Maximum Likelihood Estimation with Stata

Fourth Edition

WILLIAM GOULD
StataCorp

JEFFREY PITBLADO
StataCorp

BRIAN POI
StataCorp

A Stata Press Publication
StataCorp LP
College Station, Texas

First edition 1999
Second edition 2003
Third edition 2006
Fourth edition 2010

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845
Typeset in $\LaTeX\,2_\varepsilon$
Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN-10: 1-59718-078-5
ISBN-13: 978-1-59718-078-8

Library of Congress Control Number: 2010935284

Contents

(Pages omitted)

Preface to the fourth edition

Maximum Likelihood Estimation with Stata, Fourth Edition is written for researchers in all disciplines who need to compute maximum likelihood estimators that are not available as prepackaged routines. To get the most from this book, you should be familiar with Stata, but you will not need any special programming skills, except in chapters 13 and 14, which detail how to take an estimation technique you have written and add it as a new *command* to Stata. No special theoretical knowledge is needed either, other than an understanding of the likelihood function that will be maximized.

Stata's `ml` command was greatly enhanced in Stata 11, prescribing the need for a new edition of this book. The optimization engine underlying `ml` was reimplemented in Mata, Stata's matrix programming language. That allowed us to provide a suite of commands (not discussed in this book) that Mata programmers can use to implement maximum likelihood estimators in a matrix programming language environment; see [M-5] **moptimize( )**. More important to users of `ml`, the transition to Mata provided us the opportunity to simplify and refine the syntax of various `ml` commands and likelihood evaluators; and it allowed us to provide a framework whereby users could write their likelihood-evaluator functions using Mata while still capitalizing on the features of `ml`.

Previous versions of `ml` had just two types of likelihood evaluators. Method-`lf` evaluators were used for simple models that satisfied the linear-form restrictions and for which you did not want to supply analytic derivatives. `d`-family evaluators were for everything else. Now `ml` has more evaluator types with both long and short names:

Short name	Long name
`lf`	`linearform`
`lf0`	`linearform0`
`lf1`	`linearform1`
`lf1debug`	`linearform1debug`
`lf2`	`linearform2`
`lf2debug`	`linearform2debug`
`d0`	`derivative0`
`d1`	`derivative1`
`d1debug`	`derivative1debug`
`d2`	`derivative2`
`d2debug`	`derivative2debug`
`gf0`	`generalform0`

You can specify either name when setting up your model using `ml model`; however, out of habit, we use the short name in this book and in our own software development work. Method `lf`, as in previous versions, does not require derivatives and is particularly easier to use.

Chapter 1 provides a general overview of maximum likelihood estimation theory and numerical optimization methods, with an emphasis on the practical implications of each for applied work. Chapter 2 provides an introduction to getting Stata to fit your model by maximum likelihood. Chapter 3 is an overview of the `ml` command and the notation used throughout the rest of the book. Chapters 4–10 detail, step by step, how to use Stata to maximize user-written likelihood functions. Chapter 11 shows how to write your likelihood evaluators in Mata. Chapter 12 describes how to package all the user-written code in a do-file so that it can be conveniently reapplied to different datasets and model specifications. Chapter 13 details how to structure the code in an ado-file to create a new Stata estimation command. Chapter 14 shows how to add survey estimation features to existing `ml`-based estimation commands.

Chapter 15, the final chapter, provides examples. For a set of estimation problems, we derive the log-likelihood function, show the derivatives that make up the gradient and Hessian, write one or more likelihood-evaluation programs, and so provide a fully functional estimation command. We use the estimation command to fit the model to a dataset. An estimation command is developed for each of the following:

- Logit and probit models
- Linear regression
- Weibull regression
- Cox proportional hazards model
- Random-effects linear regression for panel data
- Seemingly unrelated regression

Appendices contain full syntax diagrams for all the `ml` subroutines, useful checklists for implementing each maximization method, and program listings of each estimation command covered in chapter 15.

We acknowledge William Sribney as one of the original developers of `ml` and the principal author of the first edition of this book.

College Station, TX
September 2010

William Gould
Jeffrey Pitblado
Brian Poi

(Pages omitted)

2 Introduction to ml

`ml` is the Stata command to maximize user-defined likelihoods. Obtaining maximum likelihood (ML) estimates requires the following steps:

1. Derive the log-likelihood function from your probability model.
2. Write a program that calculates the log-likelihood values and, optionally, its derivatives. This program is known as a likelihood evaluator.
3. Identify a particular model to fit using your data variables and the `ml model` statement.
4. Fit the model using `ml maximize`.

This chapter illustrates steps 2, 3, and 4 using the probit model for dichotomous (0/1) variables and the linear regression model assuming normally distributed errors.

In this chapter, we fit our models explicitly, handling each coefficient and variable individually. New users of `ml` will appreciate this approach because it closely reflects how you would write down the model you wish to fit on paper; and it allows us to focus on some of the basic features of `ml` without becoming overly encumbered with programming details. We will also illustrate this strategy's shortcomings so that once you become familiar with the basics of `ml` by reading this chapter, you will want to think of your model in a slightly more abstract form, providing much more flexibility.

In the next chapter, we discuss `ml`'s probability model parameter notation, which is particularly useful when, as is inevitably the case, you decide to change some of the variables appearing in your model. If you are already familiar with `ml`'s θ-parameter notation, you can skip this chapter with virtually no loss of continuity with the rest of the book.

Chapter 15 contains the derivations of log-likelihood functions (step 1) for models discussed in this book.

2.1 The probit model

Say that we want to fit a probit model to predict whether a car is foreign or domestic based on its weight and price using the venerable `auto.dta` that comes with Stata. Our statistical model is

$$
\begin{aligned}
\pi_j &= \Pr(\texttt{foreign}_j \mid \texttt{weight}_j, \texttt{price}_j) \\
&= \Phi(\beta_1 \texttt{weight}_j + \beta_2 \texttt{price}_j + \beta_0)
\end{aligned}
$$

where we use the subscript j to denote observations and $\Phi(\cdot)$ denotes the standard normal distribution function. The log likelihood for the jth observation is

$$
\ln \ell_j = \begin{cases} \ln \Phi(\beta_1 \texttt{weight}_j + \beta_2 \texttt{price}_j + \beta_0) & \text{if } \texttt{foreign}_j = 1 \\ 1 - \ln \Phi(\beta_1 \texttt{weight}_j + \beta_2 \texttt{price}_j + \beta_0) & \text{if } \texttt{foreign}_j = 0 \end{cases}
$$

Because the normal density function is symmetric about zero, $1 - \Phi(w) = \Phi(-w)$, and computers can more accurately calculate the latter than the former. Therefore, we are better off writing the log likelihood as

$$
\ln \ell_j = \begin{cases} \ln \Phi(\beta_1 \texttt{weight}_j + \beta_2 \texttt{price}_j + \beta_0) & \text{if } \texttt{foreign}_j = 1 \\ \ln \Phi(-\beta_1 \texttt{weight}_j - \beta_2 \texttt{price}_j - \beta_0) & \text{if } \texttt{foreign}_j = 0 \end{cases} \tag{2.1}
$$

With our log-likelihood function in hand, we write a program to evaluate it:

begin `myprobit_gf0.ado`

```
program myprobit_gf0
        args todo b lnfj
        tempvar xb
        quietly generate double `xb' = `b'[1,1]*weight + `b'[1,2]*price + ///
                                       `b'[1,3]
        quietly replace `lnfj' = ln(normal(`xb')) if foreign == 1
        quietly replace `lnfj' = ln(normal(-1*`xb')) if foreign == 0
end
```

end `myprobit_gf0.ado`

We named our program `myprobit_gf0.ado`, but you could name it anything you want as long as it has the extension `.ado`. The name without the `.ado` extension is what we use to tell `ml model` about our likelihood function. We added `gf0` to our name to emphasize that our evaluator is a general-form problem and that we are going to specify no (0) derivatives. We will return to this issue when we use the `ml model` statement.

Our program accepts three arguments. The first, `todo`, we can safely ignore for now. In later chapters, when we discuss other types of likelihood-evaluator programs, we will need that argument. The second, `b`, contains a row vector containing the parameters of our model (β_0, β_1, and β_2). The third argument, `lnfj`, is the name of a temporary variable that we are to fill in with the values of the log-likelihood function evaluated at the coefficient vector `b`. Our program then created a temporary variable to hold the values of $\beta_1 \texttt{weight}_j + \beta_2 \texttt{price}_j + \beta_0$. We created that variable to have storage type `double`; we will discuss this point in greater detail in the next chapter, but for now you should remember that when coding your likelihood evaluator, you must create temporary variables as `double`s. The last two lines replace `lnfj` with the values of the log likelihood for $\texttt{foreign}_j$ equal to 0 and 1, respectively. Because `b` and `lnfj` are arguments passed to our program and `xb` is a temporary variable we created with the `tempvar` commands, their names are local macros that must be dereferenced using left- and right-hand single quote marks to use them; see [U] **18.7 Temporary objects**.

The next step is to identify our model using `ml model`. To that end, we type

```
. sysuse auto
(1978 Automobile Data)
. ml model gf0 myprobit_gf0 (foreign = weight price)
```

We first loaded in our dataset, because `ml model` will not work without the dataset in memory. Next we told `ml` that we have a method-`gf0` likelihood evaluator named `myprobit_gf0`, our dependent variable is `foreign`, and our independent variables are `weight` and `price`. In subsequent chapters, we examine all the likelihood-evaluator types; method-`gf0` (general form) evaluator programs most closely follow the mathematical notation we used in (2.1) and are therefore perhaps easiest for new users of `ml` to grasp, but we will see that they have disadvantages as well. General-form evaluators simply receive a vector of parameters and a variable into which the observations' log-likelihood values are to be stored.

The final step is to tell `ml` to maximize the likelihood function and report the coefficients:

```
. ml maximize
initial:       log likelihood = -51.292891
alternative:   log likelihood = -45.055272
rescale:       log likelihood = -45.055272
Iteration 0:   log likelihood = -45.055272
Iteration 1:   log likelihood = -20.770386
Iteration 2:   log likelihood = -18.023563
Iteration 3:   log likelihood = -18.006584
Iteration 4:   log likelihood = -18.006571
Iteration 5:   log likelihood = -18.006571
                                                  Number of obs   =         74
                                                  Wald chi2(2)    =      14.09
Log likelihood = -18.006571                       Prob > chi2     =     0.0009

------------------------------------------------------------------------------
     foreign |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      weight |  -.003238    .0008643    -3.75   0.000     -.004932   -.0015441
       price |   .000517    .0001591     3.25   0.001     .0002052    .0008287
       _cons |  4.921935    1.330066     3.70   0.000     2.315054    7.528816
------------------------------------------------------------------------------
```

You can verify that we would obtain identical results using `probit`:

```
. probit foreign weight price
```

This example was straightforward because we had only one equation and no auxiliary parameters. Next we consider linear regression with normally distributed errors.

2.2 Normal linear regression

Now suppose we want to fit a linear regression of `turn` on `length` and `headroom`:

$$\texttt{turn}_j = \beta_1 \texttt{length}_j + \beta_2 \texttt{headroom}_j + \beta_3 + \epsilon_j$$

where ϵ_j is an error term. If we assume that each ϵ_j is independent and identically distributed as a normal random variable with mean zero and variance σ^2, we have what is often called normal linear regression; and we can fit the model by ML. As derived in section 15.3, the log likelihood for the jth observation assuming homoskedasticity (constant variance) is

$$\ln \ell_j = \ln \phi \left(\frac{\texttt{turn}_j - \beta_1 \texttt{length}_j - \beta_2 \texttt{headroom}_j - \beta_3}{\sigma} \right) - \ln \sigma$$

There are four parameters in our model: β_1, β_2, β_3, and σ, so we will specify our `ml model` statement so that our likelihood evaluator receives a vector of coefficients with four columns. As a matter of convention, we will use the four elements of that vector in the order we just listed so that, for example, β_2 is the second element and σ is the fourth element. Our likelihood-evaluator program is

begin `mynormal1_gf0.ado`

```
program mynormal1_gf0
        args todo b lnfj
        tempvar xb
        quietly generate double `xb' = `b'[1,1]*length +                ///
                                       `b'[1,2]*headroom + `b'[1,3]
        quietly replace `lnfj' = ln(normalden((turn - `xb')/`b'[1,4])) - ///
                                 ln(`b'[1,4])
end
```

end `mynormal1_gf0.ado`

In our previous example, when we typed

```
. ml model gf0 myprobit_gf0 (foreign = weight price)
```

`ml` knew to create a coefficient vector with three elements because we specified two right-hand-side variables, and by default `ml` includes a constant term unless we specify the `noconstant` option, which we discuss in the next chapter. How do we get `ml` to include a fourth parameter for σ? The solution is to type

```
. ml model gf0 mynormal1_gf0 (turn = length headroom) /sigma
```

The notation `/sigma` tells `ml` to include a fourth element in our coefficient vector and to label it `sigma` in the output. Having identified our model, we can now maximize the log-likelihood function:

```
. ml maximize
initial:       log likelihood =     -<inf>  (could not be evaluated)
feasible:      log likelihood =  -8418.567
rescale:       log likelihood = -327.16314
rescale eq:    log likelihood = -215.53986
Iteration 0:   log likelihood = -215.53986  (not concave)
Iteration 1:   log likelihood = -213.33272  (not concave)
Iteration 2:   log likelihood = -211.10519  (not concave)
Iteration 3:   log likelihood =  -209.6059  (not concave)
Iteration 4:   log likelihood = -207.93809  (not concave)
Iteration 5:   log likelihood = -206.43891  (not concave)
Iteration 6:   log likelihood =  -205.1962  (not concave)
Iteration 7:   log likelihood = -204.11317  (not concave)
Iteration 8:   log likelihood = -203.00323  (not concave)
Iteration 9:   log likelihood =  -202.1813  (not concave)
Iteration 10:  log likelihood = -201.42353  (not concave)
Iteration 11:  log likelihood = -200.64586  (not concave)
Iteration 12:  log likelihood =  -199.9028  (not concave)
Iteration 13:  log likelihood = -199.19009  (not concave)
Iteration 14:  log likelihood = -198.48271  (not concave)
Iteration 15:  log likelihood = -197.78686  (not concave)
Iteration 16:  log likelihood = -197.10722  (not concave)
Iteration 17:  log likelihood = -196.43923  (not concave)
Iteration 18:  log likelihood = -195.78098  (not concave)
Iteration 19:  log likelihood = -195.13352  (not concave)
Iteration 20:  log likelihood = -194.49664  (not concave)
Iteration 21:  log likelihood = -193.86938  (not concave)
Iteration 22:  log likelihood = -193.25148  (not concave)
Iteration 23:  log likelihood = -192.64285  (not concave)
Iteration 24:  log likelihood = -192.04319  (not concave)
Iteration 25:  log likelihood = -191.45242  (not concave)
Iteration 26:  log likelihood = -190.87034  (not concave)
Iteration 27:  log likelihood = -190.29685  (not concave)
Iteration 28:  log likelihood = -189.73203  (not concave)
Iteration 29:  log likelihood = -189.17561  (not concave)
Iteration 30:  log likelihood = -188.62745
Iteration 31:  log likelihood = -177.20678  (backed up)
Iteration 32:  log likelihood = -163.35109
Iteration 33:  log likelihood = -163.18766
Iteration 34:  log likelihood = -163.18765
                                                  Number of obs   =         74
                                                  Wald chi2(2)    =     219.18
Log likelihood = -163.18765                       Prob > chi2     =     0.0000

------------------------------------------------------------------------------
        turn |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
eq1          |
      length |   .1737845   .0134739    12.90   0.000     .1473762    .2001929
    headroom |  -.1542077   .3546293    -0.43   0.664    -.8492684     .540853
       _cons |   7.450477   2.197352     3.39   0.001     3.143747    11.75721
-------------+----------------------------------------------------------------
sigma        |
       _cons |   2.195259   .1804491    12.17   0.000     1.585585    2.548932
------------------------------------------------------------------------------
```

The point estimates match those we obtain from typing

```
. regress turn length headroom
```

The standard errors differ by a factor of sqrt(71/74) because `regress` makes a small-sample adjustment in estimating the error variance. In the special case of linear regression, the need for a small-sample adjustment is not difficult to prove. However, in general ML estimators are only justified asymptotically, so small-sample adjustments have dubious value.

❑ Technical note

The log-likelihood function is only defined for $\sigma > 0$—standard deviations must be nonnegative and ln(0) is not defined. `ml` assumes that all coefficients can take on any value, but it is designed to gracefully handle situations where this is not the case. For example, the output indicates that at the initial values (all four coefficients set to zero), `ml` could not evaluate the log-likelihood function; but `ml` was able to find alternative values with which it could begin the optimization process. In other models, you may have coefficients that are restricted to be in the range $(0,1)$ or $(-1,1)$, and in those cases, `ml` is often unsuccessful in finding feasible initial values. The best solution is to reparameterize the likelihood function so that all the parameters appearing therein are unrestricted; subsequent chapters contain examples where we do just that.

❑

2.3 Robust standard errors

Robust standard errors are commonly reported nowadays along with linear regression results because they allow for correct statistical inference even when the tenuous assumption of homoskedasticity is not met. Cluster–robust standard errors can be used when related observations' errors are correlated. Obtaining standard errors with most estimation commands is trivial: you just specify the option `vce(robust)` or `vce(cluster` *id*`)`, where *id* is the name of a variable identifying groups. Using our previous regression example, you might type

```
. regress turn length headroom, vce(robust)
```

For the evaluator functions we have written so far, both of which have been method `gf0`, obtaining robust or cluster–robust standard errors is no more difficult than with other estimation commands. To refit our linear regression model, obtaining robust standard errors, we type

```
. ml model gf0 mynormal1_gf0 (turn = length headroom) /sigma, vce(robust)
. ml maximize, nolog
initial:       log pseudolikelihood =     -<inf>  (could not be evaluated)
feasible:      log pseudolikelihood =  -8418.567
rescale:       log pseudolikelihood = -327.16314
rescale eq:    log pseudolikelihood = -215.53986
                                                  Number of obs   =         74
                                                  Wald chi2(2)    =     298.85
Log pseudolikelihood = -163.18765                 Prob > chi2     =     0.0000

------------------------------------------------------------------------------
             |               Robust
        turn |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
eq1          |
      length |   .1737845   .0107714    16.13   0.000      .152673    .1948961
    headroom |  -.1542077   .2955882    -0.52   0.602      -.73355    .4251345
       _cons |   7.450477   1.858007     4.01   0.000      3.80885     11.0921
-------------+----------------------------------------------------------------
sigma        |
       _cons |   2.195259   .2886183     7.61   0.000     1.629577     2.76094
------------------------------------------------------------------------------
```

`ml model` accepts `vce(cluster` *id*`)` with method-`gf0` evaluators just as readily as it accepts `vce(robust)`.

Being able to obtain robust standard errors just by specifying an option to `ml model` should titillate you. When we discuss other types of evaluator programs, we will see that in fact there is a lot of work happening behind the scenes to produce robust standard errors. With method-`gf0` evaluators (and other linear-form evaluators), `ml` does all the work for you.

2.4 Weighted estimation

Stata provides four types of weights that the end-user can apply to estimation problems. Frequency weights, known as `fweights` in the Stata vernacular, represent duplicated observations; instead of having five observations that record identical information, `fweights` allow you to record that observation once in your dataset along with a frequency weight of 5, indicating that observation is to be repeated a total of five times. Analytic weights, called `aweights`, are inversely proportional to the variance of an observation and are used with group-mean data. Sampling weights, called `pweights`, denote the inverse of the probability that an observation is sampled and are used with survey data where some people are more likely to be sampled than others. Importance weights, called `iweights`, indicate the relative "importance" of the observation and are intended for use by programmers who want to produce a certain computation.

Obtaining weighted estimates with method-`gf0` likelihood evaluators is the same as with most other estimation commands. Suppose that in `auto.dta`, `rep78` is actually a frequency weight variable. To obtain frequency-weighted estimates of our probit model, we type

```
. ml model gf0 myprobit_gf0 (foreign = weight price) [fw = rep78]
. ml maximize
initial:       log likelihood = -162.88959
alternative:   log likelihood = -159.32929
rescale:       log likelihood = -156.55825
Iteration 0:   log likelihood = -156.55825
Iteration 1:   log likelihood = -72.414357
Iteration 2:   log likelihood =  -66.82292
Iteration 3:   log likelihood = -66.426129
Iteration 4:   log likelihood = -66.424675
Iteration 5:   log likelihood = -66.424675
                                                  Number of obs   =        235
                                                  Wald chi2(2)    =      58.94
Log likelihood = -66.424675                       Prob > chi2     =     0.0000

------------------------------------------------------------------------------
     foreign |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      weight |  -.0027387   .0003576    -7.66   0.000    -.0034396   -.0020379
       price |   .0004361   .0000718     6.07   0.000     .0002953    .0005768
       _cons |   4.386445   .5810932     7.55   0.000     3.247523    5.525367
------------------------------------------------------------------------------
```

Just like with obtaining robust standard errors, we did not have to do anything to our likelihood-evaluator program. We just added a weight specification, and `ml` did all the heavy lifting to make that work. You should be impressed. Other evaluator types require you to account for weights yourself, which is not always a trivial task.

2.5 Other features of method-gf0 evaluators

In addition to easily obtaining robust standard errors and weighted estimates, method-`gf0` likelihood evaluators provide several other features. By specifying the `svy` option to `ml model`, you can obtain results that take into account the complex survey design of your data. Before using the `svy` option, you must first `svyset` your data; see [U] **26.19 Survey data**.

You can restrict the estimation sample by using `if` and `in` conditions in your `ml model` statement. Again, method-`gf0` evaluators require you to do nothing special to make them work. See [U] **11 Language syntax** to learn about `if` and `in` qualifiers.

2.6 Limitations

We have introduced `ml` using method-`gf0` evaluators because they align most closely with the way you would write the likelihood function for a specific model. However, writing your likelihood evaluator in terms of a particular model with prespecified variables severely limits your flexibility.

For example, say that we had a binary variable `good` that we wanted to use instead of `foreign` as the dependent variable in our probit model. If we simply change our `ml model` statement to read

```
. ml model gf0 myprobit_gf0 (good = weight price)
```

the output from `ml maximize` will label the dependent variable as `good`, but the output will otherwise be unchanged! When we wrote our likelihood-evaluator program, we hardcoded in the name of the dependent variable. As far as our likelihood-evaluator program is concerned, changing the dependent variable in our `ml model` statement did nothing.

When you specify the dependent variable in your `ml model` statement, `ml` stores the variable name in the global macro `$ML_y1`. Thus a better version of our `myprobit_gf0` program would be

```
begin myprobit_gf0_good.ado
program myprobit_gf0_good
        args todo b lnfj
        tempvar xb
        quietly generate double `xb' = `b'[1,1]*weight + `b'[1,2]*price + ///
                                        `b'[1,3]
        quietly replace `lnfj' = ln(normal(`xb')) if $ML_y1 == 1
        quietly replace `lnfj' = ln(normal(-1*`xb')) if $ML_y1 == 0
end
end myprobit_gf0_good.ado
```

With this change, we can specify dependent variables at will.

Adapting our program to accept an arbitrary dependent variable was straightforward. Unfortunately, making it accept an arbitrary set of independent variables is much more difficult. We wrote our likelihood evaluator assuming that the coefficient vector `` `b' `` had three elements, and we hardcoded the names of our independent variables in the likelihood-evaluator program. If we were hell-bent on making our method-`gf0` evaluator work with an arbitrary number of independent variables, we could examine the column names of `` `b' `` and deduce the number of variables, their names, and even the number of equations. In the next chapter, we will learn a better way to approach problems using `ml` that affords us the ability to change regressors without having to modify our evaluator program in any way.