# 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 $u_1 < u_0$

Denote $cdf(z) = F(z)$. Let choice $y_i \in \{0,1\}$.

Likelihood is

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

Score is

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

Information matrix is

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

We can use the following functions from **`StatsFuns.jl`**

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

```
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 $y_i \in \{0,1\}$.

Likelihood is

```math
\log L(y|X) = \sum_i \log F\left( 2(y_i-1) x_i^\top \beta \right)
```

Score is

```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 use the following functions from [`StatsFuns.jl`]
(https://github.com/JuliaStats/StatsFuns.jl)

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

```julia
begin
    using Random: seed!
    using StatsFuns: logsumexp, softmax
    using LinearAlgebra: diag
    using Optim
    using Distributions
    using DataFrames
    using StatsBase: countmap

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

```
MersenneTwister(1234)
```

```julia
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)
```

```
@assert all(cum_prob[:,2] .≈ 1)
```

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

```
# 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 ⇒ 509, 1 ⇒ 491)
```

```
countmap(y)
```

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

```
function checksizes(y, X, theta)
    @assert eltype(y) <: Integer
    n,k = size(X)
    n == length(y) || throw(DimensionMismatch())
    k == length(theta) || throw(DimensionMismatch())
    return n, k
end
```

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

```
function loglik(y, X, theta)
    n,k = checksizes(y,X,theta)
    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)
```

```
function dloglik!(grad, y, X, theta)

    n,k = checksizes(y,X,theta)
    k == length(grad) || throw(DimensionMismatch())

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

    grad .= -vec(sum(g .* X; dims=1))
    return grad
end
```

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

```
dloglik(y, X, theta) = dloglik!(similar(theta), y, X, theta)
```

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

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

    n,k = checksizes(y,X,theta)
    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[1.46633e-9,  -1.11e-8,  6.52823e-9,  -1.92486e-9]
```

```
# Check gradient against autodiff
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:     3.917137e+02

       * Found with
         Algorithm:     BFGS

       * Convergence measures
         |x - x'|               = 5.80e-10 ≰ 0.0e+00
         |x - x'|/|x'|          = 2.63e-10 ≰ 0.0e+00
         |f(x) - f(x')|         = 5.68e-14 ≰ 0.0e+00
         |f(x) - f(x')|/|f(x')| = 1.45e-16 ≰ 0.0e+00
         |g(x)|                 = 1.36e-12 ≤ 1.0e-08

       * Work counters
         Seconds run:   2  (vs limit Inf)
         Iterations:    14
         f(x) calls:    42
         ∇f(x) calls:   42
```

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

| | beta | betahat | tstat | se | pval |
|---|---|---|---|---|---|
| 1 | 1.0 | 0.991798 | 10.2166 | 0.0970772 | 1.67126e-24 |
| 2 | -2.0 | -2.20044 | -15.7587 | 0.139634 | 5.9898e-56 |
| 3 | 1.0 | 0.990966 | 9.66605 | 0.10252 | 4.20272e-22 |

| | beta | betahat | tstat | se | pval |
|---|---|---|---|---|---|
| 4 | 0.5 | 0.510733 | 5.34361 | 0.0955783 | 9.1112e-8 |

```
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
```