Notes on using Stata

John Gardner

Contents

Introduction

These notes introduce (what I consider) the most important things you need to know to start using Stata for empirical projects, illustrating some helpful tricks along the way. While I hope they will provide you with a good start, they are not comprehensive, so I suggest consulting the resources provided at the end for even more.

Interacting with Stata

You can type commands directly into the *command* pane, but it's a better idea to use a *do* file (a script ending in `.do` that contains a series of commands) to keep a record of all of your commands. Stata comes with an integrated do file editor, and you can create a new do file by going to `File > New > Do file`. In the do file editor, you can run the current line by clicking on the `do` command or the entire file by clicking on `Execute (do)` from the dropdown next to the `do` button. If you're old school, you can also write your do files in an external editor and run them by typing `do filename.do` in the *command* pane.

Before you run your do file, you need to tell Stata the directory the file is saved in, which you can do using

```
cd "path/to/file"
```

on macOS or `path\to\file` on Windows.

You can get help on any command by typing `help commandname` into the command pane.

Packages extend Stata's functionality. Many packages are hosted on the Statistical Software Components repository, and can be installed using `ssc install packagename`. If that doesn't work, type `findit packagename`, click on the the appropriate result, then scroll to the bottom and click the link that says "(click here to install)".[1]

A do file template

Here is the template that I use to start every do file:

```
capture log close
clear all
set more off
// local path "/Users/..."
// local path "C:\Users\..."
cd "`path'"
capture log using filename.log, text replace
********************************************************************************
// your commands here
********************************************************************************
capture log close
```

Here is an explanation of these commands:

- Stata allows you to keep a log of all your commands and the resulting output, and `capture log close` tells Stata to close the existing log if there is already one open. If we typed `log close` but there was currently no open log, Stata would give an error; the `capture` prefix prevents this from happening (you can use this with any Stata command).
- `clear all` tells Stata to clear any existing data and user-written programs. Stata tries to prevent you from accidentally overwriting your data, and will throw an error if you try to load a dataset and there is already one open.

[1] You can also manually install a package by downloading the files placing them your personal ado directory (see https://www.stata.com/support/faqs/programming/personal-ado-directory/).

- `set more off` tells Stata to print the output from all commands. Without this, Stata would fill the *results* pain with output, then pause until you hit enter.
- Stata allows three kinds of comments. Anything after `//` is a comment, any line starting with `*` is a comment, and anything between `/*` and `*/` is a comment, even if the contents span multiple lines.
- `local path "/Users/..."` defines a *local macro* named path, which I fill with the path to my file (I have one line for macOS paths and one for Windows paths, and I uncomment whichever one I need).
- `` cd "`path'" `` tells Stata to set the directory to the contents of the macro `path`. Note the weird quotation marks. The double quotes `""` tell Stata that we're working with a *string*, and the ‘`path'`tells Stata that string is the contents of the local macro `path` (note that this is a backtick `` ` `` followed by a normal single quote `'`) . This is called *dereferencing* the macro.
- `capture log filename using filename.log, text replace` tells Stata to open a log named `filename.log`, save it as a plain text file, and replace the existing one if there is already one by that name. By default, Stata saves logs as `.smcl` files, which uses Stata's internal formatting, but I prefer to work with plain text. Finally, `capture log close` at the end closes the log.

It's also a good idea to start every do file with a comment describing what it does, so that you can find a file that you're looking for several months after writing it.

Programming tools

Macros

A *macro* is a “variable” in the sense that x is a variable in the expression $y = a + b * x$, as opposed to the sense in which `income` is a variable in a dataset. Macros can be numeric or character, and are defined using

```
. local macro1 thing1 thing2 thing3
. local macro2 42
```

Note. Some of the examples in these notes show Stata output instead of the original commands. Before presenting the results of a command, Stata *echoes* the command itself, preceded by “`.`” (and sometimes “`>`” when a command is continued from a previous line, or numbers when the command contains a loop). You don't need to type these characters when entering Stata commands into the command pane or a do file.

You can display the contents of a macro using the `display` command:

```
. display "`macro1'"
thing1 thing2 thing3
. di `macro2'
42
```

In the second line, we used the fact that Stata only requires you to type enough of a command to uniquely identify it, so it sees `di` as `display`.

This is known as *dereferencing* the macro. To dereference the macro, we surround it with a backtick `` ` `` followed by a single quote `'`. In the example above, if we don't surround `` `macro1' `` in quotes, Stata thinks we're asking it to display the variables named `thing1`, `thing2` and `thing3`, which don't exist. We don't need to do this for `` `macro2' `` because it is numeric.

You can add to a macro iteratively, as in:

```
. local macro1 `macro1' thing4
. di "`macro1'"
thing1 thing2 thing3 thing4
```

If the entries of a macro contain spaces, they can be enclosed in quotation marks, but then the entire macro needs to be enclosed in *compound quotes* (`` `""' ``). We can also use the *macro function* `word` to refer to specific elements of a macro. Macro functions are preceded by colons when defining a new macro or dereferencing existing ones:

```
. local macro3 `" "first entry" "second entry" "'
. local macro4: word 1 of `macro3'
. di "`macro4'"
first entry
. di "`: word 1 of `macro3''"
first entry
. di "`: word count `macro3''"
2
```

Note that in the third and fourth lines we dereference the macro expression using `` `: expression' `` as well as the macro contained in the expression.

You can also evaluate mathematical expressions involving macros, either when defining new macros, or when dereferencing them (in the following, we could also use `` local n3 =  `n1' + `n2' ``):

```
. local n1 1
. local n2 2
. local n3 `n1' + `n2'
. di `n3'
3
. di `=`n1' + `n2''
3
```

There are also *global macros* which are defined using `global macroname contents` and dereferenced using `$macroname`. Roughly speaking, the difference between local and global macros is that local macros only "exist" while the current do file is being run, while global macros exist until you exit Stata (and can be accessed from other do files). *Scalars* are like macros, but can hold longer strings. You define scalars using `scalar scalar1 = 1` or `scalar scalar2 = "thing5"`, and they don't have to be dereferenced (try `di scalar2`). I usually stick with local macros for simplicity, but global macros and scalars can be more convenient in some settings.

Loops

Loops allow us to perform actions iteratively, and are incredibly useful in combination with macros. `foreach` loops perform an action for every item in a list, and there are several ways to specify them. The basic syntax is:

```
. foreach x in a b c {
  2.   di "`x'"
  3. }
a
b
c
```

The foreach statement is making an implicit macro with contents a b c, and looping through its elements. We can also loop through the elements of an existing macro:

```
. foreach x of local macro3 {
  2.   di "`x'"
  3. }
first entry
second entry
```

If we want to loop through a list of numbers, we can use numlist:

```
. foreach x of numlist 1/3 {
  2.   di "`: word `x' of `macro3''"
  3. }
first entry
second entry
```

In addition to numlist, there is also varlist for lists of variables. We will see examples of this below.

The syntax 1/3 means 1 2 3. If we wanted to count backwards, we could use 3/1, and we could use 1(2)5 to count up in increments of 2.

Alternatively, we can use forvalues instead of foreach when looping over numbers (below, I initialized sum as sum = 0, but I could have been lazy and just typed local sum):

```
. local sum = 0

. forvalues i=1/3 {
  2.   local sum = `sum' + `i'
  3.   di `sum'
  4. }
1
3
6
```

While loops and conditionals

While loops can be used to iterate while a condition holds, and if and else can be used to execute a command based on a condition. Here is a simple example for checking whether numbers are even (mod(a, 2) is the remainder after dividing a by 2):

```
. local i = 1

. while `i'<5 {
  2.   if mod(`i', 2)==0 {
  3. di "`i' is even"
  4.   }
  5.   else {
  6. di "`i' is odd"
  7.   }
  8.   local ++i
  9. }
1 is odd
2 is even
3 is odd
4 is even
```

The syntax local ++i is a shorthand for local i = `i'+1.

Here's a helpful little trick to avoid an error when you're not sure if a variable exists in your dataset:

```
. capture confirm variable z

. if _rc==0 {
.  replace x = x^2
. }
```

```
. else {
.   di "That variable doesn't exist, bub"
That variable doesn't exist, bub
. }
```

Programs*

A Stata "program" is analogous to a function: it yields output from some inputs. You may never need to write a program (so this section can be skipped on first reading), but they are useful for simulations and bootstrapping, and the basics are pretty easy.

We haven't talked about data yet, but let's generate some random data to make a sample program:

```
. clear all
. set obs 100
Number of observations (_N) was 0, now 100.
. set seed 57474
. gen y = rnormal()
. gen x = runiform(0,1)
```

This clears any existing data, sets the number of observations to 100, and generates a pseudo-normally distributed variable named `y` and a uniformly distributed variable named `x`. We *seed* the random number generator to ensure that we get the same result every time.

Now, let's write a quick-and-dirty program that calculates the mean of a variable raised to an exponent:

```
. capture program drop expmean
. program define expmean, rclass
  1.   capture drop expvar
  2.   gen expvar = `1'^`2'
  3.   quietly sum expvar
  4.   di r(mean)
  5.   return scalar result = r(mean)
  6. end
. expmean x 2
.32759205
. expmean y .5
(53 missing values generated)
.76818721
. return list
scalars:
             r(result) =  .7681872131342583
```

Here's how this works: First, we drop the program `expmean` in case it's already defined. Then we define it, and tell Stata that it is an *r class* program, which means that it will return some scalar (there are also *e class* programs which return a vector of parameters and a variance matrix, but we won't be covering those). Next, we drop the variable `expvar` in case it already exists, and then define `` expvar = `1'^`2' ``, where `` `1' `` is the first argument to our program and `` `2' `` is the second. Then we get the mean of `expvar` using the `quietly` prefix (which means it won't print the results of the command). Finally, we use `di r(mean)` to print just the mean, which we return as the scalar named result.

In our program, we ran `sum expvar`, then displayed the results as `di r(mean)`. This is because `sum` is also an r class program which returns scalars. We can type `return list` (or `ret li` for short) after any r class program to see what it returns. Running it after `expmean` shows that it saves a scalar named `result`, which we can display using `di r(result)` (note that we don't need to dereference this because it is a scalar instead of a macro).

This program gets the job done (and often that's all you need), but it also leaves room for improvement. One issue is that it leaves behind the variable `expvar` in our data, which isn't ideal. Another is that the syntax is pretty different other Stata commands. Also, the output is fairly spartan. Here is a fancier version that addresses these issues:

```
. capture program drop expmean

. program define expmean, rclass
  1.   syntax varname [, NUMber(real 1)]
  2.   tempname expvar
  3.   tempvar `expvar'
  4.   gen `expvar' = `varlist'^`number'
  5.   quietly sum `expvar'
  6.   di "mean of `varlist'^`number': " r(mean)
  7.   return scalar result = `r(mean)'
  8. end

. expmean x, number(2)
mean of x^2: .32759205

. expmean y, num(.5)
(53 missing values generated)
mean of y^.5: .76818721

. ret li

scalars:
             r(result) =  .7681872131342583
```

Here, we use the `syntax` statement to specify what the command should look like. The `syntax` statement takes the name of our input variable, and allows an optional real number, where the default is 1 (if we got rid of the brackets and the default, this would be required; the capitalization `NUMber` allows us to use `num` as a shorthand for `number`). Next, we declare that `expvar` is a temporary name (this avoids conflicts in case we give something else this name later), and we define a temporary variable, also named `expvar` (this way, the variable won't be left over after our program runs; note that we refer to temporary variables similarly to macros). Finally, we cleaned up the output a bit.

Naturally, there is much more to writing Stata programs, but this is all (more than, really) we need for most projects. Type `help program` for more, and a link to the Stata Programming manual.

Working with data

Reading and writing data

To open a dataset in Stata's native `.dta` format, type

```
use "filename.dta", clear
```

Anything after a comma in a Stata command is an *option.* Here, the `clear` option is a precaution in case there is already a dataset loaded. Once you've loaded a dataset, you can inspect it by using

```
. describe

Contains data
 Observations:           100
    Variables:             3
------------------------------------------------------------------------------------------
Variable      Storage   Display    Value
    name         type    format    label      Variable label
------------------------------------------------------------------------------------------
y               float   %9.0g
x               float   %9.0g
expvar          float   %9.0g
------------------------------------------------------------------------------------------
Sorted by:
     Note: Dataset has changed since last saved.
```

or get basic descriptives using

```
. summarize

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
           y |        100   -.1329901    .9963762  -2.295731   2.041936
           x |        100    .4867573    .3026138   .0011245   .9981707
      expvar |         47    .7681872    .3485469   .0702013   1.428963
```

We can also browse the data in a spreadsheet-like view by typing `browse` (you could also use `browse y1 x1` to only view those variables).

To save a dataset in this format, use (the `replace` option overwrites the existing file, if one exists):

```
save "filename.dta", replace
```

Stata can read several different data formats, but I'm going to focus on the most common ones. To read a CSV (comma separated values) file, you can use:

```
. insheet using "data.csv", comma clear
(8 vars, 50 obs)
```

If these data were saved in Excel format, we could import them using

```
import excel "data.xlsx", sheet("data") firstrow clear
```

The `firstrow` option imports the first row of the data as variable names.

Note. One nice aspect of Stata is that you don't have to remember the syntax for these commands. For example, if you go to `File > Import > Excel spreadsheet`, you can use the graphical interface to import the file. When you're done, Stata will import the spreadsheet, but also print the syntax that it used to do the import, which you can copy into your do file.

To save these data in Stata's format, you can use

```
save "data.dta", replace
```

or to export them as a CSV file,

```
outsheet using "data.csv", replace
```

Variable names and labels

You can rename a variable using `rename oldname newname` or multiple variables using rename `(oldname1 newname1) (oldname2 newname2)`.

Sometimes, data are stored so that the variables are named in uppercase letters. You can do away with this using `rename *, lower` (here, the asterisk is a wildcard that represents all variables).

Stata allows you to apply labels to variable names and values. To label a variable, you can use

```
label variable y1 "Variable name"
```

to label values, use:

```
. label define mylabels 1 "Label 1" 2 "Label 2" 3 "Label 3" 4 "Label 4" 5 "Label 5", rep
> lace

. label values z2 mylabels
```

When you run a command that uses labelled variables, the output will refer to your variable and value labels. I don't use labels often, but they are handy for exporting nicely formatted tables.

You still need to refer to categorical variables by their numeric values, even when they have been labelled. To see a list of the values and corresponding labels, type `codebook z2` for one variable or `codebook` for all of them.

Subsetting data

If you wanted to drop the variables starting with `x` from your dataset, you could use any of the following:

```
drop x1 x2 x3
drop x*
keep y* z*
```

The syntax `x*` means "match anything starting with `x`".

If you wanted to drop a subset of observations, say those where `y1 < 0`, you could use

```
drop if y1 < 0
keep if y1 >= 0
keep if inrange(y1, 0, .)
```

The command `inrange(y1, a, b)` selects observations where $a \leq y_1 \leq b$.

You can also run commands on subsets of data. For example, if you wanted to summarize the data for observations where `y1 < 0`, you could use

```
. sum if y1 < 0

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          x1 |         22    .2384604    1.125911  -1.329126   2.537032
          x2 |         22   -.1554666     .863381   -1.57978   .9924132
          x3 |         22    .1326971    1.038498  -2.395694   1.894375
          y1 |         22   -.9943454    .6494489  -2.152447  -.0570929
          y2 |         22   -.1279769    .8899685  -1.367311    2.49957
-------------+---------------------------------------------------------
          y3 |         22   -.4317595    1.130035  -2.732989   1.216734
          z1 |         22    3.090909    1.305997          1          5
          z2 |         22    3.181818    1.332251          1          5
```

Similarly, if you wanted to summarize `y1` for every value of `z1`, you could use:

```
. bysort z1: sum y1

-----------------------------------------------------------------------------
-> z1 = 1

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          y1 |         10    .3884885     1.35863  -2.098264   1.904308

-----------------------------------------------------------------------------
-> z1 = 2

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          y1 |         10   -.2408193    1.012609  -1.871239   1.295517

-----------------------------------------------------------------------------
-> z1 = 3

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          y1 |         11    .2876875    .8533628   -1.63913   1.388999

-----------------------------------------------------------------------------
-> z1 = 4

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          y1 |         12   -.3418385    .7974625  -1.720355   .5203478
```

```
-> z1 = 5

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          y1 |          7   -.2600435    1.179715  -2.152447   1.131644
```

The bysort command sorts the data in order of one or more categorical variables, then performs an action within the levels of those variables. If you wanted summaries within the levels defined by z1 and z2, you could use `bysort z1 z2: sum y1`.

Transforming data

In our data, the variable z1 takes the values 1-5. Suppose we wanted to to recode the values 1-3 to 0 and 4-5 to 1. There are several ways that we could do this using the replace command:

```
. replace z1 = 0 if z1==1 | z1==2 | z1==3
(31 real changes made)

. replace z1 = 1 if z1==4 | z1==5
(19 real changes made)
```

Alternatively, we can use inlist:

```
replace z1 = 0 if inlist(z1, 1, 2, 3)
replace z1 = 1 if inlist(z1, 4, 5)
```

or recode:

```
recode z1 (1 2 3 = 0) (4 5 = 1)
```

We can use the *indicator function*, which takes the value 1 if (expression) is true, and zero otherwise:

```
replace z1 = (z1==4 | z1==5)
```

Note. Our example data don't have any missing values, but if they did, this could have caused a problem, because the condition `(z1==4 | z1==5)` evaluates to zero when z1 is missing. We can avoid this problem by using inlist or recode, or by specifying that the observation is not missing (mi is short for missing):

```
replace z1 = (z1==4 | z1==5) if !mi(z1)
```

Stata stores missing values as large numbers, and refers to these values as "." (in fact, we can replace `!mi(z)` with `if z <= .`). Hence, a condition like `y1 >= 5` will also evaluate to true if y1 is missing, so to truncate y1 from above at 5, we should use one of

```
replace y1=5 if y1>=5 & !mi(y1)
replace y1=5*(y1>=5 & y1<.)
replace y1=5 if inrange(y1, 5, .)
```

We can use generate to define new variables:

```
. gen x1_sq = x1^2

. gen x1_exp = exp(x1)
```

The beauty of Stata loops is that it's easy to use them to define new objects, which makes this easy to automate (since we already defined x1_sq and x1_exp, Stata will give an error if we try to define them again, so we drop them first):

```
. foreach x of varlist x* {
  2.   capture drop `x'_sq `x'_exp
```

```
  3.   gen `x'_sq = `x'^2
  4.   gen `x'_exp = exp(`x')
  5. }
```

Merging and reshaping data

Suppose that we had several datasets, each containing data on the same variables for different years. We could easily stack them into one large dataset using `append`:

```
use data1, replace
forvalues i=2/5 {
  append using data`i'
}
```

Now suppose that in our original (unrecoded) data, `z1` and `z2` represent "state" and "year", and that we have another dataset containing additional variables for each state and year combination. How can we merge these datasets?

First, we'll create such a dataset, and illustrate a few things along the way:

```
(8 vars, 50 obs)

. frame create newdata

. frame change newdata

. set obs 25
Number of observations (_N) was 0, now 25.

. gen z1 = .
(25 missing values generated)

. replace z1 = ceil(_n/5)
(25 real changes made)

. gen z2 = 1

. replace z2 = z2[_n-1] + 1 if z1[_n]==z1[_n-1]
(20 real changes made)

. gen w = rnormal()

. save newdata.dta, replace
file newdata.dta saved

. frame change default
```

Frames are how Stata allows you to use multiple datasets at once. Here, we create a new one named `newdata`, then switch over to that frame. You can get a list of all frames using `frames dir`. See `help frames` for more. Instead of frames, we could have used

```
preserve
// commands to make new data
restore
```

to set our original data aside, make the new data, then reload the original data (these are older commands that don't require a recent version of Stata). The advantage of frames is that we can switch between the two frames (actually, in newer versions, `preserve` uses frames under the hood to set the old data aside).

Next, we set the observations in this blank frame to be 25, and initialize `z1` to missing. Then we use the `ceil` function to replace `z1` with the first integer greater than the observation number divided by five (`_n` is a built-in scalar that gives the observation number). This results in five ones, five twos, etc. Next, we initialize `z2` to one and replace it with the value of the previous observation plus one `z2[_n-1] + 1` for observations where the value of `z1` is the same for the current and previous observation (this way, the first observation for every value of `z1` remains 1).

Now we have every combination of z1 and z2, and we create a new normally distributed variable w, save the data, and change back to the default frame. If you want to explore this new data frame, you can switch back to that frame.

Now we can merge the new dataset to our existing one:

```
. merge m:1 z1 z2 using newdata
(variable z1 was byte, now float to accommodate using data's values)
(variable z2 was byte, now float to accommodate using data's values)
    Result                      Number of obs
    ─────────────────────────────────────────
    Not matched                             2
        from master                         0  (_merge==1)
        from using                          2  (_merge==2)

    Matched                                50  (_merge==3)
    ─────────────────────────────────────────
```

Here, merge m:1 stands for "many to one," since we are potentially matching many observations in the original dataset with the same values of z1 and z2 to the same observations in the using dataset. 1:1 means that one row in the original data matches with each row in the using data, and 1:m means that a row in the original data might be matched to many in the using data.

The results of the merge tell us that there were two combinations of z1 and z2 that didn't occur in our original data. The merge creates a new variable named _merge that indicates whether an observation was matched during the merge. We can use drop if _merge!=3 to eliminate observations that didn't have a match in both datasets. You will get an error if you try to do another merge without dropping this.

Now suppose that our x and y variables (x1, x2, etc.) refer to observations on the same variables for different units at different points in time (i.e., *panel data*). Many panel data estimators require the data to be in a *long form*, with one column per variable, and multiple rows per unit, each corresponding to a different time period. To put our data into long form, we can use reshape long.

```
. gen id = _n

. reshape long x y, i(id) j(year)
(j = 1 2 3)

Data                               Wide   ->   Long
─────────────────────────────────────────────────────────────────────────────
Number of observations               52   ->   156
Number of variables                  11   ->   8
j variable (3 values)                     ->   year
xij variables:
                               x1 x2 x3   ->   x
                               y1 y2 y3   ->   y
─────────────────────────────────────────────────────────────────────────────

. sum

    Variable │        Obs        Mean    Std. dev.       Min        Max
─────────────┼─────────────────────────────────────────────────────────
          id │        156        26.5    15.05667          1         52
        year │        156           2    .8191262          1          3
           x │        150    .0086975    .9643137  -2.395694   2.728213
           y │        150   -.0507002      .97855  -2.732989    2.49957
          z1 │        156    2.884615    1.343858          1          5
─────────────┼─────────────────────────────────────────────────────────
          z2 │        156    3.019231    1.398022          1          5
           w │        156   -.0633444    .8645373  -1.915596   2.385123
      _merge │        156    2.961538     .192927          2          3
```

Now we only have one x variable and one y variable, but three observations per id, corresponding to different values of the newly created year variable. Note that we had to generate an id variable in order to do this.

To go back to *wide* form, we can use `reshape wide` (however, Stata will give an error unless we drop the `_merge` variable first):

```
. drop _merge

. reshape wide x y, i(id) j(year)
(j = 1 2 3)

Data                               Long   ->   Wide
-----------------------------------------------------------------------------
Number of observations              156   ->   52
Number of variables                   7   ->   10
j variable (3 values)              year   ->   (dropped)
xij variables:
                                      x   ->   x1 x2 x3
                                      y   ->   y1 y2 y3
-----------------------------------------------------------------------------
```

Aggregating data

Suppose that we wanted to know the sample size, mean and standard deviation of our `y` variables within groups defined by `z1` and `z2`. We could type `bysort z1 z2: sum y*`, but what if we wanted it in a format that we could add to our original (or some other) dataset?

We could also do this using the collapse command:

```
. preserve

. collapse (mean) y1_mean=y1 y2_mean=y2 y3_mean=y3 ///
>   (sd) y1_sd = y1 y2_sd = y2 y3_sd = y3 ///
>   (count) y1_n = y1 y2_n = y2 y3_n = y3, by(z1 z2)

. list y1_mean in 1/5

     +-----------+
     |   y1_mean |
     |-----------|
  1. |         . |
  2. | -.2085993 |
  3. | -.3881125 |
  4. |  .6914923 |
  5. |   .641409 |
     +-----------+

. restore
```

If we wanted, we could save this and merge it with another dataset. In the above, `///` allows you to continue a command on the next line, and I used the `list` command to print the first five observations of `mean_y1`.

Here is a little loop-and-macro trick to avoid having to type all of these new variable names:

```
local mean (mean)
local sd (sd)
local n (count)
foreach x of varlist y* {
  local mean `mean' `x'_mean=`x'
  local sd `sd' `x'_sd=`x'
  local n `n' `x'_n=`x'
}
collapse `mean' `sd' `n'
```

The idea is that the macro `mean` is initially `(mean)`, but then we add `mean_y1=y1`, `mean_y2=y2`, etc. to it, and similarly for `sd` and `n`, so that `` collapse `mean' `sd` `n' `` ends up being the same as the command that we typed manually above.

If we are mainly interested in adding these aggregate statistics to our original dataset, we can also use the `egen` (*extended generate*) commands. For example, we can add the mean of `y1` within groups using

```
. egen y1_mean = mean(y1), by(z1 z2)
(2 missing values generated)
```

and similarly for the other variables and statistics (of course we could use loops and macros to automate the process). There are many useful `egen commands`. For example, `group` creates a variable indicating membership in groups defined by multiple categorical variables, and `total` gives the sum of a variable within groups. See `help egen` for more.

Analysis

Descriptive statistics

We've already seen how to use the `summarize` command, along with the `bysort` prefix, to get summaries of variables. You can also use the `detail` option with `summarize` to get more detailed statistics:

```
. sum y1, d
                              1 y
-------------------------------------------------------------
      Percentiles      Smallest
 1%    -2.152447      -2.152447
 5%    -1.871239      -2.098264
10%    -1.545414      -1.871239       Obs                  50
25%    -.8429845      -1.720355       Sum of wgt.          50

50%     .1392209                      Mean          -.0256222
                        Largest       Std. dev.      1.043069
75%      .560402       1.388999
90%      1.28302       1.681066       Variance       1.087994
95%     1.681066       1.839705       Skewness       -.258056
99%     1.904308       1.904308       Kurtosis       2.313607
```

You can also create one- and two-way frequency tables using the `table` command. In the second example below, `row` gives row percentages and `col` column percentages:

```
. tab z1
         z1 |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         11       21.15       21.15
          2 |         10       19.23       40.38
          3 |         12       23.08       63.46
          4 |         12       23.08       86.54
          5 |          7       13.46      100.00
------------+-----------------------------------
      Total |         52      100.00
. tab z1 z2, row col

+-------------------+
| Key               |
|-------------------|
|     frequency     |
|  row percentage   |
| column percentage |
+-------------------+

           |                          z2
        z1 |         1          2          3          4          5 |     Total
-----------+-------------------------------------------------------+----------
         1 |         1          2          1          4          3 |        11
           |      9.09      18.18       9.09      36.36      27.27 |    100.00
           |      9.09      25.00       9.09      30.77      33.33 |     21.15
```

2	3 30.00 27.27	1 10.00 12.50	4 40.00 36.36	1 10.00 7.69	1 10.00 11.11	10 100.00 19.23
3	2 16.67 18.18	2 16.67 25.00	1 8.33 9.09	4 33.33 30.77	3 25.00 33.33	12 100.00 23.08
4	3 25.00 27.27	1 8.33 12.50	4 33.33 36.36	3 25.00 23.08	1 8.33 11.11	12 100.00 23.08
5	2 28.57 18.18	2 28.57 25.00	1 14.29 9.09	1 14.29 7.69	1 14.29 11.11	7 100.00 13.46
Total	11 21.15 100.00	8 15.38 100.00	11 21.15 100.00	13 25.00 100.00	9 17.31 100.00	52 100.00 100.00

There are several ways to obtain formatted tables of descriptive statistics, including the `dtable`, `table` and `collect` commands in more recent versions of Stata and the `outreg2` and `tabstat` packages (see German Rodriguez's tutorial for a nice introduction to the `dtable` and `etable` approaches). I usually find that, unless they are very simple, descriptive and regression tables need manual editing, so I just try to get the statistics into Excel so that I can format them, then export them from there.[2]

Here's how you can use `outreg2` (which is simple and works with any version of Stata) to get basic descriptives (overall, and by group; we need to clear saved estimates or they'll be included in the table):

```
estimates clear
outreg2 using descriptives.txt, replace sum(log) keep(y1-y3)
bysort z1: outreg2 using descriptives.txt, sum(log) keep(y1-y3)
```

Visualizing data

Stata has more extensive graphing capabilities than can be done justice in a short tutorial. Here, we'll highlight some basic examples. See `help graph` for more, and remember that you can always use the graphical user interface to figure out the syntax.

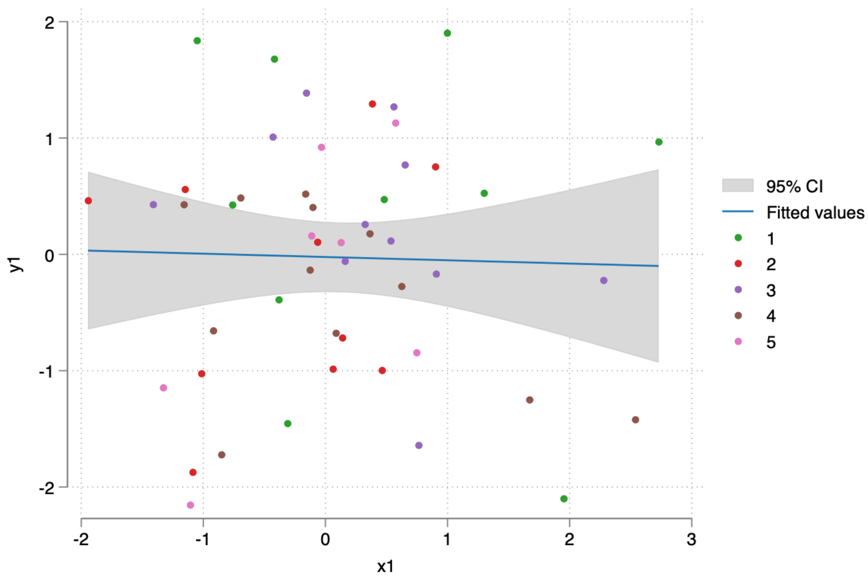
You can create a basic histogram using `hist y1` or a basic scatterplot using `scatter y1 x1`. There are many options for customizing your graphs (you can also type `help graph`, or use the graphical interface to explore them in case you forget the syntax).

Below are some more complicated examples that I adapted from *R for Data Science*. Here is a scatterplot of `y1` on `x1`, with the color of the points determined by `z1`, a linear fit with confidence intervals, with labeled axes:

```
. levelsof z1, local(z1)
1 2 3 4 5
. foreach x of local z1 {
  2.   gen y1_`x' = y1 if z1==`x'
  3.   label variable y1_`x' "`x'"
  4. }
(42 missing values generated)
(42 missing values generated)
```

[2]You can get tables from spreadsheets into Excel by pasting into LyX and exporting a LaTeX file, or by using the Excel2LaTeX plugin.

```
(41 missing values generated)
(40 missing values generated)
(45 missing values generated)
. twoway (lfitci y1 x1) (scatter y1_1-y1_5 x1), ///
>   ytitle("y1") xtitle("x1") ///
>   scheme(white_tableau)
. graph export scatter1.png, replace
file
    /Users/johngardner/Library/CloudStorage/Box-Box/OleMissTeaching/StatComp/NotesOnSt
    > ata/scatter1.png saved as PNG format
```



Scatterplot of `y1` against `x1`

Above, we used the `levelsof` command to save the levels of `z1` in a local macro, then created separate `y1` variables for each value of `z1` as a trick to customize the color of the points (it's possible to manually specify the shape and color of the points, but this approach changes the color automatically). We also used the user-written `white_tableau` scheme to style the graph. Normally, I'd save the graph as a PDF, but a PNG works better for the web.

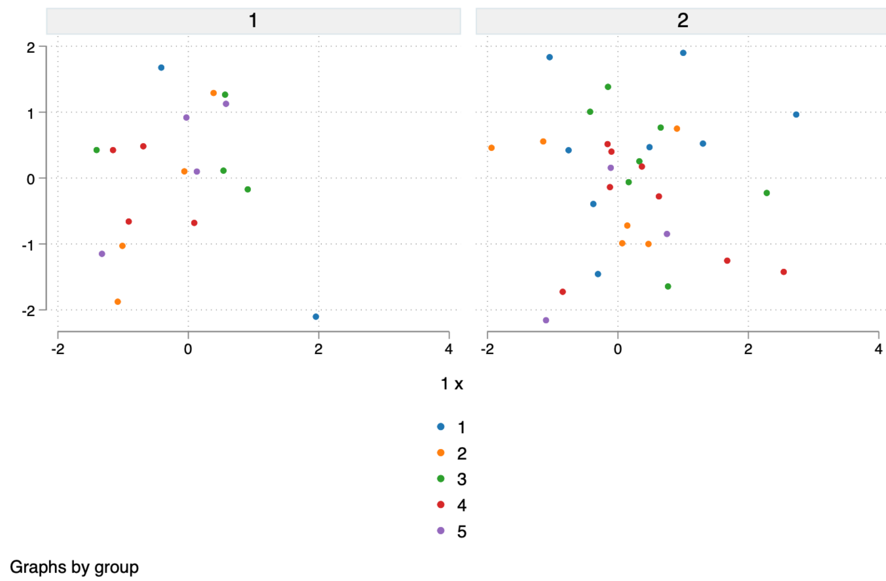
We can also create separate subplots for each level of a variable:

```
. gen group = 1*(z2<3) + 2*(z2>=3)
. tw (scatter y1_1-y1_5 x1), by(group) scheme(white_tableau)
. graph export scatter2.png, replace
file
    /Users/johngardner/Library/CloudStorage/Box-Box/OleMissTeaching/StatComp/NotesOnSt
    > ata/scatter2.png saved as PNG format
```

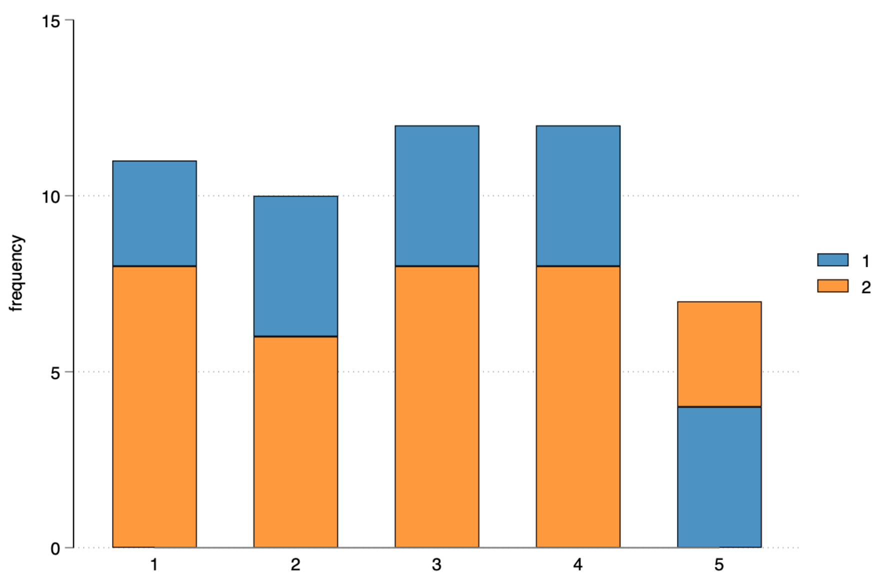
Here is a bar graph of `z1`, broken down by group:

```
. graph bar (count), over(group, sort(1) descending) over(z1) ///
>   stack asyvars scheme(white_tableau)
. graph export bar.png, replace
file
    /Users/johngardner/Library/CloudStorage/Box-Box/OleMissTeaching/StatComp/NotesOnSt
    > ata/bar.png saved as PNG format
```

Here, `over(group, sort(1) descending)` breaks the bars down by groups, sorts them in (descend-



Scatterplot of `y1` against `x1` by `group`



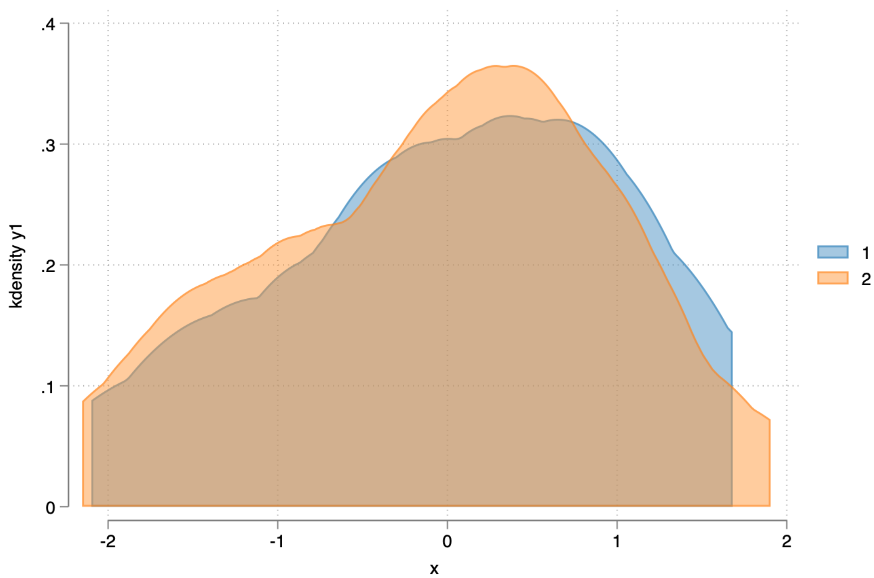
Bar graph of `z1` by `group`

ing) order of the first *yvar*, which is `group`. The second `over` command makes separate bars for each value of `z1`. The `stack` and `asyvars` options tells Stata to treat `group` as a `yvar`, but to `stack` the "sub-bars" instead of putting them side by side.

Finally, here are kernel density estimates of `y1` by `group`, using a `recast` trick to fill the graphs (I don't recall where I learned this trick):

```
. graph tw (kdensity y1 if group==1, recast(area) color(%50)) ///
>   (kdensity y1 if group==2, recast(area) color(%50)), ///
>   legend(order(1 "1" 2 "2")) scheme(white_tableau)

. graph export kdensity.png, replace
file
    /Users/johngardner/Library/CloudStorage/Box-Box/OleMissTeaching/StatComp/NotesOnSt
    > ata/kdensity.png saved as PNG format
```



Kernel densities of `y1` by `group`

If `group` had many values, we could automate this using a loop/macro trick similar to the one we used for collapse:

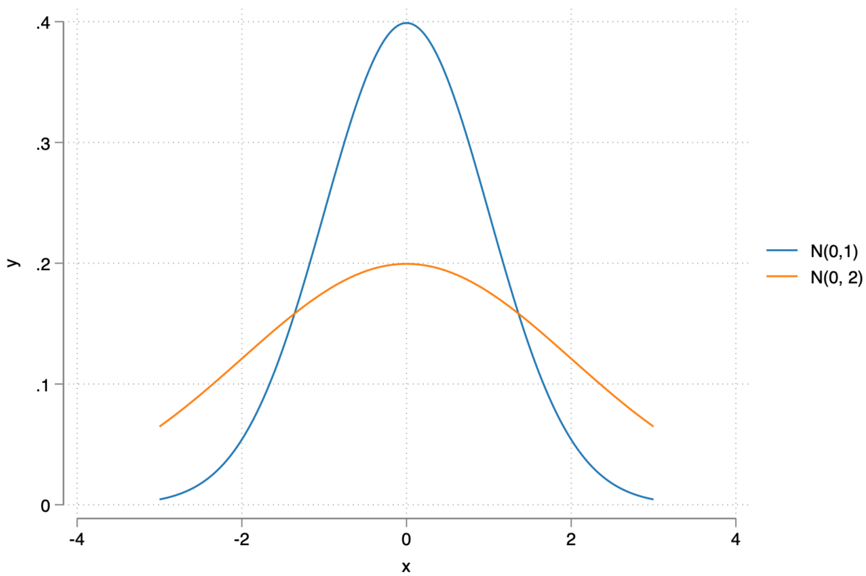
```
forvalues i=1/2 {
  local plottext  ///
    " `plottext' (kdensity y1 if group==`i', recast(area) color(%50)) "
  local legendtext `" `legendtext' `i' "`i'" "'
}
graph tw `plottext', legend(order(`legendtext')) ///
  scheme(white_tableau)
```

You can often use the `by` option to put multiple plots into the same image, as in the examples above. You can also use the `graph save graphname` and `graph combine` to save graphs in Stata's format in order to make a table of graphs (or go to `Graphics > Table of graphs`).

We can also use Stata to plot functions. As an example, here are the densities normal distributions with different variances:

```
twoway (function normalden(x, 1), range(-3 3)) ///
```

```
  (function normalden(x, 2), range(-3 3)),  ///
  scheme(white_tableau) ///
  legend(order(1 "N(0,1)" 2 "N(0, 2)"))
graph export densities.png, replace
```



Densities for two normal distributions

Regression basics

Stata has extensive faculties for regressions and other statistical and econometric estimators. Here, we will only touch on the essentials.

You can run a regression using the `regress` command:

```
. reg y1 x1 x2 x3
      Source |       SS           df       MS      Number of obs   =        50
-------------+----------------------------------   F(3, 46)        =      0.83
       Model |  2.73693139         3  .912310463   Prob > F        =    0.4843
    Residual |  50.5747722        46  1.09945157   R-squared       =    0.0513
-------------+----------------------------------   Adj R-squared   =   -0.0105
       Total |  53.3117036        49  1.08799395   Root MSE        =    1.0485

------------------------------------------------------------------------------
          y1 | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |  -.0664044    .150171    -0.44   0.660     -.368683    .2358742
          x2 |   .0154261    .157254     0.10   0.922    -.3011098     .331962
          x3 |  -.2594343   .1678957    -1.55   0.129    -.5973908    .0785221
       _cons |  -.0120578   .1498728    -0.08   0.936    -.3137361    .2896204
------------------------------------------------------------------------------
```

You can save the residuals using `predict resids_name, r` and predicted values using `predict prediction_name, xb`.

It's easy to get robust or clustered standard errors:

```
. reg y1 x*, robust
Linear regression                               Number of obs     =         50
                                                F(3, 46)          =       0.78
```

```
                                                Prob > F          =     0.5098
                                                R-squared         =     0.0513
                                                Root MSE          =     1.0485

------------------------------------------------------------------------------
             |               Robust
          y1 | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |  -.0664044    .164943    -0.40   0.689    -.3984175    .2656087
          x2 |   .0154261   .1379679     0.11   0.911     -.262289    .2931412
          x3 |  -.2594343   .1844303    -1.41   0.166    -.6306732    .1118046
       _cons |  -.0120578   .1523839    -0.08   0.937    -.3187907    .2946751
------------------------------------------------------------------------------

. reg y1 x*, vce(cluster z1)

Linear regression                               Number of obs     =         50
                                                F(3, 4)           =       1.58
                                                Prob > F          =     0.3257
                                                R-squared         =     0.0513
                                                Root MSE          =     1.0485

                                    (Std. err. adjusted for 5 clusters in z1)
------------------------------------------------------------------------------
             |               Robust
          y1 | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |  -.0664044   .1724878    -0.38   0.720    -.5453072    .4124985
          x2 |   .0154261   .0610512     0.25   0.813    -.1540791    .1849313
          x3 |  -.2594343   .1455596    -1.78   0.149    -.6635725    .1447039
       _cons |  -.0120578   .1846531    -0.07   0.951    -.5247369    .5006212
------------------------------------------------------------------------------
```

You can use `test` to perform hypothesis tests on the most recently estimated model. Stata stores the coefficients and standard errors as scalars labelled `_b[varname]` and _se[varname], which you can use with the `nlcom` command to test nonlinear hypotheses:

```
. reg y1 x*

      Source |       SS           df       MS      Number of obs   =        50
-------------+----------------------------------   F(3, 46)        =      0.83
       Model |  2.73693139         3  .912310463   Prob > F        =    0.4843
    Residual |  50.5747722        46  1.09945157   R-squared       =    0.0513
-------------+----------------------------------   Adj R-squared   =   -0.0105
       Total |  53.3117036        49  1.08799395   Root MSE        =    1.0485

------------------------------------------------------------------------------
          y1 | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |  -.0664044    .150171    -0.44   0.660     -.368683    .2358742
          x2 |   .0154261    .157254     0.10   0.922    -.3011098     .331962
          x3 |  -.2594343   .1678957    -1.55   0.129    -.5973908    .0785221
       _cons |  -.0120578   .1498728    -0.08   0.936    -.3137361    .2896204
------------------------------------------------------------------------------

. test x1=x2

 ( 1)  x1 - x2 = 0

       F(  1,    46) =    0.16
            Prob > F =    0.6871

. nlcom _b[x1]^2 - _b[x2]

       _nl_1: _b[x1]^2 - _b[x2]

------------------------------------------------------------------------------
          y1 | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
       _nl_1 |  -.0110166   .1612239    -0.07   0.946    -.3270096    .3049765
------------------------------------------------------------------------------
```

You can also store estimates to refer to them later:

```
. qui reg y1 x*

. estimates store model1

. // several commands later
```

```
. est replay model1
--------------------------------------------------------------------------------
Model model1
--------------------------------------------------------------------------------

      Source |       SS           df       MS      Number of obs   =        50
-------------+----------------------------------   F(3, 46)        =      0.83
       Model |  2.73693139         3  .912310463   Prob > F        =    0.4843
    Residual |  50.5747722        46  1.09945157   R-squared       =    0.0513
-------------+----------------------------------   Adj R-squared   =   -0.0105
       Total |  53.3117036        49  1.08799395   Root MSE        =    1.0485

------------------------------------------------------------------------------
          y1 | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |  -.0664044    .150171    -0.44   0.660     -.368683    .2358742
          x2 |   .0154261    .157254     0.10   0.922    -.3011098     .331962
          x3 |  -.2594343   .1678957    -1.55   0.129    -.5973908    .0785221
       _cons |  -.0120578   .1498728    -0.08   0.936    -.3137361    .2896204
------------------------------------------------------------------------------

. estimates restore model1
(results model1 are active now)

. test x1=x2

 ( 1)  x1 - x2 = 0

       F(  1,    46) =    0.16
            Prob > F =    0.6871
```

You can type `ereturn list` to see a list of all of the macros, scalars, and matrices that are stored after a regression or other estimation command:

```
. eret li

scalars:
               e(rank) =  4
               e(ll_0) =  -72.5502487097244
                 e(ll) =  -71.23267351705861
               e(r2_a) =  -.0105309582061381
                e(rss) =  50.57477224345187
                e(mss) =  2.736931388681448
               e(rmse) =  1.048547362072798
                 e(r2) =  .0513382841330131
                  e(F) =  .8297868567969495
               e(df_r) =  46
               e(df_m) =  3
                  e(N) =  50

macros:
            e(cmdline) : "regress y1 x*"
              e(title) : "Linear regression"
          e(marginsok) : "XB default"
                e(vce) : "ols"
             e(depvar) : "y1"
                e(cmd) : "regress"
         e(properties) : "b V"
            e(predict) : "regres_p"
              e(model) : "ols"
          e(estat_cmd) : "regress_estat"

matrices:
                  e(b) :  1 x 4
                  e(V) :  4 x 4
               e(beta) :  1 x 3

functions:
             e(sample)
```

Stata has useful features for specifying models. You can use `i.z1` to include indicators for the levels of `z1` as regressors, `i2.z1` to specify that the omitted category should be 2, and `ibn.z1` to specify that there should be no omitted category (in which case you want to suppress the constant using the `noconstant` option). You can use `#` to get the interaction between variables and `##` to get all two-way interactions. If one of these variables is continuous, you should prefix it by `c.`.

The following includes x1, its square, its interaction with x2 and x3, its interactions with indicators for the levels of z1 and z2, as well as those levels themselves:

```
. reg y1 c.x1##c.x2 c.x1#c.x1 c.x1#c.x3 c.x1#(i.z1 i.z2) i.z1 i.z2

      Source |       SS           df       MS      Number of obs   =        50
-------------+----------------------------------   F(21, 28)       =      0.63
       Model |  17.1054571        21  .814545578   Prob > F        =    0.8609
    Residual |  36.2062465        28  1.29308023   R-squared       =    0.3209
-------------+----------------------------------   Adj R-squared   =   -0.1885
       Total |  53.3117036        49  1.08799395   Root MSE        =    1.1371

------------------------------------------------------------------------------
          y1 | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |   .5119097   .7865183     0.65   0.520      -1.0992    2.123019
          x2 |  -.0607202   .1949629    -0.31   0.758    -.4600835    .3386431
             |
   c.x1#c.x2 |  -.1240336   .2620448    -0.47   0.640    -.6608081    .4127408
             |
   c.x1#c.x1 |  -.0610364   .1938742    -0.31   0.755    -.4581696    .3360968
             |
   c.x1#c.x3 |   .1742655   .3160815     0.55   0.586     -.473198     .821729
             |
     z1#c.x1 |
          2  |   .2831985   .7956912     0.36   0.725    -1.346701    1.913098
          3  |  -.4193324   .6628734    -0.63   0.532    -1.777167    .9385023
          4  |  -.5098121   .6056266    -0.84   0.407    -1.750382    .7307577
          5  |   1.111535   .8050032     1.38   0.178    -.5374398    2.760509
             |
     z2#c.x1 |
          2  |  -.9379134   .8095513    -1.16   0.256    -2.596204    .7203772
          3  |  -.9040705   .7688583    -1.18   0.250    -2.479005    .6708643
          4  |   -.341652   .7850846    -0.44   0.667    -1.949825    1.266521
          5  |  -.3200915   .9607607    -0.33   0.741    -2.288121    1.647938
             |
          z1 |
          2  |  -.8286218    .613738    -1.35   0.188    -2.085807    .4285634
          3  |  -.2705119   .5653038    -0.48   0.636    -1.428484    .8874603
          4  |  -.8410675   .5970149    -1.41   0.170    -2.063997     .381862
          5  |  -.6799663   .6402059    -1.06   0.297    -1.991369    .6314361
             |
          z2 |
          2  |  -.4950503   .6344122    -0.78   0.442    -1.794585    .8044843
          3  |  -.3736093   .5497722    -0.68   0.502    -1.499767     .752548
          4  |  -.1426655   .5691045    -0.25   0.804    -1.308423    1.023092
          5  |  -.6368414   .6884577    -0.93   0.363    -2.047083    .7734003
             |
       _cons |   .9144687   .6497817     1.41   0.170    -.4165487    2.245486
------------------------------------------------------------------------------
```

We can use the margins command to find the marginal effect of x1 after all of those terms:

```
. margins, dydx(x1)

Average marginal effects                                  Number of obs = 50
Model VCE: OLS

Expression: Linear prediction, predict()
dy/dx wrt:  x1

------------------------------------------------------------------------------
             |            Delta-method
             |      dy/dx   std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |   .0346223   .2021769     0.17   0.865    -.3795183    .4487629
------------------------------------------------------------------------------
```

To estimate a probit and the average marginal effects of its regressors, we can use

```
. gen y_bin = (y1 >= 0)

. probit y_bin x*

Iteration 0:   log likelihood =  -34.29649
```

```
Iteration 1:   log likelihood = -33.507899
Iteration 2:   log likelihood = -33.507377
Iteration 3:   log likelihood = -33.507377

Probit regression                                       Number of obs =     50
                                                        LR chi2(3)    =   1.58
                                                        Prob > chi2   = 0.6643
Log likelihood = -33.507377                             Pseudo R2     = 0.0230

------------------------------------------------------------------------------
       y_bin | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |  -.1689203   .1844929    -0.92   0.360    -.5305196    .1926791
          x2 |    .033874   .1931569     0.18   0.861    -.3447066    .4124546
          x3 |  -.1891183   .2024053    -0.93   0.350    -.5858253    .2075888
       _cons |   .1793668   .1821691     0.98   0.325    -.1776781    .5364117
------------------------------------------------------------------------------

. margins, dydx(*)

Average marginal effects                                    Number of obs = 50
Model VCE: OIM

Expression: Pr(y_bin), predict()
dy/dx wrt:  x1 x2 x3

------------------------------------------------------------------------------
             |            Delta-method
             |      dy/dx   std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
          x1 |  -.0649794   .0691144    -0.94   0.347    -.2004411    .0704823
          x2 |   .0130305    .074224     0.18   0.861    -.1324458    .1585068
          x3 |  -.0727491    .075812    -0.96   0.337    -.2213379    .0758398
------------------------------------------------------------------------------
```

There are many ways to export a table of regressions, including Stata's built-in `etable` and `collect` commands and third-party packages like `outreg2` and `estout`. The example below shows how to use `outreg2` (which works in any version of Stata) to export a table of regressions as a CSV file (you can also export to LaTeX and Word; note that we only use the `replace` option after the first model to make sure that additional models are added to the table):

```
qui reg y1 x1
outreg2 using results.txt, replace
qui reg y1 x1 x2
outreg2 using results.txt
qui reg y1 x*
outreg2 using results.txt
```

The result will look similar to the results from Stata's `etable` command:

```
. qui reg y1 x1

. est sto m1

. qui reg y1 x1 x2

. est sto m2

. qui reg y1 x*

. est sto m3

. etable, estimates(m1 m2 m3) mstat(r2)

-------------------------------------
                 y1      y1      y1
-------------------------------------
1 x          -0.028  -0.024  -0.066
            (0.147) (0.150) (0.150)
2 x                   0.040   0.015
                    (0.159) (0.157)
3 x                          -0.259
                            (0.168)
Intercept    -0.023  -0.019  -0.012
            (0.150) (0.152) (0.150)
R-squared      0.00    0.00    0.05
-------------------------------------
```

We can also loop over different dependent variables (below, we use a loop/macro trick to make sure we only use the `replace` option once by redefining the macro `replace` to be empty after the first model):

```
local replace replace
foreach y of varlist y1 y2 y3 {
  qui reg `y' x*
  outreg2 using results2.txt, `replace`
  local replace
}
```

Some other useful regression commands that you should be aware of are:

- `ivregress` and the `ivreg2` package for instrumental variables and two-stage least squares
- `xtreg` for panel data models including fixed and random effects
- `areg` for fixed effects models
- The `reghdfe` package for models with lots of fixed effects
- The `coefplot` package for plotting model coefficients

Matrix algebra basics*

There are two ways to work with matrices in Stata: using the original matrix commands, or using the matrix commands that come with Stata's programming language, Mata. If you are going to do advanced computational work in Stata, you should probably learn about Mata. Here we'll only briefly touch on the original commands. See `help matrix` for more detail.

Stata matrix commands have to be prefaced by `matrix`. You can create and view a matrix using

```
. mat A = (1, 2 \ 3, 4)

. mat list A

A[2,2]
    c1  c2
r1   1   2
r2   3   4
```

You can subset a matrix using

```
. mat B = A[1, 1..2]

. mat C = A[1..., 2]

. mat li B

B[1,2]
    c1  c2
r1   1   2

. mat li C

C[2,1]
    c2
r1   2
r2   4
```

You can get the transpose and inverse of a matrix using:

```
. mat A_t = A'

. mat A_inv = inv(A)

. mat li A_t

A_t[2,2]
    r1  r2
```

```
c1   1   3
c2   2   4

. mat li A_inv

A_inv[2,2]
     r1   r2
c1   -2    1
c2  1.5  -.5
```

Often, you want to create a matrix from variables in a dataset. You can do this using the `mkmat` command (we use the variable `one` to add a constant at the end):

```
. drop if mi(y1) // drop missing values introduced from merge
(2 observations deleted)

. gen one = 1

. mkmat x* one, matrix(X)
```

Stata has commands to efficiently create matrix products. To obtain $X'X$ or $y_1'X$ we can use the following (by default, these will also include a constant term, which can be suppressed using the `noconstant` option; there are also commands for weighted cross products, see `help matrix accum`):

```
. matrix accum XX = x*
(obs=50)

. matrix vecaccum y1X = y1 x*
```

You can also multiply and add matrices. For example, we could regress `y1` on `x1-x3` using

```
. mat beta_hat = inv(XX) * y1X'

. mat li beta_hat

beta_hat[4,1]
               y1
   x1  -.06640435
   x2   .01542611
   x3  -.25943432
_cons  -.01205783
```

and obtain the (default) standard errors as

```
. mkmat y1, matrix(y1)

. mat e = y1 - X*beta_hat

. local n: rowsof y1

. local k: colsof X

. mat vcov = inv(XX) * (e' * e)/(`n' - `k')

. mat li vcov

symmetric vcov[4,4]
               x1          x2          x3       _cons
   x1   .02255134
   x2   .00326241   .02472883
   x3   .00463959   .00263781   .02818895
_cons  -.00222657   .00211105  -.00075217   .02246185
```

We can compare this to the variance matrix from the `regress` command:

```
. qui reg y1 x*

. mat li e(V)

symmetric e(V)[4,4]
               x1          x2          x3       _cons
   x1   .02255134
   x2   .00326241   .02472883
   x3   .00463959   .00263781   .02818895
_cons  -.00222657   .00211105  -.00075217   .02246185
```

Resources for more

- German Rodriguez's Stata tutorial is a great place to learn more about Stata (and how I originally learned)[3]
- Kit Baum's A little Stata programming goes a long way is a nice introduction to basic Stata programming
- *An introduction to Stata programming* by Baum is a more comprehensive resource on Stata programming
- The official Stata documentation is available at https://www.stata.com/features/documentation/, and also comes bundled with Stata

[3] I also created this document using his Markstat package.