

# 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



© Copyright © 1999, 2003, 2006, 2010 by StataCorp LP  
All rights reserved. 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 L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>

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

No part of this book may be reproduced, stored in a retrieval system, or transcribed, in any form or by any means—electronic, mechanical, photocopy, recording, or otherwise—without the prior written permission of StataCorp LP.

Stata, Mata, NetCourse, and Stata Press are registered trademarks of StataCorp LP. L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> is a trademark of the American Mathematical Society.

# Contents

	List of tables	xiii
	List of figures	xv
	Preface to the fourth edition	xvii
	Versions of Stata	xix
	Notation and typography	xxi
<b>1</b>	<b>Theory and practice</b>	<b>1</b>
1.1	The likelihood-maximization problem . . . . .	2
1.2	Likelihood theory . . . . .	4
1.2.1	All results are asymptotic . . . . .	8
1.2.2	Likelihood-ratio tests and Wald tests . . . . .	9
1.2.3	The outer product of gradients variance estimator . . . . .	10
1.2.4	Robust variance estimates . . . . .	11
1.3	The maximization problem . . . . .	13
1.3.1	Numerical root finding . . . . .	13
	Newton's method . . . . .	13
	The Newton–Raphson algorithm . . . . .	15
1.3.2	Quasi-Newton methods . . . . .	17
	The BHHH algorithm . . . . .	18
	The DFP and BFGS algorithms . . . . .	18
1.3.3	Numerical maximization . . . . .	19
1.3.4	Numerical derivatives . . . . .	20
1.3.5	Numerical second derivatives . . . . .	24
1.4	Monitoring convergence . . . . .	25

<b>2</b>	<b>Introduction to ml</b>	<b>29</b>
2.1	The probit model . . . . .	29
2.2	Normal linear regression . . . . .	32
2.3	Robust standard errors . . . . .	34
2.4	Weighted estimation . . . . .	35
2.5	Other features of method-gf0 evaluators . . . . .	36
2.6	Limitations . . . . .	36
<b>3</b>	<b>Overview of ml</b>	<b>39</b>
3.1	The terminology of ml . . . . .	39
3.2	Equations in ml . . . . .	40
3.3	Likelihood-evaluator methods . . . . .	48
3.4	Tools for the ml programmer . . . . .	51
3.5	Common ml options . . . . .	51
3.5.1	Subsamples . . . . .	51
3.5.2	Weights . . . . .	52
3.5.3	OPG estimates of variance . . . . .	53
3.5.4	Robust estimates of variance . . . . .	54
3.5.5	Survey data . . . . .	56
3.5.6	Constraints . . . . .	57
3.5.7	Choosing among the optimization algorithms . . . . .	57
3.6	Maximizing your own likelihood functions . . . . .	61
<b>4</b>	<b>Method lf</b>	<b>63</b>
4.1	The linear-form restrictions . . . . .	64
4.2	Examples . . . . .	65
4.2.1	The probit model . . . . .	65
4.2.2	Normal linear regression . . . . .	66
4.2.3	The Weibull model . . . . .	69
4.3	The importance of generating temporary variables as doubles . . . . .	71
4.4	Problems you can safely ignore . . . . .	73

<i>Contents</i>	vii
4.5 Nonlinear specifications . . . . .	74
4.6 The advantages of lf in terms of execution speed . . . . .	75
<b>5 Methods lf0, lf1, and lf2</b>	<b>77</b>
5.1 Comparing these methods . . . . .	77
5.2 Outline of evaluators of methods lf0, lf1, and lf2 . . . . .	78
5.2.1 The todo argument . . . . .	79
5.2.2 The b argument . . . . .	79
Using mlevel to obtain values from each equation . . . . .	80
5.2.3 The lnfj argument . . . . .	82
5.2.4 Arguments for scores . . . . .	83
5.2.5 The H argument . . . . .	84
Using mlmatsum to define H . . . . .	86
5.2.6 Aside: Stata's scalars . . . . .	87
5.3 Summary of methods lf0, lf1, and lf2 . . . . .	90
5.3.1 Method lf0 . . . . .	90
5.3.2 Method lf1 . . . . .	92
5.3.3 Method lf2 . . . . .	94
5.4 Examples . . . . .	96
5.4.1 The probit model . . . . .	96
5.4.2 Normal linear regression . . . . .	98
5.4.3 The Weibull model . . . . .	104
<b>6 Methods d0, d1, and d2</b>	<b>109</b>
6.1 Comparing these methods . . . . .	109
6.2 Outline of method d0, d1, and d2 evaluators . . . . .	110
6.2.1 The todo argument . . . . .	111
6.2.2 The b argument . . . . .	111
6.2.3 The lnf argument . . . . .	112
Using lnf to indicate that the likelihood cannot be calculated	113
Using msum to define lnf . . . . .	114

6.2.4	The $g$ argument . . . . .	116
	Using <code>mlvecsum</code> to define $g$ . . . . .	116
6.2.5	The $H$ argument . . . . .	118
6.3	Summary of methods <code>d0</code> , <code>d1</code> , and <code>d2</code> . . . . .	119
6.3.1	Method <code>d0</code> . . . . .	119
6.3.2	Method <code>d1</code> . . . . .	122
6.3.3	Method <code>d2</code> . . . . .	124
6.4	Panel-data likelihoods . . . . .	126
6.4.1	Calculating <code>lnf</code> . . . . .	128
6.4.2	Calculating $g$ . . . . .	132
6.4.3	Calculating $H$ . . . . .	136
	Using <code>mlmatbysum</code> to help define $H$ . . . . .	136
6.5	Other models that do not meet the linear-form restrictions . . . . .	144
<b>7</b>	<b>Debugging likelihood evaluators</b>	<b>151</b>
7.1	<code>ml check</code> . . . . .	151
7.2	Using the debug methods . . . . .	153
7.2.1	First derivatives . . . . .	155
7.2.2	Second derivatives . . . . .	165
7.3	<code>ml trace</code> . . . . .	168
<b>8</b>	<b>Setting initial values</b>	<b>171</b>
8.1	<code>ml search</code> . . . . .	172
8.2	<code>ml plot</code> . . . . .	175
8.3	<code>ml init</code> . . . . .	177
<b>9</b>	<b>Interactive maximization</b>	<b>181</b>
9.1	The iteration log . . . . .	181
9.2	Pressing the Break key . . . . .	182
9.3	Maximizing difficult likelihood functions . . . . .	184
<b>10</b>	<b>Final results</b>	<b>187</b>
10.1	Graphing convergence . . . . .	187
10.2	Redisplaying output . . . . .	188

<b>11</b>	<b>Mata-based likelihood evaluators</b>	<b>193</b>
11.1	Introductory examples . . . . .	193
11.1.1	The probit model . . . . .	193
11.1.2	The Weibull model . . . . .	196
11.2	Evaluator function prototypes . . . . .	198
	Method-lf evaluators . . . . .	199
	lf-family evaluators . . . . .	199
	d-family evaluators . . . . .	200
11.3	Utilities . . . . .	201
	Dependent variables . . . . .	202
	Obtaining model parameters . . . . .	202
	Summing individual or group-level log likelihoods . . . . .	203
	Calculating the gradient vector . . . . .	203
	Calculating the Hessian . . . . .	204
11.4	Random-effects linear regression . . . . .	205
11.4.1	Calculating lnf . . . . .	206
11.4.2	Calculating g . . . . .	207
11.4.3	Calculating H . . . . .	208
11.4.4	Results at last . . . . .	209
<b>12</b>	<b>Writing do-files to maximize likelihoods</b>	<b>213</b>
12.1	The structure of a do-file . . . . .	213
12.2	Putting the do-file into production . . . . .	214
<b>13</b>	<b>Writing ado-files to maximize likelihoods</b>	<b>217</b>
13.1	Writing estimation commands . . . . .	217
13.2	The standard estimation-command outline . . . . .	219
13.3	Outline for estimation commands using ml . . . . .	220
13.4	Using ml in noninteractive mode . . . . .	221
13.5	Advice . . . . .	222
13.5.1	Syntax . . . . .	223
13.5.2	Estimation subsample . . . . .	225

13.5.3	Parsing with help from mlopts . . . . .	229
13.5.4	Weights . . . . .	232
13.5.5	Constant-only model . . . . .	233
13.5.6	Initial values . . . . .	237
13.5.7	Saving results in e() . . . . .	240
13.5.8	Displaying ancillary parameters . . . . .	240
13.5.9	Exponentiated coefficients . . . . .	242
13.5.10	Offsetting linear equations . . . . .	244
13.5.11	Program properties . . . . .	246
<b>14</b>	<b>Writing ado-files for survey data analysis</b>	<b>249</b>
14.1	Program properties . . . . .	249
14.2	Writing your own predict command . . . . .	252
<b>15</b>	<b>Other examples</b>	<b>255</b>
15.1	The logit model . . . . .	255
15.2	The probit model . . . . .	257
15.3	Normal linear regression . . . . .	259
15.4	The Weibull model . . . . .	262
15.5	The Cox proportional hazards model . . . . .	265
15.6	The random-effects regression model . . . . .	268
15.7	The seemingly unrelated regression model . . . . .	271
<b>A</b>	<b>Syntax of ml</b>	<b>285</b>
<b>B</b>	<b>Likelihood-evaluator checklists</b>	<b>307</b>
B.1	Method lf . . . . .	307
B.2	Method d0 . . . . .	308
B.3	Method d1 . . . . .	309
B.4	Method d2 . . . . .	311
B.5	Method lf0 . . . . .	314
B.6	Method lf1 . . . . .	315
B.7	Method lf2 . . . . .	317



<b>C</b>	<b>Listing of estimation commands</b>	<b>321</b>
C.1	The logit model . . . . .	321
C.2	The probit model . . . . .	323
C.3	The normal model . . . . .	325
C.4	The Weibull model . . . . .	327
C.5	The Cox proportional hazards model . . . . .	330
C.6	The random-effects regression model . . . . .	332
C.7	The seemingly unrelated regression model . . . . .	335
	<b>References</b>	<b>343</b>
	<b>Author index</b>	<b>347</b>
	<b>Subject index</b>	<b>349</b>

*(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-1f` 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
<code>lf</code>	<code>linearform</code>
<code>lf0</code>	<code>linearform0</code>
<code>lf1</code>	<code>linearform1</code>
<code>lf1debug</code>	<code>linearform1debug</code>
<code>lf2</code>	<code>linearform2</code>
<code>lf2debug</code>	<code>linearform2debug</code>
<code>d0</code>	<code>derivative0</code>
<code>d1</code>	<code>derivative1</code>
<code>d1debug</code>	<code>derivative1debug</code>
<code>d2</code>	<code>derivative2</code>
<code>d2debug</code>	<code>derivative2debug</code>
<code>gf0</code>	<code>generalform0</code>

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  $\theta$ -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

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

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

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

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

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

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 ( $\beta_0$ ,  $\beta_1$ , and  $\beta_2$ ). 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  $\beta_1 \text{weight}_j + \beta_2 \text{price}_j + \beta_0$ . 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 `doubles`. The last two lines replace `lnfj` with the values of the log likelihood for `foreignj` 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
                                Prob > chi2    =          0.0009

Log likelihood = -18.006571
```

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`:

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

where  $\epsilon_j$  is an error term. If we assume that each  $\epsilon_j$  is independent and identically distributed as a normal random variable with mean zero and variance  $\sigma^2$ , 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  $j$ th observation assuming homoskedasticity (constant variance) is

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

There are four parameters in our model:  $\beta_1$ ,  $\beta_2$ ,  $\beta_3$ , and  $\sigma$ , 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,  $\beta_2$  is the second element and  $\sigma$  is the fourth element. Our likelihood-evaluator program is

```

----- begin mynormal1_gf0.ad0 -----
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.ad0 -----

```

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  $\sigma$ ? 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
Prob > chi2 = 0.0000
Log likelihood = -163.18765

```

	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	1.841585	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 `gfb0`, 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
                                     Prob > chi2    =          0.0000

Log pseudolikelihood = -163.18765

```

turn	Robust		z	P> z	[95% Conf. Interval]	
	Coef.	Std. Err.				
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 `awweights`, 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
                                Prob > chi2    =      0.0000

Log likelihood = -66.424675

```

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.