離散選択モデル: 補足資料２(コード編)

応用社会科学RAブートキャンプ

講師: 遠山祐太 (早稲田大学)

最終更新: 2024-08-24 23:28:39

フルスクラッチでモデル推定

フルスクラッチで推定コードをプログラミング

- 自分自身で推定・シミュレーションのコードをプログラミングする状況が多々ある。
  - むしろパッケージで実行可能な構造推定手法は限定的
- 以下では、多項ロジットモデルの推定のコードについて、ゼロから書いていこう。
- 手順：
  - Step 1: 多項ロジットモデルの尤度関数を定義する。
  - Step 2: 最尤法による推定：定義した尤度関数について数値最適化を行う。-> 点推定値
  - Step 3: 標準誤差の計算
- 参考文献として、Matlabにはなるが以下がオススメ：
  - Adams, Clarke, and Quinn "Microeconometrics and Matlab"

多項ロジットモデルの尤度関数(再掲)

- 尤度関数は

$$L(\theta) = \prod_{i=1}^{N} \prod_{k=1}^{5} P_k \left(j = y_{i,k} | \theta\right)$$

  - ここで

$$P_k(j|\theta) = \frac{\exp(\alpha_j - \beta \cdot p_{j,k})}{1 + \exp(\alpha_{Kinoko} - \beta \cdot p_{Kinoko,k}) + \exp(\alpha_{Takenoko} - \beta \cdot p_{Takenoko,k})}$$

- パラメタは $\theta = (\beta, \alpha_{Kinoko}, \alpha_{Takenoko})$

対数尤度(再掲)

- 対数尤度関数を用いる

$$\log L(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{5} \log P_k \left(j = y_{i,k} | \theta \right)$$

- ポイント：この対数尤度を最大化するような値は解析的には得られない。数値計算の出番！！

【R分析】下準備

```r
rm(list = ls())
library("tidyverse")
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.4     ✔ readr     2.1.5
## ✔ forcats   1.0.0     ✔ stringr   1.5.1
## ✔ ggplot2   3.5.0     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.1
## ✔ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```r
library("knitr")
library("mlogit") # ロジットモデル推定のためのパッケージ
```

```
##  要求されたパッケージ dfidx をロード中です
##
##  次のパッケージを付け加えます: 'dfidx'
```

【R分析】ロジット確率計算のための関数

- パラメタ $\beta, \alpha_{Kinoko}, \alpha_{Takenoko}$ と価格を与えることで、選択確率を計算する。

```r
f_logit_prob <- function(alpha_Kinoko, alpha_Takenoko, beta, price_Kinoko, price_Takenoko){

  util_Kinoko <- alpha_Kinoko - beta*price_Kinoko
  util_Takenoko <- alpha_Takenoko - beta*price_Takenoko

  prob_Kinoko <- exp( util_Kinoko ) / ( 1 + exp( util_Kinoko ) + exp( util_Takenoko ))
  prob_Takenoko <- exp( util_Takenoko ) / ( 1 + exp( util_Kinoko ) + exp( util_Takenoko ))
  prob_Other <- 1 - (prob_Kinoko + prob_Takenoko)

  return( cbind(prob_Kinoko, prob_Takenoko, prob_Other))
}
```

【R分析】Step 1: 対数尤度関数の定義

- 関数`f_logit_prob`は上で定義済み。

```
f_likelihood_logit <- function(param, data){

  alpha_Kinoko <- param[1]
  alpha_Takenoko <- param[2]
  beta    <- param[3]

  P1 <- f_logit_prob(alpha_Kinoko, alpha_Takenoko, beta, 200, 200)
  P2 <- f_logit_prob(alpha_Kinoko, alpha_Takenoko, beta, 180, 200)
  P3 <- f_logit_prob(alpha_Kinoko, alpha_Takenoko, beta, 200, 170)
  P4 <- f_logit_prob(alpha_Kinoko, alpha_Takenoko, beta, 220, 200)
  P5 <- f_logit_prob(alpha_Kinoko, alpha_Takenoko, beta, 190, 210)
```

【R分析】続き

```
  pred <- rbind(P1, P2, P3, P4, P5)
  pred <- as_tibble(pred)
  pred$occasion <- c("Q1", "Q2", "Q3", "Q4", "Q5")

  data %>%
    left_join(pred, by = "occasion") %>%
    mutate( choice_prob = case_when( choice == 0 ~ prob_Other,
                                     choice == 1 ~ prob_Kinoko,
                                     choice == 2 ~ prob_Takenoko) ) %>%
    mutate(log_choice_prob = log(choice_prob)) %>%
    select(log_choice_prob) -> result

  likelihood <- sum(result)

  return(likelihood)

}
```

【R分析】Step 2: 対数尤度関数の最大化

- 定義した対数尤度関数を最大化するようなパラメタを求めたい。
- どうやるか？ -> 数値最適化
- まずは数値最適化について簡単に説明する。

数値最適化

- 定義した関数について、その関数を最大にするような変数を求める作業。
- 簡単な例： $f(x,a) = (x-a)^2$. a はパラメタ。

```
# 関数の定義
fx <- function(x,a){ (x-a)^2 }
```

- 当然ながら、 $x = a$ で最小化される。
- 数値最適化を行って確かめてみよう。

optimize関数による数値最適化

```
# 関数を最小化する。
xmin <- optimize( f = fx,
                  interval = c(-10,10),
                  a = 2.4)
print(xmin)
```

```
## $minimum
## [1] 2.4
##
## $objective
## [1] 0
```

- `f`: 最小化したい、自分で定義した関数。なお、`optimize`は1変数のみ。(多変数は後述)
- `interval`: 変数を探索する範囲。
- `a`: 自分で定義した関数に追加の引数が必要な場合は指定する。

数値最適化の一般論

- Rのデフォルトは`optimize`と`optim`
  - `optimize`は一変数の最適化
  - `optim`は多変数の最適化
- その他にも様々なパッケージ：`optimx`
- 数値最適化のややこしい点：様々なオプションを設定しなければならない。
  - 最適化の方法：微分を使うか否か
  - 初期値の選び方
  - 最適化停止の基準(tolerance): 目的関数や微分の値がどの程度動かなくなったら停止するか？

最適化の方法

- 微分を使わない方法：Nelder-Mead、シンプレックス法
- 微分を使う方法：BFGS (quasi-Newton法)
- また、微分を使う方法の場合も、いくつかのVariation
  - 自分で微分を計算して関数として与える
  - 数値微分を内部で計算してもらう。
- 関数が滑らか（微分可能）で、微分計算ができるならば、微分を使う方法の方が早い。
- 関数が滑らかかわからない場合は、微分を使わない方法を使う。ただし、計算時間がかかる。

初期値の選び方

- 初期値：どの値から関数の評価を始めるか。
- 非常に重要。
  - 極端な値を選ぶと、関数の内部で変なことが発生し、最適化が進まない。(例：exp関数)
  - 初期値によっては、**大域的最適点**ではなく、**局所最適点**に止まるかも。
- 実践においては
  - 初期値をうまく選ぶ(後述)
  - いろいろな初期値を試して、結果が大きく変わらないことを確かめる&目的関数が小さくなる値を選ぶ。

最適化停止の基準(tolerance)

- 目的関数や微分の値がどの程度動かなくなったら停止するか？
- 基準をゆるくすると、計算時間は早いが、得られる値は不正確かもしれない。
- 関数における引数やアウトプットのスケールとも関連する。

数値最適化の参考資料

- 数値最適化は非常に広い
  - 今回扱っているのは、制約なしの非線形最適化
  - その他：線形計画、制約つき最適化、非線形方程式、などなど
- 離散選択モデルにおける最適化については、TrainのChapter 8がよくまとまっている。
- 経済学一般におけるガイドとしては、Miranda and Fackler "Applied Computational Economics and Finance" ただしMatlabベース。
- Rにおける数値最適化のガイドとして Optimization with R –Tips and Tricks

【R分析】Step 2: 対数尤度関数の最大化

- `optimx`パッケージを用いて、対数尤度関数の最大化を行う。
  - 後ほど標準誤差計算に必要なヘシアンの計算のため。

```
# 初期パラメタ
ini <- c(5,5, -0.01)

# 最適化
result <- optimx( par = ini,
                  fn = f_likelihood_logit,
                  data = data_for_estimation,
                  control = list(fnscale=-1),
                  hessian = TRUE   )

# 推定値の取得
est_vec <- as.numeric(result[1,1:3])
```

【R分析】optimxの引数

- `par`: 初期値
- `fn`: 最適化したい関数
- `method`: 方法。ここではNelder－Mead （シンプレックス法）
- `data`: 定義した`f_likelihood_logit`で必要な引数
- `control`: 最大化なので、`fnscale = -1`としている。
- `hessian`: 数値計算で得られるヘシアン。後ほど。

どうやって初期値を選んだのか？

- 身も蓋もない説明としては：
  - `mlogit`ですでに推定値がわかっている -> パラメタのスケールの当たりがついている。
  - 多項ロジットモデルの対数尤度関数は、**大域的に凹関数** -> 初期値はそこまで重要でない。
- 初期値の選び方に関する実践(個人的経験に基づく)
  - パラメタを適当にいれて、対数尤度関数を計算し、変な値にならないことを確認する。
  - グリッドサーチをする。
  - 多次元関数における一部変数を固定して、一次元でプロットしてみる。

実践１：極端な値を入れない。

- 関数の中に`exp`が入っているときや、分母に変数が入る場合は気をつける。`Inf`が生じて、内部の計算がおかしくなる。

```
exp(2000)
```

```
## [1] Inf
```

```
1/0
```

```
## [1] Inf
```

- ロジットモデルにおいて初期値のパラメタを極端にすると、尤度が計算できない。
    - 例えば、$\beta_{Kinoko} = 10000$ を初期値にしたとき、exp()関数は発散し、確率は０／１に張り付く。
    - 尤度関数の中に$\log(P)$があるため、尤度が $-\infty$ になる。
- 選んだ初期値において、最適化したい関数の中身がちゃんと計算されているか（例：確率、尤度）を確認する！！

実践２：グリッドサーチ

- 最適化の変数が３つであれば、それぞれ１０通りの取りうる値を選び、合計１０００通りの組み合わせについて、目的関数を評価する。
- その結果、最も目的関数が大きく（小さく）なる値を初期値として選んで、数値最適化する。
- 注意点：変数が多くなると大変。いわゆる**次元の呪い**

実践３：一次元（二次元）プロット

- 一次元・二次元ならば、図としてプロット可能。
- 一部のパラメタを（適切に）止めた上で、他のパラメタについて動かしてみる。
- ある程度の当たりをつけた上で、数値最適化する。

【参考】数値最適化・シミュレーションの個人的コツ

- 目的関数の評価時間についてプロファイラーで計算する。
- モンテカルロ実験
  - 真のパラメタを知っている元でデータを生成。
  - そのデータを使って推定を行い、元のパラメタを復元できるか？
  - デバッグの手段としても有効。
- 並列計算の活用
  - いろいろな初期値をたくさん試す場合に有効

【参考】個人的コツ続き

- モデルの挙動を把握する(比較静学)
  - 経済学的思考をバグ取りに活用する。数値計算の結果が経済的な直感と合うのか？？
- パラメタの識別を考える。
- パラメタにReasonableな制約を入れる。
  - パラメタが変な値をとったときに、モデルの挙動がおかしくなるのを防ぐ。
- 最適化する変数のスケールをなるべく合わせる。
  - 最適化のToleranceとも関係。

【参考】標準誤差の計算

- 最尤推定量の標準誤差の計算。
- 漸近正規性

$$\sqrt{n}\left(\hat{\theta}-\theta_0\right) \xrightarrow{d} N(0, V)$$

- ここで、

$$A = -E\left[\left.\frac{\partial^2 \log f(y_i;\theta)}{\partial\theta\partial\theta'}\right|_{\theta=\theta_0}\right]$$

$$V = A^{-1}$$

- この結果にもとづいて、漸近分散の計算を以下のように行うことができる。
- まず、漸近分散の推定として、

$$\hat{A} = -\frac{1}{n}\sum_{i=1}^{N}\frac{\partial^2}{\partial\theta\partial\theta}\log f(y_i, \theta) = -\frac{\partial^2}{\partial\theta\partial\theta}\frac{1}{N}l(\theta)$$

- これは目的関数(をサンプルサイズで割った値)のヘシアンに相当する。なお、目的関数（対数尤度）のヘシアンは、数値計算などにおいて付随的に計算されることが多い。
- よって漸近分散の推定値 $ase(\hat{\theta})$ は、

$$ase(\hat{\theta}) = \sqrt{diag(\frac{1}{n}\hat{A}^{-1})}$$

として与えられる。ここで、diag(X)は行列Xの対角要素のみを抜き出したベクトルである。

【R分析】Step 3: 標準誤差の計算

- 以上を踏まえて、目的関数のヘシアンを数値的に計算すれば良い。
- `optimx`には、`gHgen`というヘシアンを数値計算する関数が存在する。

```
Hessian <- gHgen(par = as.numeric(result[1,1:3]),
                 fn = f_likelihood_logit,
                 data = data_for_estimation)
se_vec <- round(sqrt(diag(solve(-Hessian$Hn))), 3)
```

推定結果のまとめ

```r
print( rbind(est_vec, se_vec) )
```

```
##                [,1]     [,2]       [,3]
## est_vec 11.68466 12.22352 0.05760414
## se_vec   0.72600  0.74100 0.00400000
```