Computing Camp - Stata Lecture 3

Automating Tables and Monte Carlo Simulations

Roy Mill

September, 2009

Plan A, ver 2.0

- Class 1 - Introduction
  - Graphical user interface
  - Datasets, command syntax and do-files
- Class 2 - Programming basics
  - Macros and post-estimation variables
  - Loops and control flow
- Class 3 - Posting results and two main examples
  - Posting results to a table
  - Creating a summary statistics table
  - Monte Carlo simulation

Regression tables

This is how they usually look like - every regression has a column and interesting independent variables' coefficients are listed:

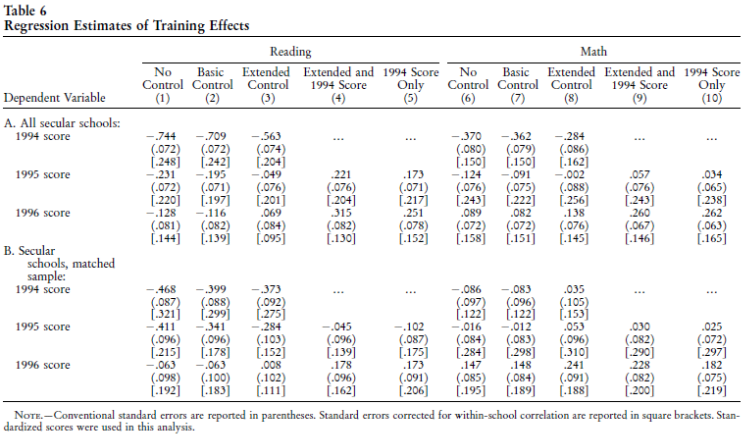

Table 6
Regression Estimates of Training Effects

	Reading					Math				
Dependent Variable	No Control (1)	Basic Control (2)	Extended Control (3)	Extended and 1994 Score (4)	1994 Score Only (5)	No Control (6)	Basic Control (7)	Extended Control (8)	Extended and 1994 Score (9)	1994 Score Only (10)
A. All secular schools:										
1994 score	−.744	−.709	−.563	...	...	−.370	−.362	−.284	...	...
	(.072)	(.072)	(.074)			(.080)	(.079)	(.086)		
	[.248]	[.242]	[.204]			[.150]	[.150]	[.162]		
1995 score	−.231	−.195	−.049	.221	.173	−.124	−.091	−.002	.057	.034
	(.072)	(.071)	(.076)	(.076)	(.071)	(.076)	(.075)	(.088)	(.076)	(.065)
	[.220]	[.197]	[.201]	[.204]	[.217]	[.243]	[.222]	[.256]	[.243]	[.238]
1996 score	−.128	−.116	.069	.315	.251	.089	.082	.138	.260	.262
	(.081)	(.082)	(.084)	(.082)	(.078)	(.072)	(.072)	(.076)	(.067)	(.063)
	[.144]	[.139]	[.095]	[.130]	[.152]	[.158]	[.151]	[.145]	[.146]	[.165]
B. Secular schools, matched sample:										
1994 score	−.468	−.399	−.373	...	...	−.086	−.083	.035	...	...
	(.087)	(.088)	(.092)			(.097)	(.096)	(.105)		
	[.321]	[.299]	[.275]			[.122]	[.122]	[.153]		
1995 score	−.411	−.341	−.284	−.045	−.102	−.016	−.012	.053	.030	.025
	(.096)	(.096)	(.103)	(.096)	(.087)	(.084)	(.083)	(.096)	(.082)	(.072)
	[.215]	[.178]	[.152]	[.139]	[.175]	[.284]	[.298]	[.310]	[.290]	[.297]
1996 score	−.063	−.063	.008	.178	.173	.147	.148	.241	.228	.182
	(.098)	(.100)	(.102)	(.096)	(.091)	(.085)	(.084)	(.091)	(.082)	(.075)
	[.192]	[.183]	[.111]	[.162]	[.206]	[.195]	[.189]	[.188]	[.200]	[.219]

Note.—Conventional standard errors are reported in parentheses. Standard errors corrected for within-school correlation are reported in square brackets. Standardized scores were used in this analysis.

Regression tables

How can we create a table like this?

1. Manually - if you are paid by the hour and you want to maximize profit
2. Using `outreg` or any other command that creates regression tables automatically
3. Creating a custom-made table with a matrix
4. Creating a custom-made table with `post` commands

Using `outreg2`

This is my favorite command, although I didn't use all others. And what do I know, anyway

This program is a user-written program. It's not part of Stata, so you need to download it.

To look for programs online, type `net search <something>`. In our case, `net search outreg2` will do. Click on the outreg2 title (in cyan). It will open a brief description, a list of files and a link to install them. Click on the link.

You have a reaaaaaally detailed help file that you can read once you installed the program (`help outreg2` as always)

Using outreg2

This is how you use it:

```
reg <dependent variable> [independent variables] [if] [in] [, options]

outreg2 [independent variables from above] using <filename> [, options]
```

Each time you will call outreg2 using filename A, `outreg2` will add a column to the right of the table in filename A.

The coefficients from the last regression will be assigned to rows of variables that are already in the table from previous calls to outreg2. Variables that are newly added to this call to `outreg2` will be added below.

Using outreg2

Suppose nothing was in the file table6 before.

```
reg ln_wage educ male age agesq
outreg2 educ male using table6

reg ln_wage educ married age agesq
outreg2 educ married using table6
```

This will create:

First...

	ln_wage (1)
educ	.239 (1.42)
male	2.30 (1.12)

And then...

	ln_wage (1)	ln_wage (2)
educ	.239 (1.42)	.420 (.120)
male	2.30 (1.12)	
married		1.21 (.652)

Using `outreg2`

Some options you might want to use:

- `auto(<integer>)` specifies precision to report.
- `excel` will save it to an XML format you can open in Excel.
- `tex` will save it to a .tex format that you can use for LaTeX.
- `e(<scalars>)` will add statistics from the ereturn list to the bottom of each column - R^2 and N are default.

Here's a tip - when you will rerun your program, unless you erased the table file you created, the new calls to `outreg2` will add columns to the tables created last time you ran the file.

To solve that, either:

- erase the tables you created at the beginning of the do file - using `erase`
- or make sure the first (and only first) call to `outreg2` in your program includes the `replace` option, to overwrite the existing table.

Reporting to a custom matrix

Regression-outputing commands such as `outreg2` are good for reporting regression results, but what if you want to report other things?

In the previous lecture we talked about matrices in Stata. A table is like a matrix, so we can create a matrix in the same structure as the table and put numbers there.

The up side is that we can write values in the table (matrix) any way we want column by column, or row by row, or in any other random order.

The down side: the code is cumbersome and we can't put strings as values. We can rename the rows and column titles, but we'll have problem exporting them to a `.dta` dataset.

Reporting to a custom matrix

```
// Initializing variables (rows) and and statistics (columns) to
// put in the table
local vars2rep "age race not_smsa c_city south"
local stats2rep "mean sd p50 min max"

// We can put these as numbers in the code, but if we will change the
// first two locals we will have to change other parts of the code
// too. Doing it this way allows one to more easily adjust the program.
local varsCount  : word count ‘vars2rep’
local statsCount : word count ‘stats2rep’

matrix table1 = J(‘varsCount’, ‘statsCount’, .)

forvalues varIndex = 1/‘varsCount’ {
  local var : word ‘varIndex’ of ‘vars2rep’

  quietly su ‘var’, detail

  forvalues statIndex = 1/‘statsCount’ {
    local stat : word ‘statIndex’ of ‘stats2rep’
    matrix table1[‘varIndex’, ‘statIndex’] = r(‘stat’)
  }
}

matrix list table1
```

Reporting to a custom matrix

To neatly save this matrix on a file:

```
// Set the current dataset in memory in a temporary place to come back
// to it later
preserve

clear

// Put table on dataset
svmat table1

// Save the dataset as a .dta file
save table1, replace

// Save the dataset as a .csv file (comma-delimited table,
// opened in Excel and other spreadsheet software packages)
outsheet using table1.csv, comma replace

// Give me my dataset back (the one with the data I'm working on)
// "restore" will restore the data from when I ran "preserve".
restore
```

Using the `post` commands

Indexes of matrices can get to you. When posting to matrices you can pretty quickly lose track of what's going on. If you won't lose track, your colleagues probably will.

A more readable and easy way to create tables is using the `post` commands.

First, we define how the columns of the table look like. The table is actually a Stata dataset constructed on the side. We need to specify the variables in this dataset

Then we "post" rows to the table - one by one - until we're done. We don't need to specify in advance how many rows. We just post them one by one until we're done and close the file.

Using the post commands

Syntax to open a post file:

```
postfile <post-ref-name> <newvarlist> using <file-on-disk-name> ///
     [, replace]
```

Here is how we define the same table we did with the matrix commands, but this time with post:

```
postfile table1 mean sd median min max using "output\table1", replace
```

New variables are numeric by default, but you can define strings by preceding the variable name with a str# where # is the maximum number of characters:

```
postfile table1 str32 myVar mean sd median min max using "output\table1" ///
    , replace
```

Variable names in Stata can not exceed 32 characters in length.

Using the post commands

Now that we opened a postfile, we can post rows into it:

```
post <post-ref-name> (<exp>) (<exp>) ... (<exp>)
```

For each column we have an expression. If we have 12 columns, we should have 12 expressions in parentheses.

```
// Specific values
post table1 ("south") (.4096) (.4918) (0) (0) (1)

// Macros and scalars
post table1 ("`var'") (r(mean)) (r(sd)) (r(p50)) (r(min)) (r(max))
```

Do not forget to close the file after you're done posting all rows:

```
postclose table1
```

Using the `post` commands

We're done - output\table1.dta was created. We can open it and export to a csv file, using `outsheet`, or a TeX file using a user-written command such as `listtex` for example.

Sometimes it's easier for your program to post results column by column instead of row by row. You must post row by row with `postfile`, but you can transpose a dataset in Stata using `xpose, clear`.

You can have more than one postfile open at the same time. This is why you need to refer to the postfile-reference-name each time you post. Not that it's very useful for posting tables, but for simulations you might need it.

The Matrix Example - Now in `post` Commands

```
// Initializing variables (rows) and and statistics (columns) to
// put in the table
local vars2rep "age race not_smsa c_city south"

// Open postfile
postfile stats str32 varname mean sd p50 min max using table1, replace

foreach var of varlist ‘vars2rep’ {
  quietly su ‘var’, detail

  post stats ("‘var’") (r(mean)) (r(sd)) (r(p50)) (r(min)) (r(max))
}

postclose stats
```

Nice, eh?

Monte Carlo Simulations - Overview

In Statistics, or Econometrics, we sometimes want simulate samples of one or more random variables to see how statistics of interest (like estimators) behave.

For k times we:

1. Simulate a sample of n observations.
2. Calculate statistics of interest from the sample (for example, an OLS vector of estimates).
3. Save the statistic somewhere for analysis.

After we're done, we can analyze the how our statistic behaved in the k samples we simulated.

For example, if it's an estimator we can check if the average estimate is close to the true parameter (unbiased), or we can calculate the standard deviation to see how "accurate" it is.

Monte Carlo Simulations - A Very Creative Example

Suppose we have a coin toss and an estimator for the probability to get heads. Our estimator is the mean of only even observations:

$$\frac{1}{n/2}\sum_{i=1}^{n/2} x_{2i}$$

(x_i is the i^{th} observation's result - 1 if heads, 0 if tails):

We can toss the fair coin 50 times and calculate our estimator - we got a number: 0.4339 - but what can we do with it? Is it a coincidence for this sample only? We could have drawn a larger sample, but it would not show us how the estimator behaves for small samples.

How about we make the 50 tosses experiment k times instead of one time? Say, 500 times.

Monte Carlo Simulations - A Very Creative Example

For $rep = 1$ to 500:

1. Simulate a sample of 50 coin tosses.
2. Calculate the mean of the even observations (the estimator of interest).
3. Calculate the mean of all observations (another estimator some wise-ass suggested).
4. Post the statistic you calculated to a table.

Now, open the dataset with the 500 observations (each observation is a mean of a 50-obs sample) and see how the estimator looks like. You can also compare estimators (2) and (3).

Monte Carlo Simulations - Simulating Samples

To simulate the sample, we need to think of the ***true*** data-generating process (DGP). After all, our estimator will try to tell us something about this "true" process.

For example, in our coin toss, we need to decide what is our true $\Pr\left[\text{Coin lands heads}\right]$ - whether it's 0.5 or some other number - and then simulate a process that will land heads (or be equal to 1) with this probability.

Computers can simulate pseudo-random numbers. Pseudo, because it's not really random. You give the computer a number to start with, and then it pours out numbers in a way that *resembles* a uniform distribution.

From this uniform distribution we can get other distributions. Stata already has functions to draw other distributions to begin with, but they always use the basic pseudo-random number generator method.

Monte Carlo Simulations - Simulating Samples

Last comments before we start (this is so exciting, isn't it?):

1. **randomization seed** - If you want to replicate the same random numbers every time you run a program, you should set the seed of the pseudo-random numbers generator at the beginning of the program. Otherwise, each run of the program will yield different results:

   ```
   set seed <number>
   ```

2. **Starting a new dataset** - Before you generate new variables in a new dataset, you need to tell Stata how many observations you want in the dataset. This can be done by running:

   ```
   set obs <number>
   ```

3. `drawnorm` **command** - lets you draw univariate and multivariate normally distributed variables.

Now we're ready to begin.

Monte Carlo Simulations - Back to the Coin

```
// Initializing values (configure your program here)
local obsInSample = 50
local repetitions = 500
local trueP = 0.4

set seed 808080

// Open postfile to save results from each repetition
postfile results estimator_a estimator_b ///
       using MonteCarloResults.dta, replace

// "Mute" operations inside repetitions (saves time)
quietly {
  // Iterate repetitions of the experiment
  forvalues rep = 1/‘repetitions’ {
      // Set up the new dataset
      clear
      set obs ‘obsInSample’

      // Toss the coin
      gen u = uniform()        // will make u~Uniform(0,1)
      gen heads = u < ‘trueP’  // will be 1 with Pr=0.4

      // Sample simulation done, see next slide for estimators’
      // calculations.
```

Monte Carlo Simulations - Back to the Coin

```
      // Now let’s calculate our two statistics of interest:
      // (a) Mean of even tosses
      su heads if mod(_n, 2) == 0
      local evenMean = r(mean)

      // (b) Mean of all tosses
      su heads
      local mean = r(mean)

      // Save our statistics in the results file
      post results (‘evenMean’) (‘mean’)
  }
}
postclose results

// Open the results file. Each observations represents
// a sample. For each sample we have two competing
// estimators
use MonteCarloResults.dta, clear
summarize
twoway (kdensity estimator_a, kernel(gaussian)) ///
       (kdensity estimator_b, kernel(gaussian))
```

Monte Carlo Simulations - Final Notes

Basically all Monte Carlo simulations you'll see, at least this year, will follow the same structure:

1. Draw a sample according to some true DGP.
2. Calculate statistics based on the sample.
3. Save statistics and then analyze their performance.

The DGP or the statistics can be more complicated, but the structure of the simulation stays the same.

Thank you and good luck with your first year!

```
#delimit;
clear all;
set more off;
```

```
STATA™ 24.0
Rockstar
Birthday Edition
Copyright 1983-2007
Matt Teschke
1255 23rd St NW
Washington DC 20037
202-466-9275 www.nera.com
24-user Stata for Nerdoids perpetual license:
Serial number: 08211983
Display "Happy Birthday Matt" if todays_date==
date("august 21 2007","mdy") & your_name==
"Matt Teschke"
Happy Birthday Matt
```

Made by some Rebecca and posted on her truly amazing blog

(`http://cakecraziness.blogspot.com`)