Title

stata.com

LinearProgram( ) — Linear programming

Description

The `LinearProgram()` class finds the parameter vector that minimizes or maximizes the linear objective function subject to some restrictions. The restrictions may be linear equality constraints, linear inequality constraints, lower bounds, or upper bounds.

Syntax

Syntax is presented under the following headings:

Step 1: Initialization

```
q = LinearProgram()
```

Step 2: Definition of linear programming problem

void	*q*.setCoefficients(*real rowvector coef*)
void	*q*.setMaxOrMin(*string scalar maxormin*)
void	*q*.setEquality(*real matrix ecmat*, *real colvector rhs*)
void	*q*.setInequality(*real matrix iemat*, *real colvector rhs*)
void	*q*.setBounds(*real rowvector lowerbd*, *real rowvector upperbd*)
void	*q*.setMaxiter(*real scalar maxiter*)
void	*q*.setTol(*real scalar tol*)
void	*q*.setTrace(*string scalar trace*)
real rowvector	*q*.getCoefficients()
string scalar	*q*.getMaxOrMin()
real matrix	*q*.getEquality()
real matrix	*q*.getInequality()
real matrix	*q*.getBounds()
real scalar	*q*.getMaxiter()
real scalar	*q*.getTol()
string scalar	*q*.getTrace()

Step 3: Perform optimization

real scalar	*q*.optimize()

Step 4: Display or obtain results

real rowvector	*q*.parameters()
real scalar	*q*.value()
real scalar	*q*.iterations()
real scalar	*q*.converged()
real scalar	*q*.errorcode()
string scalar	*q*.errortext()
real scalar	*q*.returncode()

Utility function for use in all steps

void *q*.query()

Definition of q

A variable of type LinearProgram is called an instance of the LinearProgram() class. *q* is an instance of LinearProgram(), a vector of instances, or a matrix of instances. If you are working interactively, you can create an instance of LinearProgram() by typing

```
q = LinearProgram()
```

For a row vector of *n* LinearProgram() instances, type

```
q = LinearProgram(n)
```

For an $m \times n$ matrix of LinearProgram() instances, type

```
q = LinearProgram(m, n)
```

In a function, you would declare one instance of the LinearProgram() class *q* as a scalar.

```
void myfunc()
{
    class LinearProgram scalar   q
    q = LinearProgram()
    ...
}
```

Within a function, you can declare *q* as a row vector of *n* instances by typing

```
void myfunc()
{
    class LinearProgram rowvector   q
    q = LinearProgram(n)
    ...
}
```

For an $m \times n$ matrix of instances, type

```
void myfunc()
{
    class LinearProgram matrix   q
    q = LinearProgram(m, n)
    ...
}
```

Functions defining the linear programming problem

At a minimum, you need to tell the LinearProgram() class about the coefficients of the linear objective function you wish to optimize. Optionally, you may specify whether to minimize or maximize the objective function, any equality constraints, any inequality constraints, any lower bounds, and any upper bounds. You may also specify the maximum number of iterations allowed, the convergence tolerance, and whether or not to print computational details.

Each pair of functions includes a *q*.set function that specifies a setting and a *q*.get function that returns the current setting.

q.setCoefficients( ) and q.getCoefficients( )

q.setCoefficients(*coef*) sets the linear objective function coefficients. The coefficients must be set before optimization.

q.getCoefficients() returns the linear objective function coefficients (or an empty vector if not specified).

q.setMaxOrMin( ) and q.getMaxOrMin( )

q.setMaxOrMin(*maxormin*) sets whether to perform maximization or minimization. *maxormin* may be "max" or "min". The default is maximization ("max").

q.getMaxOrMin() returns "max" or "min" according to the current setting.

q.setEquality( ) and q.getEquality( )

The equality constraints for a linear programming problem are in the form of linear system $\mathbf{A}_{\mathrm{EC}}\mathbf{x} = \mathbf{b}_{\mathrm{EC}}$, where $\mathbf{A}_{\mathrm{EC}}$ is the equality-constraints (EC) matrix and $\mathbf{b}_{\mathrm{EC}}$ is the right-hand-side (RHS) vector.

q.setEquality(*ecmat*, *rhs*) sets the EC matrix and the RHS vector.

q.getEquality() returns a matrix containing both the EC matrix and the RHS vector. The RHS vector is the last column of the returned matrix. (An empty matrix is returned if equality constraints were not specified.)

q.setInequality( ) and q.getInequality( )

The inequality constraints for a linear programming problem are in the form of linear system $\mathbf{A}_{\mathrm{IE}}\mathbf{x} \leq \mathbf{b}_{\mathrm{IE}}$, where $\mathbf{A}_{\mathrm{IE}}$ is the inequality-constraints (IE) matrix and $\mathbf{b}_{\mathrm{IE}}$ is the RHS vector.

q.setInequality(*iemat*, *rhs*) sets the IE matrix and the RHS vector.

q.getInequality() returns a matrix containing both the IE matrix and the RHS vector. The RHS vector is the last column of the returned matrix. (An empty matrix is returned if inequality constraints were not specified.)

q.setBounds( ) and q.getBounds( )

The parameters may have lower bounds or upper bounds. By default, the lower bound is $-\infty$ and the upper bound is ∞.

q.setBounds(*lowerbd*, *upperbd*) sets the lower and upper bounds. Using a missing value as the lower bound indicates $-\infty$, and using a missing value as the upper bound indicates ∞.

q.getBounds() returns a two-row matrix containing the lower and upper bounds.

q.setMaxiter( ) and q.getMaxiter( )

q.setMaxiter(*maxiter*) specifies the maximum number of iterations, which must be an integer greater than 0. The default value of *maxiter* is 16000.

q.getMaxiter() returns the current maximum number of iterations.

q.setTol( ) and q.getTol( )

q.setTol(*tol*) specifies the convergence-criterion tolerance, which must be greater than 0. The default value of *tol* is 1e-8.

q.getTol() returns the currently specified tolerance.

q.setTrace( ) and q.getTrace( )

q.setTrace(*trace*) sets whether or not to print out computation details. *trace* may be "on" or "off". The default value is "off".

q.getTrace() returns the current trace status.

Performing optimization

q.optimize( )

q.optimize() invokes the optimization process and returns the value of the objective function at the optimum.

Functions for obtaining results

After performing optimization, the functions below provide results including parameters, the value at the optimum, the number of iterations, whether convergence was achieved, error messages, and return codes.

q.parameters( )

q.parameters() returns the parameter vector that optimizes the objective function; it returns an empty vector prior to performing the optimization.

q.value( )

q.value() returns the value of the objective function at the optimum; it returns a missing value prior to performing the optimization.

q.iterations( )

q.iterations() returns the number of iterations.

q.converged( )

q.converged() returns 1 if the optimization converged and 0 if not.

q.errorcode( ), q.errortext( ), and q.returncode( )

q.errorcode() returns the error code generated during the computation; it returns 0 if no error is found.

q.errortext() returns an error message corresponding to the error code generated during the computation; it returns an empty string if no error is found.

q.returncode() returns the Stata return code corresponding to the error code generated during the computation.

The error codes and the corresponding Stata return codes are as follows:

Error code	Return code	Error text
1	430	problem is infeasible
2	430	problem is unbounded
3	430	maximum number of iterations has been reached
4	3499	dimensions of coefficients, constraints, and bounds do not conform
5	111	dimension of the parameters is 0

Utility function

You can obtain a report of all settings and results currently stored in a class LinearProgram() instance.

q.query( )

q.query() with no return value displays the information stored in the class.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

The LinearProgram() class is a Mata class for linear programming.

LinearProgram() uses Mehrotra's (1992) predictor-corrector primal-dual method to optimize the linear programming problems of the form

$$\begin{aligned} &\min_{\mathbf{x}} \text{ or } \max_{\mathbf{x}} \ \mathbf{c}\mathbf{x}' \\ &\text{such that } \mathbf{A}_{\mathrm{EC}}\mathbf{x}' = \mathbf{b}_{\mathrm{EC}} \\ &\mathbf{A}_{\mathrm{IE}}\mathbf{x}' \le \mathbf{b}_{\mathrm{IE}} \\ &\mathbf{lowerbd} \le \mathbf{x} \le \mathbf{upperbd} \end{aligned}$$

where $\mathbf{cx}'$ is the linear objective function, $\mathbf{A}_{\mathrm{EC}}\mathbf{x}' = \mathbf{b}_{\mathrm{EC}}$ specifies equality constraints, $\mathbf{A}_{\mathrm{IE}}\mathbf{x}' \leq \mathbf{b}_{\mathrm{IE}}$ specifies inequality constraints, **lowerbd** is the lower bound on **x**, and **upperbd** is the upper bound on **x**.

Mehrotra's (1992) predictor-corrector primal-dual method is much faster than the traditional simplex method for large problems. This method is a version of the interior-point method that is widely used today instead of the older simplex method that was widely used in the past. This speed comes at the cost of some accuracy, but the inaccuracy can be removed in practice by lowering the convergence tolerance. Lowering the convergence tolerance will produce answers that are practically the same as those produced by the simplex method.

Details about the interior-point method

The simplex method breaks down for large problems, because it repeatedly checks a list of candidate solutions. When the list gets too large, the solution time becomes infeasible.

Instead of repeatedly checking a list of candidate solutions, the interior-point method solves a series of approximations to the original problem. This approximation-based approach makes it feasible to solve large problems that are not feasibly solved by the simplex algorithm. The interior-point method is also much faster than the simplex method on the medium-sized and large-sized problems that are common in statistical applications of linear programming.

Because the simplex algorithm checks a list of candidate solutions, it finds the exact integer solution, when one exists. In contrast, interior-point algorithms find the exact integer solution plus or minus the small positive amount ϵ. Reducing the convergence tolerance reduces ϵ to a value that is practically 0. (Technically, it will be on the order of 10^{-16}.)

See example 1 for an illustration.

For an introduction to class programming in Mata, see [M-2] **class**.

Examples

To solve a linear programming problem, you first use `LinearProgram()` to get an instance of the class. At a minimum, you must also use `setCoefficients()` to specify the linear objective function coefficients. In the examples below, we demonstrate both basic and more advanced use of the `LinearProgram()` class.

▷ Example 1: A first example

Consider the following linear programming problem:

$$
\begin{aligned}
\max_{x_1,x_2}\ & 5x_1 + 3x_2 \\
\text{such that}\ & -x_1 + 11x_2 = 33 \\
& 0.5x_1 - x_2 \leq -3 \\
& 2x_1 + 14x_2 \leq 60 \\
& 2x_1 + x_2 \leq 14.5 \\
& x_1 - 0.4x_2 \leq 5 \\
& x_1 \geq 0 \\
& x_2 \geq 0
\end{aligned}
$$

This problem can be written in matrix form,

$$\begin{aligned} &\max_{\mathbf{x}} \ \mathbf{c}\mathbf{x}' \\ \text{such that } &\mathbf{A}_{\text{EC}}\mathbf{x}' = \mathbf{b}_{\text{EC}} \\ &\mathbf{A}_{\text{IE}}\mathbf{x}' \leq \mathbf{b}_{\text{IE}} \\ &\mathbf{x} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned}$$

where $\mathbf{c} = (5, 3)$, $\mathbf{A}_{\text{EC}} = (-1, 11)$, $\mathbf{b}_{\text{EC}} = 33$,

$$\mathbf{A}_{\text{IE}} = \begin{bmatrix} 0.5 & -1 \\ 2 & 14 \\ 2 & 1 \\ 1 & -0.4 \end{bmatrix} \quad \text{and} \quad \mathbf{b}_{\text{IE}} = \begin{bmatrix} -3 \\ 60 \\ 14.5 \\ 5 \end{bmatrix}$$

We first define all the coefficients and constraints in matrix form, respectively:

```
: c = (5, 3)
:
: Aec = (-1, 11)
: bec = 33
:
: Aie = (0.5, -1 \ 2, 14 \ 2, 1 \ 1, -0.4)
: bie = (-3 \ 60 \ 14.5 \ 5)
:
: lowerbd = (0, 0)
: upperbd = (., .)
```

Here we use missing in the upper bound to indicate an infinite upper bound. (A missing value used in the lower bound indicates a minus infinite lower bound.)

We generate `q` as an instance of the class:

```
: q = LinearProgram()
```

Then we initialize the coefficients and constraints:

```
: q.setCoefficients(c)
: q.setEquality(Aec, bec)
: q.setInequality(Aie, bie)
: q.setBounds(lowerbd, upperbd)
```

Now we can solve the problem:

```
: q.optimize()
  9.000000001
```

We can display the optimal parameters after the solution is found:

```
: q.parameters()
                 1             2
  1 | 9.89562e-11             3 |
```

As mentioned above, the exact solution for this problem is $(0, 3)$, and the simplex method would find it. Lowering the convergence tolerance produces a solution that is practically $(0, 3)$.

```
: q.setTol(1e-12)
: q.optimize()
  9
: q.parameters()
                  1              2

  1   4.94781e-15              3
```

◁

▷ Example 2: Display information about the linear programming problem

Each instance of the class contains a lot of information about the problem at hand. You can use the member function *q*`.query()` to display this information. Several other member functions display specific pieces of information. This example illustrates how to use these functions.

Consider the following linear programming problem:

$$\begin{aligned} \min_{x_1,x_2,x_3} \; & x_1 + x_2 \\ \text{such that } & x_1 + x_2 + x_3 = 5 \\ & x_1 - x_2 + 2x_3 = 8 \\ & x_1 \geq 1 \\ & 0 \leq x_2 \leq 2 \\ & x_3 \geq 0 \end{aligned}$$

We can express the problem in matrix form, as we did with the previous example:

```
: c = (1, 1, 0)
: Aec = (1, 1, 1 \ 1, -1, 2)
: bec = (5 \ 8)
: lowerbd = (1, 0, 0)
: upperbd = (., 2, .)
```

To calculate the problem, we define the class instance `q`:

```
: q = LinearProgram()
```

We can show all the default values by using `q.query()` before we perform any initialization or computation.

```
: q.query()
Settings for LinearProgram() ──────────────────────────────────────────────
Version:                                        1.00

Problem setup:
    Perform maximization of the following problem
    Objective function size:                          1 x       0
    Equality constraint matrix size:                  0 x       0
    Equality constraint right-hand-side size:         0 x       1
    Inequality constraint matrix size:                0 x       0
    Inequality constraint right-hand-side size:       0 x       1
    Lower bound size:                                 1 x       0
    Upper bound size:                                 1 x       0

Trace:                                          off

Convergence
    Maximum iterations:                         16000
    Tolerance:                                  1.0000e-08

Current status
    Objective function value:                   .
    Converged:                                  no

Note: The function setCoefficients() has not been called.
```

Now we initialize the problem with all the information required. First, we define the coefficients of the objective function:

```
: q.setCoefficients(c)
```

Because this is a minimization problem, we use `q.setMaxOrMin()` to change the setting from the default, `"max"`, to `"min"`.

```
: q.setMaxOrMin("min")
```

We then define the equality constraints and bounds, respectively.

```
: q.setEquality(Aec, bec)
: q.setBounds(lowerbd, upperbd)
```

Our default maximum number of iterations is 16,000; Stata will issue a warning message when the maximum number of iterations has been reached. For illustrative purposes, we set the maximum to 2 here and compute the approximation.

```
: q.setMaxiter(2)
: q.optimize()
Warning: Maximum number of iterations has been reached.
  .
```

The solution could not be found using only two iterations. Thus, We see a warning, and a missing value (`.`) is returned as the value of the objective function.

We switch back to a maximum of 16,000 iterations.

```
: q.setMaxiter(16000)
```

We can set the trace to "on" to see the computation details.

```
: q.setTrace("on")
: q.optimize()
Quadrature trace:
Iteration          Current function value   Current error estimate
    1               5.202919005              3.44331e+00
    2               .5872208056              6.08052e-01
    3               .3333542428              2.63268e-02
    4               .3333333396              1.29229e-06
    5               .3333333333              6.46181e-11
  1.333333333
```

We turn off the trace by typing

```
: q.setTrace("off")
```

Now we can list parameters at the minimum:

```
: q.parameters()
                    1             2             3
  1 |               1   .3333333333   3.666666667 |
```

We can also display the value of the objective function at the solution.

```
: q.value()
  1.333333333
```

And, we can display the number of iterations used to find the solution.

```
: q.iterations()
  5
```

No error was found, so the error code and error message are

```
: q.errorcode()
  0
: q.errortext()
```

We can show all the values by using `q.query()` after the computation:

```
: q.query()
Settings for LinearProgram() ──────────────────────────────────────────────

Version:                                        1.00

Problem setup:
    Perform minimization of the following problem
    Objective function size:                          1 x       3
    Equality constraint matrix size:                  2 x       3
    Equality constraint right-hand-side size:         2 x       1
    Inequality constraint matrix size:                0 x       0
    Inequality constraint right-hand-side size:       0 x       1
    Lower bound size:                                 1 x       3
    Upper bound size:                                 1 x       3

Trace:                                          off

Convergence
    Maximum iterations:                         16000
    Tolerance:                                  1.0000e-08

Current status
    Objective function value:                   1.33333333
    Converged:                                  yes
    Iterations:                                 5
```

◁

The following three examples use linear programming to solve statistical estimation problems. For some statistical estimation problems, there are many equivalent ways of specifying the corresponding linear programming problems. In our examples, we use a common way of specifying each problem.

▷ Example 3: Quantile regression

Linear programming can be used to fit quantile regression models.

We begin with a brief introduction to writing the quantile-regression (QR) estimation problem as a linear programming problem; see [R] **qreg**, Koenker and Hallock (2001), and Koenker and Bassett (1978) for more details.

Minimizing the sum of squared residuals produces an estimator of the coefficients in a mean regression model. Analogously, minimizing the sum of the check function of the residuals produces an estimator of the coefficients in a QR model. For the QR model, we estimate the coefficients of the τth conditional quantile function ($\boldsymbol{\beta}_\tau$) by solving

$$\min_{\boldsymbol{\beta}_\tau} \sum_{i=1}^{n} c_\tau(y_i - \mathbf{x}_i\boldsymbol{\beta}_\tau')$$

where y_i is the ith observation of the outcome $\mathbf{y}$, $\mathbf{x}_i$ is the ith observation of the vector of covariates $\mathbf{x}$, n is the number of observations, and $c_\tau(\cdot)$ is the check function. The check function $c_\tau(r_i)$ of the residual $r_i = y_i - \mathbf{x}_i\boldsymbol{\beta}_\tau'$ is given by

$$c_\tau(r_i) = \{\tau - \mathbb{I}(r_i < 0)\}r_i$$

where

$$\mathbb{I}(r_i < 0) = \begin{cases} 1 & \text{if } r_i < 0 \\ 0 & \text{otherwise} \end{cases}$$

This minimization problem for the estimator of the coefficients in the QR model can be written as the following linear programming problem:

$$\begin{aligned} \min_{\boldsymbol{\beta}_\tau, \mathbf{u}, \mathbf{v}} \ & \tau \mathbf{1}_n' \mathbf{u} + (1-\tau)\mathbf{1}_n' \mathbf{v} \\ \text{such that } & \mathbf{y} - \mathbf{X}\boldsymbol{\beta}_\tau = \mathbf{u} - \mathbf{v} \\ & \mathbf{u} \geq \mathbf{0}_n \\ & \mathbf{v} \geq \mathbf{0}_n \end{aligned}$$

where $\mathbf{1}_n$ is a vector of 1s, $\mathbf{0}_n$ is a vector of 0s, $\mathbf{X}$ is the matrix of observations of the covariates, $\mathbf{y}$ is the vector of observations of the outcome, and $\mathbf{u}$ and $\mathbf{v}$ are added to the inequality constraint to transform it into an equality (in other words, they are slack variables).

The above problem can be rewritten as

$$\begin{aligned} \min_{\boldsymbol{\beta}_\tau, \mathbf{u}, \mathbf{v}} \ & \mathbf{c} \begin{bmatrix} \boldsymbol{\beta}_\tau \\ \mathbf{u} \\ \mathbf{v} \end{bmatrix} \\ \text{such that } & \mathbf{A}_{\text{EC}} \begin{bmatrix} \boldsymbol{\beta}_\tau \\ \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \mathbf{y} \\ & \mathbf{u} \geq \mathbf{0}_n \\ & \mathbf{v} \geq \mathbf{0}_n \end{aligned}$$

where

$$\mathbf{c} = \begin{bmatrix} \mathbf{0}_k \\ \tau \mathbf{1}_n \\ (1-\tau)\mathbf{1}_n \end{bmatrix}' \quad \text{and} \quad \mathbf{A}_{\text{EC}} = [\, \mathbf{X} \quad \mathbf{I}_n \quad -\mathbf{I}_n \,]$$

where k is the number of covariates in $\mathbf{x}$ and $\mathbf{I}_n$ is the identity matrix.

Now let us see an example using linear programming for quantile regression.

We use an extract of the dataset on chief-executive officer (CEO) salaries from Wooldridge (2020). This extract was created and is distributed by the Boston College Economics department (see Wooldridge datasets). In addition to salary (`salary`), the dataset also contains information on the CEOs' age (`age`), whether they completed college (`college`) or graduate school (`grad`), their years of experience with the company (`comten`) and as CEOs (`ceoten`), and the company's current profits as a percentage of sales (`profmarg`).

```
. use https://www.stata-press.com/data/r18/ceosal2
(CEO salaries)
```

Let's begin by using `qreg` to perform a quantile regression for the 75th quantile using `salary` as the dependent variable, and `age`, `college`, `grad`, `comten`, `ceoten`, and `profmarg` as independent variables.

```
. qreg salary age college grad comten ceoten profmarg, quantile(.75)
Iteration 1:  WLS sum of weighted deviations =   34405.44

Iteration 1:  Sum of abs. weighted deviations =  34538.146
Iteration 2:  Sum of abs. weighted deviations =  34103.361
Iteration 3:  Sum of abs. weighted deviations =  33839.987
Iteration 4:  Sum of abs. weighted deviations =  33408.942
Iteration 5:  Sum of abs. weighted deviations =  33395.286
Iteration 6:  Sum of abs. weighted deviations =  32994.187
Iteration 7:  Sum of abs. weighted deviations =  32604.049
Iteration 8:  Sum of abs. weighted deviations =  32458.071
Iteration 9:  Sum of abs. weighted deviations =  32278.694
Iteration 10: Sum of abs. weighted deviations =  32164.004
Iteration 11: Sum of abs. weighted deviations =   32146.02
Iteration 12: Sum of abs. weighted deviations =  32098.911
Iteration 13: Sum of abs. weighted deviations =   32091.43
Iteration 14: Sum of abs. weighted deviations =  32054.647
Iteration 15: Sum of abs. weighted deviations =  32003.957
Iteration 16: Sum of abs. weighted deviations =  31837.324
Iteration 17: Sum of abs. weighted deviations =  31741.915
Iteration 18: Sum of abs. weighted deviations =  31515.552
Iteration 19: Sum of abs. weighted deviations =  31492.239
Iteration 20: Sum of abs. weighted deviations =    31488.9
Iteration 21: Sum of abs. weighted deviations =  31488.622
Iteration 22: Sum of abs. weighted deviations =  31483.585
Iteration 23: Sum of abs. weighted deviations =  31407.582
Iteration 24: Sum of abs. weighted deviations =  31407.363
Iteration 25: Sum of abs. weighted deviations =  31398.938
Iteration 26: Sum of abs. weighted deviations =  31398.217
Iteration 27: Sum of abs. weighted deviations =  31396.024

.75 Quantile regression                         Number of obs =        177
  Raw sum of deviations 32811.25 (about 1119)
  Min sum of deviations 31396.02                Pseudo R2     =     0.0431
```

salary	Coefficient	Std. err.	t	P>|t|	[95% conf.	interval]
age	5.379105	9.441549	0.57	0.570	-13.25867	24.01688
college	-554.7379	419.7605	-1.32	0.188	-1383.352	273.8764
grad	186.6717	140.3615	1.33	0.185	-90.40431	463.7478
comten	-6.826164	6.497454	-1.05	0.295	-19.65225	5.999919
ceoten	17.84588	10.23762	1.74	0.083	-2.363353	38.05511
profmarg	-.4654744	3.791003	-0.12	0.902	-7.948978	7.018029
_cons	1298.486	672.6187	1.93	0.055	-29.2744	2626.247

The output table contains the estimated coefficients that we will now obtain by linear programming. For further interpretation of the output, see [R] **qreg**.

We begin by importing the data into Mata and adding a vector of 1s to covariates for the constant term:

```
. mata:
------------------------------------------------- mata (type end to exit) -----
: X = st_data(., ("age college grad comten ceoten profmarg"))
: y = st_data(., ("salary"))
: X = (X, J(rows(X), 1, 1))
```

Now specify that τ should be 0.75.

```
: tau = 0.75
```

Then we formulate the problem as a linear programming problem using the formulas at the beginning of this example:

```
: n = rows(X)
: k = cols(X)
: c = (J(1, k, 0), tau * J(1, n, 1), (1 - tau) * J(1, n, 1))
: Aec = (X, I(n), -I(n))
: lowerbd = (J(1, k, .), J(1, 2*n, 0))
: upperbd = J(1, 2*n + k, .)
```

We then generate an instance of the class and save the required coefficients and the constraints to it:

```
: q = LinearProgram()
: q.setCoefficients(c)
: q.setEquality(Aec, y)
: q.setBounds(lowerbd, upperbd)
: q.setMaxOrMin("min")
```

Now we solve it:

```
: q.optimize()
  31396.02428
: x = q.parameters()
: x[1..k]
                 1              2              3              4
  ┌──────────────────────────────────────────────────────────────┐
1 │  5.379105375  -554.7378912   186.6717198  -6.82616387       │
  └──────────────────────────────────────────────────────────────┘
                 5              6              7
  ┌──────────────────────────────────────────────┐
1 │  17.84587946  -.4654742937   1298.486114     │
  └──────────────────────────────────────────────┘
```

These point estimates are the same as those computed by `qreg`.

◁

Example 4: Data envelopment analysis

Production theory is a fundamental component of economic analysis. When studying production, we base our analysis on the concept of a production function. The production function describes how inputs in the production process are turned into outputs. Before the work of Debreu (1951), Koopmans (1951), and Farrell (1957), it was assumed that the input–output relationship had no inefficiency. It is now common to study and measure deviations from efficient production (called efficiency analysis).

A way to measure production efficiency is data envelopment analysis DEA; see Cooper, Seiford, and Tone [2007]; Färe [1988]; Grosskopf and Knox Lovell [1994]; and Färe and Primont [1995]). DEA makes no assumptions about the functional form of the production function and is therefore robust to misspecification.

Intuitively, we know the total output produced by each firm, given their inputs. We can then measure the results of each firm relative to the most efficient firms in our sample. The most efficient firms, those that produce the most for a given set of inputs, form a frontier, an envelope. All other firms are underneath and are relatively inefficient. What we obtain when using DEA is a measure of relative efficiency, a number between 0 and 1, where inefficiency is any number below 1.

DEA uses the weighted sum of outputs over the weighted sum of inputs to measure efficiency, and it can be written as a linear programming problem. For example, to estimate the efficiency of a firm or, more generally, the efficiency of a unit k, we solve

$$\begin{aligned} \max_{\mathbf{u},\mathbf{v}} \ & \frac{\mathbf{Y}_k\mathbf{u}'}{\mathbf{X}_k\mathbf{v}'} \\ \text{such that } & \frac{\mathbf{Y}_j\mathbf{u}'}{\mathbf{X}_j\mathbf{v}'} \leq 1 \qquad j = 1,\ldots,n \\ & \mathbf{u} \geq \mathbf{0}'_p \\ & \mathbf{v} \geq \mathbf{0}'_m \end{aligned}$$

where $\mathbf{X}_j$ and $\mathbf{Y}_j$ are the inputs and outputs for unit j, respectively. n is the number of units, m is the number of inputs, and p is the number of outputs. This problem can be rewritten as a linear programming problem:

$$\begin{aligned} \max_{\mathbf{u},\mathbf{v}} \ & \mathbf{Y}_k\mathbf{u}' \\ \text{such that } & \mathbf{X}_k\mathbf{v}' = 1 \\ & \mathbf{Y}\mathbf{u}' - \mathbf{X}\mathbf{v}' \leq \mathbf{0}_n \\ & \mathbf{u} \geq \mathbf{0}'_p \\ & \mathbf{v} \geq \mathbf{0}'_m \end{aligned}$$

Using our notation,

$$\begin{aligned} \max_{\mathbf{u},\mathbf{v}} \ & \mathbf{c}\begin{bmatrix}\mathbf{u} & \mathbf{v}\end{bmatrix}' \\ \text{such that } & \mathbf{A}_{\text{EC}}\begin{bmatrix}\mathbf{u} & \mathbf{v}\end{bmatrix}' = \mathbf{b}_{\text{EC}} \\ & \mathbf{A}_{\text{IE}}\begin{bmatrix}\mathbf{u} & \mathbf{v}\end{bmatrix}' \leq \mathbf{b}_{\text{IE}} \\ & \mathbf{u} \geq \mathbf{0}'_p \\ & \mathbf{v} \geq \mathbf{0}'_m \end{aligned}$$

We have

$$\mathbf{c} = \begin{bmatrix} \mathbf{Y}_k \\ \mathbf{0}_m \end{bmatrix} \qquad \mathbf{A}_{\text{EC}} = [\,\mathbf{0}_p' \quad \mathbf{X}_k\,] \qquad \mathbf{b}_{\text{EC}} = 1 \qquad \mathbf{A}_{\text{IE}} = [\,\mathbf{Y} \quad -\mathbf{X}\,] \qquad \mathbf{b}_{\text{IE}} = \mathbf{0}_n$$

Here $\mathbf{0}_m$, $\mathbf{0}_n$, and $\mathbf{0}_p$ are vectors of 0s.

As an example, we use a sample of 756 fictional firms producing a manufactured good with capital and labor. The firms are hypothesized to use a constant returns-to-scale technology, but the sizes of the firms differ. For more details, see example 2 in [R] **frontier**. The inputs for the firms will be capital (`lncapital`) and labor (`lnlabor`); the output will be the manufactured good (`lnoutput`).

```
. use https://www.stata-press.com/data/r18/frontier1
```

We begin by importing the data into Mata:

```
. mata:
------------------------------------------------- mata (type end to exit) -----
: X = st_data(., ("lnlabor", "lncapital"))
: Y = st_data(., "lnoutput")
: n = rows(X)
: m = cols(X)
: p = cols(Y)
```

Now we write the coefficients and constraints as a linear programming problem using the equations above, and we use `id = 1` to indicate the first firm.

```
: id = 1
: c = (Y[id, .], J(1, m, 0))
: Aec = (J(1, p, 0), X[id, .])
: bec = 1
: Aie = (Y, -X)
: bie = J(n, 1, 0)
: lowerbd = J(1, m + p, 0)
: upperbd = J(1, m + p, .)
```

We then generate an instance of the `LinearProgram()` class and store all the required information.

```
: q = LinearProgram()
: q.setCoefficients(c)
: q.setEquality(Aec, bec)
: q.setInequality(Aie, bie)
: q.setBounds(lowerbd, upperbd)
```

Then we can compute the relative efficiency of unit 1, which we defined above.

```
: q.optimize()
  .191803712
```

Because the optimal value is less than 1, we conclude that the first firm is inefficient. If we want to estimate the efficiencies of other firms, we simply change `id` to the firm of interest. For example, if we change `id` to 261, we can get the relative efficiency of firm 261:

```
: q.optimize()
  1
```

This shows that firm 261 is efficient.

◁

▷ Example 5: Dantzig selector

In a linear model, we model the mean of the outcome y_i as the linear combination $\mathbf{x}_i\boldsymbol{\beta}'$, where $\mathbf{x}_i$ are the covariates and $\boldsymbol{\beta}$ are the coefficients. In the standard case, the number of covariates k is small relative to the number of observations n. In a high-dimensional regression, k is large relative to n, but we must assume that many of the coefficients $\boldsymbol{\beta}$ on $\mathbf{x}_i$ are 0.

The Dantzig selector (Candes and Tao 2007) estimates which of the coefficients are 0 and produces estimates of the coefficients that are not 0.

In a standard linear model, we estimate $\boldsymbol{\beta}$ by minimizing the sum of the squared residuals. The first-order conditions for this minimization problem are known as the normal equations, and in matrix form, they are

$$\mathbf{X}'(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \mathbf{0}$$

where $\mathbf{X}$ is a matrix containing the observations of the covariates and $\mathbf{y}$ is a vector containing the observations of the outcome.

The Dantzig selector solution to the high-dimensional linear model finds the smallest-in-magnitude coefficients that get close to solving the first-order conditions. In math, the Dantzig selector solves

$$\begin{aligned} &\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_1 \\ &\text{such that } \|\mathbf{X}'(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\|_\infty \leq \lambda \end{aligned}$$

where λ is a constant that parameterizes "close" to solving the first-order conditions. (The least absolute shrinkage and selector operator is another solution to the high-dimensional linear model; see [LASSO] **Lasso intro** for more details.)

The Dantzig selector problem can be written as the following linear programming problem:

$$\begin{aligned} &\min_{\boldsymbol{\beta},\mathbf{u}} \mathbf{1}_k'\mathbf{u} \\ &\text{such that } \mathbf{X}'(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \leq \lambda\mathbf{1}_k \\ &-\mathbf{X}'(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \leq \lambda\mathbf{1}_k \\ &\boldsymbol{\beta} - \mathbf{u} \leq \mathbf{0}_k \\ &-\boldsymbol{\beta} - \mathbf{u} \leq \mathbf{0}_k \end{aligned}$$

Here $\mathbf{u}$ is a vector of variables used as the upper bounds of the absolute values of $\boldsymbol{\beta}$.

The above version is closely related to the original motivation for the Dantzig selection, but it is not in an easy-to-implement form. A ready-to-implement form for the above problem is

$$\min_{\boldsymbol{\beta},\mathbf{u}} \mathbf{c}\begin{bmatrix}\boldsymbol{\beta}\\ \mathbf{u}\end{bmatrix}$$
$$\text{such that } \mathbf{A}_{\text{IE}}\begin{bmatrix}\boldsymbol{\beta}\\ \mathbf{u}\end{bmatrix} \leq \mathbf{b}_{\text{IE}}$$

where

$$\mathbf{c} = \begin{bmatrix}\mathbf{0}_k\\ \mathbf{1}_k\end{bmatrix}' \quad \mathbf{A}_{\text{IE}} = \begin{bmatrix}-\mathbf{X}'\mathbf{X} & \mathbf{0}_{k\times k}\\ \mathbf{X}'\mathbf{X} & \mathbf{0}_{k\times k}\\ \mathbf{I}_k & -\mathbf{I}_k\\ -\mathbf{I}_k & -\mathbf{I}_k\end{bmatrix} \quad \mathbf{b}_{\text{IE}} = \begin{bmatrix}-\mathbf{X}'\mathbf{y} + \lambda\mathbf{1}_k\\ \mathbf{X}'\mathbf{y} + \lambda\mathbf{1}_k\\ \mathbf{0}_k\\ \mathbf{0}_k\end{bmatrix}$$

In this example, we use an extract of the data used in Sunyer et al. (2017) that models how the attention of school children is affected by pollution. Our sample contains 1,096 students, and we model the relationship between the mean-hit reaction time (`htime`) and other factors, including daily nitrogen dioxide levels (`no2`), home socioeconomical vulnerability index (`sev_home`), school socioeconomical vulnerability index (`sev_sch`), school noise levels (`noise_sch`), starting age at school (`age_start_sch`), number of young siblings they live with (`youngsibl`), and home greenness (`ndvi_mn`).

We begin by using the dataset and dropping the observations that contain missing values.

```
. use https://www.stata-press.com/data/r18/breathe_lp
. generate byte touse = 1
. markout touse *
. drop if touse == 0
(18 observations deleted)
```

Next we create local macros to hold the names of the outcome variable and the covariates in the model, including some powers and interactions of the original covariates.

```
. local ccontrols "sev_home sev_sch age_start_sch ndvi_mn youngsibl noise_sch"
. local ofinterest "no2"
. local depvar htime
. local indeps ‘ofinterest’ ‘ccontrols’ c.(‘ccontrols’)#c.(‘ccontrols’)
```

Then we expand the names of covariates in the model, remove the collinear variables, and save the names back to the same macro.

```
. _rmcoll ‘indeps’, expand
. local indeps ‘r(varlist)’
```

Now we import the data into Mata, remove the mean from the outcome variable, and standardize the covariates.

```
. mata:
------------------------------------------------- mata (type end to exit) -----
: X = st_data(., ‘"‘indeps’"’, "touse")
: y = st_data(., "‘depvar’", "touse")
: ybar = mean(y)
: y = y :- ybar
```

```
: Xbar = mean(X)
: X = X :- Xbar
: sd_X = sqrt(mean(X:^2))
: X = X :/ sd_X
```

We need a value for λ to complete the problem. Prior to writing this example, we used the method of cross-validation to find a good value for λ; see Hastie, Tibshirani, and Friedman (2009) for an introduction to cross-validation. Cross-validation specifies that we should set $\lambda = 4163.558931$.

We are now ready to specify and solve the linear programming problem.

```
: tmpA = quadcross(X, X)
: tmpb = quadcross(X, y)
: lambda = 4163.558931
: k = cols(X)
: c = (J(1, k, 0), J(1, k, 1))
: Aec = (-tmpA, J(k, k, 0) \ tmpA, J(k, k, 0) \ I(k), -I(k) \ -I(k), -I(k))
: bec = (-tmpb :+ lambda \ tmpb :+ lambda \ J(k, 1, 0) \ J(k, 1, 0))
```

We now define an instance of the `LinearProgram()` class and put the required information into that instance `q`.

```
: q = LinearProgram()
: q.setCoefficients(c)
: q.setInequality(Aec, bec)
: q.setMaxOrMin("min")
```

Now solve the problem.

```
: q.optimize()
  8.21319472
```

We put the estimates of β into the Mata vector `b` by typing

```
: x = q.parameters()
: b = x[1..k]
```

Out of the 28 estimated parameters, we only want to see which variables were selected, that is, which variables have estimated coefficients that are greater than 0 in absolute value. To show the variable names that have been selected, we type

```
: indep_list = tokens(`""`indeps'"")
: sel = (abs(b) :>= 1e-6)
: "selected variables are "
  selected variables are
: select(indep_list, sel)
                              1                              2
  1 |                 sev_home   c.sev_home#c.age_start_sch |

                              3                              4
  1 |       c.sev_sch#c.ndvi_mn        c.ndvi_mn#c.noise_sch |
```

Here we consider a coefficient to be 0 when its absolute value is less than 10^{-6}. Recall from *Details about the interior-point method* above and the discussion in example 1 that the interior-point method will not produce solution values that are exactly 0. Lowering the tolerance will make the coefficients that are practically 0 have values closer to 0, but it will not change which variables are selected.

The results indicate that the mean-hit reaction time is related to the home socioeconomical vulnerability index, the home socioeconomical vulnerability index times starting age at school, the school socioeconomical vulnerability index times home greenness, and home greenness times school noise levels.

◁

Conformability

`LinearProgram()`:

input:		
		void
output:		
	result:	1×1

`LinearProgram(`*n*`)`:

input:		
	n:	1×1
output:		
	result:	$1 \times n$

`LinearProgram(`*m*`, `*n*`)`:

input:		
	m:	1×1
	n:	1×1
output:		
	result:	$m \times n$

`setCoefficients(`*coef*`)`:

input:		
	coef:	$1 \times N$
output:		
	result:	*void*

`getCoefficients()`:

input:		
		void
output:		
	result:	$1 \times N$

`setMaxOrMin(`*maxormin*`)`:

input:

	object:	1×1

output:

	result:	*void*

`getMaxOrMin()`:

input:

		void

output:

	result:	1×1

`setEquality(`*ecmat*`, `*rhs*`)`:

input:

	ecmat:	$M0 \times N$
	rhs:	$M0 \times 1$

output:

	result:	*void*

`getEquality()`:

input:

		void

output:

	result:	$(M0 + 1) \times N$

`setInequality(`*iemat*`, `*rhs*`)`:

input:

	iemat:	$M1 \times N$
	rhs:	$M1 \times 1$

output:

	result:	*void*

`getInequality()`:

input:

		void

output:

	result:	$(M1 + 1) \times N$

`setBounds(`*lowerbd*`, `*upperbd*`)`:

input:

	lowerbd:	$1 \times N$
	upperbd:	$1 \times N$

output:

	result:	*void*

`getBounds()`:

input:

void

output:

result: $2 \times N$

`setMaxiter(`*maxiter*`)`:

input:

maxiter: 1×1

output:

result: *void*

`getMaxiter()`:

input:

void

output:

result: 1×1

`setTol(`*tol*`)`:

input:

tol: 1×1

output:

result: *void*

`getTol()`:

input:

void

output:

result: 1×1

`setTrace(`*trace*`)`:

input:

trace: 1×1

output:

result: *void*

`getTrace()`:

input:

void

output:

result: 1×1

`optimize()`:

input:

void

output:

result: 1×1

`parameters()`:

input:

void

output:

result: $1 \times N$

`value()`:

input:

void

output:

result: 1×1

`iterations()`:

input:

void

output:

result: 1×1

`converged()`:

input:

void

output:

result: 1×1

`errorcode()`:

input:

void

output:

result: 1×1

`errortext()`:

input:

void

output:

result: 1×1

`returncode()`:

input:

void

output:

result: 1×1

`query()`:

input:

void

output:

void

Diagnostics

`LinearProgram()`, `q.set*()`, `q.get*()`, `q.parameters()`, `q.value()`, `q.iterations()`, `q.converged()`, `q.errorcode()`, `q.errortext()`, `q.returncode()`, and `q.query()` functions abort with an error message when used incorrectly.

`q.optimize()` aborts with an error message if it is used incorrectly. If `q.optimize()` runs into numerical difficulties, it returns a missing value and displays a warning message including some details about the problem encountered.

References

Andersen, E. D., and K. D. Andersen. 1995. Presolving in linear programming. *Mathematical Programming, Series B* 71: 221–245. https://doi.org/10.1007/BF01586000.

Andersen, E. D., J. Gondzio, and C. Mészáros. 1996. Implementation of interior-point methods for large scale linear programs. In *Interior Point Methods of Mathematical Programming*, ed. T. Terlaky, 189–252. Dordrecht, The Netherlands: Kluwer.

Badunenko, O., and P. Mozharovskyi. 2016. Nonparametric frontier analysis using Stata. *Stata Journal* 16: 550–589.

Belotti, F., S. Daidone, G. Ilardi, and V. Atella. 2013. Stochastic frontier analysis using Stata. *Stata Journal* 13: 719–758.

Brearley, A. L., G. Mitra, and H. P. Williams. 1975. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming, Series A* 8: 54–83. https://doi.org/10.1007/BF01580428.

Candes, E., and T. Tao. 2007. The Dantzig selector: Statistical estimation when p is much larger than n. *Annals of Statistics* 35: 2313–2351. https://doi.org/10.1214/009053606000001523.

Cooper, W. W., L. M. Seiford, and K. Tone. 2007. *Data Envelopment Analysis: A Comprehensive Text with Models, Applications, References and DEA-Solver Software*. 2nd ed. New York: Springer.

Czyzyk, J., S. Mehrotra, and S. J. Wright. 1997. PCx user guide. Technical Memorandum 217, Argonne National Laboratory, Mathematics and Computer Science Division.

Debreu, G. 1951. The coefficient of resource utilization. *Econometrica* 19: 273–292. https://doi.org/10.2307/1906814.

Färe, R. 1988. *Fundamentals of Production Theory*. New York: Springer.

Färe, R., and D. Primont. 1995. *Multi-Output Production and Duality: Theory and Applications*. New York: Springer.

Farrell, M. J. 1957. The measurement of productive efficiency. *Journal of the Royal Statistical Society, Series A* 120: 253–290. https://doi.org/10.2307/2343100.

Gay, D. M. 1985. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society, Committee on Algorithms Newsletter* 13: 10–12.

Gondzio, J. 1997. Presolve analysis of linear programs prior to applying an interior point method. *INFORMS Journal on Computing* 9: 73–91. https://doi.org/10.1287/ijoc.9.1.73.

Grosskopf, R. F. S., and C. A. Knox Lovell. 1994. *Production Frontiers*. Cambridge: Cambridge University Press.

Hastie, T. J., R. J. Tibshirani, and J. H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer.

Huang, X. 2004. *Preprocessing and postprocessing in linear optimization*. Master's thesis, McMaster University.

Jansen, B., J. J. de Jong, C. Roos, and T. Terlaky. 1997. Sensitivity analysis in linear programming: Just be careful! *European Journal of Operational Research* 101: 15–28. https://doi.org/10.1016/S0377-2217(96)00172-5.

Karakaplan, M. U. 2017. Fitting endogenous stochastic frontier models in Stata. *Stata Journal* 17: 39–55.

Koenker, R., and G. Bassett, Jr. 1978. Regression quantiles. *Econometrica* 46: 33–50. https://doi.org/10.2307/1913643.

Koenker, R., and K. Hallock. 2001. Quantile regression. *Journal of Economic Perspectives* 15: 143–156. https://doi.org/10.1257/jep.15.4.143.

Koopmans, T. C. 1951. Analysis of production as an efficient combination of activities. In *Activity Analysis of Production and Allocation: Proceedings of a Conference*, ed. T. C. Koopmans, 33–97. New York: Wiley.

Lustig, I. J., R. E. Marsten, and D. F. Shanno. 1994. Interior point methods for linear programming: Computational state of the art. *ORSA Journal on Computing* 6: 1–14. https://doi.org/10.1287/ijoc.6.1.1.

Marsten, R. E., M. J. Saltzman, D. F. Shanno, G. S. Pierce, and J. F. Ballintijn. 1989. Implementation of a dual affine interior point algorithm for linear programming. *ORSA Journal on Computing* 1: 287–297. https://doi.org/10.1287/ijoc.1.4.287.

Mehrotra, S. 1991. Higher order methods and their performance. Technical Report 90-16R1, Department of Industrial Engineering and Management Sciences, Northwestern University. http://users.iems.northwestern.edu/~mehrotra/TechnicalReports/HigherOrderPerformance.pdf.

——. 1992. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization* 2: 575–601. https://doi.org/10.1137/0802028.

Mészáros, C. 2005. The Cholesky factorization in interior point methods. *Computers and Mathematics with Applications* 50: 1157–1166. https://doi.org/10.1016/j.camwa.2005.08.016.

Netlib. 2013. Netlib linear programming test problems. http://www.netlib.org/lp/.

Nocedal, J., and S. J. Wright. 2006. *Numerical Optimization*. 2nd ed. New York: Springer.

Sadhana, V. V. 2002. *Efficient presolving in linear programming*. Master's thesis, University of Florida. http://etd.fcla.edu/UF/UFE1000157/sadhana_v.pdf.

Sunyer, J., E. Suades-González, R. García-Esteban, I. Rivas, J. Pujol, M. Alvarez-Pedrerol, J. Forns, X. Querol, and X. Basagaña. 2017. Traffic-related air pollution and attention in primary school children: Short-term association. *Epidemiology* 28: 181–189. https://doi.org/10.1097/EDE.0000000000000603.

Tauchmann, H. 2012. Partial frontier efficiency analysis. *Stata Journal* 12: 461–478.

Wei, H. 2006. Numerical stability in linear programming and semidefinite programming. PhD thesis, University of Waterloo. http://www.math.uwaterloo.ca/~hwolkowi/henry/reports/h3wei2006thesis.pdf.

Wooldridge, J. M. 2020. *Introductory Econometrics: A Modern Approach*. 7th ed. Boston: Cengage.

Wright, S. J. 1997. *Primal-Dual Interior-Point Methods*. Philadelphia: Society for Industrial and Applied Mathematics.

Also see

[M-2] **class** — Object-oriented programming (classes)

[M-5] **moptimize( )** — Model optimization

[M-5] **optimize( )** — Function optimization

[LASSO] **Lasso intro** — Introduction to lasso

[R] **frontier** — Stochastic frontier models

[R] **qreg** — Quantile regression

[R] **regress** — Linear regression

For suggested citations, see the FAQ on citing Stata documentation.