Logit

Suppose that agents can choose an action $y \in \{0,1\}$. The payoffs to each are

$$\begin{align*}
u_0(X,\epsilon) &= \epsilon_{i0} \\
u_1(X,\epsilon) &= x_i^\top \beta + \epsilon_{i1}
\end{align*}$$

The shocks $\epsilon_{i0},\epsilon_{i1}$ are distributed as iid Type-I extreme value with mean 0 and scale parameter 1. Agents choose $y_i=1$ if $u_1 \geq u_0$ and y_0 if $u_1 < u_0$

```
md"
# Logit

Suppose that agents can choose an action $y \in \{0,1\}$. The payoffs to each are

```math
\begin{align*}
u_0(X,\epsilon) &= \epsilon_{i0} \\
u_1(X,\epsilon) &= x_i^\top \beta + \epsilon_{i1}
\end{align*}
```

The shocks $\epsilon_{i0},\epsilon_{i1}$ are distributed as iid Type-I extreme value with mean 0 and scale parameter 1. Agents choose $y_i=1$ if $u_1 \geq u_0$ and y_0 if $u_1 < u_0$
"
```

Denote $cdf(z) = F(z)$. Let choice $y_i \in \{0,1\}$. For a symmetric distribution and a binary discrete choice model, we can use this shortcut (trick is in Greene's Econometrics tome, Greene 6th ed p. 779):

$$\log L(y|X) = \sum_i \log F\big(2(y_i-1)x_i^\top \beta\big)$$

Score is a vector

$$\nabla_\beta \log L(y_i|x_i) = \big[y_i - F\big(x_i^\top \beta\big)\big] x_i$$

Information matrix is

$$\left[\sum_i \nabla \log L_i \nabla \log L_i^\top\right] \to Var(\beta)$$

```
md"
Denote $cdf(z) = F(z)$. Let choice $y_i \in \{0,1\}$. For a symmetric distribution and a binary discrete choice model, we can use this shortcut (trick is in Greene's Econometrics tome, Greene 6th ed p. 779):
```math
```
```

```
\log L(y|X) = \sum_i \log F\left( 2(y_i-1) x_i^\top \beta \right)
```
Score is a vector
```math
\nabla_\beta \log L(y_i|x_i) = \left[y_i - F\left(x_i^\top\beta\right)\right]x_i
```
Information matrix is
```math
\left[\sum_i \nabla \log L_i \nabla \log L_i^\top\right] \to Var(\beta)
```
"
```

We can vectorize stuff to make it simpler. The . means element-by-element operations a la MATLAB/Julia. Define

$$\boldsymbol q \equiv 2.\boldsymbol y .- 1$$

Then

$$\log L(y|X) = \sum_i \log. F.\left(\boldsymbol q .* X\beta \right)$$

Matrix of scores

$$\frac{\partial \log L_i}{\partial \beta} = \left(y .- F.(X\beta)\right) .* X$$

Information matrix

$$\left(\frac{\partial \log L_i}{\partial \beta}\right)^\top \frac{\partial \log L_i}{\partial \beta} \to Var(\beta)$$

```
md"
We can vectorize stuff to make it simpler. The $.$ means element-by-element operations a la MATLAB/Julia. Define
```math
\boldsymbol q \equiv 2.\boldsymbol y .- 1
```
Then
```math
\log L(y|X) = \sum_i \log. F.\left(\boldsymbol q .* X\beta \right)
```
Matrix of scores
```math
\frac{\partial \log L_i}{\partial \beta} = \left(y .- F.(X\beta)\right) .* X
```
Information matrix
```math
\left(\frac{\partial \log L_i}{\partial \beta}\right)^\top \frac{\partial \log L_i}{\partial \beta} \to Var(\beta)
```

"
```
```

In the version below, we use the `Distributions.jl` package, which means we could actually change to a binary probit just by swapping out the distribution from `Logistic` to `Normal`.

Alternatively, for lower-level control, we can use the following functions from [`StatsFuns.jl`](https://github.com/JuliaStats/StatsFuns.jl). This is useful for more computationally intensive work with multinomial discrete choice.

```julia
logsumexp      # log(exp(x) + exp(y)) or log(sum(exp(x)))
softmax        # exp(x_i) / sum(exp(x)), for i
```

```
md"
In the version below, we use the `Distributions.jl` package, which means we could
actually change to a binary probit just by swapping out the distribution from
`Logistic` to `Normal`.

Alternatively, for lower-level control, we can use the following functions from
[`StatsFuns.jl`](https://github.com/JuliaStats/StatsFuns.jl). This is useful for more
computationally intensive work with multinomial discrete choice.

```julia
logsumexp      # log(exp(x) + exp(y)) or log(sum(exp(x)))
softmax        # exp(x_i) / sum(exp(x)), for i
```
"
```

```julia
begin
    # import the entire package
    using Optim
    using Distributions
    using DataFrames

    # import just a few functions
    using Random: seed!
    using StatsFuns: logsumexp, softmax
    using LinearAlgebra: diag
    using StatsBase: countmap

    # autodiff instead of finite diff?
    using FiniteDiff: finite_difference_gradient
end
```

```
MersenneTwister(1234)
```

```julia
# set seed for random # generator
seed!(1234)
```

```
1000×2 Matrix{Float64}:
 0.0  -2.89099
 0.0   1.2706
 0.0  -2.74435
 0.0  -2.06214
 0.0  -0.363533
 0.0   1.73134
 0.0  -0.725822
 ⋮
 0.0  -4.08263
 0.0   0.8717
```

```
 0.0  -5.16962
 0.0  -1.80468
 0.0  -0.734464
 0.0   3.2252
```

```
begin
    nobs = 1_000
    β = [1.0, -2.0, 1.0, 0.5]
    k = length(β)
    X = randn(nobs, k);

    # choice utilities
    u0 = zeros(nobs)
    u1 = X*β
    u = hcat(u0, u1)
end
```

```
prob_actions = 1000×2 Matrix{Float64}:
                 0.947399   0.0526007
                 0.219155   0.780845
                 0.939594   0.0604063
                 0.887168   0.112832
                 0.589895   0.410105
                 0.150417   0.849583
                 0.673888   0.326112
                 ⋮
                 0.983417   0.0165835
                 0.294901   0.705099
                 0.994345   0.00565458
                 0.858717   0.141283
                 0.675784   0.324216
                 0.0382285  0.961771
```

```
# multinomial logit probabilities
prob_actions = mapslices(softmax, u; dims=2)
```

```
cum_prob = 1000×2 Matrix{Float64}:
             0.947399   1.0
             0.219155   1.0
             0.939594   1.0
             0.887168   1.0
             0.589895   1.0
             0.150417   1.0
             0.673888   1.0
             ⋮
             0.983417   1.0
             0.294901   1.0
             0.994345   1.0
             0.858717   1.0
             0.675784   1.0
             0.0382285  1.0
```

```
cum_prob = cumsum(prob_actions; dims=2)
```

```
# will throw an error if we goof
@assert all(cum_prob[:,2] .≈ 1)
```

```
y =
 Int64[0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1,    more ,
```

```julia
# instead of simulating random type-1 extreme values, we just
# use a uniform variable and the CDF
y = [searchsortedfirst(row, rand()) for row in eachrow(cum_prob) ] .-1
```

```
Dict(0 ⇒ 518,  1 ⇒ 482)
```

```julia
countmap(y)
```

```
loglik (generic function with 1 method)
```

```julia
function loglik(y, X, theta)
    n,k = size(X)
    ff(z) = logcdf(Logistic(), z)

    # see footnote 6 on p. 778 in Greene 6th ed for this shortcut
    q = 2 .* y .- 1
    u1 = X*theta
    LL = sum(ff.(q.*u1))

    return -LL  # I *think* you'll need to flip sign to maximize
end
```

```
dloglik! (generic function with 1 method)
```

```julia
# note that the '!' means we're updating the first argument(s)
function dloglik!(grad, y, X, theta)

    n,k = size(X)
    u1 = X*theta

    # create function
    ff(z) = cdf(Logistic(), z)

    # all the broadcasting fuses operations into a single
    # loop instead of allocating temp vectors
    # this can help w/ speed + memory
    grad .= -vec(sum( (y .- ff.(u1)) .* X; dims=1))
    return grad
end
```

```
dloglik (generic function with 1 method)
```

```julia
# wrapper to allocate gradient vector
dloglik(y, X, theta) = dloglik!(similar(theta), y, X, theta)
```

```
informationmatrix (generic function with 1 method)
```

```julia
function informationmatrix(y, X, theta)

    n,k = size(X)
    infomatrix = zeros(k,k)

    u1 = X*theta
    ff(z) = cdf(Logistic(), z)
    g = y .- ff.(u1)   # as per Greene 6th ed p. 779

    infomatrix = (g .* X)' * (g .* X)

    return infomatrix # maybe flip signs?
end
```

```
g! (generic function with 1 method)
```

```
# closures wrap likelihood & gradient
begin
    f(thet) = loglik(y,X,thet)
    g!(grad,thet) = dloglik!(grad,y,X,thet)
end
```

```
theta0 =  Float64[0.0,  0.0,  0.0,  0.0]
```

```
# initial guess
theta0 = zeros(k)
```

```
Float64[-6.5473e-9,  1.80674e-8,  -3.56951e-9,  -1.54798e-8]
```

```
# Check gradient against finite difference
begin
    fdgrad = finite_difference_gradient(f, theta0, Val{:central})
    @assert  fdgrad ≈ dloglik(y,X,theta0)
    fdgrad .- dloglik(y,X,theta0)
end
```

```
res =  * Status: success

       * Candidate solution
         Final objective value:     3.916284e+02

       * Found with
         Algorithm:     BFGS

       * Convergence measures
         |x - x'|               = 1.69e-08 ≰ 0.0e+00
         |x - x'|/|x'|          = 8.09e-09 ≰ 0.0e+00
         |f(x) - f(x')|         = 0.00e+00 ≤ 0.0e+00
         |f(x) - f(x')|/|f(x')| = 0.00e+00 ≤ 0.0e+00
         |g(x)|                 = 3.81e-10 ≤ 1.0e-08

       * Work counters
         Seconds run:   0  (vs limit Inf)
         Iterations:    14
         f(x) calls:    39
         ∇f(x) calls:   39
```

```
res = optimize(f, g!, theta0, BFGS(), Optim.Options(;show_trace=true))
```

	beta	betahat	tstat	se	pval
1	1.0	1.06057	10.043	0.105603	9.85822e-24
2	-2.0	-2.09389	-14.426	0.145147	3.54972e-47
3	1.0	1.08814	10.415	0.104479	2.11883e-25
4	0.5	0.586291	6.33893	0.0924905	2.31366e-10

```
begin
    theta1 = res.minimizer  # should be about β
    vcov = informationmatrix(y, X, theta1)
```

```
    vcovinv = inv(vcov)
    stderr = sqrt.(diag(vcovinv))
    tstats = theta1 ./ stderr
    pvals = map(z -> 2 .* cdf(Normal(), -abs(z)), tstats)

    DataFrame(beta = β, betahat = theta1, tstat=tstats, se = stderr, pval=pvals)
end
```