Stata Introduction

Stefano Lombardi*

August 9, 2023

Note

This introduction is a crash course in how to work in a structured fashion and write efficient code. It is intended to get you started on working in Stata, and for those of you who are already familiar with the software, it will hopefully provide a useful summary of the main ingredients necessary to successfully code.

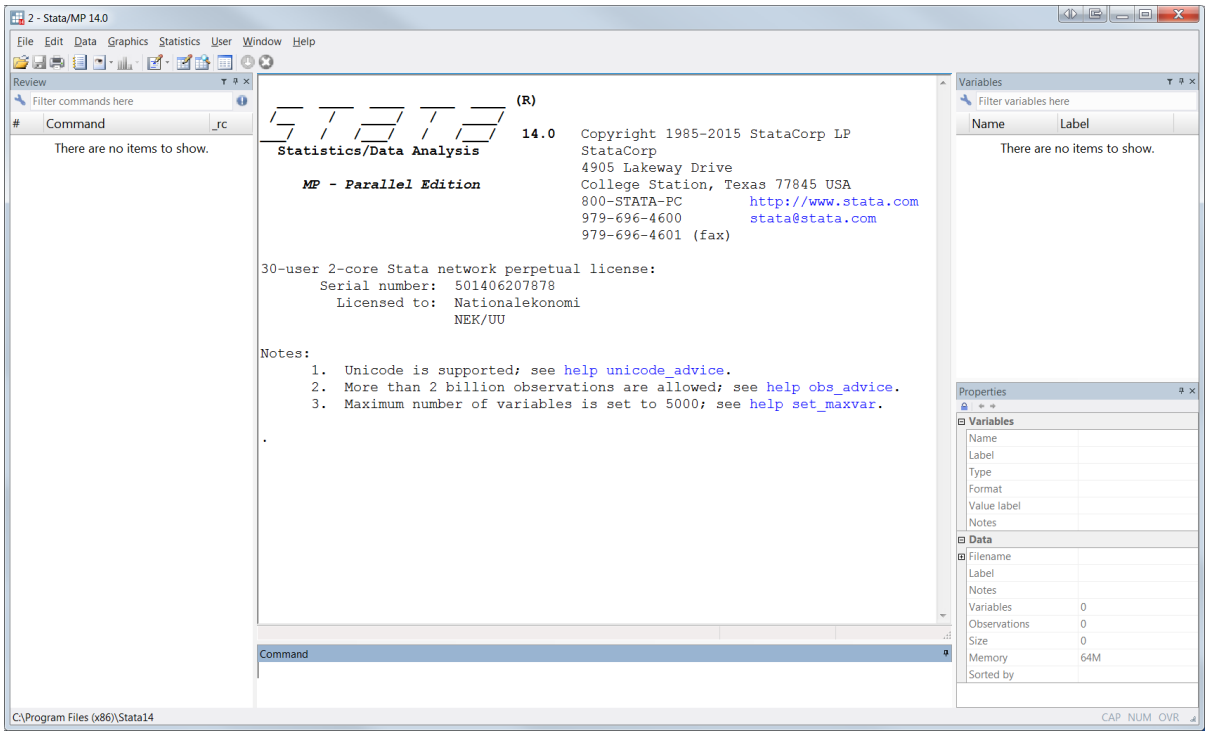
*VATT Institute for Economic Research. E-mail: stefano.lombardi@vatt.fi.

Contents

1 Introduction

1.1 Stata Interface

The first thing you encounter when opening Stata is the interface:



The main windows are:

1. the *command window* (by default positioned at the bottom of the Stata window);
2. *results window* (centre);
3. *review window* (left);
4. *variables window* (top-right);
5. *properties window* (bottom-right).

Command window

Commands are *interactively* submitted to Stata from this window (by default positioned at the bottom of the Stata window). The command window should be used **only to perform basic tasks**, like inspecting the data or, as in the following example, for using Stata as a calculator:

```
. display 2+2
4
```

The first line contains the code typed in the command window preceded by a dot (`.`). The second line shows the result after Stata interprets the `display` command typed by the user.

Type *Page Up*/*Page Down* keyboard buttons to step through the command history.

Results window

It contains all the commands and their respective results. You can use *CTRL* + *F* to search into the results window. The `cls` command cleans the screen.

Review window

It shows the history of submitted commands and displays successful commands in black and unsuccessful in red. To enter a command you can:

- Click on it once to copy it to the command window, replacing the contents you had there.
- Double-click on it to resubmit it.

Variables window

You can do different things with the variables from this window:

- Double-click a variable to send it to the command window.
- Use *SHIFT* and *CTRL* to select different variables and either (1) *drag* them into the command window or (2) *right click* on the selection to interactively drop/keep the variables selected or perform other operations.

Note that reordering the display of the variables in the Variables window does *not* affect the order of the variables in the dataset itself.

Properties window

It displays properties of variables and datasets. By default, changes to these properties are not allowed (to toggle this ability, click the lock icon in the titlebar).

1.2 Getting Help

Whether you already are an experienced Stata user or a beginner, there is absolutely no chance that you remember by heart all the commands, options, sub-options (and sub-sub-options!) relevant for your research. Fortunately Stata provides the user with a very good help command.

Just type `help` *`command name`* to access the help window of an existing command, or `findit` *`word(s)`* to search for information across all available sources (including system help, FAQs and the *Stata Journal*.

1. getting help

```
* Help for using summary command
help summary

* Help for performing Kolmogorov-Smirnov test
findit kolmogorov test distribution
```

Note that the above box contains proper Stata code, i.e. it can be entirely copy-pasted line by line in the command window (or in a do-file). Note also that we are making use of specific Stata syntax for writing comments. We will come back to do-files and comments in section 2 – *Working with do-files*.

For now, keep in mind that *every single Stata command, operator, option, syntax rule* can be searched into the Stata documentation. I encourage you to use Stata help as much as possible! For more information on how to get help in Stata, type `help help_advice`.

1.3 Basics of Stata syntax

The help command is a good way to start speaking about Stata syntax. Type `help summarize` to open the help window relative to the `summarize` command. The way the help window for this specific command is structured mirrors the general Stata syntax. What we see under **Syntax** is:

```
summarize [varlist] [if] [in] [weight] [, options]
```

The main ingredient of Stata syntax are:

- The command is in **bold**; the **underlined** letters are sufficient to use the command: Stata interprets `summar`, `summ` and `su` exactly in the same way (however, `s` will return error).
- Everything in brackets is optional, it does not have to be specified. The following optional arguments are shared by nearly all Stata estimation and data analysis commands:
  - `[varlist]`: list of variables separated by a space.
  - `[if]`: conditional statement relative to the values of a variable. It basically tells Stata to use only the data specified.[1]
  - `[in]`: tells Stata to use only the observations specified.[2]
  - `[weight]`: tells Stata how to deal with weighted data.

  Practically speaking, you will **routinely** use *if* statements, while you will seldom use *in* and *weight* statements.
- `[, options]`: command options specific to the main command. The set of the specified command options **always** follows a comma; options are separated each other by a space.

Note that command options can allow the specification of sub-options, but the syntax of built-in Stata commands follows the above general structure.

This is far from being everything you need to know about Stata syntax, but it is a necessary starting point to be able to master the use of the help editor and of the software itself. At the beginning of section 3 we will introduce *logical operators*, while more advanced topics concerning *macro variables* and *loops* will be introduced later in section 5.

[1] For instance, you can use it to summarize *weight* only if *age* is smaller than 4. In section 3.1 we will see the actual logical operators to perform such tasks.

[2] You can use it to summarize only the first 100 observations.

1.4 Setting up the work environment

Before proceeding further, let us structure the work environment by using different folders where to store raw and processed data and output.

First, go to your Desktop and create a folder using the first three letters of your name and the first two of your surname (*stelo* in my case). Then create the following 7 folders in your personal directory:

folder	**description**
do	put here all the relevant do-files.
raw	contains the raw data. **NEVER** alter your raw data.
use	contains the data cleaned and prepared for the analysis.
temp	contains *intermediary* files that are used in your code.
outreg	contains your *final* regression output and tables.
graphs	contains your *final* plots and figures.
log	contains all your logs. It is good practice to give them the same name as the do-files they are associated with.

You can rename and put your personal directory wherever you want. As you will see in the next examples, I keep it in my Dropbox folder, but you can leave it on your Desktop if you prefer.

Second, use `cd` to set your personal directory to be the Stata working directory. You can check what your current working directory is by typing `pwd`. By typing `dir` you should see the 7 sub-directories listed. Remember, you *must* use **forward slashes** if you are a Mac user.[3]

2. setting the work environment

```
// Default working directory
pwd

// Set new working directory and check sub-directories
cd "C:/Users/Stefano/Dropbox/stelo"
pwd
dir
```

Finally, put into the *raw* folder the *dta* file that you find in the compressed folder that also contains the current tutorial.

Important: *From now on I will assume that you set up the folders structure as explained above, that you have modified your Stata working directory accordingly, and that you put the data in the* raw *sub-folder.*

[3] In practice, always use forward slashes to improve portability of your scripts.

2 Working with do-files

In virtually any application you will want to **automatize** as much as possible your data management and data analysis operations. This means that you are seldom going to use Stata interactively (by typing in the command window). Accordingly, we will now see how to work in a **structured manner** using do-files.

A **do-file** is a text file that contains a list of Stata commands (i.e. a script), and is produced in a built-in text editor called the **do-file editor**. With few exceptions, the syntax used in a do-file is exactly the same that you would use in the command window. However, there are two huge advantages to work with do-files:

- It allows you to build up series of commands that can be submitted to Stata either **all at once** or **in chunks**.
- It allows you to structure your work in a way that is essential for its **reproducibility**.

2.1 A do-file template

To open a new do-file type `edit`. With few modifications to fit your needs, you can use the following template for your do-files:

3. do-file template

```
// Do-file template

clear all
set more off

capture log close
cd "C:/Users/Stefano/Dropbox/stelo"
log using "log/log_template", replace

/* ------ write your commands here ------ */

log close
```

Copy-paste the above code in a new do-file, save it and use it as a template for your future works. In a nutshell, the commands do the following:

Stata command	**description**
`clear all`	remove and close everything in the current session.
`set more off`	do not ask to continue when running a long script.
`capture log close`	close any already opened log file (if existing).
`cd "working directory path"`	define working directory.
`log using "log\log name", replace`	save a log recording all commands and results of the entire session **in the log sub-folder**.

2.2 Running do-files

The easiest way to run a do-file is to:

1. Select the line(s) of code that you want to submit to Stata.
2. Press *CTRL + D*.

You can run the entire do-file by pressing *CTRL + A* (select all) and *CTRL + D* in sequence.

2.3 Inserting comments in do-files

Do-file comments are words that are not processed as commands by Stata. They are displayed in *green* in the do-file editor. Inserting comments in essential to improve both the understanding and the readability of the code you are writing, whether you are sharing it with someone or not.

There are three ways to include comments in your do-file:

- All words *following* the `//` operator are commented out.
  - This works also if `//` is inserted *after* commands, see *Example 1* below.
  - This operator **cannot** be used interactively in the command window.
  - It only comments words on the same line where `//` is inserted.
- All words *following* the `*` operator are comments.
  - This commenting operator **cannot** be inserted after commands, see *Example 2.*
  - It **can** be used interactively in the command window.
  - As `//`, it only comments words on the same line.
- Words *included* in `/* */` are commented out. This operator can be inserted at the beginning, at the end or in the middle of a command line, and it can span across different lines. Remember to put `*/`, otherwise *all* the commands following `/*` will be commented out!

The following examples show how to include comments in Stata do-files.

4. comments in do-files

```
/* Example 1 */
// Stata can be used as a calculator:
display 2+2 // --> this is a valid comment!
display /* this works as well! */ 2+2

/*
note that
you can also span comments
across different lines
*/

/* Example 2 */
* Stata can be used as a calculator:
display 2+2 * --> NOT a valid comment! <--
```

2.4 Increasing the readability of do-files

2.4.1 Using comments to create code sections

When opened in the editor, the do-file code is automatically syntax highlighted ("coloured") according to the type of elements inserted in the script (e.g. strings are *red*). However, there is a lot you can do to improve code readability by making use of commented text:

- Always put the part of code that the user has to modify *at the very beginning* of your do-file – e.g. the `cd` command, the names of the files to be loaded, the number fo simulations to be performed by your code, etc – and in a separate section.
- Separate sections, subsections and sub-subsections by using easily recognizable titles, e.g.:

```
/*-------------- I. DATA MANAGEMENT --------------*/

// In the section we perform this and that operations
// What we get in the end is... further used in...
```

- Use lines and boxes to help the reader to go through the code:

```
*-----------------------------*
*   This is a beautiful box   *
*-----------------------------*

* This is an easily recognizable title
*=====================================

* This is a subtitle
*-------------------
```

- **Always** comment what you are doing: it saves (lots of) time when checking the code. However, do not exaggerate with the comments either! If later on you modify parts of your code, some comments might become redundant if not patently wrong since they refer to commands that do not exist anymore.

2.4.2 Breaking script lines

Consider this:

```
display "this is a long string (... a lot more words ...) to be displayed"
```

Very often the commands you write in your do-files will become awfully long. You can break strings using *three* dashes: `///`. The above could be written as:

5. breaking script lines (1)

```
display "this is " ///
"a long string " ///
"(... a lot more words ...) " ///  here you can avtually write comments if you want
"to be displayed"
```

Alternatively, you can also use `#delimit`. Remember to "close" the command when you are done breaking the line. Also, note the different use of the quotes (") as compared to above:

6. breaking script lines (2)

```
#delimit ;

display "this is
a long string
(... a lot more words ...)
to be displayed" ;

#delimit cr
```

Mind that only `#delimit` can be used interactively (in the command window).

Breaking code into different lines becomes crucial when your command has (sometimes dozens) of options and sub-options, e.g. when plotting graphs.

3 Data Management

Stata is good for performing data management tasks, also (or better, especially) thanks to the use of macro variables.[4] In what follows we show how to import, modify, save and export data in Stata. We will focus on the commands *per se*, not on how to perform the above mentioned operations efficiently. We will explore efficiency in section 5 – *Macro Variables and Loops*.

3.1 Logical operators, *varlists* and system variables

Every language syntax needs operators to perform logical comparisons. In the following table you find the main **logical operators** used by Stata. The way they are used will become clear in the next subsections.

operator	**meaning**
`a == b`	equality
`a != b`	inequality
`a >= b`	greater or equal to
`a > b`	greater than
`a & b`	and
`a | b`	or

A **varlist** is simply a list of variables names (separated by a space). Clearly, lists of variables can get very large (imagine if you have to refer to dozens of dummy variables). Two Stata *wildcard characters* can help you in such cases:

- The `*` character matches one or more characters in the name specified. Examples include:
  - `var*` (`*var`) : list composed by all variables with name *starting* (*ending*) with `var`.
  - `my*var` : all variables starting with `my`, ending with `var` and with any number of characters in between.
- The `-` character returns variables starting/ending with the left-/right-hand characters.
  - `var1-var5` : returns `var1`, `var2`, ..., `var5`.

Whenever you need to refer to observations (i.e. indexing rows) the importance of two so-called **system variables** cannot be overstated. They are `_n` and `_N` (note the underscores):

system variable	**meaning**
`_n`	observation (row) number
`_N`	last row number

If used with `by`, they respectively become running counter and group size within each `by`-defined group (more on this later).

[4] For complex data analysis steps consider using *matrix-* and *object-oriented* languages, like R or Matlab. They are way more flexible than Mata (the Stata matrix language).

3.2 Loading and saving data

Imagine you want to do the following:

1. Load a Stata dataset (i.e. in *dta* format).
2. Perform some operations and save the data in *dta*, *csv* and *xlsx* formats.
3. Import the *csv* and *xlsx* data created at the previous point.

The following do-file performs the above operations. As a general rule, note the use of the `clear` and `replace` options whenever data are imported and exported, respectively. For a detailed explanation of the commands used, see the next subsections.

7. load and save data

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

// 1. Load Stata data

use "raw/CPS_2012_micro_small.dta", clear
/* ---- some data operations here ---- */

// 2. Export data in Stata, Excel and csv formats

* Stata format (dta)
save "temp/stata_outdata.dta", replace

* Comma separated (csv) and Excel (xls) formats
export delimited using "temp/csv_outdata.csv", delimiter(",") replace
export excel using "temp/excel_outdata.xls", replace

// 3. Import csv and xls data

import delimited "temp/csv_outdata.csv", delimiter(",") clear
import excel "temp/excel_outdata.xls", sheet("Sheet1") firstrow clear
```

3.2.1 use, save and alternative use commands

The `use` command allows to load **dta** datasets (the Stata-specific format for data). Note that if a dataset has already been loaded and you try to load a new one, Stata will return error. You avoid this by using the `clear` option. Instead, to save **dta** files use `save`. The `replace` option makes sure you overwrite the file contents.

Stata comes with some example datasets automatically installed, e.g. the ever-present `auto.dta`. Instead of using `use`, type `sysuse` *`filename`* `[,` *`clear`*`]` to access them. To know which example data you can load, type `sysuse dir`. Alternatively, online example datasets can be loaded using `webuse` *`filename`* `[,` *`clear`*`]`.

3.2.2 `import` and `export`

Stata can work with files in many formats (e.g. *xlsx*, *txt*, *csv*, etc.), check `help import`.

To save a Stata dataset as a comma-separated or an Excel file you need to use `export delimited` and `export excel`, respectively. In the first case, if the delimiter type is not specified in the options, it is assumed to be a comma.

The `delimited` option is also used when adopting `import delimited`.[5] To import excel files with `import excel`, instead, you need to specify the name of the sheet containing the data (using the `sheet` option), and whether the first row of the Excel sheet contains variable names.

3.3 Inspecting and describing data

After loading data in Stata, there are some typical preliminary steps you will routinely perform. First, to check which variables are present in your dataset, and what their format is. Second, to produce informative descriptive statistics. Finally, graphical representing of the data.

We cover some of the main aspects related to these topics in the next sub-sections.

3.3.1 `browse` and `tabulate`

These two commands can be used interactively from the command window to inspect data.

If you want to visualize the data you just loaded, you can simply type `browse`.[6] This opens a spreadsheet-style **data editor** and already gives you a general an idea of how your data looks like (e.g. string variables are coloured in red, labels in blue, etc.). This command becomes even more useful when you use the *varlist* and *if* optional arguments to inspect a **subset** of the data.

The `tabulate oneway` and `tabulate twoway` commands produce one-way and two-way table of frequencies, respectively. In the next example we use the latter after browsing the data.

8. browse and tabulate

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"
use "raw/CPS_2012_micro_small.dta", clear

// Browse all data
browse

// Browse lnwage and age for females with log wage>2.5
browse lnwage age if sex==2 & lnwage>2.5

// Two-way frequencies table for sex and state
tabulate statefip sex
```

[5] The `import delimited` command has superseded `insheet`, so use the former.

[6] Altenatively, `list` can be used. The main difference is that it prints the information on the screen.

3.3.2 summary statistics, missing values and duplicate observations

The `describe` command provides a compact overview of some of all of the variables loaded in memory, their type and their **labels** (which, if well used, can be extremely useful).[7]

In order to compute the **descriptive statistics** of interest we can use either `summarize`, `tabstat`, or `codebook`. The latter also provides information on **missing values**.

Finally, `duplicates` manages **duplicate data rows**, including tagging and dropping them.

9. descriptive statistics

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

// Load Stata data
use "raw/CPS_2012_micro_small.dta", clear

// Describe all and part of the dataset variables
describe
describe lnwage age sex

// Descriptive statistics of 3 variables for men
codebook age yrseduc lnwage if sex==1
summarize age yrseduc lnwage if sex==1
summarize age yrseduc lnwage if sex==1, detail
tabstat age yrseduc lnwage if sex==1, stat(mean sd p25 p50 p75) col(s)
```

3.3.3 graphical methods

Stata offers excellent graphics capability that we cannot cover extensively. Let us focus on three types of graphs that are very useful to inspect data:[8] `histogram`, `scatter` and `line`.

Note that, for convenience, in the next examples graphs are exported in pdf format, but many other file extensions are available.

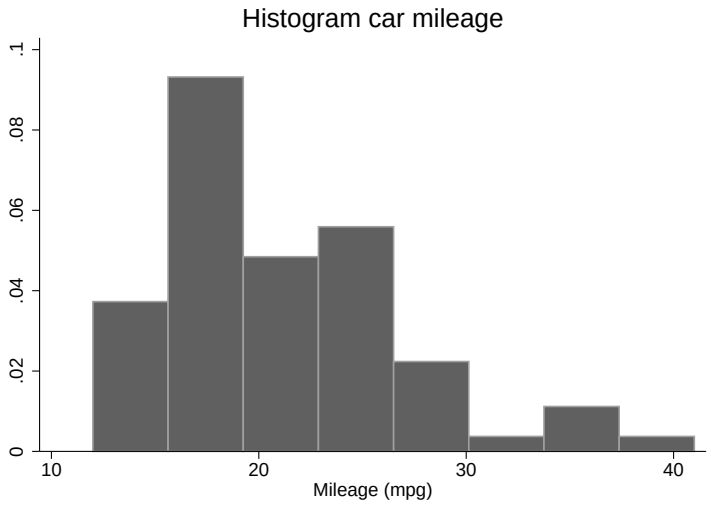
10. histogram

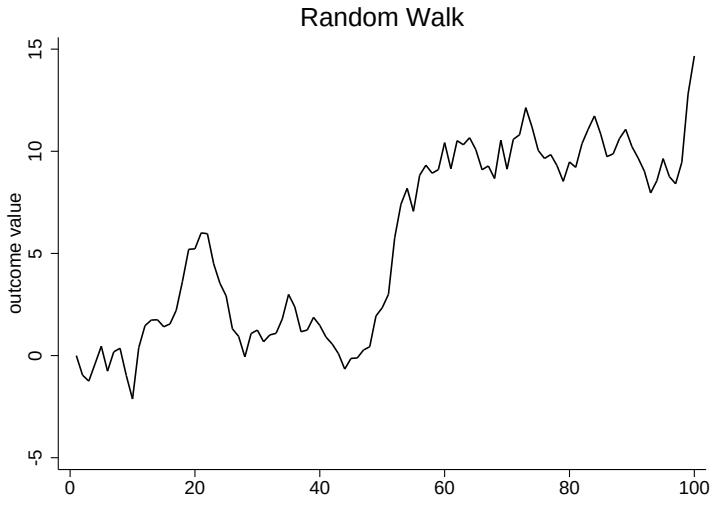
```
cd "C:/Users/Stefano/Dropbox/stelo"
sysuse "auto", clear

// histogram
hist mpg, title("Histogram car mileage", color(black)) ylabel(, nogrid) /// no grid
   graphregion(color(white) lwidth(medium)) /// white background
   lcolor(gs10) fcolor(gs6) ytitle("") // color; no y-axis title
graph export "graphs/hist.pdf", replace
```

[7] See `help label`, and in particular `help label define` and `help label variable`.
[8] For additional information, you can start checking here and here.

11. line plot

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

// Random walk creation
set seed 12345
set obs 100
gen x=_n
gen y=0 if _n==1
replace y=y[_n-1] + rnormal() if _n>=2

// Line plot
line y x, title("Random Walk", color(black)) lcolor(black) lwidth(medium) ///
   graphregion(color(white) lwidth(thick)) /// white background
   ylabel(, nogrid) xtitle("") ytitle("outcome value")
graph export "graphs/line.pdf", replace
```

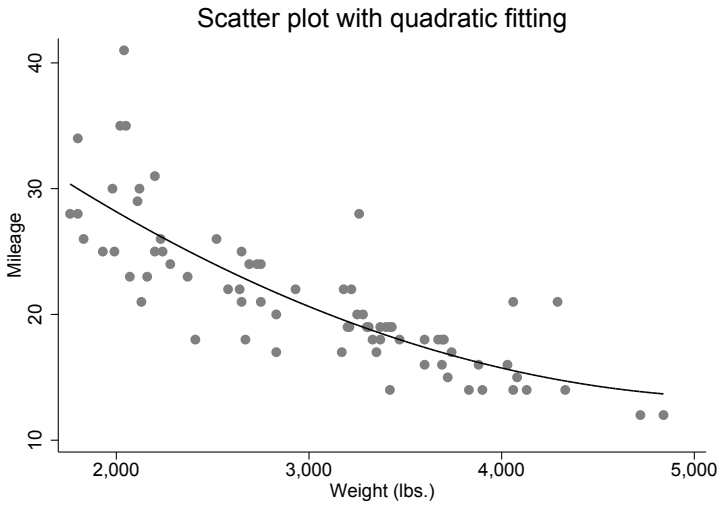


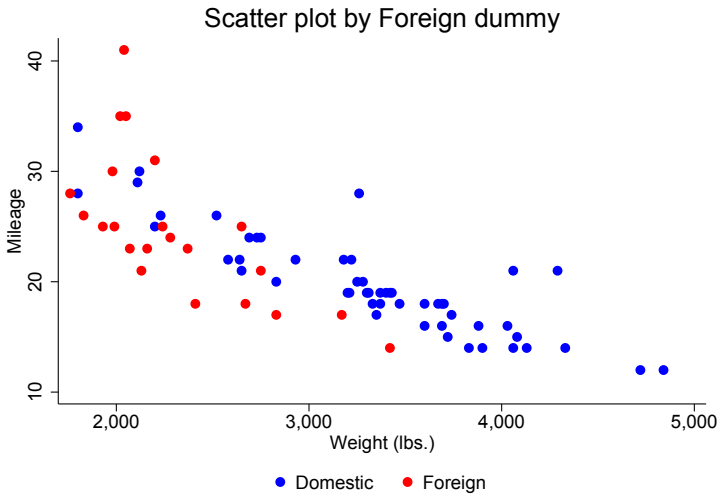


The next two **scatter plots** are examples of *twoway* graphs, i.e. graphs where we draw more than one figure at a time in the same window. In the first case we produce a scatter plot and then a quadratic line fitting the data. In the second case we plot two separate scatter plots in order to assign different colours to the observations relative to the *foreign* dummy.

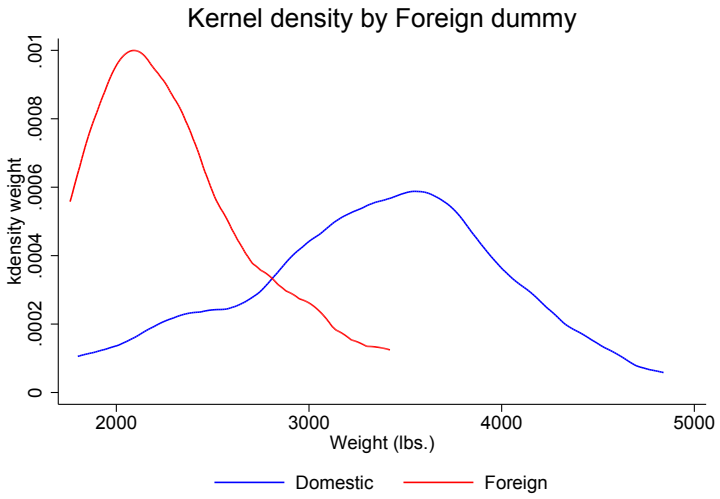
12. scatter plots

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

sysuse "auto", clear

// 1. Scatter plot with fitting line
twoway ///
( scatter mpg weight, mcolor(gs8) ) /// scatter plot
( qfit mpg weight, lcolor(black) lwidth(medium) /// quadratic fitting
      graphregion(color(white) lwidth(thick)) ylabel(, nogrid) legend(off) ///
      title("Scatter plot with quadratic fitting", color(black)) ytitle("Mileage") )
graph export "graphs\scatter1.pdf", replace

// 2. Scatter plot by "foreign" dummy
twoway ///
( scatter mpg weight if foreign==0, mcolor(blue) ) /// 1st scatter plot
( scatter mpg weight if foreign==1, mcolor(red) /// 2nd scatter plot
      graphregion(color(white) lwidth(thick)) ylabel(, nogrid) ytitle("Mileage") ///
      title("Scatter plot by Foreign dummy", color(black)) ///
      legend(lab(1 "Domestic") lab(2 "Foreign") nobox region(lcolor(white))) )
graph export "graphs/scatter2.pdf", replace
```





As an alternative to histograms, you can also use kernel densities (script not included):



13. kernel densities

```
clear all
cd "C:\Users\stelo424\Dropbox\stelo"

sysuse "auto", clear

// Kernel densities
twoway ///
( kdensity weight if foreign==0, lcolor(blue) lwidth(medium) ) /// 1st kernel density
( kdensity weight if foreign==1, lcolor(red) lwidth(medium) /// 2nd kernel density
      graphregion(color(white) lwidth(thick)) ///
      ylabel(, nogrid) xtitle("Weight (lbs.)") ///
          title("Kernel densities by dummy variable values", color(black)) ///
      legend(lab(1 "Domestic") lab(2 "Foreign") nobox region(lcolor(white))) )
graph export "graphs\kdensities.pdf", replace
```

3.4 Creating and modifying variables

In what follows, we cover the following fundamental commands:

Stata command	description
`generate`	create a new variable.
`replace`	replace contents of existing variable.
`drop`	eliminate variables or observations.
`keep`	keep variables or observations.
`destring`, `encode`	convert string variables in numeric.
`sort`	often used together with `by`, it sorts observations according to one or more variables.
`by`	*prefix* repeating the subsequent command on subsets of the data (according to the *varlist* specified).

Important: *Stata considers* ***missing values*** *– identified by a dot (.) – as* ***large*** *values. Keep this in mind when creating and modifying variables.*

3.4.1 `generate` and `replace`

To create a new variable you need to use `generate`.[9] This is often (but not necessarily) used in tandem with `replace`, which of course is a command that can be used on its own as well. Either way, the general syntax[10] used in both cases is:

command *varname* =*exp* [*if*] [*in*] [, *command_options*]

where the arguments are:

- **`command`**: either `generate` or `replace`.
- `varname`: a new variable name in case of `genereate` or an existing one for `replace`.
- `exp`: the *expression* used to generate or replace `varname`. Note that `exp` is preceded by only **one equal sign** (`=`). The expression can be:
  - a function of an already existing variable (e.g. `varname` is *age* squared).
  - newly generated numbers (typically, random variates drawn from a given $f(x)$).
  - a combination of the two.
- [*if*] and [*in*]: see sections 1.3 and 3.1. In particular, remember to use **two equal signs** (`==`) when specifying an equality statement.

Whenever you simulate random variates you should **always set the seed** in order to guarantee replicability of your results. This is done by using `set seed #`, where `#` is a constant.

[9] Sometimes `egen`, the extended version of `generate`, is actually required. Check the example at section 3.4.4.
[10] Check `help gen` to see the complete set of optional arguments and options for the two commands.

The next example will make use of some of the previously presented commands and ideas. If you do not feel comfortable, take 15 minutes to read the previous sections before continuing. You will also note that some commands are "new": we haven't encountered them yet. I promise that this won't be a problem as long as you read and understood the materials covered so far.[11]

1. Use `set obs` to create an empty dataset composed by 1000 observations.
2. Create a *female* dummy such that 60% of observations are females.
3. Set the seed equal to 12345 and use `rnormal` to create the variable *height* such that $height \sim N(\mu, \sigma^2)$ with (μ, σ) equal to $(70, 3)$ for men and $(64, 3)$ for women.
4. Codify *height* as missing if a man is shorter than 63; check missing values number.
5. Create a new variable called *height_cat*. For men, set it equal to `"tall"` if *height* is grater than or equal to 75, `"short"` if *height* is smaller or equal to 65, and `"average"` otherwise. For women, use 59 and 70 as lower and higher cut-off, respectively.
6. Visualize the joint distribution of *height_cat* and *female* in a table; save *output1.dta*.

14. generate and replace

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

// 1. Create empty dataset (N=100)
set obs 1000

// 2. Female dummy (60% observations are women)
gen female=1 if _n/_N <= 0.6
replace female=0 if female==.

// 3. Create height
set seed 12345
gen height=rnormal(70,3) if female==0
replace height=rnormal(64,3) if female==1

// 4. height missing for men shorter than 63
replace height=. if female==0 & height<63
count if height==. // alternatively use codebook

// 5. Create height_cat
// note "& height!=."!
gen height_cat=.
* men
replace height_cat=3 if female==0 & height>=75 & height!=.
replace height_cat=2 if female==0 & height<75 & height>65
replace height_cat=1 if female==0 & height<=65
```

[11] You can find all the informaiton you need and more on Stata `help`.

```
* women
replace height_cat=3 if female==1 & height>=70 & height!=.
replace height_cat=2 if female==1 & height<70 & height>59
replace height_cat=1 if female==1 & height<=59

// 6. Produce frequencies table and save
tabulate height_cat female
save "use/output1", replace
```

3.4.2 `drop` and `keep`

The `drop` command can be used in the two following ways (`keep` works symmetrically):

- **`drop`** *`varlist`*: erase *varibles* from memory according to the specified `varlist`.
- **`drop if`** *`exp`*: drop *observatios* according to *`exp`*.

Neither `drop` nor `keep` are reversible. Hence, if the aim is to perform operations on subsets of observations or variables, it is better to use the *if* statement (as we repeatedly did in the previous exercise). **`drop`** and **`keep`** are correctly used if the goal is to clean and reorganize the data in memory (to simply rename variables, use `rename`). In the next exercise we:

1. Load the example dataset `auto.dta`.
2. Transform *price* from dollars to thousand dollars without using `generate`. Call it *price_th*.
3. Clean data from cars costing more than $5000 having *rep78* missing; report sample size.
4. Drop all variables but *price* and *price_th* and save as *output2.dta*

15. drop and keep

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

// 1. Load the auto dataset
sysuse auto, clear

// 2. price in thousand dollars
replace price=price/1000
rename price price_th

// 3. Drop cars obs. costing more than 5000 withrep78 missing
drop if price_th > 5 & rep78==.
count  // alternatively, di _N

// 4. Drop all variables but price and price_th
keep price price_th
save "use/output2", replace
```

3.4.3 `destring` and `encode`

Very often, some of the variables in memory are in unwanted formats. A typical situation is when categorical variables are stored as strings and we need to transform them in numerical. While `destring` converts string variables that are inherently numeric in their contents, `encode` deals with arbitrary string variables. In the following example we:

1. Load the example dataset `auto.dta`.
2. Check if there are string variables and convert them into numerical.
3. Repeat the above for the `destring1.dta` web dataset.

16. converting string variables

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

// 1. Load the auto dataset
sysuse "auto.dta", clear

// 2. Convert string variables
// destring doesn't work since "make" contains nonnumeric characters
describe
encode make, gen(make_new)

// 3. Convert string variables of destring1.dta
webuse "destring1.dta", clear
describe
destring, replace // "replace" here replaces the old variables with the new ones
```

3.4.4 `by` and `sort`

The majority of Stata commands allow the `by` *prefix*. The syntax of the command can take one of the two following forms:

`by` *varlist*`:` *stata_cmd*

`bysort` *varlist*`:` *stata_cmd*

with the following arguments:

- *varlist*: list of variables according to which applying *stata_cmd*.
- *stata_cmd*: any Stata command allowed to be used together with `by`.

Note that `by` requires data to be sorted according to *varlist* (either by using the `sort` option or by previously using the `sort` command), while `bysort` automatically takes care of this.[12]

[12] To choose whether to sort in ascending or descending order, use `gsort`.

Among the countless possible applications of this command, a good example is the creation of group-specific running counters or means (e.g. by cross-section *id* variable in a panel dataset).

17. by and sort

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

// 1. Load National Longitudinal Survey panel
webuse "nlswork.dta", clear

// 2. Use "sort" and "generate" to create id-level running counters
sort idcode year
by idcode: gen id_counter=_n

// 3. Shuffle rows and repeat using "bysort" and "egen"
* Note: "(year)" sorts data also by year, but "gen" is then computed by idcode only
gen random=runiform()
sort random                                 // shuffles dataset rows
bysort idcode (year): gen id_counter2=_n  // id_counter2 identical to id_counter

// 4. Use "egen" to create...
* ...average weeks worked for each person
bysort idcode: egen avg_work=mean(wks_work)
* ...person-specific yearly ranking of the hours worked
bysort idcode: egen rank_work=rank(wks_work)

// 5. Check
browse idcode id_counter* year *_work
```

4 Regression Analysis

In the next subsections we first see how to estimate linear and non-linear regression models. We then see how to obtain marginal effects and other post-estimation statistics.

More specifically, the commands that we cover in what follows are:

Stata command	**description**
regression command	
`regress`	linear regression.
`xtreg`	fixed-, random- and between-effects linear models.
`ivregress`	single-equation instrumental-variables regression.
`logit`, `probit`	logit and probit estimation.
post-estimation command	
`ereturn list`	returns estimation results.
`predict`	post-estimation predictions, residuals, etc.
`margins`	marginal effects computation.

4.1 Regression commands

4.1.1 Linear regression

Generally speaking, linear models are such that $\mathrm{E}[\,y\,|\,x\,] = X\beta$ is linear *in the parameters* in the β vector (not necessarily in the regressors). The general syntax used to fit linear models is:

```
command depvar [indepvars] [if] [in] [weight] [, command_options]
```

with the following arguments:

- **`command`**: either `regress` or `xtreg`.
- *`depvar`*: dependent variable in the regression.
- `[`*`indepvars`*`]`: independent variables.
- `[`*`, command_options`*`]`:
  - `vce(`*`vce_type`*`)`: shared by most regression commands. It allows to adopt robust, clustered, bootstrap or other variance estimators.
  - `re`, `fe`, `be` (for `xtreg` only): Random-, Fixed-, Between-Effects regression.

The `ivregress` command performs single-equation Instrumental Variables (IV) regression. The syntax is similar, but it additionally requires to separately specify the list of the exogenous regressors (i.e. the instruments) and the list of endogenous variables.

4.1.2 Non-linear regression

This general class of models comprises the cases where $\mathrm{E}[\,y\,|\,x\,]$ is *not* linear in the regressors. The **logit** and **probit** models are used with *binary* and *non-censored nor truncated* outcome. The commands used in this case are `probit` and `tobit`, with the same syntax as above.

4.2 Post-estimation commands

Once estimated the model of interest, you can immediately use `ereturn list` to visualize all the quantities relative to the estimation, including (but not only) the estimated coefficients and variance-covariance matrix.[13] You can store such results in different ways (as macro variables, scalars or matrices), but htis goes beyond the current presentation.[14]

Another extremely useful command that we will not cover here is `estopost`. It allows to post any estimated quantities – displayed by `ereturn list` – so that they can be tabulated and eventually exported as, for instance, *tex* or *doc* tables.

In what follows we focus on *prediction* and *marginal effects*. You should be aware, though, that both the availability and the interpretation of post-estimation commands are highly dependent on the specific estimation strategy adopted. Hence, you should *always* use:

```
help estimation_command postestimation
```

where *estimation_command* can be `regress`, `xtreg`, etc.

4.2.1 Prediction

After fitting our model, `predict` is used to obtain in- or out-of-sample predictions, residuals, and related quantities. The syntax of `predict` used after single equation models is:

```
predict newvar [if] [in] [, options]
```

with *newvar* being the custom name of the variable that is going to be created by `predict`. The content of *newvar* is different depending on the specification of `[, options]`, e.g. :

- *xb*: calculates the *linear prediction*, i.e. $X\hat{\beta}$ for the linear model and, for instance, $F(X\hat{\beta})$ for the probit model.
- *stdp*: standard error of the linear prediction.

Note that it can be specified if to restrict the prediction to the estimation sub-sample by using `predict newvar if e(sample)`.

In addition to `predict`, also `estat` produces post-estimation statistics. The main qualitative difference is that `estat` produces scalar- and matrix-valued statistics instead of new variables. As for `predict`, each estimator allows for different uses of this command.

[13] This is similar to what can be done after generic non-estimation commands with `return list`.
[14] Check `ereturn local`, `ereturn scalar` and `ereturn matrix` if interested.

4.2.2 `margins`

This command computes marginal means and effects and predictive margins. Its syntax is:[15]

margins [*marginlist*] [, *response_options options*]

with the following arguments:

- [*marginlist*]: list of categorical variables or interactions appearing in the estimation. If not specified, the overall margin is computed.
- [, *response_options*]: main options of the command. They allow to choose the type of marginal effect or elasticity.
- [, *options*]: additional options to refine the use of the command, the main ones being:
  - `At`: family of options specifying at which covariates level to compute `margins`.
  - `over(`*varlist*`)`: estimation at the unique values taken by *varlist*.
  - `SE`: options to choose the type of standard errors.
  - `Advanced`: advanced options concerning weights, the estimation sample used, etc.

Important: *Given the high number of options and the fact that some of them only make sense with some estimated models but not with others, it is higly reccommended to carefully read Stata* `help` *and the link* ***Remarks and examples*** *there suggested.*

Keeping the above warning is mind, in what follows we see how to use post-estimation commands in the *linear regression* case.

18. predict and margins

```
clear all
cd "C:/Users/Stefano/Dropbox/stelo"

sysuse "auto", clear

// 1. Preliminary linear regression
reg price mpg trunk weight

// 2. xb prediction
predict yhat, xb  // same as using "if e(sample)"

// 3. margins
margins, dydx(*)
```

[15] We can also use the *if*, *in* and *weight* statements.

5 Macro Variables and Loops

In this section we deal with two crucial topics for programming in Stata: *macro variables* and *loops*. If you take a look at the Stata user guide, you will see that there is an entire 50-pages chapter about *programming*. So it should be clear that the materials covered here are by no means exhaustive. Nonetheless, by the end of this section, you will hopefully have good knowledge of the basics necessary to deal with relatively advanced tasks.

5.1 Macro Variables

A **macro variable** has a *name* and it is characterized by some *content*. When you define it, you assign the content to the name. In a second moment, you can access the macro content.

There are two types of macros, *local macros* and *global macros*. They differ in their scope – as suggested by their name – and in the way their content is accessed. Besides this, they can be used interchangeably.

Local macros are *defined* and *accessed* respectively with:

`local` *macroname* `=`*exp*

`` `macroname' ``

The macro is *local* since it is only available for the program or script it is written within.

Global macros are instead available anywhere within the current session. They are *defined* and *accessed* with:

`global` *macroname* `=`*exp*

`$`*macroname*

It is generally wise to use local macros as often as possible to avoid conflicts in your code. On the other side, *all local macros contents are cleared from Stata memory after the do-file (or part of it) is run by Stata*. This means that whenever you want to run a part of the code containing '*macroname*', you also need to run *at the same time* the part of the code where you define the local macro. On the contrary, given their nature, this is not necessary for *global* macros.

The next box provides some basic examples of local macros (try also to use `global`).

19. local macros

```
local x 10
di ‘x’
di "the result of 2*x is: " 2*‘x’

local name = "stefano"
di "my name is " "‘name’"  // di ‘name’ doesn’t work

local names = ‘" "first " "second " "third" "’
di ‘names’
```

5.2 Loops

Loops allow to automatize repetitive commands. Whenever you have to perform any kind of operation (loading and appending data, creating variables, replacing values, etc.) which can be thought as being based on a running counter and/or on a set of names, you can apply a loop.

The types of loops available in Stata are:

Stata command	**description**
loop type	
`foreach`	loop over *items* of a list or macro variable.
`forvalues`	loop over consecutive *numbers*.
`while`	keep looping until a specified condition is met.
logical command	
`if` *exp* {...}	evaluates *exp*. If true, commands in {...} are executed.
`else if`, `else`	can be used together with `if` (same syntax).

Important: *The above* 'if' programming command *must not be confused with the* 'if' qualifier *that can be used as optional argument in most Stata commands (see section 1.3).*

20. loops (1)

```
sysuse "auto", clear

// foreach loops

* using simple list
reg price mpg rep78 headroom trunk weight length turn
foreach x in mpg rep78 headroom trunk weight length turn  {
    di "‘x’ coefficient: " _b[‘x’]  // _b is a system variable
}

* using local variable list
local covariates mpg rep78 headroom trunk weight length turn
quietly reg price ‘covariates’

foreach x in ‘covariates’ {
    di "‘x’ coefficient: " _b[‘x’]
}

foreach x of local covariates {
    di "‘x’ coefficient: " _b[‘x’]
}
```

21. loops (2)

```
// forvalues loop

forval i = 1/5 {    // equivalent to i = 1(1)10
    if "`i'"=="0" | "`i'"=="1" {    // need parentheses if more than 1 command
        di "Printing iteration number:"
        di "`i'st iteration"
    }
    else if "`i'"=="2" di "`i'nd iteration"
    else if "`i'"=="3" di "`i'rd iteration"
    else di "`i'th iteration"
}
```

In the next example we see how to generate the first 20 numbers of the Fibonacci's sequence, i.e. a sequence such that $y_1 = y_2 = 1$ and $y_n = y_{n-1} + y_{n-2}$ for $n > 2$. This is a good example to see how to loop keeping track of values relative to past iterations. It is also an instance where explicit looping is not required at all.

22. loops (3)

```
// while loop
* first 20 numbers of Fibonacci's sequence

local i = 1
local y_1 = 1
local y_2 = 1

while `i' <= 20 {
    if `i'<=2 local y=`y_1'
    else {
        local y = `y_2'+`y_1'
        local y_2=`y_1'  // update y-2
        local y_1=`y'    // update y-1
    }
    di "`y'"
    local i=`i'+1  // update counter
}

* Note: can be obtained just like this (trick: "in 3/1")
clear all
set obs 20
gen y=1
replace y = y[_n-1] + y[_n-2] in 3/l
list y
```