

ml — Maximum likelihood estimation

Description	Syntax	Options	Remarks and examples
Stored results	Methods and formulas	References	Also see

Description

`ml model` defines the current problem.

`ml clear` clears the current problem definition. This command is rarely used because when you type `ml model`, any previous problem is automatically cleared.

`ml query` displays a description of the current problem.

`ml check` verifies that the log-likelihood evaluator you have written works. We strongly recommend using this command.

`ml search` searches for (better) initial values. We recommend using this command.

`ml plot` provides a graphical way of searching for (better) initial values.

`ml init` provides a way to specify initial values.

`ml report` reports $\ln L$'s values, gradient, and Hessian at the initial values or current parameter estimates, b_0 .

`ml trace` traces the execution of the user-defined log-likelihood evaluation program.

`ml count` counts the number of times the user-defined log-likelihood evaluation program is called; this command is seldom used. `ml count clear` clears the counter. `ml count on` turns on the counter. `ml count` without arguments reports the current values of the counter. `ml count off` stops counting calls.

`ml maximize` maximizes the likelihood function and reports results. Once `ml maximize` has successfully completed, the previously mentioned `ml` commands may no longer be used unless `noclear` is specified. `ml graph` and `ml display` may be used whether or not `noclear` is specified.

`ml graph` graphs the log-likelihood values against the iteration number.

`ml display` redisplay results.

`ml footnote` displays a warning message when the model did not converge within the specified number of iterations.

`ml score` creates new variables containing the equation-level scores. The variables generated by `ml score` are equivalent to those generated by specifying the `score()` option of `ml maximize` (and `ml model ... , ... maximize`).

programe is the name of a Stata program you write to evaluate the log-likelihood function.

funcname() is the name of a Mata function you write to evaluate the log-likelihood function.

In this documentation, *programe* and *funcname()* are referred to as the user-written evaluator, the likelihood evaluator, or sometimes simply as the evaluator. The program you write is written in the style required by the method you choose. The methods are `lf`, `d0`, `d1`, `d2`, `lf0`, `lf1`, `lf2`, and `gf0`. Thus, if you choose to use method `lf`, your program is called a method-`lf` evaluator.

Method-`lf` evaluators are required to evaluate the observation-by-observation log likelihood $\ln \ell_j$, $j = 1, \dots, N$.

Method-d0 evaluators are required to evaluate the overall log likelihood $\ln L$. Method-d1 evaluators are required to evaluate the overall log likelihood and its gradient vector $\mathbf{g} = \partial \ln L / \partial \mathbf{b}$. Method-d2 evaluators are required to evaluate the overall log likelihood, its gradient, and its Hessian matrix $H = \partial^2 \ln L / \partial \mathbf{b} \partial \mathbf{b}'$.

Method-lf0 evaluators are required to evaluate the observation-by-observation log likelihood $\ln \ell_j$, $j = 1, \dots, N$. Method-lf1 evaluators are required to evaluate the observation-by-observation log likelihood and its equation-level scores $g_{ji} = \partial \ln \ell / \partial \mathbf{x}_{ji} \mathbf{b}_i$. Method-lf2 evaluators are required to evaluate the observation-by-observation log likelihood, its equation-level scores, and its Hessian matrix $H = \partial^2 \ln \ell / \partial \mathbf{b} \partial \mathbf{b}'$.

Method-gf0 evaluators are required to evaluate the summable pieces of the log likelihood $\ln \ell_k$, $k = 1, \dots, K$.

`ml eval` is a subroutine used by evaluators of methods d0, d1, d2, lf0, lf1, lf2, and gf0 to evaluate the coefficient vector, \mathbf{b} , that they are passed.

`ml sum` is a subroutine used by evaluators of methods d0, d1, and d2 to define the value, $\ln L$, that is to be returned.

`ml vecsum` is a subroutine used by evaluators of methods d1 and d2 to define the gradient vector, \mathbf{g} , that is to be returned. It is suitable for use only when the likelihood function meets the linear-form restrictions.

`ml matsum` is a subroutine used by evaluators of methods d2 and lf2 to define the Hessian matrix, \mathbf{H} , that is to be returned. It is suitable for use only when the likelihood function meets the linear-form restrictions.

`ml matbysum` is a subroutine used by evaluator of method d2 to help define the Hessian matrix, \mathbf{H} , that is to be returned. It is suitable for use when the likelihood function contains terms made up of grouped sums, such as in panel-data models. For such models, use `ml matsum` to compute the observation-level outer products and `ml matbysum` to compute the group-level outer products. `ml matbysum` requires that the data be sorted by the variable identified in the `by()` option.

Syntax

ml model in interactive mode

```
ml model    method progname eq [eq ...] [if] [in] [weight]
              [, model_options svy diparm_options]
```

```
ml model    method funcname() eq [eq ...] [if] [in] [weight]
              [, model_options svy diparm_options]
```

ml model in noninteractive mode

```
ml model    method progname eq [eq ...] [if] [in] [weight], maximize
              [model_options svy diparm_options noninteractive_options]
```

```
ml model    method funcname() eq [eq ...] [if] [in] [weight], maximize
              [model_options svy diparm_options noninteractive_options]
```

Noninteractive mode is invoked by specifying the `maximize` option. Use `maximize` when `ml` will be used as a subroutine of another ado-file or program and you want to carry forth the problem, from definition to posting of results, in one command.

```

ml clear

ml query

ml check

ml search    [ [ / ] eqname [ : ] #lb #ub ] [ ... ] [ , search_options ]

ml plot      [ eqname : ] name [ # [ # [ # ] ] ] [ , saving(filename [ , replace ] ) ]

ml init      { [ eqname : ] name = # | / eqname = # } [ ... ]

ml init      # [ # ... ] , copy

ml init      matname [ , copy skip ]

ml report

ml trace     { on | off }

ml count     [ clear | on | off ]

ml maximize  [ , ml_maximize_options display_options eform_option ]

ml graph     [ # ] [ , saving(filename [ , replace ] ) ]

ml display   [ , display_options eform_option ]

ml footnote

ml score newvar [ if ] [ in ] [ , equation(eqname) missing ]

ml score newvarlist [ if ] [ in ] [ , missing ]

ml score [ type ] stub* [ if ] [ in ] [ , missing ]

```

where *method* is one of

lf	d0	lf0	gf0
	d1	lf1	
	d1debug	lf1debug	
	d2	lf2	
	d2debug	lf2debug	

or *method* can be specified using one of the longer, more descriptive names

<i>method</i>	Longer name
lf	linearform
d0	derivative0
d1	derivative1
d1debug	derivative1debug
d2	derivative2
d2debug	derivative2debug
lf0	linearform0
lf1	linearform1
lf1debug	linearform1debug
lf2	linearform2
lf2debug	linearform2debug
gf0	generalform0

eq is the equation to be estimated, enclosed in parentheses, and optionally with a name to be given to the equation, preceded by a colon,

([*eqname*:] [*varlist*_{*y*} =] [*varlist*_{*x*}] [, *eq_options*])

or *eq* is the name of a parameter, such as sigma, with a slash in front

/eqname which is equivalent to (*eqname*: , *freeparm*)

and *diparm_options* is one or more *diparm*(*diparm_args*) options where *diparm_args* is either `__sep__` or anything accepted by the “undocumented” `_diparm` command; see help `_diparm`.

eq_options

Description

noconstant

do not include an intercept in the equation

offset(*varname*_{*o*})

include *varname*_{*o*} in model with coefficient constrained to 1

exposure(*varname*_{*e*})

include ln(*varname*_{*e*}) in model with coefficient constrained to 1

freeparm

eqname is a free parameter

<i>model_options</i>	Description
<code>group(varname)</code>	use <i>varname</i> to identify groups
<code>vce(vctype)</code>	<i>vcetype</i> may be <code>robust</code> , <code>cluster clustvar</code> , <code>oim</code> , or <code>opg</code>
<code>constraints(numlist)</code>	constraints by number to be applied
<code>constraints(matname)</code>	matrix that contains the constraints to be applied
<code>nocnsnotes</code>	do not display notes when constraints are dropped
<code>title(string)</code>	place a title on the estimation output
<code>nopreserve</code>	do not preserve the estimation subsample in memory
<code>collinear</code>	keep collinear variables within equations
<code>missing</code>	keep observations containing variables with missing values
<code>lf0(#_k #_{ll})</code>	number of parameters and log-likelihood value of the constant-only model
<code>continue</code>	specifies that a model has been fit and sets the initial values \mathbf{b}_0 for the model to be fit based on those results
<code>waldtest(#)</code>	perform a Wald test; see <i>Options for use with ml model in interactive or noninteractive mode</i> below
<code>obs(#)</code>	number of observations
<code>crittype(string)</code>	describe the criterion optimized by ml
<code>subpop(varname)</code>	compute estimates for the single subpopulation
<code>nosvyadjust</code>	carry out Wald test as $W/k \sim F(k, d)$
<code>technique(nr)</code>	Stata's modified Newton–Raphson (NR) algorithm
<code>technique(bhhh)</code>	Berndt–Hall–Hall–Hausman (BHHH) algorithm
<code>technique(dfp)</code>	Davidon–Fletcher–Powell (DFP) algorithm
<code>technique(bfgs)</code>	Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm

<i>noninteractive_options</i>	Description
<code>init(ml_init_args)</code>	set the initial values \mathbf{b}_0
<code>search(on)</code>	equivalent to <code>ml search, repeat(0)</code> ; the default
<code>search(norescale)</code>	equivalent to <code>ml search, repeat(0) norescale</code>
<code>search(quietly)</code>	same as <code>search(on)</code> , except that output is suppressed
<code>search(off)</code>	prevents calling <code>ml search</code>
<code>repeat(#)</code>	<code>ml search</code> 's <code>repeat()</code> option; see below
<code>bounds(ml_search_bounds)</code>	specify bounds for <code>ml search</code>
<code>nowarning</code>	suppress “convergence not achieved” message of <code>iterate(0)</code>
<code>novce</code>	substitute the zero matrix for the variance matrix
<code>negh</code>	indicates that the evaluator returns the negative Hessian matrix
<code>score(newvars)</code>	new variables containing the contribution to the score
<i>maximize_options</i>	control the maximization process; seldom used

<i>search_options</i>	Description
<code>repeat(#)</code>	number of random attempts to find better initial-value vector; default is <code>repeat(10)</code> in interactive mode and <code>repeat(0)</code> in noninteractive mode
<code>restart</code>	use random actions to find starting values; not recommended
<code>nore scale</code>	do not rescale to improve parameter vector; not recommended
<i>maximize_options</i>	control the maximization process; seldom used
<i>ml_maximize_options</i>	Description
<code>nowarning</code>	suppress “convergence not achieved” message of <code>iterate(0)</code>
<code>novce</code>	substitute the zero matrix for the variance matrix
<code>negh</code>	indicates that the evaluator returns the negative Hessian matrix
<code>score(newvars stub*)</code>	new variables containing the contribution to the score
<code>nooutput</code>	suppress display of final results
<code>noclear</code>	do not clear ml problem definition after model has converged
<i>maximize_options</i>	control the maximization process; seldom used
<i>display_options</i>	Description
<code>noheader</code>	suppress header display above the coefficient table
<code>nofootnote</code>	suppress footnote display below the coefficient table
<code>level(#)</code>	set confidence level; default is <code>level(95)</code>
<code>first</code>	display coefficient table reporting results for first equation only
<code>neq(#)</code>	display coefficient table reporting first # equations
<code>showeqns</code>	display equation names in the coefficient table
<code>plus</code>	display coefficient table ending in dashes–plus–sign–dashes
<code>nocnsreport</code>	suppress constraints display above the coefficient table
<code>noomitted</code>	suppress display of omitted variables
<code>vsquish</code>	suppress blank space separating factor-variable terms or time-series–operated variables from other variables
<code>noemptycells</code>	suppress empty cells for interactions of factor variables
<code>baselevels</code>	report base levels of factor variables and interactions
<code>allbaselevels</code>	display all base levels of factor variables and interactions
<code>cformat(%fmt)</code>	format the coefficients, standard errors, and confidence limits in the coefficient table
<code>pformat(%fmt)</code>	format the <i>p</i> -values in the coefficient table
<code>sformat(%fmt)</code>	format the test statistics in the coefficient table
<code>no1stretch</code>	do not automatically widen the coefficient table to accommodate longer variable names
<code>coeflegend</code>	display legend instead of statistics

<i>eform_option</i>	Description
<u>e</u> form(<i>string</i>)	display exponentiated coefficients; column title is “ <i>string</i> ”
<u>e</u> form	display exponentiated coefficients; column title is “exp(b)”
hr	report hazard ratios
shr	report subhazard ratios
<u>i</u> rr	report incidence-rate ratios
or	report odds ratios
<u>r</u> rr	report relative-risk ratios

`fweights`, `awweights`, `iweights`, and `pweights` are allowed; see [U] 11.1.6 **weight**. With all but method `lf`, you must write your likelihood-evaluation program carefully if `pweights` are to be specified, and `pweights` may not be specified with method `d0`, `d1`, `d1debug`, `d2`, or `d2debug`. See Gould, Pitblado, and Poi (2010, chap. 6) for details.

See [U] 20 **Estimation and postestimation commands** for more capabilities of estimation commands.

To redisplay results, type `ml display`.

Syntax of subroutines for use by evaluator programs

```

mleval      newvar = vecname [ , eq(#) ]
mleval      scalarname = vecname , scalar [ eq(#) ]
mlsum       scalarnamelnf = exp [ if ] [ , noweight ]
mlvecsum    scalarnamelnf rowvecname = exp [ if ] [ , eq(#) ]
mlmatsum    scalarnamelnf matrixname = exp [ if ] [ , eq(#[,#]) ]
mlmatbysum  scalarnamelnf matrixname varnamea varnameb [ varnamec ] [ if ] ,
            by(varname) [ eq(#[,#]) ]

```

Syntax of user-written evaluator

Summary of notation

The log-likelihood function is $\ln L(\theta_{1j}, \theta_{2j}, \dots, \theta_{Ej})$, where $\theta_{ij} = \mathbf{x}_{ij} \mathbf{b}_i$, $j = 1, \dots, N$ indexes observations, and $i = 1, \dots, E$ indexes the linear equations defined by `ml model1`. If the likelihood satisfies the linear-form restrictions, it can be decomposed as $\ln L = \sum_{j=1}^N \ln \ell(\theta_{1j}, \theta_{2j}, \dots, \theta_{Ej})$.

Method-`lf` evaluators

```

program progname
  version 15.1
  args lnfj theta1 theta2 ...
  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  quietly generate double 'tmp1' = ...
  ...
  quietly replace 'lnfj' = ...
end

```

where

'lnfj' variable to be filled in with observation-by-observation values of $\ln \ell_j$
 'theta1' variable containing evaluation of first equation $\theta_{1j} = \mathbf{x}_{1j} \mathbf{b}_1$
 'theta2' variable containing evaluation of second equation $\theta_{2j} = \mathbf{x}_{2j} \mathbf{b}_2$
 ...

Method-d0 evaluators

```

program programe
  version 15.1
  args todo b lnf
  tempvar theta1 theta2 ...
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2) // if there is a  $\theta_2$ 
  ...
  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  generate double 'tmp1' = ...
  ...
  mlsum 'lnf' = ...
end

```

where

'todo' always contains 0 (may be ignored)
 'b' full parameter row vector $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_E)$
 'lnf' scalar to be filled in with overall $\ln L$

Method-d1 evaluators

```

program programe
  version 15.1
  args todo b lnf g
  tempvar theta1 theta2 ...
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2) // if there is a  $\theta_2$ 
  ...
  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  generate double 'tmp1' = ...
  ...
  mlsum 'lnf' = ...
  if ('todo'==0 | 'lnf'>=.) exit
  tempname d1 d2 ...
  mlvecsum 'lnf' 'd1' = formula for  $\partial \ln \ell_j / \partial \theta_{1j}$ , eq(1)
  mlvecsum 'lnf' 'd2' = formula for  $\partial \ln \ell_j / \partial \theta_{2j}$ , eq(2)
  ...
  matrix 'g' = ('d1', 'd2', ...)
end

```

where

'todo' contains 0 or 1
 0 \Rightarrow 'lnf' to be filled in;
 1 \Rightarrow 'lnf' and 'g' to be filled in
 'b' full parameter row vector $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_E)$
 'lnf' scalar to be filled in with overall $\ln L$
 'g' row vector to be filled in with overall $\mathbf{g} = \partial \ln L / \partial \mathbf{b}$

Method-d2 evaluators

```

program prognam
  version 15.1
  args todo b lnf g H
  tempvar theta1 theta2 ...
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2) // if there is a  $\theta_2$ 
  ...
  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  generate double 'tmp1' = ...
  ...
  mlsum 'lnf' = ...
  if ('todo'==0 | 'lnf'>=.) exit
  tempname d1 d2 ...
  mlvecsum 'lnf' 'd1' = formula for  $\partial \ln \ell_j / \partial \theta_{1j}$ , eq(1)
  mlvecsum 'lnf' 'd2' = formula for  $\partial \ln \ell_j / \partial \theta_{2j}$ , eq(2)
  ...
  matrix 'g' = ('d1', 'd2', ... )
  if ('todo'==1 | 'lnf'>=.) exit
  tempname d11 d12 d22 ...
  mlmatsum 'lnf' 'd11' = formula for  $\partial^2 \ln \ell_j / \partial \theta_{1j}^2$ , eq(1)
  mlmatsum 'lnf' 'd12' = formula for  $\partial^2 \ln \ell_j / \partial \theta_{1j} \partial \theta_{2j}$ , eq(1,2)
  mlmatsum 'lnf' 'd22' = formula for  $\partial^2 \ln \ell_j / \partial \theta_{2j}^2$ , eq(2)
  ...
  matrix 'H' = ('d11', 'd12', ... \ 'd12', 'd22', ... )
end

```

where

' <i>todo</i> '	contains 0, 1, or 2 0 \Rightarrow ' <i>lnf</i> ' to be filled in; 1 \Rightarrow ' <i>lnf</i> ' and ' <i>g</i> ' to be filled in; 2 \Rightarrow ' <i>lnf</i> ', ' <i>g</i> ', and ' <i>H</i> ' to be filled in
' <i>b</i> '	full parameter row vector $\mathbf{b}=(\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_E)$
' <i>lnf</i> '	scalar to be filled in with overall $\ln L$
' <i>g</i> '	row vector to be filled in with overall $\mathbf{g}=\partial \ln L / \partial \mathbf{b}$
' <i>H</i> '	matrix to be filled in with overall Hessian $\mathbf{H}=\partial^2 \ln L / \partial \mathbf{b} \partial \mathbf{b}'$

Method-lf0 evaluators

```

program prognam
  version 15.1
  args todo b lnfj
  tempvar theta1 theta2 ...
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2) // if there is a  $\theta_2$ 
  ...
  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  generate double 'tmp1' = ...
  ...
  quietly replace 'lnfj' = ...
end

```

where

' <i>todo</i> '	always contains 0 (may be ignored)
' <i>b</i> '	full parameter row vector $\mathbf{b}=(\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_E)$
' <i>lnfj</i> '	variable to be filled in with observation-by-observation values of $\ln \ell_j$

Method-lf1 evaluators

```

program programe
  version 15.1
  args todo b lnfj g1 g2 ...
  tempvar theta1 theta2 ...
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2) // if there is a  $\theta_2$ 
  ...
  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  generate double 'tmp1' = ...
  ...
  quietly replace 'lnfj' = ...
  if ('todo'==0) exit
  quietly replace 'g1' = formula for  $\partial \ln \ell_j / \partial \theta_{1j}$ 
  quietly replace 'g2' = formula for  $\partial \ln \ell_j / \partial \theta_{2j}$ 
  ...
end

where
'todo'      contains 0 or 1
              0  $\Rightarrow$  'lnfj' to be filled in;
              1  $\Rightarrow$  'lnfj', 'g1', 'g2', ..., to be filled in
'b'         full parameter row vector  $\mathbf{b}=(b_1, b_2, \dots, b_E)$ 
'lnfj'      variable to be filled in with observation-by-observation values of  $\ln \ell_j$ 
'g1'       variable to be filled in with  $\partial \ln \ell_j / \partial \theta_{1j}$ 
'g2'       variable to be filled in with  $\partial \ln \ell_j / \partial \theta_{2j}$ 
...

```

Method-lf2 evaluators

```

program programe
  version 15.1
  args todo b lnfj g1 g2 ... H
  tempvar theta1 theta2 ...
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2) // if there is a  $\theta_2$ 
  ...
  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  generate double 'tmp1' = ...
  ...
  quietly replace 'lnfj' = ...
  if ('todo'==0) exit
  quietly replace 'g1' = formula for  $\partial \ln \ell_j / \partial \theta_{1j}$ 
  quietly replace 'g2' = formula for  $\partial \ln \ell_j / \partial \theta_{2j}$ 
  ...
  if ('todo'==1) exit
  tempname d11 d12 d22 lnf ...
  mlmatsum 'lnf' 'd11' = formula for  $\partial^2 \ln \ell_j / \partial \theta_{1j}^2$ , eq(1)
  mlmatsum 'lnf' 'd12' = formula for  $\partial^2 \ln \ell_j / \partial \theta_{1j} \partial \theta_{2j}$ , eq(1,2)
  mlmatsum 'lnf' 'd22' = formula for  $\partial^2 \ln \ell_j / \partial \theta_{2j}^2$ , eq(2)
  ...
  matrix 'H' = ('d11', 'd12', ... \ 'd12', 'd22', ... )
end

```

where

```

'todo'      contains 0 or 1
            0⇒ 'lnfj' to be filled in;
            1⇒ 'lnfj', 'g1', 'g2', ..., to be filled in
            2⇒ 'lnfj', 'g1', 'g2', ..., and 'H' to be filled in
'b'         full parameter row vector  $\mathbf{b}=(b_1, b_2, \dots, b_E)$ 
'lnfj'      scalar to be filled in with observation-by-observation  $\ln L$ 
'g1'       variable to be filled in with  $\partial \ln \ell_j / \partial \theta_{1j}$ 
'g2'       variable to be filled in with  $\partial \ln \ell_j / \partial \theta_{2j}$ 
...
'H'        matrix to be filled in with overall Hessian  $\mathbf{H}=\partial^2 \ln L / \partial \mathbf{b} \partial \mathbf{b}'$ 

```

Method-gf0 evaluators

```

program programe
  version 15.1
  args todo b lnfj

  tempvar theta1 theta2 ...
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2) // if there is a  $\theta_2$ 
  ...

  // if you need to create any intermediate results:
  tempvar tmp1 tmp2 ...
  generate double 'tmp1' = ...
  ...

  quietly replace 'lnfj' = ...

end

where
'todo'      always contains 0 (may be ignored)
'b'         full parameter row vector  $\mathbf{b}=(b_1, b_2, \dots, b_E)$ 
'lnfj'      variable to be filled in with the values of the log-likelihood  $\ln \ell_j$ 

```

Global macros for use by all evaluators

```

$ML_y1      name of first dependent variable
$ML_y2      name of second dependent variable, if any
...
$ML_samp    variable containing 1 if observation to be used; 0 otherwise
$ML_w       variable containing weight associated with observation or 1 if no weights specified

```

Method-lf evaluators can ignore `$ML_samp`, but restricting calculations to the `$ML_samp==1` subsample will speed execution. Method-lf evaluators must ignore `$ML_w`; application of weights is handled by the method itself.

Methods d0, d1, d2, lf0, lf1, lf2, and gf0 can ignore `$ML_samp` as long as `ml model's nopreserve` option is not specified. These methods will run more quickly if `nopreserve` is specified. These evaluators can ignore `$ML_w` only if they use `mlsum`, `mlvecsum`, `mlmatsum`, and `mlmatbysum` to produce all final results.

Options

Options are presented under the following headings:

- Options for use with ml model in interactive or noninteractive mode*
- Options for use with ml model in noninteractive mode*
- Options for use when specifying equations*
- Options for use with ml search*
- Option for use with ml plot*
- Options for use with ml init*
- Options for use with ml maximize*
- Option for use with ml graph*
- Options for use with ml display*
- Options for use with mleval*
- Option for use with mlsum*
- Option for use with mlvecsum*
- Option for use with mlmatsum*
- Options for use with mlmatbysum*
- Options for use with ml score*

Options for use with ml model in interactive or noninteractive mode

`group(varname)` specifies the numeric variable that identifies groups. This option is typically used to identify panels for panel-data models.

`vce(vcetype)` specifies the type of standard error reported, which includes types that are robust to some kinds of misspecification (`robust`), that allow for intragroup correlation (`cluster clustvar`), and that are derived from asymptotic theory (`oim`, `opg`); see [R] [vce_option](#).

`vce(robust)`, `vce(cluster clustvar)`, `pweight`, and `svy` will work with evaluators of methods `lf`, `lf0`, `lf1`, `lf2`, and `gf0`; all you need to do is specify them.

These options will not work with evaluators of methods `d0`, `d1`, or `d2`, and specifying these options will produce an error message.

`constraints(numlist | matname)` specifies the linear constraints to be applied during estimation. `constraints(numlist)` specifies the constraints by number. Constraints are defined by using the `constraint` command; see [R] [constraint](#). `constraint(matname)` specifies a matrix that contains the constraints.

`nocnsnotes` prevents notes from being displayed when constraints are dropped. A constraint will be dropped if it is inconsistent, contradicts other constraints, or causes some other error when the constraint matrix is being built. Constraints are checked in the order in which they are specified.

`title(string)` specifies the title for the estimation output when results are complete.

`nopreserve` specifies that `ml` need not ensure that only the estimation subsample is in memory when the user-written likelihood evaluator is called. `nopreserve` is irrelevant when you use method `lf`.

For the other methods, if `nopreserve` is not specified, `ml` saves the data in a file (preserves the original dataset) and drops the irrelevant observations before calling the user-written evaluator. This way, even if the evaluator does not restrict its attentions to the `$ML_samp==1` subsample, results will still be correct. Later, `ml` automatically restores the original dataset.

`ml` need not go through these machinations for method `lf` because the user-written evaluator calculates observation-by-observation values, and `ml` itself sums the components.

`ml` goes through these machinations if and only if the estimation sample is a subsample of the data in memory. If the estimation sample includes every observation in memory, `ml` does not preserve the original dataset. Thus programmers must not alter the original dataset unless they `preserve` the data themselves.

We recommend that interactive users of `ml` not specify `nopreserve`; the speed gain is not worth the possibility of getting incorrect results.

We recommend that programmers specify `nopreserve`, but only after verifying that their evaluator really does restrict its attentions solely to the `$ML_samp==1` subsample.

`collinear` specifies that `ml` not remove the collinear variables within equations. There is no reason to leave collinear variables in place, but this option is of interest to programmers who, in their code, have already removed collinear variables and do not want `ml` to waste computer time checking again.

`missing` specifies that observations containing variables with missing values not be eliminated from the estimation sample. There are two reasons you might want to specify `missing`:

Programmers may wish to specify `missing` because, in other parts of their code, they have already eliminated observations with missing values and do not want `ml` to waste computer time looking again.

You may wish to specify `missing` if your model explicitly deals with missing values. Stata's `heckman` command is a good example of this. In such cases, there will be observations where missing values are allowed and other observations where they are not—where their presence should cause the observation to be eliminated. If you specify `missing`, it is your responsibility to specify an `if exp` that eliminates the irrelevant observations.

`lfo(#k #ll)` is typically used by programmers. It specifies the number of parameters and log-likelihood value of the constant-only model so that `ml` can report a likelihood-ratio test rather than a Wald test. These values may have been analytically determined, or they may have been determined by a previous fitting of the constant-only model on the estimation sample.

Also see the `continue` option directly below.

If you specify `lfo()`, it must be safe for you to specify the `missing` option, too, else how did you calculate the log likelihood for the constant-only model on the same sample? You must have identified the estimation sample, and done so correctly, so there is no reason for `ml` to waste time rechecking your results. All of which is to say, do not specify `lfo()` unless you are certain your code identifies the estimation sample correctly.

`lfo()`, even if specified, is ignored if `vce(robust)`, `vce(cluster clustvar)`, `pweight`, or `svy` is specified because, in that case, a likelihood-ratio test would be inappropriate.

`continue` is typically specified by programmers and does two things:

First, it specifies that a model has just been fit by either `ml` or some other estimation command, such as `logit`, and that the likelihood value stored in `e(l1)` and the number of parameters stored in `e(b)` as of that instant are the relevant values of the constant-only model. The current value of the log likelihood is used to present a likelihood-ratio test unless `vce(robust)`, `vce(cluster clustvar)`, `pweight`, `svy`, or `constraints()` is specified. A likelihood-ratio test is inappropriate when `vce(robust)`, `vce(cluster clustvar)`, `pweight`, or `svy` is specified. We suggest using `lrtest` when `constraints()` is specified; see [R] [lrtest](#).

Second, `continue` sets the initial values, b_0 , for the model about to be fit according to the `e(b)` currently stored.

The comments made about specifying `missing` with `lfo()` apply equally well here.

`waldtest(#)` is typically specified by programmers. By default, `ml` presents a Wald test, but that is overridden if the `lfo()` or `continue` option is specified. A Wald test is performed if `vce(robust)`, `vce(cluster clustvar)`, or `pweight` is specified.

`waldtest(0)` prevents even the Wald test from being reported.

`waldtest(-1)` is the default. It specifies that a Wald test be performed by constraining all coefficients except the intercept to 0 in the first equation. Remaining equations are to be unconstrained. A Wald test is performed if neither `lfo()` nor `continue` was specified, and a Wald test is forced if `vce(robust)`, `vce(cluster clustvar)`, or `pweight` was specified.

`waldtest(k)` for $k \leq -1$ specifies that a Wald test be performed by constraining all coefficients except intercepts to 0 in the first $|k|$ equations; remaining equations are to be unconstrained. A Wald test is performed if neither `lfo()` nor `continue` was specified, and a Wald test is forced if `vce(robust)`, `vce(cluster clustvar)`, or `pweight` was specified.

`waldtest(k)` for $k \geq 1$ works like the options above, except that it forces a Wald test to be reported even if the information to perform the likelihood-ratio test is available and even if none of `vce(robust)`, `vce(cluster clustvar)`, or `pweight` was specified. `waldtest(k)`, $k \geq 1$, may not be specified with `lfo()`.

`obs(#)` is used mostly by programmers. It specifies that the number of observations reported and ultimately stored in `e(N)` be `#`. Ordinarily, `ml` works that out for itself. Programmers may want to specify this option when, for the likelihood evaluator to work for N observations, they first had to modify the dataset so that it contained a different number of observations.

`crittype(string)` is used mostly by programmers. It allows programmers to supply a string (up to 32 characters long) that describes the criterion that is being optimized by `ml`. The default is "log likelihood" for nonrobust and "log pseudolikelihood" for robust estimation.

`svy` indicates that `ml` is to pick up the `svy` settings set by `svyset` and use the robust variance estimator. This option requires the data to be `svyset`; see [SVY] [svyset](#). `svy` may not be specified with `vce()` or `weights`.

`subpop(varname)` specifies that estimates be computed for the single subpopulation defined by the observations for which `varname` \neq 0. Typically, `varname = 1` defines the subpopulation, and `varname = 0` indicates observations not belonging to the subpopulation. For observations whose subpopulation status is uncertain, `varname` should be set to missing (`.'`). This option requires the `svy` option.

`nosvyadjust` specifies that the model Wald test be carried out as $W/k \sim F(k, d)$, where W is the Wald test statistic, k is the number of terms in the model excluding the constant term, d is the total number of sampled PSUs minus the total number of strata, and $F(k, d)$ is an F distribution with k numerator degrees of freedom and d denominator degrees of freedom. By default, an adjusted Wald test is conducted: $(d - k + 1)W/(kd) \sim F(k, d - k + 1)$. See [Korn and Graubard \(1990\)](#) for a discussion of the Wald test and the adjustments thereof. This option requires the `svy` option.

`technique(algorithm_spec)` specifies how the likelihood function is to be maximized. The following algorithms are currently implemented in `ml`. For details, see [Gould, Pitblado, and Poi \(2010\)](#).

`technique(nr)` specifies Stata's modified Newton–Raphson (NR) algorithm.

`technique(bhhh)` specifies the Berndt–Hall–Hall–Hausman (BHHH) algorithm.

`technique(dfp)` specifies the Davidon–Fletcher–Powell (DFP) algorithm.

`technique(bfgs)` specifies the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm.

The default is `technique(nr)`.

You can switch between algorithms by specifying more than one in the `technique()` option. By default, `ml` will use an algorithm for five iterations before switching to the next algorithm. To specify a different number of iterations, include the number after the technique in the option. For example, `technique(bhhh 10 nr 1000)` requests that `ml` perform 10 iterations using the BHHH algorithm, followed by 1,000 iterations using the NR algorithm, and then switch back to BHHH for

10 iterations, and so on. The process continues until convergence or until reaching the maximum number of iterations.

Options for use with ml model in noninteractive mode

The following extra options are for use with `ml model` in noninteractive mode. Noninteractive mode is for programmers who use `ml` as a subroutine and want to issue one command that will carry forth the estimation from start to finish.

`maximize` is required. It specifies noninteractive mode.

`init(ml_init_args)` sets the initial values, \mathbf{b}_0 . *ml_init_args* are whatever you would type after the `ml init` command.

`search(on | norescale | quietly | off)` specifies whether `ml search` is to be used to improve the initial values. `search(on)` is the default and is equivalent to separately running `ml search, repeat(0)`. `search(norescale)` is equivalent to separately running `ml search, repeat(0) norescale`. `search(quietly)` is equivalent to `search(on)`, except that it suppresses `ml search`'s output. `search(off)` prevents calling `ml search`.

`repeat(#)` is `ml search`'s `repeat()` option. `repeat(0)` is the default.

`bounds(ml_search_bounds)` specifies the search bounds. *ml_search_bounds* is specified as

`[eqn_name] lower_bound upper_bound ... [eqn_name] lower_bound upper_bound`

for instance, `bounds(100 100 lnsigma 0 10)`. The `ml model` command issues `ml search ml_search_bounds, repeat(#)`. Specifying search bounds is optional.

`nowarning`, `novce`, `negh`, and `score()` are `ml maximize`'s equivalent options.

maximize_options: `difficult`, `technique(algorithm_spec)`, `iterate(#)`, `[no]log`, `trace`, `gradient`, `showstep`, `hessian`, `showtolerance`, `tolerance(#)`, `ltolerance(#)`, `nrtolerance(#)`, `nonrtolerance`, and `from(init_specs)`; see [R] [maximize](#). These options are seldom used.

Options for use when specifying equations

`noconstant` specifies that the equation not include an intercept.

`offset(varnameo)` specifies that the equation be $\mathbf{x}\mathbf{b} + \text{varname}_o$ —that it include *varname_o* with coefficient constrained to be 1.

`exposure(varnamee)` is an alternative to `offset(varnameo)`; it specifies that the equation be $\mathbf{x}\mathbf{b} + \ln(\text{varname}_e)$. The equation is to include $\ln(\text{varname}_e)$ with coefficient constrained to be 1.

`freeparm` specifies that the associated *eqname* is a free parameter. The corresponding full column name on `e(b)` will be */eqname* instead of *eqname: _cons*. This option is not allowed with *varlist_x*.

Options for use with ml search

`repeat(#)` specifies the number of random attempts that are to be made to find a better initial-value vector. The default is `repeat(10)`.

`repeat(0)` specifies that no random attempts be made. More precisely, `repeat(0)` specifies that no random attempts be made if the first initial-value vector is a feasible starting point. If it is not, `ml search` will make random attempts, even if you specify `repeat(0)`, because it has no alternative. The `repeat()` option refers to the number of random attempts to be made to improve the initial values. When the initial starting value vector is not feasible, `ml search` will make up to 1,000 random attempts to find starting values. It stops when it finds one set of values that works and then moves into its `improve-initial-values` logic.

`repeat(k)`, $k > 0$, specifies the number of random attempts to be made to improve the initial values.

`restart` specifies that random actions be taken to obtain starting values and that the resulting starting values not be a deterministic function of the current values. Generally, you should not specify this option because, with `restart`, `ml search` intentionally does not produce as good a set of starting values as it could. `restart` is included for use by the optimizer when it gets into serious trouble. The random actions ensure that the optimizer and `ml search`, working together, do not cause an endless loop.

`restart` implies `nore scale`, which is why we recommend that you do not specify `restart`. In testing, sometimes `rescale` worked so well that, even after randomization, the rescaler would bring the starting values right back to where they had been the first time and thus defeat the intended randomization.

`nore scale` specifies that `ml search` not engage in its rescaling actions to improve the parameter vector. We do not recommend specifying this option because rescaling tends to work so well.

maximize_options: `[no]log` and `trace`; see [R] [maximize](#). These options are seldom used.

Option for use with ml plot

`saving(filename[, replace])` specifies that the graph be saved in `filename.gph`.
See [G-3] [saving_option](#).

Options for use with ml init

`copy` specifies that the list of numbers or the initialization vector be copied into the initial-value vector by position rather than by name.

`skip` specifies that any parameters found in the specified initialization vector that are not also found in the model be ignored. The default action is to issue an error message.

Options for use with ml maximize

`nowarning` is allowed only with `iterate(0)`. `nowarning` suppresses the “convergence not achieved” message. Programmers might specify `iterate(0) nowarning` when they have a vector `b` already containing the final estimates and want `ml` to calculate the variance matrix and postestimation results. Then specify `init(b) search(off) iterate(0) nowarning nolog`.

`novce` is allowed only with `iterate(0)`. `novce` substitutes the zero matrix for the variance matrix, which in effect posts estimation results as fixed constants.

`negh` indicates that the evaluator returns the negative Hessian matrix. By default, `ml` assumes `d2` and `lf2` evaluators return the Hessian matrix.

`score(newvars|stub*)` creates new variables containing the contributions to the score for each equation and ancillary parameter in the model; see [U] 20.23 **Obtaining scores**.

If `score(newvars)` is specified, the *newvars* must contain k new variables. For evaluators of methods `lf`, `lf0`, `lf1`, and `lf2`, k is the number of equations. For evaluators of method `gf0`, k is the number of parameters. If `score(stub*)` is specified, variables named *stub1*, *stub2*, ..., *stubk* are created.

For evaluators of methods `lf`, `lf0`, `lf1`, and `lf2`, the first variable contains $\partial \ln \ell_j / \partial (\mathbf{x}_{1j} \mathbf{b}_1)$, the second variable contains $\partial \ln \ell_j / \partial (\mathbf{x}_{2j} \mathbf{b}_2)$, and so on.

For evaluators of method `gf0`, the first variable contains $\partial \ln \ell_j / \partial \mathbf{b}_1$, the second variable contains $\partial \ln \ell_j / \partial \mathbf{b}_2$, and so on.

`nooutput` suppresses display of results. This option is different from prefixing `ml maximize` with `quietly` in that the iteration log is still displayed (assuming that `nolog` is not specified).

`noclear` specifies that the `ml` problem definition not be cleared after the model has converged. Perhaps you are having convergence problems and intend to run the model to convergence. If so, use `ml search` to see if those values can be improved, and then restart the estimation.

maximize_options: `difficult`, `iterate(#)`, `[no]log`, `trace`, `gradient`, `showstep`, `hessian`, `showtolerance`, `tolerance(#)`, `ltolerance(#)`, `nrtolerance(#)`, `nonrtolerance`; see [R] **maximize**. These options are seldom used.

display_options; see *Options for use with ml display* below.

eform_option; see *Options for use with ml display* below.

Option for use with ml graph

`saving(filename[, replace])` specifies that the graph be saved in *filename.gph*. See [G-3] *saving_option*.

Options for use with ml display

`noheader` suppresses the header display above the coefficient table that displays the final log-likelihood value, the number of observations, and the model significance test.

`nofootnote` suppresses the footnote display below the coefficient table, which displays a warning if the model fit did not converge within the specified number of iterations. Use `ml footnote` to display the warning if 1) you add to the coefficient table using the `plus` option or 2) you have your own footnotes and want the warning to be last.

`level(#)` is the standard confidence-level option. It specifies the confidence level, as a percentage, for confidence intervals of the coefficients. The default is `level(95)` or as set by `set level`; see [U] 20.8 **Specifying the width of confidence intervals**.

`first` displays a coefficient table reporting results for the first equation only, and the report makes it appear that the first equation is the only equation. This option is used by programmers who estimate ancillary parameters in the second and subsequent equations and who wish to report the values of such parameters themselves.

`neq(#)` is an alternative to `first`. `neq(#)` displays a coefficient table reporting results for the first $\#$ equations. This option is used by programmers who estimate ancillary parameters in the $\# + 1$ and subsequent equations and who wish to report the values of such parameters themselves.

`showeqns` is a seldom-used option that displays the equation names in the coefficient table. `ml display` uses the numbers stored in `e(k_eq)` and `e(k_aux)` to determine how to display the coefficient table. `e(k_eq)` identifies the number of equations, and `e(k_aux)` identifies how many of these are for ancillary parameters. The `first` option is implied when `showeqns` is not specified and all but the first equation are for ancillary parameters.

`plus` displays the coefficient table, but rather than ending the table in a line of dashes, ends it in dashes–plus–sign–dashes. This is so that programmers can write additional display code to add more results to the table and make it appear as if the combined result is one table. Programmers typically specify `plus` with the `first` or `neq()` options. This option implies `nofootnote`.

`nocnsreport` suppresses the display of constraints above the coefficient table. This option is ignored if constraints were not used to fit the model.

`noomitted` specifies that variables that were omitted because of collinearity not be displayed. The default is to include in the table any variables omitted because of collinearity and to label them as “(omitted)”.

`vsquish` specifies that the blank space separating factor-variable terms or time-series–operated variables from other variables in the model be suppressed.

`noemptycells` specifies that empty cells for interactions of factor variables not be displayed. The default is to include in the table interaction cells that do not occur in the estimation sample and to label them as “(empty)”.

`baselevels` and `allbaselevels` control whether the base levels of factor variables and interactions are displayed. The default is to exclude from the table all base categories.

`baselevels` specifies that base levels be reported for factor variables and for interactions whose bases cannot be inferred from their component factor variables.

`allbaselevels` specifies that all base levels of factor variables and interactions be reported.

`cformat(%fmt)` specifies how to format coefficients, standard errors, and confidence limits in the coefficient table.

`pformat(%fmt)` specifies how to format p -values in the coefficient table.

`sformat(%fmt)` specifies how to format test statistics in the coefficient table.

`no!stretch` specifies that the width of the coefficient table not be automatically widened to accommodate longer variable names. The default, `!stretch`, is to automatically widen the coefficient table up to the width of the Results window. To change the default, use `set !stretch off`. `no!stretch` is not shown in the dialog box.

`coeflegend` specifies that the legend of the coefficients and how to specify them in an expression be displayed rather than displaying the statistics for the coefficients.

eform_option: `eform(string)`, `eform, hr, shr, irr, or,` and `rrr` display the coefficient table in exponentiated form: for each coefficient, $\exp(b)$ rather than b is displayed, and standard errors and confidence intervals are transformed. *string* is the table header that will be displayed above the transformed coefficients and must be 11 characters or shorter in length—for example, `eform("Odds ratio")`. The options `eform, hr, shr, irr, or,` and `rrr` provide a default *string* equivalent to “`exp(b)`”, “Haz. Ratio”, “SHR”, “IRR”, “Odds Ratio”, and “RRR”, respectively. These options may not be combined.

`ml display` looks at `e(k_eform)` to determine how many equations are affected by an *eform_option*; by default, only the first equation is affected. Type `ereturn list, all` to view `e(k_eform)`; see [P] [ereturn](#).

Options for use with mlevel

`eq(#)` specifies the equation number, i , for which $\theta_{ij} = \mathbf{x}_{ij}\mathbf{b}_i$ is to be evaluated. `eq(1)` is assumed if `eq()` is not specified.

`scalar` asserts that the i th equation is known to evaluate to a constant, meaning that the equation was specified as `()`, `(name:)`, or `/name` on the `ml model` statement. If you specify this option, the new variable created is created as a scalar. If the i th equation does not evaluate to a scalar, an error message is issued.

Option for use with mlsum

`noweight` specifies that weights (`$ML_w`) be ignored when summing the likelihood function.

Option for use with mlvecsum

`eq(#)` specifies the equation for which a gradient vector $\partial \ln L / \partial \mathbf{b}_i$ is to be constructed. The default is `eq(1)`.

Option for use with mlmatsum

`eq(#[,#])` specifies the equations for which the Hessian matrix is to be constructed. The default is `eq(1)`, which is the same as `eq(1,1)`, which means $\partial^2 \ln L / \partial \mathbf{b}_1 \partial \mathbf{b}'_1$. Specifying `eq(i,j)` results in $\partial^2 \ln L / \partial \mathbf{b}_i \partial \mathbf{b}'_j$.

Options for use with mlmatbysum

`by(varname)` is required and specifies the group variable.

`eq(#[,#])` specifies the equations for which the Hessian matrix is to be constructed. The default is `eq(1)`, which is the same as `eq(1,1)`, which means $\partial^2 \ln L / \partial \mathbf{b}_1 \partial \mathbf{b}'_1$. Specifying `eq(i,j)` results in $\partial^2 \ln L / \partial \mathbf{b}_i \partial \mathbf{b}'_j$.

Options for use with ml score

`equation(eqname)` identifies from which equation the observation scores are to come. This option may be used only when generating one variable.

`missing` specifies that observations containing variables with missing values not be eliminated from the estimation sample.

Remarks and examples

[stata.com](http://www.stata.com)

For a thorough discussion of `ml`, see the fourth edition of *Maximum Likelihood Estimation with Stata* (Gould, Pitblado, and Poi 2010). The book provides a tutorial introduction to `ml`, notes on advanced programming issues, and a discourse on maximum likelihood estimation from both theoretical and practical standpoints. See [Survey options and ml](#) at the end of *Remarks and examples* for examples of the new `svy` options. For more information about survey estimation, see [\[SVY\] survey](#), [\[SVY\] svy estimation](#), and [\[SVY\] variance estimation](#).

ml requires that you write a program that evaluates the log-likelihood function and, possibly, its first and second derivatives. The style of the program you write depends upon the method you choose. Methods lf, lf0, d0, and gf0 require that your program evaluate the log likelihood only. Methods d1 and lf1 require that your program evaluate the log likelihood and its first derivatives. Methods d2 and lf2 requires that your program evaluate the log likelihood and its first and second derivatives. Methods lf, lf0, d0, and gf0 differ from each other in that, with methods lf and lf0, your program is required to produce observation-by-observation log-likelihood values $\ln \ell_j$ and it is assumed that $\ln L = \sum_j \ln \ell_j$; with method d0, your program is required to produce only the overall value $\ln L$; and with method gf0, your program is required to produce the summable pieces of the log likelihood, such as those in panel-data models.

Once you have written the program—called an evaluator—you define a model to be fit using ml model and obtain estimates using ml maximize. You might type

```
. ml model ...
. ml maximize
```

but we recommend that you type

```
. ml model ...
. ml check
. ml search
. ml maximize
```

ml check verifies your evaluator has no obvious errors, and ml search finds better initial values.

You fill in the ml model statement with 1) the method you are using, 2) the name of your program, and 3) the “equations”. You write your evaluator in terms of $\theta_1, \theta_2, \dots$, each of which has a linear equation associated with it. That linear equation might be as simple as $\theta_i = b_0$, it might be $\theta_i = b_1\text{mpg} + b_2\text{weight} + b_3$, or it might omit the intercept b_3 . The equations are specified in parentheses on the ml model line.

Suppose that you are using method lf and the name of your evaluator program is myprog. The statement

```
. ml model lf myprog (mpg weight)
```

would specify one equation with $\theta_i = b_1\text{mpg} + b_2\text{weight} + b_3$. If you wanted to omit b_3 , you would type

```
. ml model lf myprog (mpg weight, nocons)
```

and if all you wanted was $\theta_i = b_0$, you would type

```
. ml model lf myprog ()
```

With multiple equations, you list the equations one after the other; so, if you typed

```
. ml model lf myprog (mpg weight) ()
```

you would be specifying $\theta_1 = b_1\text{mpg} + b_2\text{weight} + b_3$ and $\theta_2 = b_4$. You would write your likelihood in terms of θ_1 and θ_2 . If the model was linear regression, θ_1 might be the **xb** part and θ_2 the variance of the residuals.

When you specify the equations, you also specify any dependent variables. If you typed

```
. ml model lf myprog (price = mpg weight) ()
```

price would be the one and only dependent variable, and that would be passed to your program in `$ML_y1`. If your model had two dependent variables, you could type

```
. ml model lf myprog (price displ = mpg weight) ()
```

Then `$ML_y1` would be `price` and `$ML_y2` would be `displ`. You can specify however many dependent variables are necessary and specify them on any equation. It does not matter on which equation you specify them; the first one specified is placed in `$ML_y1`, the second in `$ML_y2`, and so on.

► Example 1: Method lf

Using method `lf`, we want to produce observation-by-observation values of the log likelihood. The probit log-likelihood function is

$$\ln \ell_j = \begin{cases} \ln \Phi(\theta_{1j}) & \text{if } y_j = 1 \\ \ln \Phi(-\theta_{1j}) & \text{if } y_j = 0 \end{cases}$$

$$\theta_{1j} = \mathbf{x}_j \mathbf{b}_1$$

The following is the method-`lf` evaluator for this likelihood function:

```
program myprobit
    version 15.1
    args lnf theta1
    quietly replace `lnf' = ln(normal(`theta1')) if $ML_y1==1
    quietly replace `lnf' = ln(normal(-`theta1')) if $ML_y1==0
end
```

If we wanted to fit a model of `foreign` on `mpg` and `weight`, we would type the following commands. The `'foreign ='` part specifies that y is `foreign`. The `'mpg weight'` part specifies that $\theta_{1j} = b_1 \text{mpg}_j + b_2 \text{weight}_j + b_3$.

```
. use http://www.stata-press.com/data/r15/auto
(1978 Automobile Data)

. ml model lf myprobit (foreign = mpg weight)
. 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 = -27.904125
Iteration 2:  log likelihood = -26.858643
Iteration 3:  log likelihood = -26.844199
Iteration 4:  log likelihood = -26.844189
Iteration 5:  log likelihood = -26.844189

Log likelihood = -26.844189
```

	Number of obs	=	74
	Wald chi2(2)	=	20.75
	Prob > chi2	=	0.0000

foreign	Coef.	Std. Err.	z	P> z	[95% Conf. Interval]	
mpg	-.1039503	.0515689	-2.02	0.044	-.2050235	-.0028772
weight	-.0023355	.0005661	-4.13	0.000	-.003445	-.0012261
_cons	8.275464	2.554142	3.24	0.001	3.269438	13.28149

► Example 2: Method lf for two-equation, two-dependent-variable model

A two-equation, two-dependent-variable model is a little different. Rather than receiving one θ , our program will receive two. Rather than there being one dependent variable in `$ML_y1`, there will be dependent variables in `$ML_y1` and `$ML_y2`. For instance, the Weibull regression log-likelihood function is

$$\begin{aligned}\ln \ell_j &= -(t_j e^{-\theta_{1j}})^{\exp(\theta_{2j})} + d_j \{ \theta_{2j} - \theta_{1j} + (e^{\theta_{2j}} - 1)(\ln t_j - \theta_{1j}) \} \\ \theta_{1j} &= \mathbf{x}_j \mathbf{b}_1 \\ \theta_{2j} &= s\end{aligned}$$

where t_j is the time of failure or censoring and $d_j = 1$ if failure and 0 if censored. We can make the log likelihood a little easier to program by introducing some extra variables:

$$\begin{aligned}p_j &= \exp(\theta_{2j}) \\ M_j &= \{t_j \exp(-\theta_{1j})\}^{p_j} \\ R_j &= \ln t_j - \theta_{1j} \\ \ln \ell_j &= -M_j + d_j \{ \theta_{2j} - \theta_{1j} + (p_j - 1)R_j \}\end{aligned}$$

The method-lf evaluator for this is

```
program myweib
  version 15.1
  args lnf theta1 theta2
  tempvar p M R
  quietly generate double 'p' = exp('theta2')
  quietly generate double 'M' = ($ML_y1*exp(-'theta1'))^'p'
  quietly generate double 'R' = ln($ML_y1)-'theta1'
  quietly replace 'lnf' = -'M' + $ML_y2*('theta2'-'theta1' + ('p'-1)*'R')
end
```

We can fit a model by typing

```
. ml model lf myweib (studytime died = i.drug age) ()
. ml maximize
```

Note that we specified `()` for the second equation. The second equation corresponds to the Weibull shape parameter s , and the linear combination we want for s contains just an intercept. Alternatively, we could type

```
. ml model lf myweib (studytime died = i.drug age) /s
```

Typing `/s` means the same thing as typing `(s:, freeparm)`. The `s`, either after a slash or in parentheses before a colon, labels the equation. The leading slash, `/`, is a shortcut for specifying that “`s`” is a free parameter. Free parameters are labeled using this shortcut notation, `/s`, in the column names of `e(b)` and in the estimation results. If we instead specified `(s:)`, then `s` would be the equation label for a constant linear equation and would be labeled as `s:_cons` in the column names of `e(b)` and in the estimation results.

```

. use http://www.stata-press.com/data/r15/cancer, clear
(Patient Survival in Drug Trial)
. ml model lf myweib (studytime died = i.drug age) /s
. ml maximize
initial:      log likelihood =      -744
alternative:  log likelihood = -356.14276
rescale:     log likelihood = -200.80201
rescale eq:  log likelihood = -136.69232
Iteration 0:  log likelihood = -136.69232 (not concave)
Iteration 1:  log likelihood = -124.11726
Iteration 2:  log likelihood = -113.9591
Iteration 3:  log likelihood = -110.30683
Iteration 4:  log likelihood = -110.26748
Iteration 5:  log likelihood = -110.26736
Iteration 6:  log likelihood = -110.26736

```

```

Log likelihood = -110.26736
Number of obs   =          48
Wald chi2(3)    =          35.25
Prob > chi2     =          0.0000

```

	Coef.	Std. Err.	z	P> z	[95% Conf. Interval]	
drug						
2	1.012966	.2903917	3.49	0.000	.4438086	1.582123
3	1.45917	.2821195	5.17	0.000	.9062261	2.012114
age						
_cons	-.0671728	.0205688	-3.27	0.001	-.1074868	-.0268587
/s	.5573333	.1402154	3.97	0.000	.2825162	.8321504

◀

► Example 3: Method d0

Method-d0 evaluators receive $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_E)$, the coefficient vector, rather than the already evaluated $\theta_1, \theta_2, \dots, \theta_E$, and they are required to evaluate the overall log-likelihood $\ln L$ rather than $\ln \ell_j$, $j = 1, \dots, N$.

Use `mleval` to produce the thetas from the coefficient vector.

Use `mlsum` to sum the components that enter into $\ln L$.

In the case of Weibull, $\ln L = \sum \ln \ell_j$, and our method-d0 evaluator is

```

program weib0
  version 15.1
  args todo b lnf
  tempvar theta1 theta2
  mlevel 'theta1' = 'b', eq(1)
  mlevel 'theta2' = 'b', eq(2)
  local t "$ML_y1"          // this is just for readability
  local d "$ML_y2"
  tempvar p M R
  quietly generate double 'p' = exp('theta2')
  quietly generate double 'M' = ('t'*exp(-'theta1'))^'p'
  quietly generate double 'R' = ln('t')-'theta1'
  mlsum 'lnf' = -'M' + 'd'*('theta2'-'theta1' + ('p'-1)*'R')
end

```

To fit our model using this evaluator, we would type

```

. ml model d0 weib0 (studytime died = i.drug age) /s
. ml maximize

```

◀

□ Technical note

Method d0 does not require $\ln L = \sum_j \ln \ell_j$, $j = 1, \dots, N$, as method lf does. Your likelihood function might have independent components only for groups of observations. Panel-data estimators have a log-likelihood value $\ln L = \sum_i \ln L_i$, where i indexes the panels, each of which contains multiple observations. Conditional logistic regression has $\ln L = \sum_k \ln L_k$, where k indexes the risk pools. Cox regression has $\ln L = \sum_{(t)} \ln L_{(t)}$, where (t) denotes the ordered failure times.

To evaluate such likelihood functions, first calculate the within-group log-likelihood contributions. This usually involves `generate` and `replace` statements prefixed with `by`, as in

```

tempvar sumd
by group: generate double 'sumd' = sum($ML_y1)

```

Structure your code so that the log-likelihood contributions are recorded in the last observation of each group. Say that a variable is named `'cont'`. To sum the contributions, code

```

tempvar last
quietly by group: generate byte 'last' = (_n==_N)
mlsum 'lnf' = 'cont' if 'last'

```

You must inform `mlsum` which observations contain log-likelihood values to be summed. First, you do not want to include intermediate results in the sum. Second, `mlsum` does not skip missing values. Rather, if `mlsum` sees a missing value among the contributions, it sets the overall result, `'lnf'`, to missing. That is how `ml maximize` is informed that the likelihood function could not be evaluated at the particular value of `b`. `ml maximize` will then take action to escape from what it thinks is an infeasible area of the likelihood function.

When the likelihood function violates the linear-form restriction $\ln L = \sum_j \ln \ell_j$, $j = 1, \dots, N$, with $\ln \ell_j$ being a function solely of values within the j th observation, use method d0. In the following examples, we will demonstrate methods d1 and d2 with likelihood functions that meet this linear-form restriction. The d1 and d2 methods themselves do not require the linear-form restriction, but the utility routines `mlvecsum` and `mlmatsum` do. Using method d1 or d2 when the restriction is violated is difficult; however, `mlmatbysum` may be of some help for method-d2 evaluators. □

► Example 4: Method d1

Method-d1 evaluators are required to produce the gradient vector $\mathbf{g} = \partial \ln L / \partial \mathbf{b}$, as well as the overall log-likelihood value. Using `mlvecsum`, we can obtain $\partial \ln L / \partial \mathbf{b}$ from $\partial \ln L / \partial \theta_i$, $i = 1, \dots, E$. The derivatives of the Weibull log-likelihood function are

$$\frac{\partial \ln \ell_j}{\partial \theta_{1j}} = p_j(M_j - d_j)$$

$$\frac{\partial \ln \ell_j}{\partial \theta_{2j}} = d_j - R_j p_j(M_j - d_j)$$

The method-d1 evaluator for this is

```

program weib1
  version 15.1
  args todo b lnf g                                // g is new
  tempvar t1 t2
  mlevel 't1' = 'b', eq(1)
  mlevel 't2' = 'b', eq(2)

  local t "$ML_y1"
  local d "$ML_y2"

  tempvar p M R
  quietly generate double 'p' = exp('t2')
  quietly generate double 'M' = ('t'*exp(-'t1'))^'p'
  quietly generate double 'R' = ln('t')-'t1'

  mlsum 'lnf' = -'M' + 'd'*('t2'-'t1' + ('p'-1)*'R')
  if ('todo'==0 | 'lnf'>=.) exit                    /* <-- new */

  tempname d1 d2                                    /* <-- new */
  mlvecsum 'lnf' 'd1' = 'p'*('M'-'d'), eq(1)        /* <-- new */
  mlvecsum 'lnf' 'd2' = 'd' - 'R'*'p'*('M'-'d'), eq(2) /* <-- new */
  matrix 'g' = ('d1','d2')                          /* <-- new */
end

```

We obtained this code by starting with our method-d0 evaluator and then adding the extra lines that method d1 requires. To fit our model using this evaluator, we could type

```

. ml model d1 weib1 (studytime died = drug2 drug3 age) /s
. ml maximize

```

but we recommend substituting `method d1debug` for `method d1` and typing

```

. ml model d1debug weib1 (studytime died = drug2 drug3 age) /s
. ml maximize

```

Method `d1debug` will compare the derivatives we calculate with numerical derivatives and thus verify that our program is correct. Once we are certain the program is correct, then we would switch from method `d1debug` to method `d1`. ◀

► Example 5: Method d2

Method-d2 evaluators are required to produce $\mathbf{H} = \partial^2 \ln L / \partial \mathbf{b} \partial \mathbf{b}'$, the Hessian matrix, as well as the gradient and log-likelihood value. `mlmatsum` will help calculate $\partial^2 \ln L / \partial \mathbf{b} \partial \mathbf{b}'$ from the second derivatives with respect to θ . For the Weibull model, these second derivatives are

$$\frac{\partial^2 \ln \ell_j}{\partial \theta_{1j}^2} = -p_j^2 M_j$$

$$\frac{\partial^2 \ln \ell_j}{\partial \theta_{1j} \partial \theta_{2j}} = p_j (M_j - d_j + R_j p_j M_j)$$

$$\frac{\partial^2 \ln \ell_j}{\partial \theta_{2j}^2} = -p_j R_j (R_j p_j M_j + M_j - d_j)$$

The method-d2 evaluator is

```

program weib2
  version 15.1
  args todo b lnf g H // H added
  tempvar t1 t2
  mllevel 't1' = 'b', eq(1)
  mllevel 't2' = 'b', eq(2)
  local t "$ML_y1"
  local d "$ML_y2"
  tempvar p M R
  quietly generate double 'p' = exp('t2')
  quietly generate double 'M' = ('t'*exp(-'t1'))^'p'
  quietly generate double 'R' = ln('t')-'t1'
  mlsum 'lnf' = -'M' + 'd'*('t2'-'t1' + ('p'-1)*'R')
  if ('todo'==0 | 'lnf'>=.) exit
  tempname d1 d2
  mlvecsum 'lnf' 'd1' = 'p'*('M'-'d'), eq(1)
  mlvecsum 'lnf' 'd2' = 'd' - 'R'*'p'*('M'-'d'), eq(2)
  matrix 'g' = ('d1','d2')
  if ('todo'==1 | 'lnf'>=.) exit // new from here down
  tempname d11 d12 d22
  mlmatsum 'lnf' 'd11' = -'p'^2 * 'M', eq(1)
  mlmatsum 'lnf' 'd12' = 'p'*('M'-'d' + 'R'*'p'*'M'), eq(1,2)
  mlmatsum 'lnf' 'd22' = -'p'*'R'*('R'*'p'*'M' + 'M' - 'd'), eq(2)
  matrix 'H' = ('d11','d12' \ 'd12','d22')
end

```

We started with our previous method-d1 evaluator and added the lines that method d2 requires. We could now fit a model by typing

```

. ml model d2 weib2 (studytime died = drug2 drug3 age) /s
. ml maximize

```

but we would recommend substituting method d2debug for method d2 and typing

```

. ml model d2debug weib2 (studytime died = drug2 drug3 age) /s
. ml maximize

```

Method d2debug will compare the first and second derivatives we calculate with numerical derivatives and thus verify that our program is correct. Once we are certain the program is correct, then we would switch from method d2debug to method d2.

◀

As we stated earlier, to produce the robust variance estimator with method lf, there is nothing to do except specify `vce(robust)`, `vce(cluster clustvar)`, or `pweight`. For methods d0, d1, and d2, these options do not work. If your likelihood function meets the linear-form restrictions, you can use methods lf0, lf1, and lf2, then these options will work. The equation scores are defined as

$$\frac{\partial \ln \ell_j}{\partial \theta_{1j}}, \frac{\partial \ln \ell_j}{\partial \theta_{2j}}, \dots$$

Your evaluator will be passed variables, one for each equation, which you fill in with the equation scores. For *both* method lf1 and lf2, these variables are passed in the fourth and subsequent positions of the argument list. That is, you must process the arguments as

```
args todo b lnf g1 g2 ... H
```

Note that for method lf1, the ‘H’ argument is not used and can be ignored.

▷ Example 6: Robust variance estimates

If you have used `mlvecsum` in your evaluator of method d1 or d2, it is easy to turn it into an evaluator of method lf1 or lf2 that allows the computation of the robust variance estimator. The expression that you specified on the right-hand side of `mlvecsum` is the equation score.

Here we turn the program that we gave earlier in the method-d1 example into a method-lf1 evaluator that allows `vce(robust)`, `vce(cluster clustvar)`, or `pweight`.

```
program weib1
  version 15.1
  args todo b lnfj g1 g2          // g1 and g2 are new
  tempvar t1 t2
  mlevel 't1' = 'b', eq(1)
  mlevel 't2' = 'b', eq(2)
  local t "$ML_y1"
  local d "$ML_y2"
  tempvar p M R
  quietly generate double 'p' = exp('t2')
  quietly generate double 'M' = ('t'*exp(-'t1'))^'p'
  quietly generate double 'R' = ln('t')-'t1'
  quietly replace 'lnfj' = -'M' + 'd'*('t2'-'t1' + ('p'-1)*'R')
  if ('todo'==0) exit
  quietly replace 'g1' = 'p'*('M'-'d')          /* <-- new */
  quietly replace 'g2' = 'd' - 'R'*'p'*('M'-'d') /* <-- new */
end
```

To fit our model and get the robust variance estimates, we type

```
. ml model lf1 weib1 (studytime died = drug2 drug3 age) /s, vce(robust)
. ml maximize
```

◀

Survey options and ml

`ml` can handle stratification, poststratification, multiple stages of clustering, and finite population corrections. Specifying the `svy` option implies that the data come from a survey design and also implies that the survey linearized variance estimator is to be used; see [\[SVY\] variance estimation](#).

▷ Example 7

Suppose that we are interested in a probit analysis of data from a survey in which `q1` is the answer to a yes/no question and `x1`, `x2`, `x3` are demographic responses. The following is a lf2 evaluator for the probit model that meets the requirements for `vce(robust)` (linear form and computes the scores).

```

program mylf2probit
  version 15.1
  args todo b lnfj g1 H
  tempvar z Fz lnf
  mlevel 'z' = 'b'
  quietly generate double 'Fz' = normal('z') if $ML_y1 == 1
  quietly replace 'Fz' = normal(-'z') if $ML_y1 == 0
  quietly replace 'lnfj' = log('Fz')
  if ('todo'==0) exit
  quietly replace 'g1' = normalden('z')/'Fz' if $ML_y1 == 1
  quietly replace 'g1' = -normalden('z')/'Fz' if $ML_y1 == 0
  if ('todo'==1) exit
  mlmatsum 'lnf' 'H' = -'g1'*('g1'+ 'z'), eq(1,1)
end

```

To fit a model, we `svyset` the data, then use `svy` with `ml`.

```

. svyset psuid [pw=w], strata(strid)
. ml model lf2 mylf2probit (q1 = x1 x2 x3), svy
. ml maximize

```

We could also use the `subpop()` option to make inferences about the subpopulation identified by the variable `sub`:

```

. svyset psuid [pw=w], strata(strid)
. ml model lf2 mylf2probit (q1 = x1 x2 x3), svy subpop(sub)
. ml maximize

```

◀

Stored results

For results stored by `ml` without the `svy` option, see [\[R\] maximize](#).

For results stored by `ml` with the `svy` option, see [\[SVY\] svy](#).

Methods and formulas

`ml` is implemented using `moptimize()`; see [\[M-5\] moptimize\(\)](#).

References

- Gould, W. W., J. S. Pitblado, and B. P. Poi. 2010. *Maximum Likelihood Estimation with Stata*. 4th ed. College Station, TX: Stata Press.
- Korn, E. L., and B. I. Graubard. 1990. Simultaneous testing of regression coefficients with complex survey data: Use of Bonferroni *t* statistics. *American Statistician* 44: 270–276.
- Royston, P. 2007. Profile likelihood for estimation and confidence intervals. *Stata Journal* 7: 376–387.

Also see

- [\[R\] maximize](#) — Details of iterative maximization
- [\[R\] mlexp](#) — Maximum likelihood estimation of user-specified expressions
- [\[R\] nl](#) — Nonlinear least-squares estimation
- [\[M-5\] moptimize\(\)](#) — Model optimization
- [\[M-5\] optimize\(\)](#) — Function optimization