Understanding the mechanics of a simple MNL

Daniel Heimgartner

April 20, 2023

1 Introduction

This should illustrate the mechanics of discrete choice modeling and explain the gist of packages such as `apollo` and `mixl`. More complicated models are not much more difficult to code (however usually involve taking random draws since there exists no closed-form solution for the likelihood function (integral). If you understand the implementation of a simple MNL it is not too difficult to generalize the idea to more "complicated" models!

At the core, we use maximum likelihood estimation: We maximize the probability of predicting right. What does this mean? The derivation of the choice probabilities P_{nit} follows from random utility theory and answers the question *What is the probability of observing individual n choosing alternative i in choice situation t* (given our assumptions about the underlying data generating process - our model). It should be intuitive to realize, that maximizing the sum of these probabilities is the same as finding the model parameters that maximize the probability of observing all the revealed choices...

For a fantastic introduction to maximum likelihood estimation consult the vignette *Getting started with maximum likelihood and maxLik*

```
> browseVignettes("maxLik")
```

2 The choice probability

Recall that the choice probability P_{nit} is

$$P_{nit} = \frac{X_{nit}}{\sum_j X_{njt}} \tag{1}$$

where $X_{njt}\beta$ reflects the assumed utility relation $V_{njt} = U_{njt} - \epsilon_{njt}$.

With a sample of N decision makers, the probability that the choice of person n is observed can be represented as

$$L(\beta) = \prod_n^N \prod_t^{T_n} \prod_i^{I_{nt}} (P_{nit})^{y_{nit}} \tag{2}$$

where $y_{ni} = 1$ if person n chooses i and zero otherwise. Note, that T_n indicates that not every individual must have the same number of choice situations and I_{nt} indicates, that not all alternatives must be available to all individuals in all

choice situations. In our simple example however, these so-called availabilities are constant...

And as mentioned above this is exactly the funtion we want to maximize. Computationally, it is usually beneficial to remove the product operators by taking the logarithm which is a monotonic transformation, i.e. the maximum does not change: $\max f(x) = \max \log(f(x))$.

$$LL(\beta) = \sum_{n}^{N} \sum_{t}^{T_n} \sum_{i}^{I_{nt}} y_{nit} \log(P_{nit}) \tag{3}$$

So far, this has only been some algebra and statistics. Almost any econometric model yields a (log-) likelihood function and if we do not care about the mathematics we can take it for granted. The only thing we are left to do is find the parameter values which maximize the above log-likelihood function! Recall, that our parameters of interest are the utility weights, i.e. the βs in (1).

3 Implementation in R

We use some sample data for illustration

```
> data("Train", package = "mlogit")
> # consult the data documentation
> ?mlogit::Train
```

We use the following utility (here U_A really is V_{nit}...) specification where individuals face the two mode alternatives A and B

$$U_A = \beta_{price} * price_A/1000 + \beta_{time} * time_A/60$$
$$U_B = ASC + \beta_{price} * price_B/1000 + \beta_{timeB} * time_B/60$$

The function in (3) translates almost verbatim to `R` code.

```
> loglik <- function(param) {
+   U_A <- param["B_price"] * Train$price_A / 1000 +
+     param["B_time"] * Train$time_A / 60
+
+   U_B <- param["asc"] + param["B_price"] * Train$price_B / 1000 +
+     param["B_timeB"] * Train$time_B / 60
+
+   # helpers
+   exp_A <- exp(U_A)
+   exp_B <- exp(U_B)
+   y_A <- as.numeric(Train$choice == "A")  # 1 if A is chosen, 0 otherwise
+   y_B <- as.numeric(Train$choice == "B")  # similar (1-y_A)
+
+   P_Ant <- exp_A / (exp_A + exp_B)
+   P_Bnt <- exp_B / (exp_A + exp_B)
+   sum(y_A * log(P_Ant) + y_B * log(P_Bnt))
+ }
```

Now, we do not need to implement the (numerical) maximization routines ourselves but simply pass it to `maxLik` specifying the use of the Broyden-Fletcher-Goldfarb-Shanno algorithm

```
> # define parameter vector (and starting values)
> param <- c(1, 1, 1, 1)
> param <- setNames(param, c("asc", "B_price", "B_time", "B_timeB"))
> # maximize the loglik function
> m <- maxLik::maxLik(loglik, start = param, method = "BFGS")
> # print model results
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
BFGS maximization, 36 iterations
Return code 0: successful convergence
Log-Likelihood: -1845.244
4  free parameters
Estimates:
        Estimate Std. error t value  Pr(> t)
asc      0.12578    0.18930   0.664    0.506
B_price -1.02556    0.05944 -17.253  < 2e-16 ***
B_time  -0.81155    0.14069  -5.768 8.00e-09 ***
B_timeB -0.87990    0.14693  -5.989 2.12e-09 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
--------------------------------------------
```

4 mixl comparison

The `mixl` package does more or less exactly this but codes the log-likelihood function in `C++` with the `Rcpp` package and parallelizes the computations (since the LL for each individal can be computed independently - however, in this simple MNL example, the vectorized computation is probably not much slower). Also the neat thing about `mixl` is the way one can specify utilities (as string) which are then parsed to the corresponding indirect utilities V.

Here is the equivalent code that reproduces (almost) identical results using `mixl`

```
> Train$ID <- Train$id
> Train$CHOICE <- as.numeric(Train$choice)
> mnl_test <- "
+         U_A = @B_price * $price_A / 1000 + @B_time * $time_A / 60;
+         U_B = @asc + @B_price * $price_B / 1000 + @B_timeB * $time_B / 60;
+         "
> model_spec <- mixl::specify_model(mnl_test, Train, disable_multicore=T)
> # only take starting values that are needed
> est <- stats::setNames(c(1, 1, 1, 1), c("asc", "B_price", "B_time", "B_timeB"))
> availabilities <- mixl::generate_default_availabilities(
```

```
+   Train, model_spec$num_utility_functions)
> model <- mixl::estimate(model_spec, est, Train, availabilities = availabilities)

Initial function value: -2956.237
Initial gradient value:
       asc    B_price     B_time    B_timeB
 -617.8363  -706.2634  1289.3497 -1306.7643
initial  value 2956.236634
iter   2 value 2810.098282
iter   3 value 2261.279763
iter   4 value 2133.407799
iter   5 value 1894.521594
iter   6 value 1874.450028
iter   7 value 1845.800674
iter   8 value 1845.261963
iter   9 value 1845.244132
iter   9 value 1845.244128
iter   9 value 1845.244128
final  value 1845.244128
converged
```

Compare the model coefficients to our own implementation!

```
> summary(model)

Model diagnosis: successful convergence
Runtime: 0.0428 secs

Number of decision makers: 235
Number of observations: 2929

LL(null):  -2030.228
LL(init):  -2956.237
LL(final):  -1845.244
LL(choice):  -1845.244
Rho2:  0.09111487

AIC:  3698.49
AICc:  3698.66
BIC:  3722.42
Estimated parameters:  4

Estimates:
            est     se trat_0 trat_1  robse robtrat_0
asc      0.1258 0.1892   0.66  -4.62 0.1785      0.70
B_price -1.0256 0.0595 -17.25 -34.06 0.1047     -9.79
B_time  -0.8116 0.1413  -5.74 -12.82 0.1687     -4.81
B_timeB -0.8799 0.1474  -5.97 -12.75 0.1669     -5.27
        robtrat_1 rob_pval0 rob_pval1
asc         -4.90      0.48         0
B_price    -19.35      0.00         0
```

```
B_time     -10.74      0.00         0
B_timeB    -11.26      0.00         0
```

5 Exercises

1. Benchmark the runtime of our all-in-R implementation to `mixl`.
2. If you know some `C++` try to write the `loglik` function with `Rcpp::cppFunction` and check the performance gain!
3. Try to implement a mixed MNL.