Stata Statistics 506

Stata

Stata is statistical analysis software used commonly in social sciences. It is known for it's ease of use, robust support for complex survey design, and comprehensive and clear documentation.

Stata (pronounced either of stay-ta or stat-ta, the official FAQ supports both) has a robust GUI similar to RStudio, or can be interacted with at the command line to run scripts similar to R.

Once you have Stata up and running, the simplest form of use is as a calculator.

```
. display 2 + 4
```

The `.` is often used for documents like this to note a command; whenever you run a command in Stata, the first thing it does is echo back the command you ran prefixed by the `.`.

The command name is "display" and it outputs the requested expression. This is a trivial example, but `display` is much more flexible.

Terminology

Stata uses some programming terminology in unique ways. Specifically:

- "command" - the primary operation, for example `regress` or `display`. This is what R (and most programming languages) call a "function".
- "variable" - refers **only** to columns in the data set.
- "macro" - what most other languages would call "variables". Can store numbers or strings.
- "function" - An operation performed on a value or a variable. For example, `log(salary)` applies the function `log()` to the variable "salary".

One Data Set

One quirk of Stata is that until the most recent version of Stata, only a single data set could be open at a time. Because of this, in earlier versions of Stata if you wanted to operate on multiple data sets, you either need to be constantly switching between them, or merge them into one file (for help with merging, look into the `append` and `merge` commands).

The benefit of having a single data set open is that you never need to refer to it - any command you give must operate on the open data set. This greatly reduces the code complexity as commands do not need the equivalent of R's `data =` argument.

The most recent version of Stata introduced the concept of frames, – rectangular data sets much like data.frames in R – that can be used to store multiple data sets in memory. I would strongly recommend you do not start with them, instead operating as Stata prefers - a single data set at a time.

Do-files

Stata scripts are called "Do-files", named for their extensions (.do). In the GUI, you can open a new Do-file for editing by typing `doedit`. You can also select a series of commands in the history, right click, and choose the "Send selected to Do-file Editor".

You can run Do-files interactively by highlighting the desired code and hitting the "Execute (Do)" (Windows) or "Do" (Mac) buttons or using an associated shortcut key. Note that unlike RStudio, you do not need to highlight the entire line of code, any lines of code which have any highlighting will be run.

Do-files can also be run from start to finish by using the `do` command. Alternatively, if you are accessing Stata at the command line via SSH, you can simply run `stata mydofile.do` to launch Stata, run the Do-file, and exit.

Common Syntax

Most Stata commands follow the same basic format:

```
. command <variable(s)>, <options>
```

The command can have sub-commands. For example, the `duplicates` command has a `drop` subcommand, so the full "command" would be `duplicates drop`.

The number of variables which needs to be/can be passed obviously varies by command. The order of variables may matter (for example, any regression model treats the first listed variable as the response.)

The options are space-separated words (e.g. `robust`) or words with options (e.g. `level(.90)`). Almost all commands support some number of options, most commands do not require any options.

For example, a regression would be as simple as

```
. regress y x1 x2, noconstant
```

This is defining a linear regression model predicting `y` based upon the continuous variables `x1` and `x2`. The `noconstant` operation forces ($\beta_0 = 0$).

Stata supports abbreviations for most commands. For example, the regression command could have been typed as:

```
. reg y x1 x2, nocons
```

I would not recommend using abbreviations in most cases (a lot aren't as clear as `reg` - what do you think the `d` command does?), but you may see others use them.

Most commands support an `if` or `in` option to operate on a subset of the data. For example,

```
. regress y x1 x2 if female == 1
```

performs linear regression only on the female subsample. The `in` option operates on specific rows, and may be useful for testing slow running code:

```
. regress y x1 x2 in 1/100
```

The `1/100` syntax is equivalent to writing `1 2 3 4 .... 99 100` - most lists in Stata are space-separated, as opposed to R's comma-separated, or +-separated in formulas.

Help Files

Stata has extremely well-written, comprehensive documentation. Use the `help _____` syntax to look up documentation on any command. Help files contain the syntax of the command, a list of options allowed, a description of the command, and a number of examples. Additionally, the very top of each help file contains a link to the appropriate location in the full Stata manual which typically has further examples. Use these help files liberally.

Note that running `help` on an abbreviation (e.g. `help reg`) pulls up the help correctly. This is useful when reading others code. Additionally, in the syntax and options listing, you'll see partial underlining. These indicate the required abbreviations.

I/O

You can access files with:

- `use` to load a local file, or when passed a URL, an online file.
- `sysuse` or `webuse` to access Stata example data sets (often used in Help examples)
- `import excel` and `import delimited` for importing Excel or CSV files.

For importation, it's often easier to use the dialog box from File -> Import menu instead of trying to type the command. The Import dialog box has a live preview as you change settings which can be useful. After you run the import, the command it generated is echoed and you can save it in your Do-file to run it again.

You can save a data set with the `save` command which works as

```
. save newfilename
```

Passing the `replace` option allows Stata to overwrite an existing file. Stata will throw an error if you try to overwrite an existing data sets without specifying replace.

To make your analyses reproducible, I'd recommend *always* specifying replace for data sets created by your scripts, and *never* saving to data files obtained from elsewhere.

If you try and open a new data set when there are un-saved changes in the existing data, Stata will refuse with an error. The `clear` command will remove the existing data and allow you to load a new data set. You can also pass `clear` as an option to the commands `use`/`sysuse`/`webuse`/`import` to save one step.

The `preserve` and `restore` commands can be very useful for switching between files or making destructive modifications to an existing data set. The `preserve` command takes a snapshot of the data as it currently is, and `restore` switches back to it. To see an example of it, consider the `collapse` command. The `collapse` command generates a summary data set by collapsing the existing data by a given variable and creating summary statistics. For example, if we have some sort of census data,

```
. collapse (mean) bmi (percent) female, by(state)
```

This would replace the existing data set with a new one with one row per state, containing a variable indicating which state, a variable indicating the average BMI in that state, and a variable indicating the percent of the state which is female.

The command `collapse` is destructive; after running it the original data (and any unsaved changes) is lost. We can wrap this in `preserve` and `restore` to save the descriptive data and recover the original data.

```
. use data1
. preserve
. collapse (median) x1 x2 (min) x3 (sum) x4, by(group)
. save summarydata
. restore
```

After running the above five lines, “data1” will still be open, but we’ll have a new file “summarydata”.

Data Inspection

The following commands may be useful for exploring the data:

- `browse` opens the data browser. `browse <variable name(s)>` restricts the window to only the requested variables. `if` and `in` can reduce the rows displayed, and can be combined: `browse age weight if gender == "male"`.
  - `edit` opens the data editor, which looks identical to the data browser, except it allows you to edit the data like a spreadsheet. (Doing so produces code which you can copy into your Do-file.
- `describe` displays some information about the variable, including it’s type (string or numeric) and whether it has labels attached to it.
- `summarize`, `codebook`, `tabulate`, `mean` display some descriptive statistics.

Manipulating data

- `drop`/`keep` - Removing columns (using `drop age gender`) or rows (using `keep if gender == "female"` or `drop in 1/20`).
- `generate` - Creating new variables using syntax `generate <newvarname> = <expression>`. The expression can depend on other variables. If the expression is logical (e.g. `age > 40`), this generates a 0/1 binary - logicals don’t exist as anything but numeric 0/1.
  - There’s a separate commend, `egen` (*not* `egenerate`, just `egen`) that supports grouping operations and a number of unique functions, such as `rowmean`. See `help egen` for a full list.
- `replace` - Modify an existing variable. Commonly used with `if`.
- `destring`/`tostring` - Convert strings that are really numbers to strings, and vice-versa
- `encode`/`decode` - Convert strings with words to numbers with associated labels, and vice-versa.

Estimation commands

In addition to the one data principal, Stata also operates on one estimation command at a time. An estimation command is any command which produces some inference, for example `mean` to obtain the mean and a confidence interval, or `regress` as we discussed above. All models are estimation commands.

After running an estimation command, Stata provides access to post-estimation commands and returned objects.

Post-estimation commands

Estimation commands support commands that follow them and use their results. For example, after a regression command we can obtain the AIC and BIC by running

```
. estat ic
```

Note that we do not refer to a specific model. You can see a list of all post-estimation commands supported by running `help ____ postestimation`. For example, `help regress postestimation`.

Replaying estimation commands

If you run an estimation command, then want to display the results again *before you've run another estimation command*, you can simply run the command again without any input. For example,

```
regress y x1 x2
<a bunch of commands which are NOT estimation commands>
regress
```

That second call to `regress` will simply redisplay the results without re-running the whole model - useful if your model is slow.

You can also sometimes pass additional options to change the display. For example, the `logit` command runs logistic regression but produces estimates of log-odds. The `or` option produces odds ratios instead - if you neglect to add `or` to the original `logit` call, you can just add it to the replay:

```
logit z x1 x2
logit, or
```

Saving and restoring estimation results

You can store estimation models with `estimates store` and restore them with `estimates restore`. Once you restore a model, its as if you just ran it - you can replay it or run post-estimation commands against it. This can be very useful if you are jumping between several models that are slow to run. There's also `estimates save` and `estimates use` for writing the models to a file.

Returned objects

The code run by issuing an estimation command, typically creates several objects which can be examined. Estimation commands are either of R-class or E-class (the distinction is not very important). What is important is that E-class commands store things in the `ereturn` while R-class commands store in the `return`.

For example, after running `regress`, the object `e(r2)` contains the (R^2) for the model.

```
. regress y x
. display e(r2)
```

Matrices can also be returned, but cannot be accessed with `display`, instead we use `matrix list`:

```
. matrix list e(V)
```

which will return the variance-covariance matrix of the estimators.

You can see a full list of the returned objects via `return list` or `ereturn list`. The help file for each command will also describe what the command returns at the very bottom of the help document.

We will see how these objects can be manipulated after we discuss macros.

Macros

Macros in Stata use a simple substitution evaluation system. We define a macro with the `local` command.

```
. local myvars x1 x2 x3
. regress y `myvars'
```

The `local` command stores whatever is passed as the name of the macro. When it is referenced via the backtick-single quote syntax, the stored value is substituted into the command before the command is executed. For example, when you run the `regress` line above, Stata will replace "'myvars"' with "x1 x2 x3" and then execute the regression model.

Macros can also store numeric values and can be operated on.

```
. local x 3
. display `x'
. local y = `x' + 2
. display `y'
```

You'll notice the first `local` call contains no equal sign, whereas the second does. The equal forces immediate evaluation. You can see the difference by running:

```
. display `y'
. display "`y'"
. local y2 `x' + 2
. display `y2'
. display "`y2'"
```

You can store returned objects and operate or display them.

```
. regress y x
. local r2 e(r2)
. display "The model R^2 is " `r2'
```

Global vs local macros

A downside to `local` macros is that they're local - they are in a local scope. If you run a chunk of code from a Do-file that defines a local, you'll notice the local does not exist when the code finishes. Try it - run this inside a Do-file:

```
local z 5
```

Then run this at the command window:

```
display `z'
```

Global macros are global - they exist in permanence across Stata.

```
global a 4
display $a + 2
```

Note the use of `$` to access the global, as opposed to the backtick-quote for locals.

There's a longer discussion on the technical details between locals and globals on the Stata blog.

I would suggest using local macros whenever you can, falling back to globals only for debugging or other limited scenarios. This is similar to R's usage of `=` vs `<-` for arguments.

Matrices

Matrices can be stored as well

```
. regress y x
. matrix v = e(V)
. display "The variance/covariance matrix:"
. matrix list v
```

Note that when referring to matrices, no tick/quote is needed. However, matrices can be referred to in more limited contexts than can macros.

You can subset matrices with `[` as in R. E.g. this would return the top 3x3 matrix:

```
matrix B = A[1..3, 1..3]
```

and this would return only the first column:

```
matrix C = A[1..., 1]
```

You can store a single entry from a matrix into a macro as well:

```
local m1 = A[3,5]
```

Loops

Loops are another place where macros are used often. The syntax is very similar to other languages.

```
foreach <macro name> of  <list of numbers/words/variables> {
  ..
}
```

The list can be:

- numbers: `numlist 1/5` or `numlist 1 4 29 192` or `numlist 1/5 10 20`
- variables: `varlist x y z` (`x` `y` and `z` must be variables in the current data)

- words: `a b c` (`a` `b` and `c` need not correspond to any existing variables/macros)

For example, to regress over a series of outcome variables:

```
foreach var of varlist y1 y2 y3 {
    regress `var' x1 x2
    estimates store reg_`var'
}
```

Alternatively, if the variables names are that clean, you could use the numerical suffix as the iterator:

```
foreach i of numlist 1/3 {
    regress y`i' x1 x2
    estimates store reg_y`i'
}
```

Mata

Mata is a matrix programming language which is part of Stata. It has two primary uses:

- Perform matrix algebra more directly.
- Manipulate Stata results.

Matrices in Stata can be passed into Mata and vice-versa, but for the most part Mata is a complete separate language to Stata.

Mata is entered by using the `mata` command.

```
. mata
```

When in Mata mode, the prompt changes from a `.` to `:`. Stata commands will not be accepted inside Mata. To exit Mata, enter the `end` command.

```
: end
```

You can enter mathematical expressions directly into Mata:

```
: 5 - 4
```

When inside Mata, you can define and use "variables" in the same sense as R.

```
: x = 4
: x + 2
```

It also supports functions, e.g.

- `sqrt(4)`
- `log(x)`

Mata sessions have permanence. If you `end` a Mata session and then invoke a new session, it retains the same variables.

You can also run a single line of Mata with the `mata:` preface. After running this command, you will be in Stata, not Mata.

```
. mata: 2 + 2
```

To define a Mata matrix, we can combine the column-join operator `,` and the row-join operator `\`. We can print a matrix by calling it alone.

```
: M = (1, 2\3, 4)
: M
```

To help keep track of object dimensions I'd recommend using lowercase letters for scalars and upper case letters for matrices.

Matrix operations work as expected. We can:

- Matrix by scalar operations: `mata: 4 * M`, `mata: M :+ 2`
- Matrix by matrix operations: `mata: A + B`, `mata: A * B` (an error if dimensions are not compatible)
- Matrix transposition: `mata: A'`
- Matrix concatenation: `mata: A\B` would stack A on top of B , `mata: A,B` would place A next to B - both assuming dimension compatibility.
- Element-by-element matrix multiplication: `mata: A :* B`
- Quickly define an identify matrix: `mata: C = I(5)`
- Quickly define a matrix of a given size with a constant element in each entry: `mata: D = J(4, 2, 0)`
- The `diagonal` function extracts only the diagonal of the matrix. Confusingly, as opposed to the `diag`, which only sets off-diagonal elements to 0.

We can pass matrices between Stata and Mata using the `st_matrix()` function. This functions works inside of Mata. `st_matrix(<stata matrix name>)` obtains a matrix in Stata; `st_matrix(<stata matrix name>, <mata matrix name>)` places a mata matrix into Stata.

Let's say we run a regression in Stata:

```
. sysuse auto
. regress mpg headroom
```

We saw before that `e(V)` contains the variance/covariance matrix, but let's obtain the standard errors for the coefficients.

```
. matrix v = e(V)
. matrix list v
```

To manipulate these, we can pass them into Mata.

```
. mata:
: V = st_matrix("v")
: SE = diagonal(sqrt(V))
: st_matrix("se", SE)
: end
. matrix list se
```

Regression and `margins`

One of Stata's most popular tools is the `margins` command. It is a very complex command (the help manual for this one command is 58 pages long), but extremely powerful. We'll explore two common uses.

First, when running a regression with a categorical variable, one level of the categorical variable is the reference, and all other groups are tested against it. If you wanted to test between variables which weren't the reference, you need to either change the reference category or develop a contrast. `margins` does this second step automatically.

```
. regress y x i.z
. margins z
. margins z, pwcompare
```

First, note the `i.z` variable in the `regress` command. All Stata models assume by default that variables are continuous. To treat a variable as categorical, preface it with `i.`: E.g. variable `race` would be, in the model, `i.race`.

The first margins command will estimate the marginal mean for each level in `z`. This is done by taking the original data, assuming in every row of data that `z` takes on it's first group, and then using the regression equation to estimate each response and generate the average. Then it repeats this process for the second group of `z` and so on.

The second margins command adds the `pwcompare` option which will generate all contrasts of comparison between each level of `z`. By default it produces a confidence interval for each contrast; you can obtain a p-value by passing `pwcompare(pv)` instead.

The second use of `margins` is for the creation of interaction plots. Interactions are entered into regressions via `x##z`; this includes the main effects of both x and z as well as their interaction. (`x#z` includes *only* the interaction term; `x z x#z` is equivalent to `x##z`.) When a variable is involved in an interaction, Stata assumes it is categorical; you can use `c.` to treat it as continuous.

```
. regress y c.x##i.z
. margins z, at(x = (1 2 3 4 5))
. marginsplot
```

This `margins` call is obtaining the marginal mean of z at each of those 5 values of x. `marginsplot` is a post-post-estimation command which can be run after `margins` to produce a plot (another extremely powerful yet complex command, another 35 pages of documentation).

We can test whether the slope is the same across several subgroups with `margins` as well:

```
. margins z, dydx(x)
. margins z, dydx(x) pwcompare(pv)
```

The `dydx()` option estimates the slope on a continuous variable by taking the slope of the regression equation relative to the continuous variable. By passing the categorical variable `z` we're asking for the slope in each group, and then testing them against each other next.