length(peeps)
    #since people draw at random without replacement, the process of drawing slips is exactly a random permutation of the range from 1 to n. This line finds one such permutation
    slips_drawn = randperm(n)
    #this next line creates a vector whose components are 1 if that player's number matches their drawn slip and 0 otherwise
    match = peeps - slips_drawn .==0
    #take the sum of all successes to get the total number of successes
    successes = sum(match)
    #return the total number of successes in this simulation
    successes
end

function hist_gen(n::Int64, sim::Int64)
    #initialize array of people
    people = collect(1:1:n)
    #initialize a blank array for successes
    sim_data = zeros(sim)
    #iterate simulations
    for i=1:sim
        #store the number of successes in iteration i
        sim_data[i] = draw_sim(people)
    end
    #return both the histogram and the mean number of matches (I included the mean because I was curious)
    hist,mean = histogram(sim_data), sum(sim_data)/sim
end

#generate histograms and means for n = 10 and 20 and save the histograms
hist_10,mean_10 = hist_gen(10, 10000)
```

```
savefig(hist_10, "hist10.png")
hist_20,mean_20 = hist_gen(20, 10000)
savefig(hist_20, "hist20.png")

mean_10
>>0.9998
mean_20
>>1.0056
```

□

Problem 3.

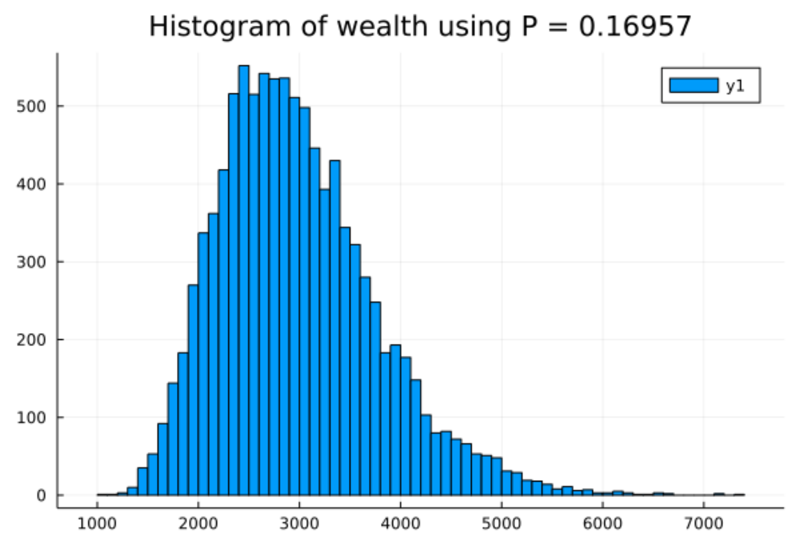
Solution: Note: throughout this problem I make the assumption that income is invested at the beginning of each period and that the goal is to have savings be 10x the earnings at age 67, not 10x the earnings at age 30.

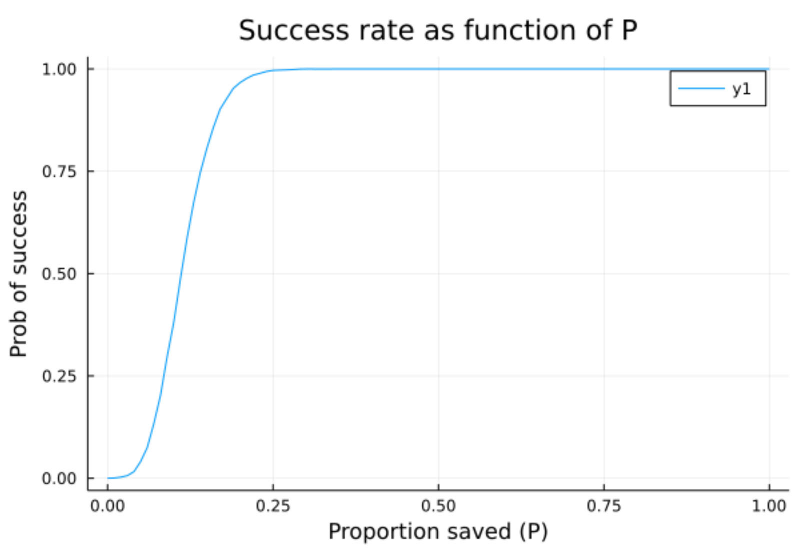
a) Given a deterministic 6% return on stocks and 3% annual raise, it is necessary to save 10.626% of one's earnings in order to meet this goal. To obtain this, I use the shooting method (described in part c).

b) Assuming stock returns are normally-distributed with mean of 6% and variance of 6% and raises are uniform on the interval $[0, 6]$, I find that across 1,000,000 simulations an agent saving 10.626% of their earnings each period will be successful in achieving the goal only 45.41% of the time. In other words, they will fail to meet their goal with probability 0.5459.

c) We now seek to determine the level of savings required for the agent to accomplish their goal 90% of the time. To do this, I used the shooting method. I start with a low value of 0, a high value of 1, and a guess of 0.5. If the success rate with the guessed p is higher than the target (0.90), then I lower the guess to the midpoint between the low value and the current guess and make the current guess the new high value. If the success rate is lower than the target, then I raise the guess to the midpoint between the current guess and the high value, and I make the current guess the new low value. Finally, once the difference between the high value and the current guess is lower than some specified tolerance level (I used 0.000001), the function stops (in an appeal to Cauchy convergence).

I find that in order to attain their savings goal 90% of the time, the agent must save 16.957% of their income each period. For kicks, I also included a histogram of wealth outcomes when the agent follows this savings rule, as well as a plot of success rate across 10,000 iterations vs the proportion saved.





```julia
using Distributed, SharedArrays, Distributions
addprocs(5)

@everywhere using SharedArrays, Distributions, Random

Random.seed!(0);

#function which takes initial earnings, savings, and saving rule as well as a sequence of returns and raises and outputs the amount of savings and whether the goal was satisfied
@everywhere function growth_path(earn::Float64, saved::Float64, prop::Float64, return_seq::Vector{Float64}, raise_seq::Vector{Float64})
    val = saved
    for i=1:37
        #Finds the total amount of current earnings that are saved in this period. I am assuming earnings that are not saved are consumed
        inv = prop*earn
        #Add amount invested in period t to the value of the portfolio. I assume earnings are invested at the start of the period - the wording in the question is ambiguous as to whether it is invested at the start or end of the period
        val += inv
        #find the value of the portfolio in the next period based on this period's return
        val = val*return_seq[i]
        #find next period's earnings based on this period's raise percent
        earn = earn*raise_seq[i]
    end
    #create boolean variable which determines whether the goal of having 10x of earnings in savings by retirement is attained. I assume that this means 10x of the agent's earnings when they are 67, not 10x of their earnings when they are 30.
    success = val >= 10*earn
```

```
    val, success
end


@everywhere function tester(determ::Int64, sims::Int64, earn::Float64, saved::Float64, prop::Float64)
    #create empty arrays for the wealth levels and success variable
    vals = zeros(sims)
    successes = zeros(sims)
    if determ==1
        vals, successes = growth_path(earn, saved, prop, fill(1.06, 37), fill(1.03, 37))
        #convert boolean to rate
        rate = Float64(successes)
        return rate,vals
    else
        return_dist = Normal(0.06, 0.06)
        raise_dist = Uniform(0.0, 0.06)
        #loop over simulation count
        for i=1:sims
            #draw new return and raise sequences following their specified distributions
            returns =1 .+ rand(return_dist, 37)
            raises = 1 .+ rand(raise_dist, 37)
            #store wealth level and outcome in their respective locations
            vals[i], successes[i] = growth_path(earn, saved, prop, returns, raises)
        end
        #sum outcomes and divide by number of simulations to get success rate
        rate = sum(successes)/sims
        return rate, vals
    end
```

```
end

#parallelized version of the above function, significant speed improvement
    for large number of simulations
@everywhere function tester2(determ::Int64, sims::Int64, earn::Float64,
   saved::Float64, prop::Float64)
    #check for whether we want the deterministic process or the random one
    if determ==1 #if we choose the deterministic process
        #find the ending wealth level and outcome variable
        vals, successes = growth_path(earn, saved, prop, fill(1.06, 37),
   fill(1.03, 37))
        #convert boolean to float
        rate = Float64(successes)
        return rate,vals
    elseif determ==0
        #initialize empty shared arrays
        vals = SharedArray{Float64}(sims,1)
        successes = SharedArray{Float64}(sims,1)
        #loop over simulation count
        @sync @distributed for i=1:sims
            #draw new return and raise sequences following their specified
    distributions
            returns =1 .+ rand(return_dist, 37)
            raises = 1 .+ rand(raise_dist, 37)
            #store wealth level and outcome in their respective locations
            vals[i], successes[i] = growth_path(earn, saved, prop, returns
   , raises)
        end
        #sum outcomes and divide by number of simulations to get success
   rate
        rate = sum(successes)/sims
        return rate, vals
    end
```

```
end

#@elapsed tester(10000000, 100.0, 100.0, 0.1, return_dist, raise_dist)
#@elapsed tester2(10000000, 100.0, 100.0, 0.1, return_dist, raise_dist)

@everywhere function binary_searcher(determ::Int64, target::Float64, tol::Float64, sims::Int64, earn::Float64, saved::Float64)
    p_guess = 0.5
    p_low = 0.0
    p_high = 1.0
    #continue until difference between max and guess is lower than the tolerance parameter
    while abs(p_high - p_guess) >= tol
        #find the success rate given the current guessed p
        succ_rate, val_data = tester2(determ, sims, earn, saved, p_guess)
        if succ_rate >= target #if success rate is too high, revise guess downward
            println("Too high!", p_guess, succ_rate)
            p_high = p_guess
            p_guess = (p_low + p_guess)/2
        elseif succ_rate < target #if success rate is too low, revise guess upward
            println("Too low!", p_guess, succ_rate)
            p_low = p_guess
            p_guess = (p_high + p_guess)/2
        end
    end
    #return the guessed p
    p_guess
end

#create distributions
return_dist = Normal(0.06, 0.06)
```

```
raise_dist = Uniform(0.0, 0.06)

#find savings rate required to achieve the goal in a deterministic setting
binary_searcher(1, 1.0, 0.0000001, 1, 100.0, 100.0)

#find probability of attaining goal in random setting using the answer to part a (p = 0.1062)
tester2(100000000,100.0, 100.0, 0.10626, return_dist, raise_dist)

#find savings rate required to achieve the goal when returns and raises are random
binary_searcher(0, 0.9, 0.0000001, 1000000, 100.0, 100.0)


#plot histogram of wealth using optimal saving rule
using Plots
wealth_hist = histogram(val_data_wrong, title = "Histogram of wealth using P = 0.16957")
savefig(wealth_hist, "wealthhist.png")

#find probability of attaining goal in random setting using the correct savings level - will be very close to 90%
succ_opt, val_data_opt = tester2(0,1000000,100.0, 100.0, 0.16957)

#creates plot of success rate vs p
p_grid = collect(0.0:0.01:1.0)
np = length(p_grid)
succ_p = zeros(np)
for i=1:np
    succ_vals, val_datas = tester2(0,1000000,100.0, 100.0, p_grid[i])
    succ_p[i] = succ_vals
end
using Plots
```

```
succ_plot = plot(p_grid, succ_p, title= "Success rate as function of P",
    xlabel="Proportion saved (P)", ylabel="Prob of success")
savefig(succ_plot, "succ_plot.png")
```

$\square$