Notes on *Discrete Choice Methods with Simulation* (Train, 2009)

Chapter 8. Numerical maximization

- The log-likelihood function takes the form $\mathrm{LL}(\beta) = \sum_{n=1}^{N} \ln P_n(\beta)/N$, where $P_n(\beta)$ is the probability of observed outcome for decision maker n, N is the sample size. The goal is to find the value of β that maximizes $\mathrm{LL}(\beta)$.
- Numerically, the maximum can be found by "walking up" the likelihood function until no further increase can be found. The researcher specifies starting values β_0. Each iteration, or step, moves to a new value of the parameters at which $\mathrm{LL}(\beta)$ is higher than at the previous value. Denote the current value of β as β_t , which is attained after t steps from the starting values.
- The gradient at β_t is the K-dimensional vector of first derivatives of $\mathrm{LL}(\beta)$ evaluated at β_t:

  $$g_t = \left(\frac{\partial \mathrm{LL}\left(\beta\right)}{\partial \beta} \right)_{\beta=\beta_t}.$$

  This vector tells us *which way to step* in order to go up the likelihood function. The Hessian is the $K \times K$ matrix of second derivatives:

  $$H_t = \left(\frac{\partial g_t}{\partial \beta'} \right)_{\beta=\beta_t} = \left(\frac{\partial^2 \mathrm{LL}\left(\beta\right)}{\partial \beta \partial \beta'} \right)_{\beta=\beta_t}.$$

  Hessian can help us to know *how far to step*, given that the gradient tells us in which direction to step.

PYTHON

```python
import numpy as np
import scipy
```

Monte Carlo: Before the details of algorithms, we first draw a sample with 9999 observations based on a logit model. We will use different algorithms to estimate those preset parameters with this simulated sample. Suppose that

$$u_{nj} = x_j^1 + 2x_j^2 + \varepsilon_{nj},$$

where $x_j = [x_j^1, x_j^2]'$ are two observed attributes of alternative j and ε_{nj} are drawn from i.i.d. standard type-I extreme value distribution. Suppose that $x_1 = [1, 2]'$, $x_2 = [2, 1]'$, and $x_3 = [3, 3]'$. Each choice maker chooses the alternative that provides the largest utility and all the final decisions are observed by the researcher. Suppose that the coefficient vector $\beta = [1, 2]'$ is unknown for the researcher and is what to be estimated.

$$\mathrm{LL} = \frac{1}{9999} \sum_{n=1}^{9999} \ln P_n = \sum_{n=1}^{9999} \sum_{i=1}^{3} y_{ni} \ln \frac{\mathrm{e}^{x_i'\beta}}{\sum_j \mathrm{e}^{x_j'\beta}},$$

$$\nabla \ln P_n = \sum_{i=1}^{3} y_{ni} \left(x_i - \sum_{j=1}^{3} x_j \frac{\mathrm{e}^{x_j'\beta}}{\sum_{j'} \mathrm{e}^{x_{j'}'\beta}} \right) = \sum_{i=1}^{3} y_{ni} \left(x_i - \sum_{j=1}^{3} x_j P_n^j \right),$$

$$\nabla^2 \ln P_n = \sum_{j=1}^{3} x_j P_n^j \left(x_j' - \sum_{j'=1}^{3} P_n^{j'} x_{j'}' \right),$$

where $y_{ni} = 1$ if decision maker n chose alternative i and 0 otherwise.

PYTHON

```
# Drawing a simulated sample with 9999 observations

N = 9999
J = 3
K = 2
np.random.seed(123456789)
x1 = np.array([1, 2]).reshape([-1, 1])
x2 = np.array([2, 1]).reshape([-1, 1])
x3 = np.array([3, 3]).reshape([-1, 1])
beta = np.array([1, 2]).reshape([-1, 1])

## drawing type-I extreme value using inverse transform sampling
draws_uniform = np.random.uniform(0, 1, N * J)
eps = -np.log(-np.log(draws_uniform)).reshape([-1, J])

## determining the final decisions
V_n1 = x1.T @ beta
V_n2 = x2.T @ beta
V_n3 = x3.T @ beta

V = np.tile(np.array([V_n1, V_n2, V_n3]).reshape([-1, 1]).T, N).reshape([N, -1])
U = V + eps
y = np.argmax(U, axis=1).reshape([-1, 1]) # find the alternative with the highest utility level
Y = np.eye(J)[y.T][0, :, :] # expand y to a (N, J) matrix containing 0-1 indicators
```

PYTHON

```
# Defineing the log likelihood, gradient, and Hessian

def LL(b, Y):
    '''Objective function'''
    b = np.array(b).reshape([-1, 1])
    exp_V_n1 = np.exp(x1.T @ b)
    exp_V_n2 = np.exp(x2.T @ b)
    exp_V_n3 = np.exp(x3.T @ b)
    P = np.tile(np.array([exp_V_n1, exp_V_n2, exp_V_n3]).reshape([-1, 1]).T, N).reshape([N, -1])
/ np.tile(exp_V_n1 + exp_V_n2 + exp_V_n3, N).reshape([N, -1])
    LL = np.log((P ** Y).prod(axis=1)).sum() / N

    return LL
```

```python
def LL_minus(b, Y):
    '''
    Minus of the objective function.
    This function is defined for the BLGS and Nelder-Mead methods as they are designed for
seeking the minimizer.
    '''
    return - LL(b, Y)

def g_n(b, Y):
    '''
    The gradient of of lnP_n.
    It returns a (N, K) matrix, thus we need to average by column to derive g_t.
    '''
    b = np.array(b).reshape([-1, 1])
    exp_V_n1 = np.exp(x1.T @ b)
    exp_V_n2 = np.exp(x2.T @ b)
    exp_V_n3 = np.exp(x3.T @ b)
    P = np.array([exp_V_n1, exp_V_n2, exp_V_n3]).reshape([-1, 1]) / (exp_V_n1 + exp_V_n2 +
exp_V_n3)

    s1_n = x1 - x1 * P[0] - x2 * P[1] - x3 * P[2]
    s2_n = x2 - x1 * P[0] - x2 * P[1] - x3 * P[2]
    s3_n = x3 - x1 * P[0] - x2 * P[1] - x3 * P[2]
    g_n = np.tile(s1_n.T, (N, 1)) * np.tile(Y[:, 0].reshape([N, 1]), (1, 2)) + np.tile(s2_n.T,
(N, 1)) * np.tile(Y[:, 1].reshape([N, 1]), (1, 2)) + np.tile(s3_n.T, (N,  1)) * np.tile(Y[:,
2].reshape([N, 1]), (1, 2))

    return g_n

def g(b, Y):
    '''The gradient.'''
    g_n0 = g_n(b, Y)
    g = (g_n0.sum(axis=0) / N).reshape([-1, 1])
    return g

def g_minus(b, Y):
    '''Minus of the gradient.'''
    return -g(b, Y).flatten()

def H(b, Y):
    '''The Hessian.'''
    b = np.array(b).reshape([-1, 1])
    exp_V_n1 = np.exp(x1.T @ b)
    exp_V_n2 = np.exp(x2.T @ b)
    exp_V_n3 = np.exp(x3.T @ b)
    P = np.array([exp_V_n1, exp_V_n2, exp_V_n3]).reshape([-1, 1]) / (exp_V_n1 + exp_V_n2 +
exp_V_n3)
    H = - (P[0] * x1 @ (x1.T - P[0] * x1.T - P[1] * x2.T - P[2] * x3.T) + P[1] * x2 @ (x2.T -
P[0] * x1.T - P[1] * x2.T - P[2] * x3.T) + P[2] * x3 @ (x3.T - P[0] * x1.T - P[1] * x2.T - P[2]
* x3.T))

    return H
```

8.1. Algorithms

8.1.1. Newton–Raphson (gradient + Hessian)

- Use a second-order Taylor's approximation around $\mathrm{LL}(\beta_t)$,

$$\mathrm{LL}\left(\beta_{t+1}\right)=\mathrm{LL}\left(\beta_t\right)+\left(\beta_{t+1}-\beta_t\right)^{\prime} g_t+\frac{1}{2}\left(\beta_{t+1}-\beta_t\right)^{\prime} H_t\left(\beta_{t+1}-\beta_t\right),$$

  to approximate the objective function, and step to the maximizer of this quadratic approximation:

$$\frac{\partial \mathrm{LL}\left(\beta_{t+1}\right)}{\partial \beta_{t+1}}=g_t+H_t\left(\beta_{t+1}-\beta_t\right)=0.$$

  That is,

$$\beta_{t+1}=\beta_t-H_t^{-1} g_t,$$

  which is the formula used by the Newton–Raphson (NR) algorithm.
- There are some useful comments for the NR procedure:
  - If $\mathrm{LL}(\beta)$ were exactly quadratic in β, then the NR procedure would reach the maximum in one step from any starting value (as what we will see in the following exercise). This implies that the NR procedure tries to approach the maximizer in a quadratic manner. That is, the NR procedure first fits a quadratic function at the starting value, β_t, in each steps and then finds the maximizer of the quadratic function as the next value, β_{t+1}.
  - If $\mathrm{LL}(\beta)$ were not quadratic in β, finding the maximizer in a quadratic manner could sometimes go beyond the maximum and lower LL (see Figure 8.4 in the textbook). To solve this problem, consider an adjusted procedure containing a scalar $\lambda>0$ to control the step size:

$$\beta_{t+1}=\beta_t-\lambda H_t^{-1} g_t.$$

    Starting with $\lambda=1$, if $\mathrm{LL}(\beta_{t+1})>\mathrm{LL}(\beta)$, move to β_{t+1} and start a new iteration. Otherwise, set $\lambda=1/2$ and try again. If, with $\lambda=1/2$, $\mathrm{LL}(\beta_{t+1})$ is still lower than $\mathrm{LL}(\beta)$, then set $\lambda=1/4$ and try again, and so on. Even though $\mathrm{LL}(\beta_{t+1})>\mathrm{LL}(\beta)$ with $\lambda=1$, we can also try a larger value of λ (e.g., $\lambda=2,4,\cdots$, as long as $\mathrm{LL}(\beta_{t+1})>\mathrm{LL}(\beta)$ still holds) to approach the maximizer within less iterations. This helps save time since less g_t and H_t need to be calculated.
  - If $\mathrm{LL}(\beta)$ is not globally concave in β, the NR procedure may fail to find an increase in $\mathrm{LL}(\beta)$. This occurs because, around β_t, the update step satisfies:

$$\begin{aligned}\mathrm{LL}\left(\beta_{t+1}\right)&=\mathrm{LL}\left(\beta_t\right)+\left(\beta_{t+1}-\beta_t\right)^{\prime} g_t\\&=\mathrm{LL}\left(\beta_t\right)-\lambda g_t^{\prime} H_t^{-1} g_t.\end{aligned}$$

    Now, suppose $\mathrm{LL}(\beta)$ is convex in a specific interval, and the starting value β_t lies within this interval. In this case, H_t^{-1} is positive semi-definite, meaning that $\mathrm{LL}\left(\beta_{t+1}\right)$ is never higher than $\mathrm{LL}\left(\beta_t\right)$ for any scalar $\lambda > 0$.

- Accordingly, there are two drawbacks of the NR procedure:
  1. Calculation of the Hessian is usually computation-intensive. Procedures that avoid calculating the Hessian at every iteration can be much faster.
  2. The NR procedure does not guarantee an increase in each step if the log-likelihood function is not globally concave.

> Exercise: Find the maximizer of $f(x_1, x_2) = -x_1^2 - x_2^2 + (x_1 - 1)(x_2 + 2)$ numerically with the NR algorithm.

PYTHON

```python
# NR algorithm

b0 = np.array([-100, -100]).reshape([-1, 1]) # suppose that the initial guess is far from the maximizer
m = 1
o = 10e-08

def calculate_g(b):
    g = np.array([-2 * b[0, 0] + b[1, 0] + 2, b[0, 0] - 2 * b[1, 0] - 1]).reshape([-1, 1])
    return g

def calculate_H():
    H = np.array([[-2, 1],
                  [1, -2]])
    return H

iter_times = 0
while abs(m) > o:
    g0 = calculate_g(b0)
    H0 = calculate_H()
    b1 = b0 - np.linalg.inv(H0) @ g0

    b0 = b1
    m = calculate_g(b1).T @ (-np.linalg.inv(calculate_H())) @ calculate_g(b1)
    iter_times += 1

print(f'Iteration: {iter_times}') # the NR procedure finds the maximizer within a single iteration
print(f'Estimated maximizer: \n{b0}')
```

```
Iteration: 1
Estimated maximizer:
```

```
[[1.]
 [0.]]
```

> Exercise: Use NR procedure to estimate the parameters with the simulated sample.

PYTHON

```python
# Estimating parameters using NR procedure

b0 = [0, 0] # It is observed that this method is very sensitive to the initial value. For example, if we use [10, 10] as the initial guess, it will report errors.
b0 = np.array(b0).reshape([-1, 1])
m = 1
o = 10e-08
lbd = 0.1

iter_times = 0
while abs(m) > o:
    g0 = g(b0, Y)
    H0 = H(b0, Y)
    b1 = b0 - lbd * np.linalg.inv(H0) @ g0
    b0 = b1

    g1 = g(b1, Y)
    H1 = H(b1, Y)
    m = g1.T @ (-np.linalg.inv(H1)) @ g1
    iter_times += 1

print(f'Iteration: {iter_times}')
print(f'Estimated maximizer: \n{b0}')
```

```
Iteration: 92
Estimated maximizer:
[[0.96805049]
 [1.94683163]]
```

8.1.2. BHHH (use scores at β_t to approximate the Hessian)

- Maximization can be faster if we utilize the fact that the function being maximized is a sum of terms in a sample. Define the following notations:
  - *Score* of an observation is the derivative of that observation's log likelihood w.r.t. the parameters: $s_n(\beta_t) = \partial \ln P_n(\beta)/\partial\beta$ evaluated at β_t.
  - The gradient is the average score: $g_t = \sum_n s_n(\beta_t)/N$.

- The outer product of observation n's score:

$$s_n\,(\beta_t)s_n(\beta_t)' = \begin{bmatrix} s_n^1 s_n^1 & s_n^1 s_n^2 & \cdots & s_n^1 s_n^K \\ s_n^1 s_n^2 & s_n^2 s_n^2 & \cdots & s_n^2 s_n^K \\ \vdots & \vdots & & \vdots \\ s_n^1 s_n^K & s_n^2 s_n^K & \cdots & s_n^K s_n^K \end{bmatrix}.$$

- The average outer product in the sample: $B_t = \sum_n s_n(\beta_t)s_n(\beta_t)'/N$. Often B_t is called the "outer product of the gradient." This term can be confusing, since B_t is not the outer product of g_t, but it does reflect the fact that the score is an observation-specific gradient and B_t is the average outer product of these observation-specific gradients.

- Since $\mathrm{LL}(\beta) = \sum_{n=1}^N \ln P_n(\beta)$, the average score g_t is indeed zero at the parameters that maximize the likelihood function.
- B_t is related to the covariance matrix: if the average score were zero, then B_t would be the covariance matrix of scores in the sample. This matrix provides a measure of the curvature of the log-likelihood function, similar to the Hessian. According to the famous *information identity*, the covariance of the scores at the true parameters is equal to the negative of the expect Hessian. Specifically, for a correctly specified model at the true parameters, $B \to -H$ as $N \to \infty$.
- Berndt, Hall, Hall, and Hausman (1974), hereafter referred to as BHHH (and commonly pronounced B-triple H), proposed using this relationship in the numerical search for the maximum of the log-likelihood function:

$$\beta_{t+1} = \beta_t + \lambda B_t^{-1} g_t,$$

  where $-H_t$ in the NR procedure is replaced by B_t.
- Two advantages of the BHHH producer over NR:
  1. B_t is far faster to calculate that H_t.
  2. B_t is necessarily positive definite.
- Some drawbacks of the BHHH procedure:
  1. The procedure can give small steps that raise $\mathrm{LL}(\beta)$ very little. This occurs especially when the iteration process is far from the maximum, because B_t is not a good approximation to $-H_t$.
  2. If the function is highly nonquadratic, NR does not perform well, and so does BHHH even if B_t were a good approximation to $-H_t$, since BHHH is an approximation to NR.

> Exercise: Use BHHH procedure to estimate the parameters with the simulated sample.

PYTHON

```python
# Estimating parameters using BHHH procedure

b0 = np.array([10, 10]).reshape([-1, 1])
m = 1
o = 10e-08
lbd = 0.1

iter_times = 0
while abs(m) > o:
    g_n0 = g_n(b0, Y)
    g0 = (g_n0.sum(axis=0) / N).reshape([-1, 1])
    B0 = (g_n0.T @ g_n0) / N
    b1 = b0 + lbd * np.linalg.inv(B0) @ g0
    b0 = b1

    g_n1 = g_n(b1, Y)
    g1 = (g_n1.sum(axis=0) / N).reshape([-1, 1])
    B1 = (g_n1.T @ g_n1) / N
    m = g1.T @ np.linalg.inv(B1) @ g1
    iter_times += 1

print(f'Iteration: {iter_times}')
print(f'Estimated maximizer: \n{b0}')
```

```
Iteration: 319
Estimated maximizer:
[[0.96893675]
 [1.94835398]]
```

8.1.3. BHHH-2

- The BHHH procedure relies on the matrix B_t , which captures the covariance of the scores when the average score is zero (i.e., at the maximizing value of β). When the iterative process is not at the maximum, the average score is not zero and B_t does not represent the covariance of the scores. A variant on the BHHH procedure is obtained by subtracting out the mean score before taking the outer product. For any level of the average score, the covariance of the scores over the sampled decision makers is

$$W_t = \sum_n \frac{(s_n(\beta_t) - g_t)(s_n(\beta_t) - g_t)'}{N}.$$

- The maximization procedure can use W_t instead of B_t:

$$\beta_{t+1} = \beta_t + \lambda W_t^{-1} g_t.$$

> Exercise: Use BHHH-2 procedure to estimate the parameters with the simulated sample.

PYTHON

```
# Estimating parameters using BHHH-2 procedure

b0 = np.array([10, 10]).reshape([-1, 1])
m = 1
o = 10e-08
lbd = 0.1

iter_times = 0
while abs(m) > o:
```

```python
    g_n0 = g_n(b0, Y)
    g0 = (g_n0.sum(axis=0) / N).reshape([-1, 1])
    W0 = ((g_n0.T - np.tile(g0, (1, 9999))) @ (g_n0 - np.tile(g0.T, (9999,1)))) / N
    b1 = b0 + lbd * np.linalg.inv(W0) @ g0
    b0 = b1

    g_n1 = g_n(b1, Y)
    g1 = (g_n1.sum(axis=0) / N).reshape([-1, 1])
    W1 = ((g_n1.T - np.tile(g1, (1, 9999))) @ (g_n1 - np.tile(g1.T, (9999,1)))) / N
    m = g1.T @ np.linalg.inv(W1) @ g1
    iter_times += 1

print(f'Iteration: {iter_times}')
print(f'Estimated maximizer: \n{b0}')
```

```
Iteration: 313
Estimated maximizer:
[[0.96883486]
 [1.94837155]]
```

8.1.4. *Steepest Ascent (we might not need to calculate or approximate the Hessian)*

- This procedure is defined by the iteration formula:

$$\beta_{t+1} = \beta_t + \lambda g_t,$$

  which can be viewed as that the defining matrix (H_t in NR, B_t in BHHH, and W_t in BHHH-2) is I.
- Considering the first-order Taylor approximation around β_t, the steepest ascent method provides the greatest possible increase in $LL(\beta)$ (actually, its approximation). However, first-order approximation is only accurate in a neighborhood of β_t. Thus, we often have to set a small step, λ, for it. Usually, BHHH and BHHH-2 converge more quickly than the method of steepest ascent.

> Exercise: Use steepest ascent procedure to estimate the parameters with the simulated sample.

PYTHON

```python
# Estimating parameters using steppest ascent procedure

b0 = np.array([10, 10]).reshape([-1, 1])
m = 1
o = 10e-08
lbd = 0.1

iter_times = 0
while abs(m) > o:
    gn0 = g_n(b0, Y)
    g0 = (gn0.sum(axis=0) / N).reshape([-1, 1])
    b1 = b0 + lbd * g0
    b0 = b1

    gn1 = g_n(b1, Y)
    g1 = (gn1.sum(axis=0) / N).reshape([-1, 1])
    m = g1.T @ g1
    iter_times += 1

print(f'Iteration: {iter_times}')
print(f'Estimated maximizer: \n{b0}')
```

```
Iteration: 5910
Estimated maximizer:
[[0.95035543]
 [1.9730307 ]]
```

8.1.5. *BFGS (use more information to approximate the Hessian)*

- Recall that NR uses the actual Hessian at βt to determine the step to β_{t+1}, and BHHH and BHHH-2 use the scores at β_t to approximate the Hessian. Only information at β_t is being used to determine the step in these procedures.
- In contrast, the BFGS procedure use information at several points to obtain a sense of the curvature of the log-likelihood function. The gradient is calculated at each step in the iteration process. The difference in the gradient between the various points that have been reached is used to calculate an *arc* Hessian over these points. This arc Hessian reflects the change in gradient that occurs for actual movement on the curve, as opposed to the Hessian, which simply reflects the change in slope for infinitesimally small steps around that point. When the log-likelihood function is nonquadratic, the Hessian at any point provides little information about the shape of the function. The arc Hessian provides better information.

> Exercise: Use BFGS procedure to estimate the parameters with the simulated sample.

PYTHON

```
# Estimating parameters using BFGS procedure

b0 = [10, 10]
options = {'maxiter': 99999,
           'disp': True}

result = scipy.optimize.minimize(fun = LL_minus,
                                 jac = g_minus,
                                 x0 = b0,
                                 args = (Y,),
                                 method = 'BFGS',
                                       options = options)
print(f'Estimated maximizer: \n{result.x}')
```

```
Optimization terminated successfully.
         Current function value: 0.142067
         Iterations: 12
         Function evaluations: 26
         Gradient evaluations: 26
Estimated maximizer:
[0.96795911 1.94822073]
```

8.1.6. Supplement: Nelder–Mead Algorithm (we might not even need the gradient)

- The Nelder–Mead algorithm is a simplex-based, *derivative-free* optimization method. Its key intuition is to iteratively reshape a geometric figure—the simplex (a triangle in 2D, tetrahedron in 3D, and so on)—based on how the function behaves at its vertices. This method is used in the original paper of BLP.
- Brief summary of its procedure:
  1. Nelder–Mead begins by constructing a simplex of $n + 1$ points (for an n-dimensional problem). Each vertex represents a candidate solution.
  2. Evaluates the objective function at these points.
  3. Find the best (lowest function value), second-worst, and worst (highest function value) vertices.
  4. Perform one of the following geometric transformations to replace the worst vertex:
     - Reflection: Reflect the worst point through the centroid of the other points.
     - Expansion: If the reflected point is better than the best, expand further in that direction.
     - Contraction: If the reflection does not give improvement, contract towards the centroid.
     - Shrinkage: If contraction also fails, shrink the whole simplex towards the best point.
  5. Repeat steps 2–4 until convergence criteria are met (e.g., simplex size or function values change is sufficiently small).
- Advantage:
  - Nelder–Mead doesn't require gradient information, making it useful for functions that are noisy, discontinuous, or expensive to evaluate.
  - By reshaping and moving the simplex in response to local landscape changes, it can efficiently "home in" on a region containing a local minimum—even when the objective function is complex.
  - The geometric moves allow the algorithm to balance exploration (searching widely) and exploitation (refining near better points).
- Limitations:
  - Nelder–Mead can get stuck in local minima and does not guarantee finding the global optimum.
  - Its performance can be sensitive to the initial simplex, especially in high dimensions.
- Some practical tips:
  - Initial guess selection matters. Start with a reasonable initial guess; random sampling can help, but it's smart to filter by function value (e.g., avoid guesses leading to very poor values).
  - Multi-stage refinement is effective. Using moderate stopping criteria (max_iter, tolerances) in the first round helps quickly "escape" bad regions and makes subsequent fine-tuning more effective.

  Then use the result of an initial round as the starting point for a more demanding second round (with tighter tolerances and higher iteration limits).
  - Monitor function value drop. Significant drops in function value after stage one often signal you are approaching a “basin” of attraction for a local minimum. If progress stalls, re-examining initial conditions or trying a different starting point may help.
  - Global or local optima. Nelder-Mead finds local minima only. For global optimization, consider multi-start approaches or combining with other algorithms.

PYTHON

```python
# Estimating parameters using Nelder–Mead Algorithm

b0 = [10, 10]
options = {'maxiter': 99999,
           'maxfev': 999999,
           'disp': True,
           'fatol': 1e-10,
           'xatol': 1e-10}


result = scipy.optimize.minimize(fun = LL_minus,
                                 x0 = b0,
                                 args = (Y,),
                                 method = 'Nelder-Mead',
                                 options = options)
print(f'Estimated maximizer: \n{result.x}')
```

```
Optimization terminated successfully.
         Current function value: 0.142067
         Iterations: 96
         Function evaluations: 198
Estimated maximizer:
[0.9677352  1.94856456]
```

8.2. Convergence criterion

- The statistic $m_t = g_t'(-H_t^{-1})g_t$ is often used to evaluate convergence. The research specifies a small value for m, such as $\breve{m} = 0.0001$, and determines in each iteration whether $g_t'(-H_t^{-1})g_t < \breve{m}$. If this inequality is satisfied, the iterative process stops and the parameters at that iteration are considered the estimates. For procedures other than NR that use an approximate Hessian in the iterative process, the approximation is used in the convergence statistic to avoid calculating the actual Hessian.
- Convergence is sometimes assessed on the basis of the gradient vector itself rather than through the test statistic m_t. There are two procedures:
  1. Determine whether each element of the gradient vector is smaller in magnitude than some value that the researcher specifies.

2. Divide each element of the gradient vector by the corresponding element of β, and determine whether each of these quotients is smaller in magnitude than some value specified by the researcher. This approach normalizes for the units of the parameters, which are determined by the units of the variables that enter the model.

8.3. *Local versus global maximum*

- All of the methods that we have discussed are susceptible to converging at a local maximum that is not the global maximum. When the log-likelihood function is globally concave, as for logit with linear-in-parameters utility, then there is only one maximum and the issue doesn't arise. However, most discrete choice models are not globally concave.
- A way to investigate the issue is to use a variety of starting values and observe whether convergence occurs at the same parameter values.

8.4. *Variance of the estimates*

8.4.1. When the model is correctly specified

- In standard econometric courses, it is shown that, for a correctly specified model

$$\sqrt{N}\left(\hat{\beta}-\beta^*\right) \rightarrow_d N\left(0,(-\mathbf{H})^{-1}\right),$$

  as $N \rightarrow \infty$, where β^* is the true parameter vector, $\hat{\beta}$ is the maximum likelihood estimator, and $\mathbf{H}$ is the expected Hessian in the population.
- The boldface type in these expressions indicates that $\mathbf{H}$ is the average in the population, as opposed to H, which is the average Hessian in the sample. The researcher calculates the asymptotic covariance by using H as an estimate of $\mathbf{H}$. That is, the asymptotic covariance of $\hat{\beta}$ is calculated as $-H^{-1}/N$, where H is evaluated at $\hat{\beta}$.
- As discussed earlier, W is the covariance of the scores in the sample. At the maximizing values of β, B is also the covariance of the scores. By the information identity, $-H$, which is the (negative of the) average Hessian in the sample, converges to the covariance of the scores for a correctly specified model at the true parameters. In calculating the asymptotic covariance of the estimates β, any of these three matrices can be used as an estimate of $-\mathbf{H}$. The asymptotic variance of β is calculated as W^{-1}/N, B^{-1}/N, or $(-H)^{-1}/N$, where each of these matrices is evaluated at $\hat{\beta}$.

PYTHON

```python
B = (g_n(result.x, Y).T @ g_n(result.x, Y)) / N
se = np.sqrt(np.diag(np.linalg.inv(B) / N))
print(se)
```

```
[0.06095187 0.08057858]
```

8.4.2. When the model is not correctly specified: Asymptotic inference

- If the model is not correctly specified, then the asymptotic covariance of $\hat{\beta}$ is more complicated. In particular, for any model for which the expected score is zero at the true parameters,

$$\sqrt{N}\left(\hat{\beta}-\beta^*\right) \rightarrow_d N\left(0,\mathbf{H}^{-1}\mathbf{V}\mathbf{H}^{-1}\right),$$

where $\mathbf{V}$ is the variance of the scores in the population and $\mathbf{H}$ is again the expected Hessian in the population. Therefore, The asymptotic variance of $\hat{\beta}$ is $\mathbf{H}^{-1}\mathbf{V}\mathbf{H}^{-1}/N$. This matrix is called the *robust covariance matrix*, since it is valid whether or not the model is correctly specified.

- To estimate the robust covariance matrix, the researcher must calculate the Hessian H. If a procedure other than NR is being used to reach convergence, the Hessian need not be calculated at each iteration; however, it must be calculated at the final iteration. Then the asymptotic covariance is calculated as $H^{-1}WH^{-1}$, or with B instead of W. This formula is sometimes called the "sandwich" estimator of the covariance, since the Hessian inverse appears on both sides.

PYTHON

```python
B = (g_n(result.x, Y).T @ g_n(result.x, Y)) / N
H1 = H(result.x, Y)
se = np.sqrt(np.diag(np.linalg.inv(H1) @ B @ np.linalg.inv(H1) / N))
print(se)
```

```
[0.06089404 0.08549375]
```

8.4.3. When the model is not correctly specified: Bootstrap inference

- An alternative way to estimate the covariance matrix is through bootstrapping, as suggested by Efron (1979). Under this procedure, the model is re-estimated numerous times on different samples taken from the original sample. Let the original sample be labeled A, which consists of the decision-makers that we have been indexing by $n = 1, \cdots, N$. That is, the original sample consists of N observations. The estimate that is obtained on this sample is $\hat{\beta}$. Bootstapping consists of the following steps:
  1. Randomly sample with replacement N observations from the original sample A. Since the sampling is with replacement, some decision-makers might be represented more than once in the new sample and others might not be included at all. This new sample is the same size as the original, but looks different from the original because some decision-makers are repeated and others are not included.
  2. Re-estimate the model on this new sample, and label the estimate β_r with $r = 1$ for this first new sample.
  3. Repeated steps 1 and 2 numerous times, obtaining estimates β_r for $r = 1, \cdots, R$ where R is the number of times the estimation is repeated on a new sample.
  4. Calculate the covariance of the resulting estimates around the original estimate: $V = \frac{1}{R}\sum_r(\beta_r - \hat{\beta})(\beta_r - \hat{\beta})'$.
- The V derived from the above procedure is an estimate of the asymptotic covariance matrix. The logic of the procedure is the following. The sampling covariance of an estimator is, by definition, a measure of the amount by which the estimates change when different samples are taken from the population. Our original sample is one sample from the population. However, if this sample is large enough, then it is probably similar to the population, such that drawing from it is similar to drawing from the population itself. The bootstrap does just that: draws from the original sample, with replacement, as a proxy for drawing from the population itself. The estimates obtained on the bootstrapped samples provide information on the distribution of estimates that would be obtained if alternative samples had actually been drawn from the population.
- The advantage of the bootstrap is that it is conceptually straightforward and does not rely on formulas that hold asymptotically but might not be particularly accurate for a given sample size. Its disadvantage

is that it is computer-intensive since it entails estimating the model numerous times. Efron and Tibshirant (1993) and Vinod (1993) provide useful discussions and applications.

PYTHON

```python
R = 9999
b_hat = result.x
b_r = []
np.random.seed(123456789)

for i in range(R):
    Y_sample_index = np.random.choice(N, size=N, replace=True)
    Y_sample = Y[Y_sample_index, :] # construct bootstrap sample

    b0 = [10, 10]
    result = scipy.optimize.minimize(fun = LL_minus,
                                     jac = g_minus,
                                     x0 = b0,
                                     args = (Y_sample,), # use the bootstrap sample
                                     method = 'BFGS')
    b_r.append(result.x)
    print(f'Bootstrap sample: {i+1}/{R}', end='\r')

b_diff = np.array(b_r) - np.tile(b_hat, [R, 1])
se = np.sqrt(np.diag((b_diff.T @ b_diff) / R))
print(f'\n{se}')
```

```
Bootstrap sample: 9999/9999
[0.08394596 0.10027597]
```