1. The routine is coded in Julia.

   The obtained log likelihood is -6942.805.

   The transpose of the score is

$$\begin{bmatrix} -2605.9082518892865 \\ -556.3196848948379 \\ -1156.8594262530135 \\ -222.81767101773977 \\ -933.039979318137 \\ -1215.1317422401712 \\ -2109.626213790837 \\ -948.0740374410863 \\ -5049.875617650256 \\ -4534.790470404961 \\ -19401.89853086738 \\ -19164.659456830384 \\ -918.8553971099844 \\ -351.75306280921296 \\ -466.6888493111424 \\ -582.4690752990825 \\ -546.4113143620349 \end{bmatrix} \tag{1}$$

   The Hessian matrix is too large to display, but is computed in the routine.

2. We don't obtain the same results using the two approaches. The score especially is very different depending on the choice of method. The Euclidean norm of the difference between scores obtained by using the two approaches was around 2.3×10^5 while the norm between the two Hessians was 0.35.

3. The results obtained from implementing the Newton algorithm is displayed in part 4. The associated code is attached in the appendix. The *NewtonAlg* function within "functions.jl" file details the algorithm.

4. The computation speed between the three methods is compared below

Newton Method	838.489 ms
Quasi-Newton (BFGS)	4.703 s
Quasi-Newton (Simplex)	2.935 s

   The estimates of β under each of these methods is tabulated below. We can observe that the Newton Method yields estimates closest to the true values, and does so fastest among the three methods.

Newton	BFGS	Simplex
$\begin{bmatrix} -1.000 \\ 1.530e-7 \\ 8.4296e-8 \\ 1.086e-7 \\ 3.010e-8 \\ 1.569e-7 \\ -1.938e-8 \\ 4.086e-8 \\ 1.278e-7 \\ 5.152e-8 \\ -5.411e-8 \\ 3.651e-8 \\ 1.4033e-7 \\ 1.988e-7 \\ 1.170e-7 \\ 4.438e-8 \\ 2.374e-8 \end{bmatrix}$	$\begin{bmatrix} -6.056 \\ 0.867 \\ 0.527 \\ 0.595 \\ 0.163 \\ 0.871 \\ -0.052 \\ 0.215 \\ 1.007 \\ 0.335 \\ -0.284 \\ 0.189 \\ 0.758 \\ 1.152 \\ 0.770 \\ 0.379 \\ 0.2406 \end{bmatrix}$	$\begin{bmatrix} -1.953 \\ 0.686 \\ 0.318 \\ 0.411 \\ 0.008 \\ -0.591 \\ -0.074 \\ -0.379 \\ 0.279 \\ 0.523 \\ -0.481 \\ 0.233 \\ 0.680 \\ 0.344 \\ 0.064 \\ -0.407 \\ -0.486 \end{bmatrix}$

Appendix

The first codefile named "runfile.jl" runs the code.

```
#==
    This file conducts the analyses for JF's PS1
==#

using CSV, DataFrames, Optim, BenchmarkTools, Latexify
# We can use BenchmarkTools for better precision. Just need to replace time
# with btime. The runtime of the overall code get longer as btime runs the
# code multiple times to reduce noise

# include("./PS1b/JuliaCode/functions.jl")
include("../JuliaCode/functions.jl")
## load the mortgage data as a DataFrame
# df = DataFrame(CSV.File("../data/mortgage.csv"))

# Use this if you are loading data from the root folder.
df = DataFrame(CSV.File("PS1b/Data/mortgage.csv"))
#df = DataFrame(CSV.File("C:/Users/ryana/OneDrive/Documents/School/PhD Economics/Research/GitHub/ECON-899/PS1b/

## Separate data into independent variable matrix X and
## dependent variable vector Y
X = df[!,[:i_large_loan,:i_medium_loan,:rate_spread,
          :i_refinance,:age_r,:cltv,:dti, :cu,
          :first_mort_r,:score_0,:score_1, :i_FHA,
          :i_open_year2,:i_open_year3, :i_open_year4,
          :i_open_year5]] |> Matrix;

Y = df[!, :i_close_first_year]; #|> Matrix
```

```
## 1. Evaluate functions at β0 = −1 and β = 0
β = [-1; zeros(size(X, 2), 1)];
LL = likelihood(β, Y, X);


gβ = score(β, Y, X)
# The transpose of the score evaluated at β is
latexify(gβ')

H  = Hessian(X, β)
# The Hessian evaluated at β is
latexify(H)

## 2. Compare score and hessian from (1) with numerical
##    first and second derivative of the log-likelihood
gβ_num=∂F(β,Y,X)
# gβ_num=score_num(β,Y,X)
diff_gβ=gβ.-gβ_num

H_num=Find_H_num(β,Y,X)
diff_H=H-H_num


## 3. Write a routine that solves the maximum likelihood
##    using a Newton algorithm
@btime β_Newton = NewtonAlg(Y, X); #Newton(Y, X; β0 = β);
# β_Newton = NewtonAlg(Y, X);

## 4. Compare the solution and speed with  BFGS and Simplex
#f(b) = likelihood(b, Y, X);
#Optimize minimizes the function, so we need to use the negative of liklihood to maximize
println("\n For Quasi-Newton Methods:")
print("\n The BFGS algorithm takes")
# @btime β_BFGS    = optimize(b->-likelihood(b, Y, X), β, BFGS(),abs_tol=1e-12).minimizer
@btime    β_BFGS    = optimize(b->-likelihood(b, Y, X), β, method=BFGS(),
        f_tol=1e-32,g_tol=1e-32).minimizer

print("\n The Simplex algorithm takes")
@btime β_simplex = optimize(b->-likelihood(b, Y, X), β, NelderMead()).minimizer;
    # β_simplex = optimize(b->-likelihood(b, Y, X), β, method=NelderMead(),
    # f_tol=1e-32,g_tol=1e-32).minimizer
```

The second codefile named "functions.jl" contains the relevant functions.

```
#==
    This file defines functions used in JF's PS1
==#
using Optim

# Calculate log-likelihood at β
function likelihood(β, Y, X)

    X = [ones(size(X, 1), 1) X] # add constant to X

    return sum(Y.*log.(exp.(X*β) ./ (1 .+ exp.(X*β))) +
        (1 .- Y).*log.(1 ./ (1 .+ exp.(X*β))))
end # log-likelihood function

# calculate the log-likelihood score, given β
function score(β, Y, X)
```

```
    X = [ones(size(X, 1), 1) X] # add constant to X

    return sum((Y .- (exp.(X*β) ./ (1 .+ exp.(X*β)))) .* X, dims = 1)

end # end log-likelihood score

# Calculate the Hessian matrix given β
function Hessian(X, β)

    X = [ones(size(X, 1), 1) X] # add constant to X

    A = (exp.(X*β) ./ (1 .+ exp.(X*β))) .*
        (1 ./ (1 .+ exp.(X*β)))

    #==
    B = zeros(size(X,2), size(X,2), size(X,1))

    for i = 1:size(X,1)

        B[:,:,i] = A[i] .* X[i,:] * transpose(X[i,:])

    end

    dropdims(sum(B, dims = 3), dims = 3)
    ==#

    # Alternative method (saves memory):
    H = 0;
    for i = 1:size(X,1)
        H = H .+ (A[i] .* X[i,:] * transpose(X[i,:]))
    end

    return -H
end # Hessian matrix

####################################################################
#Calculate First Derivate numerically
function ∂F(β,Y,X;h=1e-5)
    ∂=zeros(length(β))
    for ii=1:length(β)
        hi=zeros(length(β))
        hi[ii]=copy(h)
        ∂[ii]=(likelihood(β.+h,Y,X)-likelihood(β,Y,X))/h
    end
    return transpose(∂)
end

## I think to calculate partial derivative, we should think about X.+h
## The following code yields this version, but, after column 2 becomes 0.0
## So, there may be errors.

function score_num(β,Y,X;h=1e-5)

    X1 = [ones(size(X, 1), 1) X] # add constant to X

    function likelihood2(β, Y, X)
        return sum(Y.*log.(exp.(X*β) ./ (1 .+ exp.(X*β))) +
            (1 .- Y).*log.(1 ./ (1 .+ exp.(X*β))))
    end # log-likelihood function

    partial = zeros(length(β))
    for i =1:length(β)
        X2=copy(X1)
        X2[:,i] .-= h
        partial[i]=(likelihood2(β,Y,X2)-likelihood2(β,Y,X1))/h
    end
    return transpose(partial)
```

```
end

######################################################################

#Calculate the Hessian numerically
function Find_H_num(β,Y,X;h=1e-5)
    H_num=zeros(length(β),length(β))
    d=1
    for i1=1:length(β)
        for i2=d:length(β)
            h1=zeros(length(β))
            h2=copy(h1)
            h1[i1], h2[i2] = copy(h),copy(h)
            #This formula was taken from http://www.holoborodko.com/pavel/2014/11/04/computing-mixed-derivative
            #H_num[i1,i2]=(likelihood(β.-h1.-h2,Y,X)+likelihood(β.+h1.+h2,Y,X)+
            #    likelihood(β.+h1.-h2,Y,X)+likelihood(β.-h1.+h2,Y,X))/(4*(h^2))
            #Alternate, more accurate formula also from the above link
            H_num[i1,i2]=( 8*(likelihood(β.+h1.-2 .*h2,Y,X)+likelihood(β.+2 .* h1.-h2,Y,X)+
                            likelihood(β.-2 .*h1.+h2,Y,X)+likelihood(β.-h1.+2 .*h2,Y,X))-
                            8*(likelihood(β.-h1.-2 .*h2,Y,X)+likelihood(β.-2 .* h1.-h2,Y,X)+
                            likelihood(β.+h1.+ 2 .*h2,Y,X)+likelihood(β.+ 2 .* h1.+h2,Y,X))-
                            (likelihood(β.+2 .*h1.-2 .*h2,Y,X)+likelihood(β.-2 .*
h1.+ 2 .*h2,Y,X)-
                            likelihood(β.-2 .*h1.- 2 .*h2,Y,X)-likelihood(β.+ 2 .*h1.+2 .*h2,Y,X))+
                            64*(likelihood(β.-h1.-h2,Y,X)+likelihood(β.+h1.+h2,Y,X)-
                            likelihood(β.+h1.-h2,Y,X)-likelihood(β.-h1.+h2,Y,X)))/(144*(h^2))
        end
        d+=1
    end
    #Exploit Hessian symmetry to find the remaining entries
    d=length(β)-1
    for i1=length(β):-1:2
        for i2=d:-1:1
            H_num[i1,i2]=H_num[i2,i1]
        end
        d-=1
    end
    return H_num
end
# Define the Newton convergence algorithm
function NewtonAlg(Y, X; β0::Matrix{Float64} = [-1.0; ones(size(X, 2), 1)],
        err::Float64 = 100.0, tol::Float64 = 1e-32, sk::Float64=1e-7)
    β_out=0
    iter=1;
    print("="^35,"\n","Newton's Method","\n")
    while err > tol

        # update β
        β_out = β0 - sk*inv(Hessian(X, β0))*transpose(score(β0, Y, X))
        #If you have made β_out NaN, you've gone too far
            while isnan((transpose(β_out)ones(size(X, 2)+1, 1))[1])
                sk=sk/10;
                β_out = β0 - sk*inv(Hessian(X, β0))*transpose(score(β0, Y, X))
            end
        # calculate error and update β0
        err_new = maximum(abs.(β_out - β0))
        β0 = copy(β_out)
        if iter % 5==0
            println("Newton Iteration $(iter) with error $(err_new)")
        end
        iter+=1
        #Update sk depending on whether things are going well or not
            if err_new<err
                sk=sk*2;
            else
                sk=sk/10
            end
```

```
            err=copy(err_new)
    end # err > tol loop

    # return converged β
    print("\nThe Newton algorithm takes")
    return β_out
end # Newton's algorithm
```