Programming MLE models in STATA[1]

Andreas Beger
30 November 2008

1 Overview

You have learned about a lot of different MLE models so far, and most of them are available as pre-defined commands in STATA.[2] However, there is a good chance that at some point you will come across a problem for which there is no pre-defined solution. You might be interested in party choice in situations where the IIA assumption does not hold and choices are nested. You might be interested in the incidence of torture, but think that there are countries in which torture occurs but is not reported. You might need a count model for situation where the truncation point is something other than 0. Many people at that point will choose the simple expedient of using the next best model for any given situation. For example, most published articles that look at the number of fatalities in wars use the log of fatalities in an OLS regression, rather than a count or truncated count model. A more difficult, but also more appropriate alternative would be to write down whatever process you think generated the data you are dealing with and derive a likelihood function from it. Let's say you have done that, i.e. you have written down a likelihood function for the particular problem you are dealing with. What now? By the end of this you should roughly know the answer, and looking through the notes you should be able to estimate (almost) any likelihood function using Stata.

The next section is a quick review of Maximum Likelihood Estimation, and the maximization problem involved in particular. This helps introduce some of the terminology involved with programming MLE commands in Stata, but is also generally helpful. The sections after that go into the nuts and bolts of writing Stata programs for maximum likelihood estimators.

2 Review of Maximum Likelihood Estimation

Say we have magically developed a log-likelihood function with the generic form:

$$\ln L = f(\theta) \tag{1}$$

or in words, where our log-likelihood for the data at hand is a function of θ, where θ is the vector of parameters in our model. For a logit regression model, this would in more detail look like this:

$$\ln L = \sum_{i=1}^{N} \left(y_i \ln \left(\frac{1}{1+e^{-\theta}} \right) + (1-y_i) \ln \left(1 - \frac{1}{1+e^{-\theta}} \right) \right) \tag{2}$$

In this logit example, we will probably be interested in estimating μ, the mean of our distribution $f()$, and we do so using a vector of coefficients β that includes a constant term and coefficients for several independent variables. Unless there is an analytical solution to such a model, we will usually use maximum likelihood estimation to solve the problem.

[1] These notes are closely based on Gould, Pitblado and Sribney (2006).

[2] Quite a few if you think back... logit, probit, normal (OLS in MLE), heteroscedastic probit, rare events logit, multinomial logit, conditional logit, multinomial probit, ordered probit, ordered logit, poisson, negative binomial, hurdle poisson, zero-inflated poisson, zero-inflated negative binomial, exponential, weibull, log-logistic, generalized gamme, Cox, etc.

The idea behind MLE is to find the values of θ, or more specifically in our case to find the values of the coefficients β, that maximize the log-likelihood for the data we use to estimate the parameters. To do this, we start with some initial values for our parameters, and adjust them in a way that should result in a higher log-likelihood, until we have reach the highest log-likelihood value possible (hopefully). To adjust any single parameter, we need to know which direction to adjust it in (i.e. make it smaller or larger) and by how much to adjust it. For our vector of parameters, this problem more generally takes the form:

$$\hat{\theta}_{k+1} = \hat{\theta}_k + \lambda_k \Delta_k \tag{3}$$

where k indicates that we are in the k-th iteration in our process of finding parameters that maximize the log-likelihood. In words, we adjust the previous parameter estimates by $\lambda_k \Delta_k$, where Δ is a vector telling us which direction to adjust the parameters by, and where λ is a vector that specifies the "step size", i.e. by how much to adjust each coefficient. We want to find a Δ and λ such that the new vector of parameters results in a higher log-likelihood than we had before (i.e. we want to pick Δ and λ such that $\ln L = f(\beta_{k+1}) > f(\beta_k)$.). How can we do this?

What we need is information on the slope(s) of the current point θ_k (so we know which direction to go), and also information on how fast the slope is changing at our current point. There are four different algorithms of doing this in Stata. All of them calculate the direction vector Δ by using the gradient vector, which is a vector of first derivatives of the log-likelihood function in respect to each of the parameters:

$$g(\theta_k) = \frac{\partial \ln L}{\partial \theta_k} \tag{4}$$

However, in addition to the gradient vector, all four algorithms also use additional information to refine the direction vector. Using only the gradient vector is not a very good direction finding tool, since it only tells you whether the log-likelihood is increasing in one direction or not. To further refine the direction vector, the four algorithms use additional information, and they differ only in what that additional information is. They only differ in how they calculate the direction vector Δ, not in how they calculate step size λ.

2.1 Newton-Raphson

The Newton-Raphson algorithm calculates the direction vector using the the inverse of the negative Hessian matrix, which itself is a matrix of partial second derivatives and cross-derivatives:

$$H(\theta_k) = \frac{\partial^2 \ln L}{\partial \theta_k \partial \theta_k'} \tag{5}$$

Specifically, Newton-Raphson calculates the direction vector as:

$$\Delta = [-H(\theta_k)]^{-1} g(\theta_i) \tag{6}$$

As it turns out, calculating the inverse of the negative Hessian is computationally quite intensive, and thus finding the direction vector is the most complicated part. Calculating a log-likelihood by comparison is easy, and so for the stepsize, Newton-Raphson (and the other algorithms) use a much simpler approach: they start with $\lambda = 1$, calculate $f(\theta_k)$ and $f(\theta_{k+1})$, and keep increasing λ as long as

$f(\theta_{k+1}) > f(\theta_k)$. Sometimes it might go to far, in which case it backs up and decreases λ, until finding an optimal step size. Once it has found the optimal step size, the algorithm updates the current estimate for θ and then searches for a new direction vector. It repeats this until a maximum is reached.

2.2 Other algorithms

Because the inverse of the negative Hessian is intensive to calculate, statisticians have developed other algorithms that use various approximations for the Hessian matrix that are computationally easier to calculate. In practice, these algorithms will do calculations faster, but because their direction vectors usually are not as good as Newton-Raphson's, they will usually also take longer to find a maximum.

Since I don't have time to write this, you'll have to take my word that the other three algorithms use various approximations for the Hessian.

3 Writing your own program in STATA

Stata comes with a lot of predefined statistical models like logit, probit, etc. At first glance, it might seem that writing your own commands can in no way come close to matching Stata's predefined commands. That is not true. Given enough time and effort, you can *exactly* replicate what Stata's predefined commands do with your own program, and more importantly, you can write your own commands that match Stata's expectations for programs.

To make this a little bit more clear, when you type `logit` in Stata, you will get something like this:[3]

```
 . logit y x1 x2

initial:       log likelihood = -693.14718
alternative:   log likelihood = -722.07698
rescale:       log likelihood = -693.11518
Iteration 0:   log likelihood = -693.11518
Iteration 1:   log likelihood = -497.20157
Iteration 2:   log likelihood = -482.37494
Iteration 3:   log likelihood = -481.75462
Iteration 4:   log likelihood = -481.75293

Logistic regression                                     Number of obs   =       1000
                                                        LR chi2(2)      =     422.72
                                                        Prob > chi2     =     0.0000
Log likelihood = -481.75293                             Pseudo R2       =     0.3049

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   1.164907   .0869981    13.39   0.000     .9943943     1.33542
          x2 |   2.163466   .1628737    13.28   0.000     1.844239    2.482692
       _cons |  -2.210133   .1852701   -11.93   0.000    -2.573256    -1.84701
------------------------------------------------------------------------------
```

[3] Actually you would not get the first three lines in the output and the output would start with "Iteration 0…", but many other commands will give you three lines like that.

By the end of this lecture, you should understand what all the lines showing you log-likelihoods mean, and we will replicate these results using our own command for Stata.

To write your own estimation command in Stata, you can more or less follow this sequence of steps shown below. They use the easiest method for writing your own commands (`lf`), and it works with most, but not all, likelihood functions. These steps provide an outline for the rest of these notes, although there will be a few major excursions into other areas that you need to know or that are useful to know.

1. Make sure you can use the `lf` method. (3.1)
2. Write a program that calculates likelihood values. (3.2)
3. Make sure the program works correctly. (3.4)
4. Specify a model. (3.5)
5. Set or search for starting values (optional). (3.6)
6. Maximize. (3.7)
7. Check for errors. (3.8)

Stata has a set of commands like `ml model` and `ml maximize` that make all of this a lot easier, so throughout the following discussion I will introduce these commands when appropriate. There will also be two lengthy tangents on programming in Stata and on generating "synthetic" data that follows a known data-generating process.

3.1 Make sure you can use the `lf` method.

We have not talked about what the `lf` method is yet–roughly, it is the easiest way of programming your own MLE commands into Stata. In order to be able to use this method, your likelihood function must meet the linear-form restriction that:

$$\ln L = \ln l_1 + \ln l_2 + \ldots + \ln l_N \tag{7}$$

or in words, the overall log likelihood must equal the sum of log likelihoods for each individual observation in your data. Examples of likelihood functions that do not meet this restriction include those that model groups of observations, rather than individual, independent observations, like conditional logit. In those cases, the easiest method to use is `d0`.

3.2 Write a program that calculates likelihood values.

The first practical step towards writing a new regression command for Stata is to write a specific type of program that calculates log likelihood values for the observations in your estimation sample. Essentially this is how you define the likelihood function. The program receives your model specification as input (i.e. what is the dependent variable, what are you explanatory variables, at minimum) and provides specific lnL values as output:

```
program define logit2_lf
    [input for $\mu_i$ and other parameters if used]
    [output is $\ln L_i$ at a minimum]
    end
```

Note that this particular program literally calculates a $\ln L$ value for each individual observation in the sample. The rest of the `ml` commands can take care of summing the log-likelihoods.

So, to do this for a logit regression model, we start with the log likelihood function:

$$\ln L = \sum_{i=1}^{N} \left(y_i \ln \left(\frac{1}{1+e^{-x_i\beta}} \right) + (1-y_i) \ln \left(1 - \frac{1}{1+e^{-x_i\beta}} \right) \right) \tag{8}$$

We essentially need to put this function into a program that, given a list of variables and coefficients (or just a constant term), literally creates a new variable containing the resulting log-likelihood values for each observation. So we might write a program like this:

```
program define logit2_lf
version 10.1
args lnf xb
tempvar lnlj
qui {
    gen double ‘lnlj’ = ln(invlogit( ‘xb’)) if $ML_y1 == 1
    replace    ‘lnlj’ = ln(invlogit(-‘xb’)) if $ML_y1 == 0
    replace ‘lnf’ = ‘lnlj’
    }
end
```

Most of this program is stuff that is not directly related to actually calculating the log-likelihood values. All of that is done by the two lines in the middle that generate and replace `lnlj`.

So what is going on in this program? The first line simply tells Stata that we want to create a new program (i.e. a new command) called `logit2_lf`. We can name our program anything we want, as long as it does not conflict with existing Stata commands (more below). The next line defines which version of Stata I am using–10.1 in this case. This is important for backwards comparability. The command `args` (short for `arguments`) defines local names by which we want to refer to the input our program receives. This just makes it slightly easier to remember what is what. The first argument our command receives is labelled `lnf`, and the second `xb`. Next, we define a temporary variable called `lnlj` that will hold log-likelihood values for individual observations. Generally, using temporary variables means that Stata will drop the variable automatically when the program concludes, and since we do not care about individual $\ln L$ values, we might as well do this. Actually we could leave this step out since the rest of Stata's `ml` routines would have done this anyways, but if we ever want to move on to writing a `d0` version of this program, we will need this. Now we can actually calculate the log-likelihood. I could have just put equation 8 into one line and calculate all log-likelihood values at once. Generally though, that will probably increase the chances that there is an error, so it's always a good idea to split problems into small parts. To that end, the program uses a couple of shortcuts. First, we calculate log-likelihood values first for observations where $y_i = 1$, then for when $y_i = 0$. When you use Stata's `ml` routines, they will automatically create a global macro that tells you whether that is the case or not–`$ML_y1`.[4] Note, and

[4]Global macro's are generally referenced by putting a `$` in front of their name, and the name `ML_y1` in this case tells us that this is the first dependent variable in the current `ml` problem. Sometimes you can have multiple dependent variables, in which case they would be referenced by `$ML_y2` and so on.

this is important, that when we generate the temporary variable holding our log-likelihoods, we specify that it is to be stored as double format. This is more precise than Stata's default, and not doing this is an easy way of creating regression models that do not converge. Second, rather than calculating the logit function by hand, we can use Stata's built-in function `invlogit`. Again, this reduces the chances of a coding error. Finally, rather than calculating $1-\frac{1}{1+e^{-x_i\beta}}$, we note that this is equal to $\frac{1}{1+e^{x_i\beta}}$. This is simpler to code and incidentally also increases accuracy. Finally, we replace `lnf` to equal `lnlj`. When you write programs to use with `ml` and method `lf`, you must produce `lnf` as the final output. Because generating and replacing variables in Stata produces output, we put `quietly` around this part of the program to suppress it.

3.3 Some notes on programming in Stata

The `program` command is generally useful for writing commands that you intend to use a lot. The program above is only written so that the other `ml` commands can use it, but you can also easily write your own. For example:

```
program define hello
    version 10.1
    di as result _new(1) "Hello World"
end
```

A couple of notes on this. You can name a program/command anything you want as long as it does not conflict with an existing Stata command. There are many commands in Stata that you either never use or that are internal (used by other programs), so it is usually a good idea to check (although Stata will produce an error if the name is taken already too) by using the command `which` followed by the program name. Also, the first line in the program specifies what version of Stata I am using. This is important for backwards comparability. For example, if the syntax for the `display` command changes in some future version of Stata, the program will potentially not work if the version is not specified. However, if I am using Stata 12, the program here will still work because Stata 12 will know that this program was written for Stata 10.1, and that it should use Stata 10.1's commands. Anyways, the command will produce the following output when you type `hello` in Stata's command prompt:

```
. hello

Hello World
```

That probably is not very useful, and in general, you will probably prefer to write programs that can also receive input. By default, you can specify input for a program and Stata will store the results as local macros, that are generically numbered from 0:

```
program define write
    version 10.1
    di as result _new(1) "`0'"
    di as result "Element1: `1'; Element2: `2'"
end
```

```
. write Hallo Welt

Hallo Welt
Element1: Hallo; Element2: Welt
```

That is great, but for programs that have more complicated input relying on Stata's default is not the best option. Instead, we can use the `syntax` command to explicitly define what sort of input our command needs. For example, throughout this semester you have had to present quantities of interest in the same format, with a mean and 95% confidence interval. Most of you did that by using the `sum` and `centile` commands, which does the job but is also somewhat crude. Instead you could use this program:

```
program define qici
    version 10.1
    syntax varname, Level(cilevel)
    tempname ll ul mean
    local ll = round( (100 - c(level))/2 , .01)
    local ul = round( (100 + c(level))/2 , .01)
    qui sum ‘varlist’
    local mean = round( r(mean) , .001)
    _pctile ‘varlist’ , p(‘ll’ ‘ul’)
    local ll = round( r(r1) , .001)
    local ul = round( r(r2) , .001)
    di _new(1) as result "‘c(level)’ % CI (percentile-based) for ‘varlist’:"
    di _col(5) as result "‘mean’ [‘ll’ to ‘ul’]"
end
```

The terms `c(level)`, `r(r1)`, and `r(r2)` refer to Stata macros that are generated by other programs, Stata default setup and the `_pctile` command respectively in this case. This is the output you will get from using it:

```
. qici x1

95 % CI (percentile-based) for x1:
    1.962 [.108 to 3.89]
```

Wonderful, but as you can notice, the work needed to create this program would probably only have been worth it at the beginning of the semester. Anyways, what do you do with programs? There are essentially three options: (1) you can enter the program manually when you are playing around with Stata, (2) you can put the program in a do file, (3) you can put the program in an ado file. Option 1 is utterly useless and defeats the purpose of programming your own commands. Putting programs in your do files is a little more sensible. Note that a program and all of its subroutines (i.e. other programs that that program calls on) will have to be defined before the first time the actual command is issues. Finally, you can also save the text of a program as an ado file, and put it in a directory where Stata can find the command. That is how the Clarify and BTSCS packages work. Type `sysdir` to see the directories that Stata is looking in; `C:\ado\personal` is a good starting point.

3.4 Specify a model.

After you have defined how the likelihood is calculated, you can specify a model using the `ml model` command. The basic syntax is:

```
ml model method progname (eq1) ..., options
```

The first argument specifies which method the program uses to calculate the likelihood values. There are four different methods: `lf`, `d0`, `d1`, and `d2`. The most important difference between these methods is in the information you provide Stata when it calculates the likelihood values. The `lf` method is the easiest– your program only includes the log likelihood function and you let Stata calculate numerical derivatives to get the gradient vector and Hessian matrix. This method is by far the easiest to code, and in most cases is sufficient.

The next three methods include increasingly more information on the gradient vector and Hessian. The `d0` method also only requires a log likelihood function, but it has somewhat more complicated syntax because instead of letting `ml` sum individual ln likelihood values, your program does so manually. This is useful for certain types of likelihood functions that otherwise will not work with method `lf`. In `d1` and `d2` your program specifies functions for the gradient vector and gradient vector as well as Hessian respectively. Stata has complicated syntax to do this, i.e. it is not just as straightforward as entering the equations for each of these. The subroutines Stata adds are not just meant to complicate things, but to take account of several common mistakes that might occur if you used regular commands to calculate certain things (e.g. summing likelihood values in your dataset without dropping observations that are not in the estimation subsample). Ultimately there are only two reasons to use a `d0`, `d1`, or `d2` method: (1) the `lf` method is no appropriate for you likelihood function, or (2) you are working your way up to a program that uses the `d2` method. The `lf` method is faster and more accurate (see the bit above about machine precision) than either the `d0` or `d1` method, but the `d2` method is the fastest and most accurate. Thus for commands that are commonly used like logit and probit it may be worth the extensive coding effort to produce a faster and more accurate program.

The main point to remember is that you generally will want to use the `lf` method. It is the simplest to program, and fairly fast and accurate. Furthermore, all four of these methods build on one another, so even if you were interested in using the `d2` method, something close to the `lf` method would be your starting point anyways.

After you specify what method to use, you need to identify the program that calculates the log-likelihood values. In case it is not obvious, your program needs to match the method you are using, i.e. if you are using the `d1` method your output will have to include the log-likelihoods and values for the gradient vector.

The final step (unless you care about specifying options, which we do not at this point) is to specify your actual model. Stata does this by breaking the command up into sections for different equations. Each equation is your model specification for the parameter(s) you are estimating. In what is about to become our running logit example, we will only estimate μ, the mean of our logit distribution. Sometimes you are estimating more than one parameter though. For example, in a heteroscedastic probit model, we not only estimate the mean μ of a logit distribution, but also its variance σ. Each of those parameters would have a separate equation for it. By the way, you can name equations something other than the Stata default of `eq1`, `eq2`, etc. by putting the equation name followed by a colon before the actual equation.

Also, Stata by default includes a constant term unless you specify in the options that you do not want one. Anyways, our command would look something like this:

```
ml model lf logit2_lf (y = x1 x2)
```

We are going to model the binary outcome y as a function of two explanatory variables, $x1$ and $x2$. Stata will also automatically include a constant term, unless we tell it not to.

3.5 Make sure the program works correctly.

Even with a simple likelihood function however, there is a good chance that something will go wrong because we made a coding mistake. Thus it is a good idea to debug first. Stata provides a tool for this, the `ml check` command.

```
. ml check

Test 1:  Calling logit2_lf to check if it computes log likelihood and
         does not alter coefficient vector...
         Passed.

Test 2:  Calling logit2_lf again to check if the same log likelihood value
         is returned...
         Passed.

Test 3:  Calling logit2_lf to check if 1st derivatives are computed...
         test not relevant for method lf.

Test 4:  Calling logit2_lf again to check if the same 1st derivatives are
         returned...
         test not relevant for method lf.

Test 5:  Calling logit2_lf to check if 2nd derivatives are computed...
         test not relevant for method lf.

Test 6:  Calling logit2_lf again to check if the same 2nd derivatives are
         returned...
         test not relevant for method lf.

------------------------------------------------------------------------------
Searching for alternate values for the coefficient vector to verify that
logit2_lf returns different results when fed a different coefficient vector:

Searching...
initial:       log likelihood =     -<inf>  (could not be evaluated)
searching for feasible values +

feasible:      log likelihood = -911.81381
improving initial values ......+...
improve:       log likelihood = -828.12024

continuing with tests...
------------------------------------------------------------------------------
```

```
Test 7:  Calling logit2_lf to check log likelihood at the new values...
         Passed.

Test 8:  Calling logit2_lf requesting 1st derivatives at the new values...
         test not relevant for method lf.

Test 9:  Calling logit2_lf requesting 2nd derivatives at the new values...
         test not relevant for method lf.

------------------------------------------------------------------------------
                         logit2_lf HAS PASSED ALL TESTS
------------------------------------------------------------------------------

Test 10: Does logit2_lf produce unanticipated output?
         This is a minor issue.  Stata has been running logit2_lf with all
         output suppressed.  This time Stata will not suppress the output.
         If you see any unanticipated output, you need to place quietly in
         front of some of the commands in logit2_lf.

-------------------------------------------------------------- begin execution
---------------------------------------------------------------- end execution
```

Most of this test is self explanatory. The last section at the end runs the command without suppressing any output. Had we not used the `quietly` command in our program `logit2_lf`, this section might display unwanted output such as "`xxx missing values generated`".

What if the test returns an error? You can go back through the program and see if there are any obvious errors. Sometimes you may not be able to find one though. In those situations, it can be useful to use the `ml trace` command. There is a regular trace command in Stata (invoked by typing `set trace on`) that will show you *very* detailed output of what is going on in the background. Because the `ml` commands themselves to things in the background though, it is better to use the `ml trace` command. This will show you, line by line, how commands in your program are executed, and what output they produce. Usually this will let you pin a problem down to a specific line in your program, which makes it much easier to find mistakes.

Note that this test will only check to see whether our program calculates log-likelihood values, not whether these log-likelihood values are correct. If there is a syntax error in our program or in the likelihood function we entered in it, this test might catch it. However, it will not catch errors that lead our program to calculate false log-likelihood values. More on this issue later.

3.6 Set or search for starting values.

Now that we have defined the problem at hand, and checked the program for obvious errors, we could just type `ml maximize` and let Stata deal with the rest. We would get output like this:

```
. ml maximize

initial:       log likelihood = -693.14718
alternative:   log likelihood = -722.07698
rescale:       log likelihood =  -693.1152
```

```
Iteration 0:   log likelihood =  -693.1152
Iteration 1:   log likelihood = -482.44656
Iteration 2:   log likelihood = -481.75499
Iteration 3:   log likelihood = -481.75293
Iteration 4:   log likelihood = -481.75293

[rest of output omitted]
```

This is very similar to what happens when you use a regular Stata regression command. Actually there are two different things going on here though. When you type `ml maximize` right after issuing the `ml model` command, `ml` will in addition also search for initial starting values. That is what the first three lines in the output are about: `ml` calculates the log-likelihood for the initial starting values (usually zero), looks for alternative starting values, and then rescales those alternative starting values before starting the maximization.

Let's go back one step. After issuing the `ml model` command again, we can type `ml query` or `ml report` to look at the current maximization problem defined. The command `ml query` shows you the problem definition (i.e. how many equations, what are the varibles) and also the current starting values for each coefficient:

```
. ml query

Method:           lf
Program:          logit2_lf
Dep. variable:    y
1 equation:
        eq1:      x1 x2
Search bounds:
        eq1:           -inf       +inf
Current (initial) values:
        (zero)
```

The command `ml report` does not show you the problem definition, but it does provide exact values for the current coefficient vector, gradient vector, negative Hessian, and maximization direction:

```
. ml report

Current coefficient vector:
      eq1:   eq1:   eq1:
       x1     x2  _cons
r1      0      0      0

Value of log likelihood function = -693.14718

Gradient vector (length = 264.7477):
         eq1:      eq1:      eq1:
          x1        x2     _cons
r1  236.6323  118.6613         4

Negative Hessian matrix (concave; matrix is full rank):
                 eq1:       eq1:       eq1:
                  x1         x2      _cons
```

```
   eq1:x1   1265.164
   eq1:x2  -15.48785   84.51683
eq1:_cons   490.5204  -4.911229   250.0011

Steepest-ascent direction:
         eq1:      eq1:      eq1:
          x1        x2     _cons
r1  .8938029  .4482054  .0151087

Newton-Raphson direction (length before normalization =  2.23146):
          eq1:       eq1:       eq1:
           x1         x2      _cons
r1   .3513312    .654672  -.6693063
```

As you can see, the starting values for our coefficients right now are all zero. We can do better than that by using the `ml search` command, which does exactly the same as the maximization command did for us above:

```
. ml search
initial:       log likelihood = -693.14718
improve:       log likelihood = -693.14718
alternative:   log likelihood = -722.07698
rescale:       log likelihood =  -693.1152

. ml query

Method:           lf
Program:          logit2_lf
Dep. variable:    y
1 equation:
        eq1:      x1 x2
Search bounds:
        eq1:           -inf       +inf
Current (initial) values:
   eq1:_cons            .015625
    remaining values are zero
lnL(current values) =  -693.1152
```

Usually you can just rely on Stata to find good starting values by itself. If you feel like playing around though, or if you have a good guess at what the coefficient estimates should be, you have two other options to set starting values.

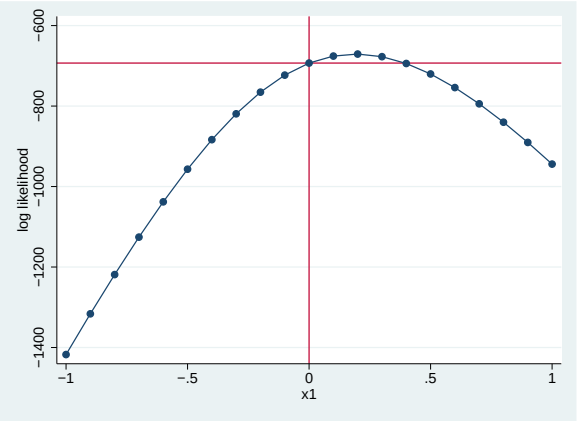
The first is by using the `ml plot` command, which graphs the log-likelihood function against *one* of the coefficients in your model:

```
. ml plot x1

                  reset x1 =         .2  (was          0)
             log likelihood = -670.75096  (was -693.14718)
```

Note that this automatically changed the starting value for β_{x1} from 0 to 0.2. You can do the same for each of the other variables. Since the overall log-likelihood depends on all three coefficients in our

Figure 1: Graph of log-likelihood values against x1



model, when you plot a coefficient and it's starting value is changed, the plots for all other coefficients will change as a result. Technically, at least in some cases, you could "maximize" a model by plotting each of the coefficients over and over again. Mostly this is just for entertainment value though.

Potentially somewhat more useful is manually setting the starting values. If we had a good idea of what our coefficient values should be, roughly, we could explicitly specify our guesses as the starting values:

```
. ml init 1 2 -2, copy

. ml query

Method:           lf
Program:          logit2_lf
Dep. variable:    y
1 equation:
        eq1:      x1 x2
Search bounds:
        eq1:           -inf       +inf
Current (initial) values:
   eq1:x1                      1
   eq1:x2                      2
   eq1:_cons                  -2
```

Typically you do not know what your coefficients roughly are though. This command can however still be useful when you have a problem at hand. If your model has problems converging (more on this later), or if you suspect that there are local maxima or something like that, you can try specifying different starting values and check if the coefficients remain the same.

3.7 Maximize.

In most cases, after you have specified a model, you can just go ahead and maximize, which is done with the `ml maximize` command:

```
. ml maximize

initial:       log likelihood = -693.14718
alternative:   log likelihood = -722.07698
rescale:       log likelihood =  -693.1152
Iteration 0:   log likelihood =  -693.1152
Iteration 1:   log likelihood = -482.44656
Iteration 2:   log likelihood = -481.75499
Iteration 3:   log likelihood = -481.75293
Iteration 4:   log likelihood = -481.75293

                                                Number of obs   =       1000
                                                Wald chi2(2)    =     243.54
Log likelihood = -481.75293                     Prob > chi2     =     0.0000

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   1.164907   .0869987    13.39   0.000     .9943932    1.335422
          x2 |   2.163466   .1628748    13.28   0.000     1.844237    2.482695
       _cons |  -2.210133   .1852712   -11.93   0.000    -2.573258   -1.847008
------------------------------------------------------------------------------
```

That is slightly different from the output you get when you use Stata's logit command–there is not model title, the results show a Wald χ^2 test rather than a likelihood-ratio test, and there is no Pseudo-R^2. The first two we can easily fix by specifying some options on the `ml model` command (the LR test is between our full model and a constant-only model, so we run that first to get a log-likelihood value):

```
. ml model lf logit2_lf (y =), maximize

initial:       log likelihood = -693.14718
alternative:   log likelihood = -722.07698
rescale:       log likelihood =  -693.1152
Iteration 0:   log likelihood =  -693.1152
Iteration 1:   log likelihood = -693.11518

. ml model lf logit2_lf (y = x1 x2), title(Logistic regression) lf0(1 'e(ll)')

. ml maximize

initial:       log likelihood = -693.14718
alternative:   log likelihood = -722.07698
rescale:       log likelihood =  -693.1152
Iteration 0:   log likelihood =  -693.1152
Iteration 1:   log likelihood = -482.44656
Iteration 2:   log likelihood = -481.75499
Iteration 3:   log likelihood = -481.75293
Iteration 4:   log likelihood = -481.75293

Logistic regression                             Number of obs   =       1000
                                                LR chi2(2)      =     422.72
Log likelihood = -481.75293                     Prob > chi2     =     0.0000

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
```

```
          x1 |   1.164907   .0869987    13.39   0.000     .9943932    1.335422
          x2 |   2.163466   .1628748    13.28   0.000     1.844237    2.482695
       _cons |  -2.210133   .1852712   -11.93   0.000    -2.573258   -1.847008
------------------------------------------------------------------------------
```

Minus the Pseudo-R^2, that is exactly what using Stata's logit command produced.

3.8 Check for errors.

In this example, everything went smoothly, and we had no convergence problems. In practice, especially with more complicated likelihood functions, you will usually get difficulties. There are a couple of notes in the output that can give you and indication that there might be problems. These are the most common indications you will see:

```
initial:       log likelihood =     -<inf>  (could not be evaluated)
...
Iteration #:   log likelihood = -482.44656  (not concave)
...
Iteration #:   log likelihood = -481.75293  (backed up)
```

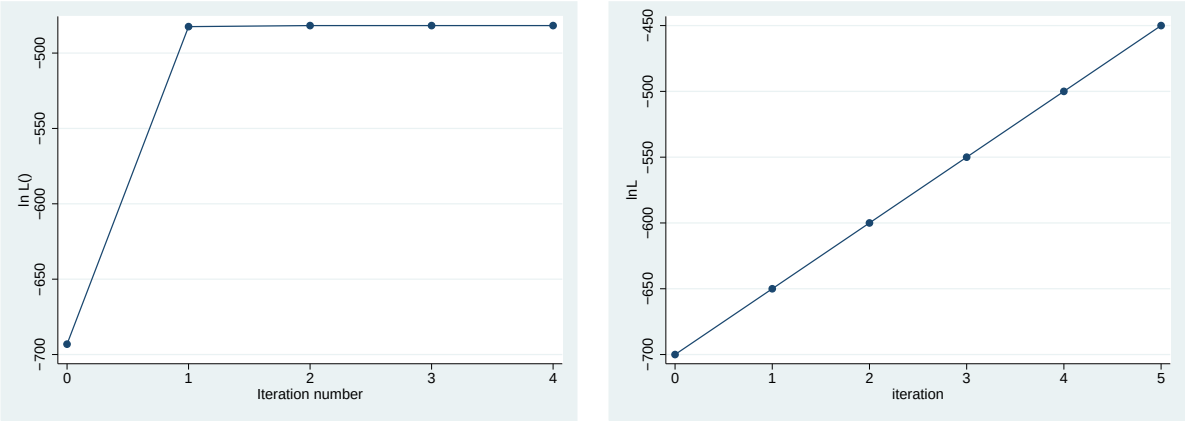
The first line more or less just shows you that the initial starting values (default is zero) were just so incredibly bad that the resulting log-likelihood was too small for Stata to keep track of it. Usually this is not an issue, although you might want to manually set different starting values next time.

The second line shows that at the current position in the parameter space, $-H(\theta_i)$ is not invertible. In words, given the current vector of coefficient estimates, some part of the gradient vector appears to be zero, like in a ridge, saddle point, or flat section. Usually this is not because the actual likelihood function has these features (nice likelihood functions do not), but because the slope is so small that Stata cannot distinguish it due to roundoff error. So Stata uses different mechanisms to determine which direction to go in. This is not a problem unless it occurs in the last few iterations, where it might indicate convergence problems.

The third line indicates that Stata backed up multiple times in the original step it took. Given a direction and step size, Stata expects the log-likelihood for the new location to be higher than that for the old location, or formally, it expects that $f(\beta_k) = f(\beta_{k-1} + \lambda_{k-1}\Delta_{k-1}) > f(\beta_{k-1})$, where $f(x)$ indicates the log-likelihood function. If that is not the case, it decreases θ_{k-1}, or the step size, until the new log-likelihood is satisfactory. Again, this is not a problem if it occur early on in maximization, but could indicate convergence problems if it happens later on.

Another way to check for potential convergence problems is to graph how the log-likelihood was maximized using the command `ml graph`. Figure 2 shows the result. The graph on the left shows convergence for the model we just maximized, and the result looks very good. You want the slope in the last few iterations to approach zero (indicating we are close to a maximum), and it does in our case. The graph on the right is made up, but shows what bad convergence would look like–the slope of the line is not even close to zero in the last few iterations, indicating that we have not yet reached a maximum.

Figure 2: Graphing convergence



3.9 What to do with convergence problems.

First, if there is a convergence problem or you suspect that there is one, you should disregard any results you get. Second, you can try to fix the convergence problem. The best problem to have is that something is wrong with your log-likelihood function or how you coded it. So if you have constant convergence problems, you might want to go back and check them (especially the coding, since it is easy to make small but significant mistakes) for accuracy. Sometimes you might just be dealing with a difficult maximization problem even though your likelihood function itself is sound and implemented well. There are three basic solutions to this in Stata. You should try all three of them (in any combination) to see if your final results agree (they should).

First, you can try using a different maximization routine. Newton-Rhapson is the default and usually works well, but sometimes is may not handle a particular likelihood function very well. To switch likelihood functions, you have to specify the `technique()` option for the `ml model` command, with the possible choices being `nr`, `bhhh`, `dfp`, and `bfgs`. Newton-Raphson is the default. You can also switch between different algorithms and tell Stata after how many iterations to switch:

```
ml model ... , tech(bhhh 2 bfgs 2 nr 5)
```

When you type `ml maximize`, this would lead Stata to use BHHH for 2 iterations, use BFGS for 2 iterations, use Newton-Raphson for 5 iterations, and repeat from start until convergence is achieved.

Second, you can specify the `difficult` option for the `ml maximize` command. This causes Stata to undertake some additional computations involving the negative Hession matrix (additional computation = slower) that may help for difficult maximization problems. Simply type `ml maximize, difficult` to do this.

Third, you can try setting different starting values, as you saw above.

4 Syntax and ado files.

Everything so far has been contained in a do file which defined each program before it was called in the command prompt. Alternatively, we can write and ado file and place it somewhere Stata can find it. Most of the work in writing ado files for estimation commands is in parsing syntax. To that end, Stata estimation commands typically have at least two ado files–one for parsing syntax, and another that contains the log-likelihood program. For example, if you look at the ado file for Stata's logit command (`logit.ado`), you will find a very complicated file whose sole purpose is to parse syntax and display results. It has no log-likelihood function or anything like that. So if we want to put our logit command into a separate do file, we can do the same by (1) putting our log-likelihood program in an ado file named "`logit2_lf.ado`", and (2) by creating a new, syntax-parsing "`logit2.ado`". The later, in a very simple version, looks like this:

```
program logit2
    version 10.1
    syntax varlist [if] [in] [, robust]

    //  Separate the d.v. from i.v.(s).
    gettoken y xb : varlist

    //  Set type of vce to estimate (i.e. oim or robust).
    local vcetype = "oim"
    if "`robust'" != "" {
        local vcetype = "robust"
    }

    //  Fit constant-only model for LR Test.
    qui ml model lf logit2_lf (`y' =), maximize

    //  Specify and fit full model.
    ml model lf logit2_lf (`y' = `xb') ///
        `if' `in' ///
        , title(Logistic) lf0(1 `e(ll)') vce(`vcetype')
    ml maximize
end
```

This program parses the variable list to get a dependent variable and explanatory variables, it works with the `if` and `in` options, and it also lets us specify whether we want robust standard errors. Note that it additionally incorporates the likelihood ratio test against a constant-only model that we used above. Now if we place the two ado files we created in a directory Stata can find, we can open Stata and estimate our model using the command `logit2`:

```
. logit2 y x1 x2

initial:       log likelihood = -693.14718
alternative:   log likelihood = -722.07698
rescale:       log likelihood =  -693.1152
Iteration 0:   log likelihood =  -693.1152
Iteration 1:   log likelihood = -482.44656
Iteration 2:   log likelihood = -481.75499
Iteration 3:   log likelihood = -481.75293
```

```
Iteration 4:   log likelihood = -481.75293

Logistic                                          Number of obs   =       1000
                                                  LR chi2(2)      =     422.72
Log likelihood = -481.75293                       Prob > chi2     =     0.0000

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   1.164907   .0869987    13.39   0.000     .9943932    1.335422
          x2 |   2.163466   .1628748    13.28   0.000     1.844237    2.482695
       _cons |  -2.210133   .1852712   -11.93   0.000    -2.573258   -1.847008
------------------------------------------------------------------------------
```

Obviously this would be much more useful had we not just replicated an existing Stata command. But at least I know that the ado files are correct (assuming Stata is correct with it's logit command).

5 Simulation

I mentioned above that there are some errors that Stata does not pick up because they are not syntax errors. One way of finding such errors is through simulation: generate junk ("synthetic") data that follow a data-generating process that is known to you, meaning that you also know what the true coefficient values of a fully specified model are, and then estimate the corresponding maximum-likelihood estimator. The estimated coefficients should asymptotically be unbiased, i.e. as your sample size gets very large your estimated coefficients should be very close to the "true" values that you know because you generated the data using them.

As it turns out, the data I have used for the example so far was generated in that way (hence the nice convergence), using this program:

```
program define syndata
    version 10.1
    args obs
    qui {
        set obs `obs'
        gen x1 = uniform()*4
        gen x2 = uniform()*2 - 1
        gen e = logit(uniform())
        scalar b0 = -2
        scalar b1 = 1
        scalar b2 = 2
        gen l_y  = scalar(b0)+scalar(b1)*x1+scalar(b2)*x2+e
        gen pr_y = invlogit(scalar(b0)+scalar(b1)*x1+scalar(b2)*x2+e)
        gen y = (pr_y>=0.5)     // (l_y>=0) produces the exact same.
    }
end
```

The program accepts requires that you specify how many observations you want there to be in your new junk dataset, and then creates data that follows the dgp $y = f(-2 + 1 \times x_1 + 2 \times x_2)$, where x_1 is uniformly distributed from 0 to 4, and x_2 is uniformly distributed from -1 to 1. By the way, the error

term needs to be logistically distributed for this to work. With a normally distributed error term we get a data-generating process appropriate for probit.

Since we know what the true coefficient values should be, we can now see if there are errors by checking whether our estimated coefficient values are close to what they should be (since MLE is only asymptotically unbiased and we have randomly-generated error, we can expect some divergence). They have been so far, and if we increase the number of observations, they get closer.

```
. clear

. syndata 100000

. logit y x1 x2

Iteration 0:   log likelihood = -69314.606
Iteration 1:   log likelihood = -51326.404
Iteration 2:   log likelihood = -50167.294
Iteration 3:   log likelihood = -50130.593
Iteration 4:   log likelihood = -50130.536

Logistic regression                                     Number of obs   =     100000
                                                        LR chi2(2)      =   38368.14
                                                        Prob > chi2     =     0.0000
Log likelihood = -50130.536                             Pseudo R2       =     0.2768

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   .9950037    .007789   127.74   0.000     .9797375     1.01027
          x2 |   1.992261   .0156321   127.45   0.000     1.961623      2.0229
       _cons |  -1.994104   .0174458  -114.30   0.000    -2.028297   -1.959911
------------------------------------------------------------------------------
```

References

Gould, William, Jeffrey Pitblado and William Sribney. 2006. *Maximum Likelihood Estimation with Stata*. 3rd ed. College Station, TX: Stata Press.