STATA SKILL PORTFOLIO

THE GUIDE

Stata: Line-by-Line

An Empirical Economics Guide

from the universal header to every command in the eleven-script portfolio

DUC V. LE

Lê Vĩnh Đức

PhD Candidate, Department of Economics · Georgetown University

Office ICC-580 · 3700 O St NW, Washington, DC 20057, USA

dvl11@georgetown.edu

Updated June 2026

Contents

How to read this guide

Knowing what a regression *is* is one thing; *saying it in Stata* is another. That gap is pure syntax, and this guide closes it one line at a time. Chapter 1 teaches the fundamentals that recur in every script — the grammar of a command, the one-dataset-in-memory model, and the boilerplate "header" at the top of each file. Chapters 2 onward then walk through each of the twelve scripts, block by block.

Its companion — *A Stata Reference for Empirical Economics* — states each method crisply with code and verified results: the place to look things up once the syntax here is second nature.

The conventions. Throughout, three kinds of boxes appear:

```
summarize price, detail // Stata code -- commands in ochre, comments in green
```

```
      price |         74    6165.257    2949.496       3291      15906     <- what Stata prints
```

R / pandas a quick translation to the familiar pandas / R idiom.

Gotcha a place beginners trip — read these twice.

Tip a habit that makes working in Stata easier.

How to use it. Open Stata, open the matching `.do` file, and read this guide alongside it: run each block, watch the output, and check it against the explanation here. Stata is learned by *typing* it, not by reading about it — so retype, don't copy-paste.

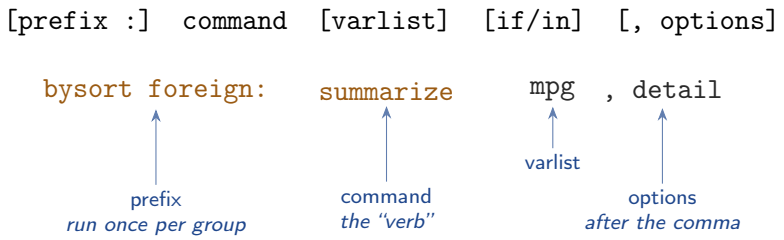
Code availability. The twelve `.do` files this guide walks through — and this document — are openly available at github.com/duc-v-le/stata-empirical-methods.

Chapter 1

Stata fundamentals & the universal header

1.1 The anatomy of a command

Almost every line of Stata follows *one* grammar. Learn this skeleton and each new command is just "fill in the blanks":



- **prefix** (optional, ends with :) — repeats the command, e.g. `bysort foreign:` runs it once for domestic and once for foreign cars.
- **command** — the verb: `summarize`, `regress`, `generate`.
- **varlist** — the variables it acts on. Omit it and many commands act on *all* variables.
- **if/in** — row filters: `if price>6000`, `in 1/5` (rows 1–5).
- **options** — everything after a *single* comma fine-tunes the command.

Gotcha there is only *one* comma in a command — everything after it is options. `summarize price, detail` has the option `detail`; a second comma is a syntax error.

Every slot at once. This single command fills *all five* slots — a prefix (`bysort foreign:`), a command (`summarize`), a varlist (`mpg`), an `if` filter (`if price>5000`), and an option (`, detail`):

```
1 bysort foreign: summarize mpg if price>5000, detail
```

It runs the detailed summary *once per origin*, over only the cars priced above $5000 — giving two full distributions, on $23 + 14$ cars rather than all 74:

```
-> foreign = Domestic

                        Mileage (mpg)
-------------------------------------------------------------
      Percentiles      Smallest
 1%           12             12
 5%           12             12
10%           14             14       Obs                  23
25%           14             14       Sum of wgt.          23

50%           16                      Mean           16.91304
                        Largest       Std. dev.       3.46296
75%           19             21
90%           21             21       Variance       11.99209
95%           22             22       Skewness       .7731511
99%           26             26       Kurtosis       3.215946

-> foreign = Foreign

                        Mileage (mpg)
-------------------------------------------------------------
      Percentiles      Smallest
 1%           14             14
 5%           14             17
10%           17             17       Obs                  14
25%           18             18       Sum of wgt.          14

50%           23                      Mean           22.42857
                        Largest       Std. dev.      6.441623
75%           25             25
90%           25             25       Variance       41.49451
95%           41             25       Skewness       1.602363
99%           41             41       Kurtosis       6.093446
```

1.2 One dataset in memory

This is the biggest mental shift from pandas or R. Stata holds **one dataset in memory at a time**, and commands act on "the data" implicitly — it is almost never named. `use` or `sysuse` loads a dataset (replacing whatever was there), `save` writes it back, `clear` empties memory.

R / pandas in pandas, many `DataFrames` are juggled by name (`df1`, `df2`); in R, many objects. In Stata there is just *the* dataset, transformed in place with `generate` / `replace` rather than assigning to a new object. (Stata 16+ has *frames* for multiple datasets, but the portfolio uses the classic single-dataset model.)

1.3 The universal header

Every script in the portfolio opens with the same handful of lines. They guarantee that running the file from a clean slate gives the *same* result every time. Here it is, decoded:

```
version 17 // interpret commands using Stata-17 semantics
clear all // wipe memory: data, programs, stored results
set more off // don't pause for --more-- in long output
set linesize 90 // fix the log/output width at 90 characters
set seed 1010 // fix the random-number stream (reproducibility)
capture log close // close any stray open log; stay quiet if none
```

```
7 log using "logs/01_basics.log", replace text // open a fresh plain-text log
```

- `version 17` — forward-compatibility. Tells *any* Stata running the file to use version-17 behaviour, so the code keeps working identically on future releases.
- `clear all` — empties memory completely (data, value labels, matrices, stored `e()`/`r()` results, programs) so nothing from a previous run leaks in.
- `set more off` — by default Stata pauses when the screen fills and waits for a key; this lets output scroll freely (essential in batch mode).
- `set linesize 90` — wraps output at 90 characters so the log looks identical regardless of window width.
- `set seed 1010` — seeds the random-number generator; every later `runiform()` / `rnormal()` draw is then identical on every run. This is *why* the simulated results reproduce exactly.
- `capture log close` — `log close` errors if no log is open; `capture` swallows that error, so this safely closes a leftover log before the next line opens a fresh one.
- `log using "...", replace text` — starts recording everything to a *plain-text* log file, overwriting (`replace`) any previous one.

Tip no need to memorise this — copying the seven lines to the top of any new `.do` file gives a clean, reproducible, logged session.

1.4 Comments, line breaks, and macros

- **Comments:** `* whole-line` (only at the start of a line), `// inline`, and `/* ...*/` for a block.
- **Long commands:** end a line with `///` to continue onto the next — handy for a `twoway` graph with many options.
- **Macros (later chapters):** a *local* is a temporary variable for *numbers or text*, not data: `local h = 0.5` stores it, and `‘h’` refers back to it. We meet these in the advanced scripts, where they carry an estimate from one command into the next.

Chapter 2

A first script: `01_basics.do`

The workflow. Load a bundled dataset, look at it, build and label new variables, draw and export two graphs, and save a cleaned copy — the full "load → explore → transform → save" loop in one file.

2.1 Load the data

```
sysuse auto, clear // 1978 automobile data (74 cars)
describe // structure: variables, types, labels  (~ R str())
codebook mpg foreign, compact // quick audit of a few variables
list make price mpg in 1/5 // peek at the first 5 rows  (~ head())
```

- `sysuse auto, clear` — loads one of the example datasets shipped *inside* Stata (works offline). `clear` discards anything already in memory first.
- `describe` — the structure: each variable's name, storage type, display format, and label.
- `codebook ... , compact` — a compact audit (range, missing, unique values).
- `list ... in 1/5` — prints rows 1–5; `in` filters by row *position*.

`describe` prints:

```
Contains data from .../auto.dta
 Observations:            74                  1978 automobile data
    Variables:            12
-------------------------------------------------------------------------------
Variable      Storage   Display    Value
    name         type    format    label      Variable label
-------------------------------------------------------------------------------
make            str18   %-18s                 Make and model
price           int     %8.0gc                Price
mpg             int     %8.0g                 Mileage (mpg)
foreign         byte    %8.0g      origin     Car origin
-------------------------------------------------------------------------------
```

And `list ... in 1/5` prints the first five rows as a clean table:

```
     +-----------------------------+
     | make            price   mpg |
     |-----------------------------|
```

```
  1. | AMC Concord     4,099    22 |
  2. | AMC Pacer       4,749    17 |
  3. | AMC Spirit      3,799    22 |
  4. | Buick Century   4,816    20 |
  5. | Buick Electra   7,827    15 |
     +-----------------------------+
```

R / pandas `sysuse` ≈ a toy dataset (`sns.load_dataset()`, R's `data(mtcars)`);
`describe` ≈ `df.info()` / `str(df)`;
`list in 1/5` ≈ `df.head()`.

Gotcha the storage type matters. `price` is an `int`, `mpg` an `int`, `foreign` a `byte` with a *value label* (`origin`) — so it prints as "Domestic"/"Foreign" even though it is stored as 0/1.

2.2 Summarize and explore

```
summarize // mean, SD, min, max for all numeric variables
summarize price, detail // adds percentiles, skewness, kurtosis
tabulate foreign // one-way frequency table
tabulate foreign rep78, row // two-way table, with row percentages
bysort foreign: summarize mpg // a separate summary for each group
```

- `summarize` (no varlist) — summarises *every* numeric variable.
- `, detail` — the long form: percentiles, median, skewness, kurtosis.
- `tabulate` — frequency tables; with two variables, a cross-tab (add `row`/`col` for percentages).
- `bysort foreign:` — the *prefix* from Chapter 1: run `summarize` once per origin.

`summarize` prints one row per variable:

```
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
       price |         74    6165.257    2949.496       3291      15906
         mpg |         74     21.2973    5.785503         12         41
       rep78 |         69    3.405797    .9899323          1          5    <- 5 missing (69<74)
      weight |         74    3019.459    777.1936       1760       4840
```

`summarize price, detail` expands one variable into its full distribution:

```
                          Price
-------------------------------------------------------------
      Percentiles      Smallest
 1%         3291           3291
25%         4195           3748       Obs                  74
50%       5006.5                      Mean           6165.257
                        Largest       Std. dev.      2949.496
75%         6342          13466       Variance        8699526
99%        15906          15906       Skewness       1.653434
                                       Kurtosis       4.819188
```

`tabulate foreign` is a one-way frequency table:

```
 Car origin |      Freq.     Percent        Cum.
------------+-----------------------------------
   Domestic |         52       70.27       70.27
    Foreign |         22       29.73      100.00
------------+-----------------------------------
      Total |         74      100.00
```

With two variables, `tabulate foreign rep78, row` becomes a cross-tab (here with row %):

```
           |              Repair record 1978
Car origin |     1      2      3      4      5 |   Total
-----------+-------------------------------------+--------
  Domestic |     2      8     27      9      2 |      48
           |  4.17  16.67  56.25  18.75   4.17 |  100.00
   Foreign |     0      0      3      9      9 |      21
           |  0.00   0.00  14.29  42.86  42.86 |  100.00
```

And `bysort foreign: summarize mpg` runs the summary *once per group*:

```
-> foreign = Domestic
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         52    19.82692    4.743297         12         34

-> foreign = Foreign
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         22    24.77273    6.611187         14         41
```

R / pandas `summarize` ≈ `df.describe()`; `tabulate` ≈ `value_counts()` / `crosstab()`; `bysort g: summarize` ≈ `df.groupby("g").describe()`.

2.3 Create and transform variables

This is the heart of data work. Three verbs do almost everything:

```
generate gpm = 1/mpg // NEW variable
label variable gpm "Gallons per mile" // attach a human-readable label
generate price_k = price/1000
generate byte expensive = price > 6000 // a logical test -> 0/1 indicator
egen mpg_z = std(mpg) // egen: "extended" generate (standardise)
egen price_mean_by_for = mean(price), by(foreign) // group mean (a window function)
replace size_class = "large" if weight > 3500 // overwrite, conditionally
```

- `generate` — creates a *new* variable from a formula. Errors if the name already exists.
- `replace` — *overwrites* an existing variable (often with an `if` condition).
- `generate byte expensive = price > 6000` — a logical test returns **1 if true, 0 if false**; `byte` stores that 0/1 efficiently.
- `egen` — "extended" generate: functions `generate` can't do alone, like `std()` (standardise) or `mean(...), by()` (group statistics).

`egen` prints nothing — it just *creates* the variables. To see what the two above produced, `list` three cars from each origin. `mpg_z` is standardised (centred at 0, scaled to SD 1), and `price_mean_by_for` (shown abbreviated as `price_~r`) holds the *group mean* — the *same* value on every car of a given origin:

```
     +-----------------------------------------+
     | mpg   mpg_z    foreign   price   price_~r |
     |-----------------------------------------|
  1. |  22    0.12   Domestic   4,099      6,072 |   every domestic car -> 6,072
  2. |  17   -0.74   Domestic   4,749      6,072 |
  3. |  22    0.12   Domestic   3,799      6,072 |
 53. |  17   -0.74    Foreign   9,690      6,385 |   every foreign car  -> 6,385
 54. |  23    0.29    Foreign   6,295      6,385 |
 55. |  25    0.64    Foreign   9,735      6,385 |
     +-----------------------------------------+
```

Gotcha `generate` vs `replace` is strict: `generate` only *creates* (it refuses to overwrite), `replace` only *changes* (it refuses to create). Pick the right verb or Stata complains.

R / pandas `generate`/`replace` ≈ `df["x"]=...` / `mutate()`;
`egen ... , by(g)` ≈ `groupby("g").transform("mean")`.

2.4 Labels make output self-documenting

```
label define yesno 0 "No" 1 "Yes" // define a value-label map
label values expensive yesno // attach it to the variable
tabulate expensive // now prints "No"/"Yes", not 0/1
```

`tabulate expensive` now prints the *labels*, not the raw 0/1:

```
  expensive |      Freq.     Percent        Cum.
------------+-----------------------------------
         No |         51       68.92       68.92
        Yes |         23       31.08      100.00
------------+-----------------------------------
      Total |         74      100.00
```

A *value label* maps the stored numbers (0/1) to readable text without changing the data — a Stata habit that makes every table and graph self-explanatory.

2.5 Two graphs, saved to disk

```
histogram price, frequency title("Distribution of price")
graph export "output/01_hist_price.png", replace width(1200)

twoway (scatter price weight) (lfit price weight), ///
    title("Price vs. weight with linear fit") legend(off)
graph export "output/01_scatter_price_weight.png", replace width(1200)
```

- `histogram price` — a histogram of one variable.

- `twoway` — the workhorse plot: layer "plot types" in parentheses. Here `(scatter price weight)` draws points and `(lfit price weight)` overlays the OLS line. The `///` continues the command onto the next line.
- `graph export` — writes the current graph to a file (no on-screen window needed, so it works in batch mode).

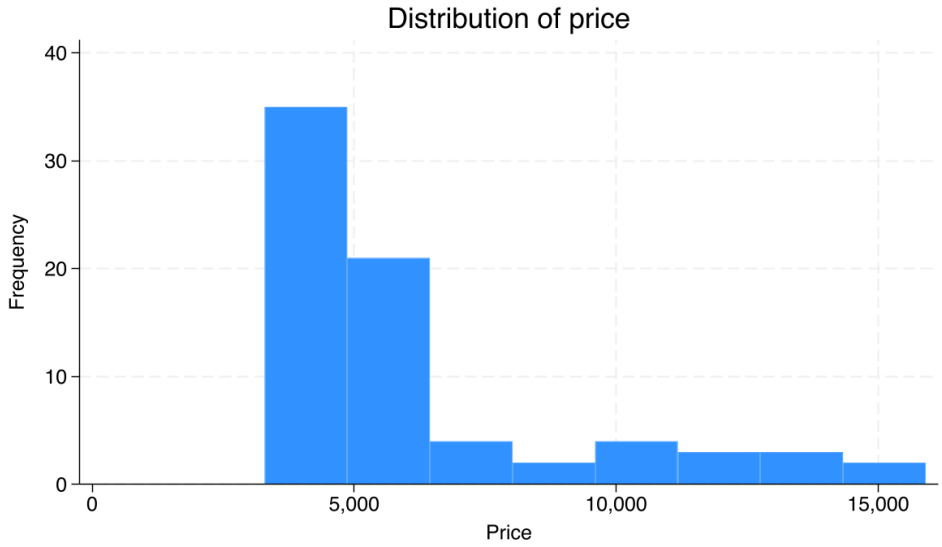


Figure 2.1: `histogram price` — the exported `01_hist_price.png`.

2.6 Save a cleaned copy

```
1 save "output/auto_clean.dta", replace // Stata's native binary format
```

`save` writes the dataset (with all the new variables and labels) to a `.dta` file; `replace` overwrites any previous copy.

R / pandas `.dta` ≈ `.parquet`/`.rds` — a native binary that preserves types and labels, unlike a flat CSV.

That is the whole "load → explore → transform → label → plot → save" loop. Every later script is a variation on it with a different model in the middle.

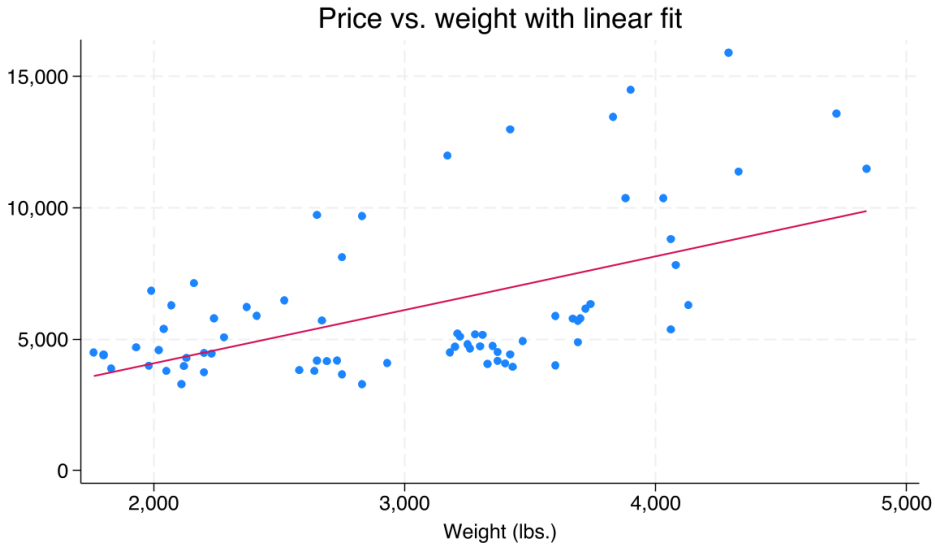


Figure 2.2: `twoway (scatter)(lfit)` — points plus the fitted line.

Chapter 3

Joining, reshaping, aggregating: 02_data_management.do

The workflow. The everyday plumbing of empirical work: join files with `merge`, pivot between wide and long with `reshape`, aggregate with `collapse`, and tidy dates, strings, and duplicate rows.

3.1 Merge: joining two files

```
use "output/_left.dta", clear // master data (in memory)
merge 1:1 make using "output/_right.dta" // join on the key variable ‘make‘
tabulate _merge // 3 = matched; 1 = master only; 2 = using only
assert _merge==3 // abort if any row failed to match
drop _merge
```

- `merge 1:1 make using "..."` — a one-to-one join: each `make` appears once in each file. The relationship (`1:1`, `1:m`, `m:1`) *must* be stated; Stata will not guess.
- `_merge` — after any merge, Stata creates this variable flagging each row: `1`=master only, `2`=using only, `3`=matched both.
- `assert _merge==3` — a guard that *stops the script* if the join wasn't clean. A good habit: fail loudly rather than silently keep bad rows.

`merge` first reports how the two files lined up — here every `make` found its partner, so all 74 rows land in `_merge==3`:

```
    Result                      Number of obs
    -----------------------------------------
    Not matched                             0
    Matched                                74  (_merge==3)
    -----------------------------------------
```

`tabulate _merge` confirms a clean join (all 74 rows matched):

```
   Matching result from |
                  merge |      Freq.     Percent        Cum.
------------------------+-----------------------------------
```

```
            Matched (3) |         74      100.00      100.00
------------------------+-----------------------------------
                  Total |         74      100.00
```

That join was perfectly clean. When one *isn't*, `_merge` flags the strays — here a master keyed {CAN, MEX, BRA} meets a using keyed {USA, CAN, MEX}:

```
   Matching result from |
                  merge |      Freq.     Percent        Cum.
------------------------+-----------------------------------
        Master only (1) |          1       25.00       25.00
         Using only (2) |          1       25.00       50.00
            Matched (3) |          2       50.00      100.00
------------------------+-----------------------------------
                  Total |          4      100.00
```

```
  +---------------------------------------+
  | country   pop    gdp            _merge |
  |---------------------------------------|
  |     BRA     .   1600   Master only (1) |
  |---------------------------------------|
  |     USA   331      .    Using only (2) |
  |---------------------------------------|
  |     CAN    38   1700       Matched (3) |
  |     MEX   126   1100       Matched (3) |
  +---------------------------------------+
```

`BRA` sat only in the master (so its `pop` is missing, `.`); `USA` only in the using (so its `gdp` is missing) — an unmatched row keeps its own columns and gets `.` for the other file's. That is exactly why `assert _merge==3` is a good reflex.

R / pandas `merge 1:1` ≈ `pd.merge(..., validate="1:1", indicator=True)`
`_merge` is that `indicator` column.

Gotcha the file in memory is the *master*; the one named in `using` is the *using* dataset. `merge` matches the using *into* the master.

Which relationship — `1:1`, `m:1`, or `1:m`? The two numbers say how many rows share a key in the *master : using*. The type is declared, and `merge` *enforces* it, erroring if the data disagree.

type	the key appears…	typical use
`1:1`	once in each file	join two unit-level files (here, one row per car)
`m:1`	many in master, once in using	attach a group-level value onto every member row
`1:m`	once in master, many in using	the mirror of `m:1`

`m:1` is the everyday case — e.g. merging a country-level GDP file *into* a country–year panel, so every row of a country picks up the same GDP. Avoid `m:m`: it rarely does what is expected.

Concretely: a six-row country–year panel (the master) and a three-row file holding one GDP figure per country (the using). `merge m:1 country` copies each country's GDP onto *all* of its rows:

```
merge m:1 country using "gdp.dta" // 6 panel rows take one GDP per country
sort country year
list country year gdp _merge, sepby(country) noobs
```

```
    Result                      Number of obs
    -----------------------------------------
    Not matched                             0
    Matched                                 6  (_merge==3)
    -----------------------------------------
```

```
  +-------------------------------------+
  | country   year     gdp        _merge |
  |-------------------------------------|
  |     CAN   2020    1700   Matched (3) |
  |     CAN   2021    1700   Matched (3) |
  |-------------------------------------|
  |     MEX   2020    1100   Matched (3) |
  |     MEX   2021    1100   Matched (3) |
  |-------------------------------------|
  |     USA   2020   21000   Matched (3) |
  |     USA   2021   21000   Matched (3) |
  +-------------------------------------+
```

Both of `CAN`'s rows pick up `1700`, both of `USA`'s pick up `21000` — one GDP value fanned out to every year of its country. That copy-down is the whole point of `m:1`.

3.2 Reshape: wide ↔ long

```
reshape wide income, i(id) j(year) // long  -> wide  (one column per year)
reshape long income, i(id) j(year) // wide  -> long  (back again)
```

- `i(id)` — the *unit* identifier (the row in wide form).
- `j(year)` — the variable that *spreads into columns* (wide) or indexes the long form.
- the *stub* is `income`: wide form holds `income2010`, `income2011`, ...

`reshape` reports exactly what it did — here turning 6 long rows into 2 wide ones:

```
Data                               Long   ->   Wide
-----------------------------------------------------------------------------
Number of observations                6   ->   2
Number of variables                   3   ->   4
xij variables:
                                 income   ->   income2010 income2011 income2012
-----------------------------------------------------------------------------
```

The same six numbers, two shapes:

long

id	year	income
1	2010	100
1	2011	110
2	2010	200
2	2011	220

`reshape wide` →
← `reshape long`

wide

id	income2010	income2011
1	100	110
2	200	220

R / pandas `reshape wide` ≈ `pivot` / `pivot_wider`; `reshape long` ≈ `melt` / `pivot_longer` (the form panel commands want).

3.3 Collapse: aggregate to groups

```
collapse (mean) mean_price=price (sd) sd_price=price (count) n=price, by(foreign)
```

`collapse` reduces the whole dataset to *one row per group* of `by()`, computing the requested statistics. Here: mean and SD of price, and a count, for domestic vs foreign. The data in memory is now just two rows — one per group:

```
     +-----------------------------------------------+
     |  foreign   mean_p~e   mean_mpg   sd_price    n |
     |-----------------------------------------------|
  1. | Domestic    6,072.4    19.8269    3,097.1   52 |
  2. |  Foreign    6,384.7    24.7727    2,621.9   22 |
     +-----------------------------------------------+
```

Gotcha `collapse` *replaces* the data in memory with the aggregated rows — the individual observations are gone. `save` first (or `preserve`/`restore`) if they are still needed.

R / pandas ≈ `df.groupby("foreign").agg(...)` — but in place, not a new object.

3.4 Dates, strings, duplicates

```
generate edate = date(raw, "YMD") // string "2020-01-15" -> a numeric date
format edate %td // display it as a calendar date
generate brand = word(make, 1) // first word of the make string
duplicates report // how many exact duplicate rows?
duplicates drop // remove them
```

- Stata stores a date as an *integer* (days since 1 Jan 1960); `date(str,"YMD")` parses a string into that integer, and `format ... %td` makes it *display* as a date.
- String helpers: `word()`, `upper()`, `strpos()`.
- `duplicates report/drop` — audit and remove repeated rows.

Gotcha a human-readable date ("2020-01-15") is a *string*; a date Stata can compute with is a *number* with a `%td` format. `date()` converts the first into the second.

Dates: a number first, a calendar after. `date` returns a plain integer; `format %td` changes only how it *prints* (same column, two displays):

```
  +--------------------+          +------------------------+
  |        raw   edate |          |        raw       edate |
  |--------------------|          |------------------------|
  | 2020-01-15   21929 |   -->    | 2020-01-15   15jan2020 |
  | 2020-06-30   22096 |          | 2020-06-30   30jun2020 |
  | 2021-12-01   22615 |          | 2021-12-01   01dec2021 |
  +--------------------+          +------------------------+
```

21929 *is* 15 Jan 2020 (days since 1 Jan 1960). The three string helpers, run on `make`:

```
+-------------------------------------------+
| make            brand         make_up   spc |
|-------------------------------------------|
| AMC Concord       AMC     AMC CONCORD     4 |
| AMC Pacer         AMC       AMC PACER     4 |
| AMC Spirit        AMC      AMC SPIRIT     4 |
| Buick Century   Buick   BUICK CENTURY     6 |
| Buick Electra   Buick   BUICK ELECTRA     6 |
| Buick LeSabre   Buick   BUICK LESABRE     6 |
+-------------------------------------------+
```

`brand` is `word(make,1)` (first token), `make_up` is `upper(make)`, and `spc` is `strpos(make," ")` — the position of the first space (`4` in "AMC Concord", `6` in "Buick Century"; `0` would mean no space at all).

Chapter 4

Regression: 03_regression.do

The workflow. Fit OLS with robust standard errors and factor-variable interactions, export a publication table, read the model with marginal effects, and run post-estimation diagnostics.

4.1 OLS, robust SEs, and factor variables

```
1 regress price mpg weight_t, vce(robust)
2 regress price mpg weight_t i.foreign, vce(robust)
3 regress price c.mpg##c.weight_t i.foreign, vce(robust)
```

- `regress y x1 x2` — OLS of the first variable on the rest.
- `vce(robust)` — heteroskedasticity-robust (White) standard errors.
- `i.foreign` — *factor* notation: treat `foreign` as categorical and add its dummies automatically (no manual `generate`).
- `c.mpg##c.weight_t` — `c.` marks a variable *continuous*; `##` is a *full interaction* (both main effects *and* their product).

Each regression line corresponds to one model:

$$(1)\quad \texttt{price} = \beta_0 + \beta_1\,\texttt{mpg} + \beta_2\,\texttt{weight\_t} + \varepsilon,$$
$$(2)\quad \texttt{price} = \beta_0 + \beta_1\,\texttt{mpg} + \beta_2\,\texttt{weight\_t} + \beta_3\,\mathbf{1}[\texttt{foreign}] + \varepsilon,$$
$$(3)\quad \texttt{price} = \beta_0 + \beta_1\,\texttt{mpg} + \beta_2\,\texttt{weight\_t} + \beta_3(\texttt{mpg}\cdot\texttt{weight\_t}) + \beta_4\,\mathbf{1}[\texttt{foreign}] + \varepsilon.$$

Line 2 adds the `foreign` dummy $\mathbf{1}[\texttt{foreign}]$; line 3 adds the `mpg` $\times$ `weight_t` interaction on top of it.

R / pandas `vce(robust)` $\approx$ `cov_type="HC1"`; `i.foreign` $\approx$ `C(foreign)`; `c.x##c.z` $\approx$ `x*z` in `patsy` / `feols`.

Gotcha the `i./c.` prefix is not optional cosmetics — it tells Stata whether a variable is categorical (dummies) or continuous (a slope). Get it wrong and the model changes.

4.2 Export a publication table

```
eststo clear
eststo m1: regress price mpg weight_t, vce(robust) // store model m1
eststo m2: regress price mpg weight_t i.foreign, vce(robust)
esttab m1 m2 m3 using "output/03_regression_table.tex", ///
    replace se star(* 0.10 ** 0.05 *** 0.01) r2 label booktabs
```

`eststo` stores each fitted model under a name; `esttab` then prints a side-by-side comparison and *exports* it to LaTeX (and CSV) — the clean, typeset table that goes straight into a paper or report. `star()` adds significance stars; `r2` adds R^2; `label` uses the variable labels. `esttab` prints the three models side by side (and writes the same to LaTeX/CSV):

```
                           (1)             (2)             (3)
                          Base        +Foreign    +Interaction
--------------------------------------------------------------------
Mileage (mpg)           -49.51           21.85          292.83*
                       (95.81)         (80.75)        (153.90)
Weight (1000 lbs)     1746.56**       3464.71***      5382.75***
                      (777.84)        (777.62)       (1084.04)
Foreign                                3673.06***      3369.81***
                                      (664.94)        (757.93)
Constant              1946.07         -5853.70       -10105.04***
--------------------------------------------------------------------
Observations               74              74              74
R-squared                0.293           0.500           0.524
--------------------------------------------------------------------
* p<0.10, ** p<0.05, *** p<0.01
```

Tip `eststo`/`esttab` (the `estout` package) is the single most useful thing to learn for producing real deliverables. Readers want a clean, typeset table, not a screenshot of the console.

4.3 Marginal effects

```
quietly regress price c.mpg##c.weight_t i.foreign, vce(robust) // re-fit model (3)
margins, dydx(mpg) // average marginal effect of mpg
margins foreign // predicted price by foreign status
margins, dydx(mpg) at(weight_t=(2 3 4)) // how the mpg slope varies with weight
```

`margins` always works on the *last model Stata fitted*, so the first line re-fits model (3) (the `+Interaction` column from the table above); the `quietly` prefix just hides its regression output, which has already appeared above. That model carries the interaction `c.mpg##c.weight_t`, so the effect of `mpg` on `price` is *not* a single number. Differentiating model (3),

$$\frac{\partial\,\texttt{price}}{\partial\,\texttt{mpg}} = \beta_1 + \beta_3\,\texttt{weight\_t},$$

so the slope *depends on* `weight_t`. `margins` does this calculus automatically — the three lines ask three different questions.

Line 1, `dydx(mpg)` — the slope, averaged over the cars. Stata plugs each car's own weight into that derivative and averages:

```
             |            Delta-method
             |      dy/dx   std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
         mpg |  -66.21965   138.8938    -0.48   0.635    -343.3052    210.8659
```

So a 1-mpg increase is worth about \$66 *less* on average — but with a standard error of \$139 ($p = 0.64$), essentially zero. "On average" is hiding something; line 3 shows what.

Line 2, `margins foreign` — predicted *levels*, not a slope. With no `dydx`, `margins` reports the predicted `price` at each value of `foreign` (holding mileage and weight at their sample values):

```
             |            Delta-method
             |     Margin   std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     foreign |
    Domestic |    5163.42   297.9123    17.33   0.000     4569.101    5757.739
     Foreign |   8533.234   627.0342    13.61   0.000     7282.335    9784.133
```

A domestic car is predicted at about \$5,163, a foreign one at \$8,533.

Line 3, `dydx(mpg) at(weight_t=(2 3 4))` — the slope *at* chosen weights. The same derivative, now evaluated at `weight_t` $= 2, 3, 4$ (2,000, 3,000, 4,000 lbs):

```
1._at: weight_t = 2
2._at: weight_t = 3
3._at: weight_t = 4
             |            Delta-method
             |      dy/dx   std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           1 |   55.00606   76.20992     0.72   0.473    -97.02856    207.0407
           2 |  -63.90569   137.4455    -0.46   0.643     -338.102    210.2906
           3 |  -182.8174   216.0004    -0.85   0.400    -613.7265    248.0916
```

The slope *flips sign*: $+55$ for a light car (row 1, 2,000 lbs) down to -183 for a heavy one (row 3, 4,000 lbs). That sign change *is* the interaction — and it is exactly why the sample average in line 1 came out near zero (the positive and negative slopes cancel).

R / pandas `margins` ≈ `.get_margeff()` in statsmodels — but far more flexible.

4.4 Post-estimation diagnostics

```
quietly regress price mpg weight_t i.foreign // re-fit
test mpg weight_t // joint F-test: are both coefficients zero?
estat hettest // Breusch-Pagan test for heteroskedasticity
estat vif // variance inflation factors (multicollinearity)
predict yhat // fitted values
predict ehat, residuals // residuals
rvfplot, yline(0) // residual-vs-fitted plot
```

Like `margins`, every line reads the *last-fitted* model, so line 1 re-fits it. The rest do four jobs — a joint test, two model checks, and the residual plot.

`test mpg weight_t` **— a joint hypothesis.** `test` asks whether several coefficients are *jointly* zero — here $H_0: \beta_{\texttt{mpg}} = \beta_{\texttt{weight\_t}} = 0$ — and reports a joint F test:

```
 ( 1)  mpg = 0
 ( 2)  weight_t = 0

       F(  2,    70) =   34.77
            Prob > F =    0.0000
```

$F(2, 70) = 34.77$, $p < 0.0001$: the two are jointly highly significant. (Each coefficient's *individual* test is already in the regression table; `test` is for *joint* restrictions — e.g. "are all the event-time dummies zero?")

`estat hettest` **— is the variance constant?** The Breusch–Pagan test, with H_0 = constant variance:

```
Breusch-Pagan/Cook-Weisberg test for heteroskedasticity
H0: Constant variance

    chi2(1) =   6.34
Prob > chi2 = 0.0118        <- reject: variance is NOT constant
```

$p = 0.012$ rejects it: the errors *are* heteroskedastic — exactly why every model in this chapter used `vce(robust)`.

`estat vif` **— is there multicollinearity?** Variance inflation factors measure how much each coefficient's variance is inflated by correlation among the regressors; the rule of thumb is that `VIF` > 10 is worrying:

```
    Variable |       VIF       1/VIF
-------------+----------------------
         mpg |      2.96    0.337297
    weight_t |      3.86    0.258809
   1.foreign |      1.59    0.627761
-------------+----------------------
    Mean VIF |      2.81
```

All well under 10 — no collinearity problem here.

`predict` **and** `rvfplot` **— eyeball the residuals.** `predict yhat` and `predict ehat, residuals` write the fitted values and residuals as new variables (no console output — they are the *inputs* to the plot). `rvfplot` then draws residuals against fitted values; the fan-out below — a wider spread at higher fitted values — is the same heteroskedasticity `estat hettest` just flagged, now visible.

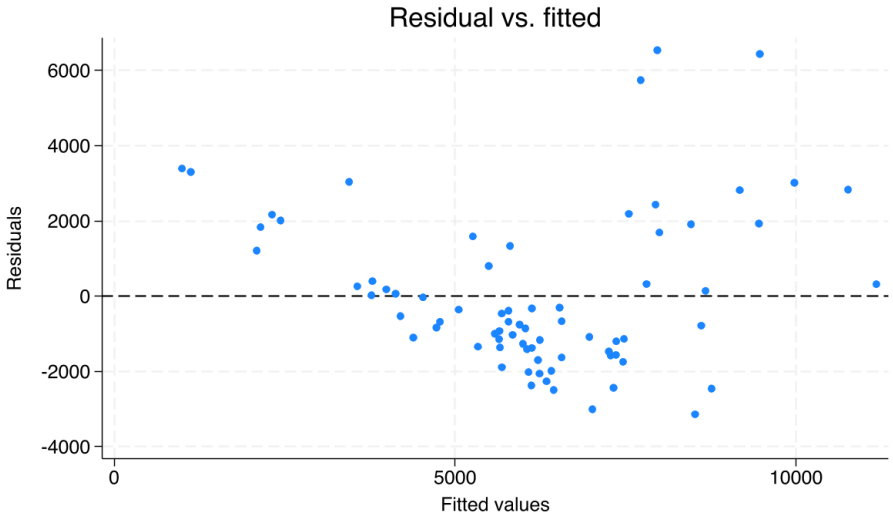


Figure 4.1: `rvfplot` — residuals against fitted values, the first thing to eyeball after OLS.

Chapter 5

Panel data, DiD, event studies: `04_panel_did_eventstudy.do`

The workflow. Declare a firm-by-year panel, estimate difference-in-differences three ways (pooled, firm FE, two-way FE), and trace the dynamic treatment path with an event study.

5.1 Declare the panel

```
xtset firm year // declare: panel id = firm, time = year
xtdescribe // is it balanced? how many units, periods?
```

`xtset` tells Stata the data is a panel — which variable is the unit, which is time. After this, the `xt*` commands and the time operators (`L.`, `F.`) work.

R / pandas `xtset firm year` ≈ `df.set_index(["firm","year"])` before `PanelOLS`.

5.2 Difference-in-differences three ways

```
regress y i.treat##i.post, vce(cluster firm) // pooled OLS
xtreg   y i.treat##i.post, fe vce(cluster firm) // firm fixed effects
reghdfe y 1.treat#1.post, absorb(firm year) vce(cluster firm) // two-way FE
```

- `i.treat##i.post` — the DiD design: the coefficient on the `treat`×`post` cell (`1.treat#1.post`) *is* the effect.
- `xtreg ..., fe` — adds unit (firm) fixed effects.
- `reghdfe ..., absorb(firm year)` — absorbs *many* fixed effects (firm *and* year) efficiently — the modern workhorse.
- `vce(cluster firm)` — standard errors clustered by firm.

The three lines estimate the same DiD coefficient β under progressively richer fixed effects — none, firm only, then firm and year:

$$\text{(1)} \quad y_{it} = \beta_0 + \beta_1 \, \texttt{treat}_i + \beta_2 \, \texttt{post}_t + \beta \, (\texttt{treat}_i \times \texttt{post}_t) + \varepsilon_{it},$$
$$\text{(2)} \quad y_{it} = \alpha_i + \beta_2 \, \texttt{post}_t + \beta \, (\texttt{treat}_i \times \texttt{post}_t) + \varepsilon_{it},$$
$$\text{(3)} \quad y_{it} = \alpha_i + \gamma_t + \beta \, (\texttt{treat}_i \times \texttt{post}_t) + \varepsilon_{it}.$$

The firm effects α_i absorb the time-constant `treat` main effect — which is why line 3 can write `1.treat#1.post`, the interaction alone — and the year effects γ_t absorb `post`. All three return the DiD estimate ≈ 1.97 (the planted truth is an average effect of 2.0). `esttab` shows the DiD coefficient is identical across all three — and right on target:

```
                           (1)             (2)             (3)
                     Pooled OLS        Firm FE      Two-way FE
------------------------------------------------------------------------
treat=1 # post=1          1.973***        1.973***        1.973***
                         (0.104)         (0.104)         (0.104)
------------------------------------------------------------------------
Observations                1800            1800            1800
------------------------------------------------------------------------
* p<0.10, ** p<0.05, *** p<0.01    (true average effect = 2.0)
```

Gotcha under firm fixed effects the time-invariant `treat` main effect *drops* — it is collinear with the firm dummies. Only the interaction survives, which is exactly the desired result.

R / pandas `reghdfe` ≈ `PanelOLS` / `feols(... | firm + year)`; `vce(cluster)` ≈ cluster-robust SEs.

5.3 The event study

```
forvalues k = -5/6 { // loop over event time
    if `k'==-1 continue // omit the reference period (eve of treatment)
    local j = `k' + 6 // a legal variable-name index
    generate evt`j' = (treat==1 & rel==`k') // a dummy for each event time
}
reghdfe y evt*, absorb(firm year) vce(cluster firm)
coefplot, keep(evt*) vertical yline(0) // plot the coefficient path
```

This introduces two new ideas: `forvalues` *loops* and *locals*. `` `k' `` is a local macro holding the loop counter; `evt`j'` builds a variable name like `evt7`. The loop creates one dummy per event-time but `continues` past $k = -1$ — the single *reference period*. An effect cannot be estimated for *every* event time at once: the full set of dummies sums to the treated indicator (which the firm effects absorb), so one must be the dropped baseline, pinned to zero, and every other coefficient is read *relative to* it. The natural choice is $k = -1$, the period just before treatment. Keeping the *leads* in the loop (the negative k, not just $k \geq 0$) is deliberate: with no treatment yet they should be zero, so they act as a placebo test — the built-in pre-trend check an event study provides that a single DiD number can't. `reghdfe` then estimates the rest jointly and `coefplot` draws the path: **leads flat near zero** (no pre-trend), **lags rising** (the effect growing).

Tip a local macro `` `k' `` holds *text or a number*, not data — it's how a value is carried from one line to the next, and the backbone of every loop.

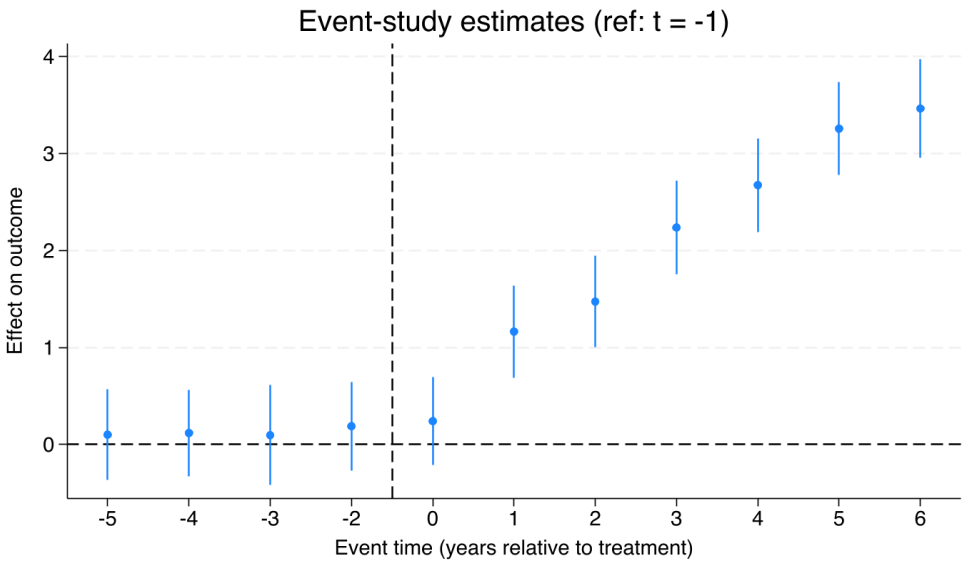


Figure 5.1: The event study from `coefplot`: flat pre-trend, rising post-treatment effect.

Chapter 6

Time series: 05_timeseries.do

The workflow. Declare a monthly series, use the lag/difference operators, inspect its autocorrelation, and fit a lagged regression with Newey–West (HAC) standard errors.

6.1 Declare a time series

```
tsset mdate // declare: time variable = mdate (a monthly date)
```

`tsset` is to time series what `xtset` is to panels: it tells Stata which variable is time, so the lag/lead-/difference operators work and gaps are understood.

6.2 Lag, difference, lead operators

```
generate dy   = D.y // first difference: y(t) - y(t-1)
generate y_l1 = L.y // first lag:        y(t-1)
```

After `tsset`, the prefixes "just work": `L.` = lag, `D.` = difference, `F.` = lead. No manual shifting, and they respect the time gaps.

R / pandas `L.x` ≈ `x.shift(1)`; `D.x` ≈ `x.diff()`.

6.3 Visualise: the series and its autocorrelation

```
tsline y, tline(2015m1, lpattern(dash)) // line plot, with a marker at the break
ac y // autocorrelation function
```

6.4 HAC standard errors (Newey–West)

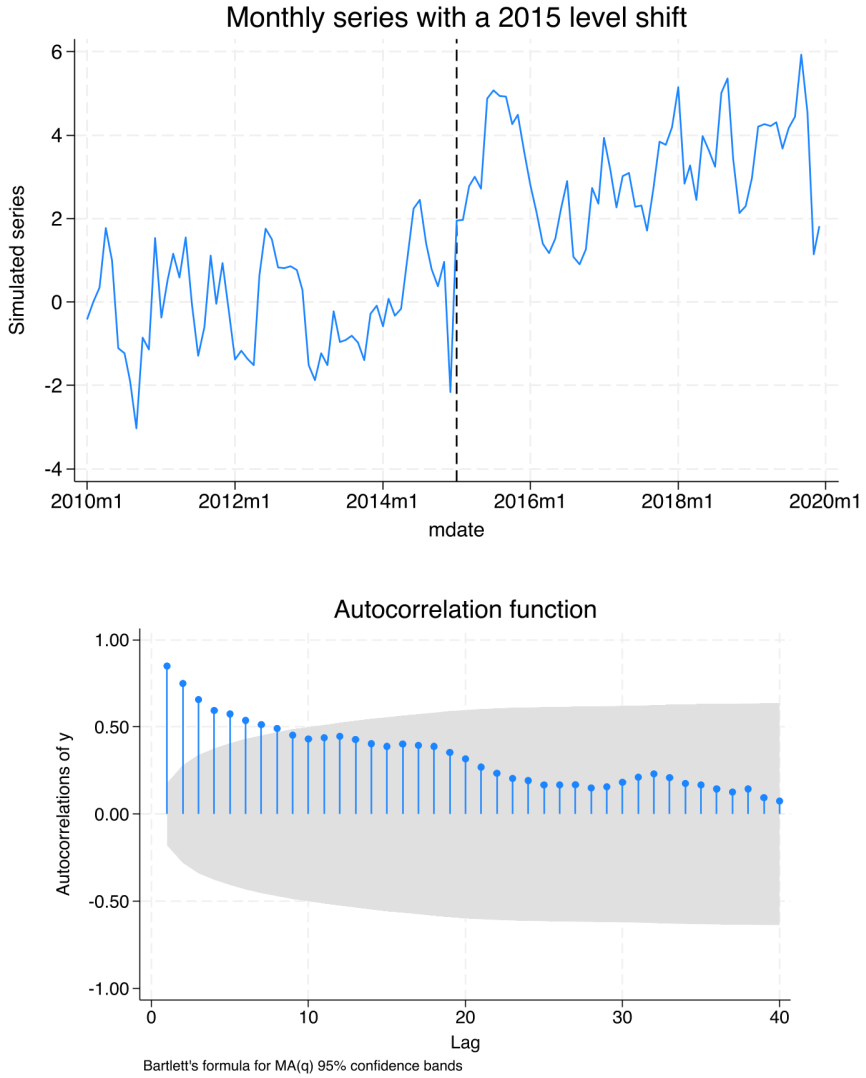


Figure 6.1: `tsline` (the series with its 2015 level shift) and `ac` (slowly-decaying autocorrelation — the signature of persistence).

```
newey y L.y, lag(3) // AR(1) regression, HAC SEs up to 3 lags
regress y L.y i.post, vce(robust) // add the level-break indicator
```

Both lines fit a first-order autoregression — the second adds the level-break dummy:

$$y_t = c + \phi\, y_{t-1} + \delta\, \mathbf{1}[t \geq \texttt{2015m1}] + \varepsilon_t,$$

with the lag operator `L.y` standing in for y_{t-1} and `i.post` for the break indicator $\mathbf{1}[t \geq \texttt{2015m1}]$ (`newey` omits the δ term, `regress` keeps it). `newey` fits the regression but reports Newey–West standard errors — robust to serial correlation (and heteroskedasticity) up to `lag()` periods. The series was built with true persistence $\phi = 0.6$ plus a level shift: regressing on the lag *without* the break gives an inflated $\hat{\phi} \approx 0.85$; *adding* `i.post` recovers $\hat{\phi} \approx 0.57$. `newey` *without* the break inflates the lag coefficient to 0.85:

```
Regression with Newey-West standard errors      Number of obs = 119
          y | Coefficient  std. err.      t    P>|t|
------------+------------------------------------------------
        L1. |   .8492528    .048781    17.41   0.000     <- biased up by the 2015 break
      _cons |   .2527267   .1230766     2.05   0.042
```

Adding the break indicator `i.post` recovers $\hat{\phi} \approx 0.57$ (close to the true 0.6):

```
          y | Coefficient  std. err.      t    P>|t|
------------+------------------------------------------------
        L1. |   .5724343   .0785252     7.29   0.000     <- ~0.6, the true persistence
     1.post |   1.458096   .3051262     4.78   0.000     <- the level shift, now modelled
      _cons |  -.0528752   .1286068    -0.41   0.682
```

Gotcha an unmodelled level break makes a series look more persistent than it is. If $\hat{\phi}$ seems suspiciously close to 1, check for a structural shift before concluding "near unit root".

Chapter 7

Importing public data:
06_import_public_data.do

The workflow. Get data *into* Stata three ways: round-trip a CSV, import a real public series from FRED and turn it into a usable time series, and (for reproducible work) the official API route.

7.1 The CSV round-trip

```
export delimited using "output/_cars.csv", replace // write a CSV
import delimited "output/_cars.csv", varnames(1) clear // read it back
```

`import`/`export delimited` are the CSV workhorses. This is a *throwaway round-trip* — we write the built-in `auto` data out and read it right back, just to exercise the commands; the data is irrelevant here, the file I/O is the point. `varnames(1)` says the first row holds the column names.

R / pandas `import delimited` ≈ `pd.read_csv` / `read_csv`; `export delimited` ≈ `to_csv`.

7.2 A real FRED series, made usable

```
import delimited "data/UNRATE.csv", varnames(1) clear
rename observation_date datestr
generate obsdate = date(datestr, "YMD") // FRED dates "YYYY-MM-DD" -> numeric date
format obsdate %td
generate mdate = mofd(obsdate) // collapse the date to monthly
format mdate %tm
tsset mdate // declare it as a monthly time series
```

This is the **universal recipe** for any downloaded series: read it, parse the date string into a numeric date with `date(...,"YMD")`, give it a display `format`, reduce to the right frequency (`mofd()` = "month of date"), and `tsset` it. After that, everything in Chapter 6 applies.

`summarize unrate` and `list` confirm the import worked:

```
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      unrate |        940    5.659362    1.704323        2.5       14.8

     +---------------------+      <- list ... in -3/L  : the three most recent months
     |    datestr   unrate |
     |---------------------|
939. | 2026-03-01      4.3 |
940. | 2026-04-01      4.3 |
941. | 2026-05-01      4.3 |
     +---------------------+
```

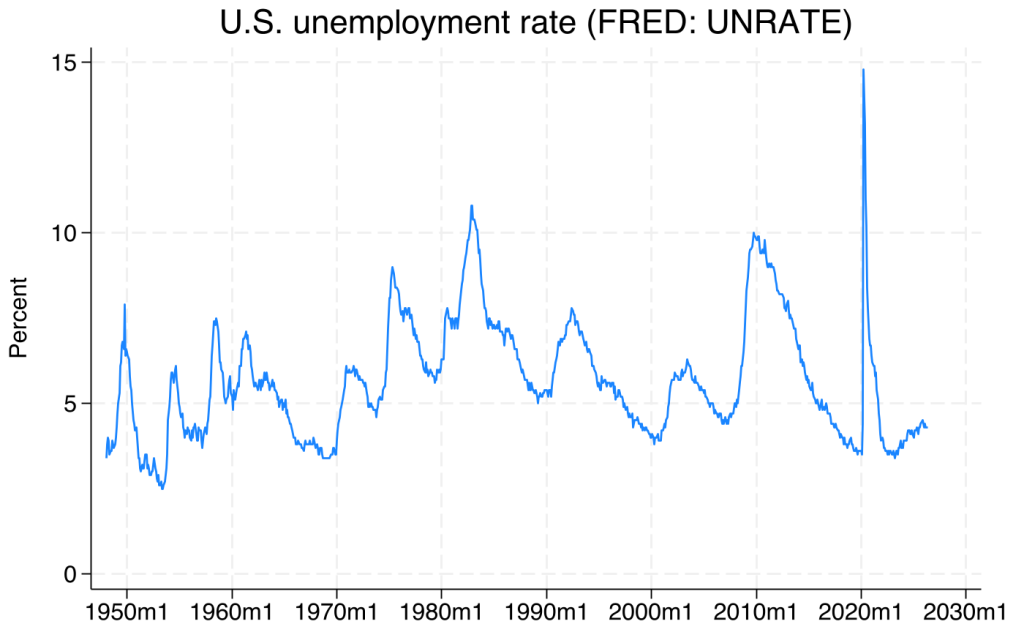


Figure 7.1: The imported FRED series (`tsline`) — U.S. unemployment, monthly since 1948.

Tip memorise the date pattern `date()` → `format` → `mofd()` → `tsset`. Nine times out of ten, the hardest part of using public data is turning the date *string* into something Stata can plot.

7.3 The live API route (`import fred`)

```
set fredkey YOUR_KEY, permanently // once: free key at fredaccount.stlouisfed.org
import fred UNRATE, clear          // pull the series live, straight from FRED's API
```

Stata has a *built-in* FRED importer that pulls series directly over the API — the most reproducible route, since there's no manual download step. The free key is set *once* with `set fredkey` (it lives in the Stata config, never in the script); after that `import fred` just works. It even provides a ready-made numeric date (`daten`), skipping the `date()` parsing of the previous section:

```
. import fred UNRATE, clear
Series ID                   Nobs    Date range               Frequency
UNRATE                      940     1948-01-01 to 2026-05-01  Monthly
```

This portfolio's script runs exactly this (keeping the CSV of the previous section as an offline fallback), so the unemployment figure above is a genuine live pull.

R / pandas `import fred` ≈ `pandas_datareader.get_data_fred` / `fredapi`.

Chapter 8

Instrumental variables: `07_iv_2sls.do`

The workflow. Show that OLS is biased when a regressor is endogenous, fix it with 2SLS, run the standard IV validity diagnostics, and refit absorbing high-dimensional fixed effects. The data are simulated with a *known* true slope of 2.0, so each estimator can be checked against it.

8.1 OLS (biased) vs. 2SLS

```
1 regress y x, vce(robust) // OLS -- biased: x is endogenous
2 ivregress 2sls y (x = z1 z2), vce(robust) first // 2SLS, showing the first stage
```

Line 1, `regress y x`, fits OLS on the *raw* regressor — which is the very problem here, since `x` is correlated with the error:

$$\text{(1) OLS:} \quad y_i = \beta_0 + \beta_1 \, \mathtt{x}_i + \varepsilon_i.$$

Line 2, `ivregress 2sls y (x = z1 z2)`, runs *two stages*: the endogenous regressor and its instruments go inside the parentheses, `(endogenous = instruments)`, and the `first` option also prints stage one. Stage one regresses `x` on the instruments; stage two puts the fitted $\widehat{\mathtt{x}}$ in place of the contaminated `x`:

$$\begin{aligned} \text{first stage:} \quad & \widehat{\mathtt{x}}_i = \hat{\pi}_0 + \hat{\pi}_1 \, \mathtt{z1}_i + \hat{\pi}_2 \, \mathtt{z2}_i, \\ \text{(2) 2SLS:} \quad & y_i = \beta_0 + \beta_1 \, \widehat{\mathtt{x}}_i + \varepsilon_i. \end{aligned}$$

Result: OLS returns ≈ 2.17 (biased upward by the confounder), 2SLS returns ≈ 1.96 — back on the true 2.0 once the contaminated variation in `x` is purged.

R / pandas `ivregress` ≈ `linearmodels.IV2SLS`
the `(x = z1 z2)` syntax means "x instrumented by `z1`, `z2`".

8.2 The three IV diagnostics

```
1 quietly ivregress 2sls y (x = z1 z2), vce(robust) // re-fit
2 estat firststage // weak-instrument F  (rule of thumb: > 10)
3 estat endogenous // is x actually endogenous?
4 estat overid // are the extra instruments valid?
```

Each `estat` reads the last 2SLS fit, so we re-fit it first. These are exactly the questions a careful reader will ask: are the instruments *strong* (first-stage F), is `x` *really* endogenous (Durbin–Wu–Hausman), and — with more instruments than endogenous regressors — are they *jointly valid* (overidentification/Hansen J)?

Gotcha a low first-stage F (weak instruments) makes 2SLS *worse* than OLS, not better. Always report it. The popular "$F > 10$" is a rule of thumb, not a guarantee.

8.3 Efficient GMM

```
ivregress gmm y (x = z1 z2), vce(robust) // efficient two-step GMM
```

2SLS is itself GMM with a particular weight matrix; `ivregress gmm` swaps in the *optimal* weight matrix (robust to heteroskedasticity), so it is *more efficient* (tighter standard errors) when the errors are heteroskedastic. With the homoskedastic errors here it simply matches 2SLS:

```
           y | Coefficient  std. err.      z    P>|z|
-------------+----------------------------------------------
           x |   1.958975   .027070    72.37   0.000     <- ~1.96, same as 2SLS
```

The over-identification test `ivregress` reports for GMM is Hansen's J — the same validity check as `estat overid` above.

R / pandas `ivregress gmm` ≈ `linearmodels.IVGMM`; remember 2SLS ⊂ GMM.

8.4 IV with high-dimensional fixed effects

```
ivreghdfe y (x = z1 z2), absorb(grp) vce(robust)
```

`ivreghdfe` marries `ivreg2` and `reghdfe`: it runs the *same* 2SLS — still using the same first-stage $\widehat{x}$ — but *while absorbing* the group fixed effects `grp`:

$$\text{(3) 2SLS+FE:}\quad y_i = \alpha_{g(i)} + \beta_1 \widehat{x}_i + \varepsilon_i \qquad (\alpha_g \text{ absorbed}).$$

It returns ≈ 1.99, again on target. The `esttab` comparison tells the whole story — OLS biased upward, both IV estimators back on the true 2.0:

```
                    (1)             (2)             (3)
             OLS (biased)            2SLS         IV + FE
------------------------------------------------------------
x                 2.173***        1.960***        1.994***
                 (0.015)         (0.027)         (0.020)
------------------------------------------------------------
N                    5000            5000            5000
------------------------------------------------------------
* p<0.10, ** p<0.05, *** p<0.01    (true slope = 2.0)
```

Chapter 9

Binary outcomes: `08_logit_probit.do`

The workflow. Model a 0/1 outcome (is a car foreign?) three ways, then — the key lesson — interpret it correctly: raw logit/probit coefficients are *not* effects; the average marginal effect is.

9.1 Three models for a 0/1 outcome

```
regress foreign mpg weight_t price_k, vce(robust) // linear probability model
logit   foreign mpg weight_t price_k, vce(robust) // logistic
probit  foreign mpg weight_t price_k, vce(robust) // normal
```

`logit` and `probit` push a linear index through an S-shaped curve so predicted probabilities stay in $[0, 1]$; the LPM (`regress`) is the straight-line approximation. All three wrap the *same* linear index $x_i'\beta = \beta_0 + \beta_1\,\texttt{mpg} + \beta_2\,\texttt{weight\_t} + \beta_3\,\texttt{price\_k}$ in a different link:

$$\begin{aligned} \texttt{regress (LPM)}: &\ P(\texttt{foreign}_i = 1) = x_i'\beta, \\ \texttt{logit}: &\ P(\texttt{foreign}_i = 1) = \Lambda(x_i'\beta), \\ \texttt{probit}: &\ P(\texttt{foreign}_i = 1) = \Phi(x_i'\beta), \end{aligned}$$

with Λ the logistic CDF and Φ the normal CDF. Because the links differ, so do the coefficient scales — which is exactly why the raw numbers aren't comparable.

Gotcha the raw coefficients are on *different scales* across the three models — a logit coefficient is roughly $1.6\times$ its probit twin and unrelated to the LPM. **Never compare the raw numbers directly.** Compare marginal effects (below).

9.2 Odds ratios

```
logit foreign mpg weight_t price_k, or // report exp(beta) instead of beta
```

The `or` option reports *odds ratios* e^{β}. For `mpg` it's ≈ 0.886: each extra mpg multiplies the odds of being foreign by 0.886 (i.e. lowers them $\sim$11%).

9.3 Average marginal effects — the comparable quantity

```
quietly logit foreign mpg weight_t price_k
margins, dydx(*) post // average marginal effect of every regressor
estimates store ame_logit
```

`margins , dydx(*)` averages the slope of the probability curve over the sample — a number on the same footing as an OLS coefficient. The logit AME of `mpg` is ≈ -0.009, essentially identical to the probit AME (-0.009): the textbook "logit ≈ probit" result. The `post` option leaves the AMEs as the active estimates so `esttab` can tabulate them. The two AME columns are nearly identical — the "logit ≈ probit" result, in numbers:

```
               Logit AME      Probit AME
--------------------------------------------
mpg              -0.0090         -0.0094
                (0.0066)        (0.0067)
weight_t         -0.5069***      -0.4974***
                (0.0550)        (0.0510)
price_k            0.0686***       0.0675***
                (0.0136)        (0.0131)
--------------------------------------------
```

R / pandas `margins , dydx(*)` ≈ `.get_margeff()` (statsmodels). Always the right way to report a logit/probit.

9.4 Predicted probabilities, plotted

```
quietly logit foreign mpg weight_t price_k
margins, at(mpg=(12(4)40)) // predicted P at mpg = 12, 16, ..., 40
marginsplot // plot it
```

`margins at()` traces predicted $P(\text{foreign})$ across the range of `mpg`, and `marginsplot` draws it — the most intuitive way to show what a nonlinear model implies.

9.5 In-sample fit

```
quietly logit foreign mpg weight_t price_k
estat classification // sensitivity / specificity at a 0.5 cutoff
lroc, nograph // area under the ROC curve
```

`estat classification` gives the sensitivity/specificity table; `lroc` reports the ROC area — standard summaries of how well a binary model separates the classes:

```
Sensitivity     Pr( +| D)   86.36%      <- caught 86% of the foreign cars
Specificity     Pr( -|~D)   90.38%      <- caught 90% of the domestic cars

Area under ROC curve   =   0.9572        <- 1.0 = perfect separation; 0.96 is excellent
```

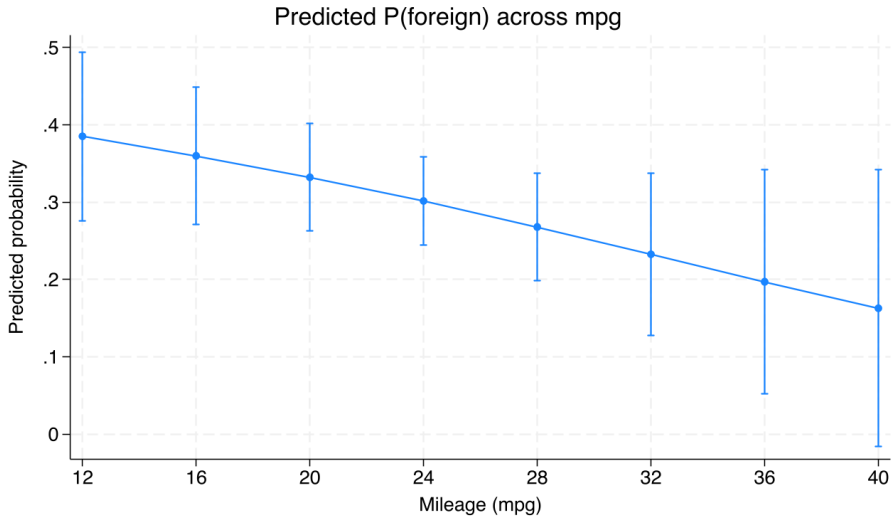


Figure 9.1: `marginsplot`: predicted probability a car is foreign, across `mpg`.

Chapter 10

Staggered-adoption DiD: `09_staggered_did_csdid.do`

The workflow. Build a panel where units are treated in *different* years, estimate clean group-time effects with Callaway–Sant'Anna's `csdid`, aggregate them, and show how the naive two-way FE estimate is biased on the very same data.

10.1 The cohort variable

The data simulate three groups: first-treated in 2014, first-treated in 2017, and never-treated. The key variable is the *cohort*:

```
generate g = 0 // 0 = never treated (the csdid convention)
replace  g = 2014 if id <= 100 // early cohort
replace  g = 2017 if id > 100 & id <= 200 // late cohort
```

Gotcha `csdid` expects the cohort variable to be the *year of first treatment*, with `0` for never-treated — not a 0/1 "treated" dummy. Getting this coding right is the whole game.

10.2 The Callaway–Sant'Anna estimator

```
csdid y, ivar(id) time(year) gvar(g) method(dripw)
```

`csdid` estimates a clean 2×2 DiD for every (cohort, time) pair, always using not-yet-treated or never-treated units as controls. `ivar()`=unit, `time()`=calendar time, `gvar()`=the cohort, `method(dripw)`=doubly-robust.

10.3 Aggregating the group-time effects

```
quietly csdid y, ivar(id) time(year) gvar(g) method(dripw) // re-fit
estat simple // one overall ATT
estat group // ATT by cohort
estat event // the dynamic / event-study path
csdid_plot, title("Callaway-Sant'Anna event study")
```

The raw output is one effect per (cohort, time); these `estat` commands *average* them in different ways. `csdid_plot` draws the event-study aggregation. `estat simple` collapses them to a single overall ATT — ≈ 1.49, the true average:

```
Average Treatment Effect on Treated
             | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
         ATT |   1.494672    .112354    13.30   0.000     1.274462    1.714881
```

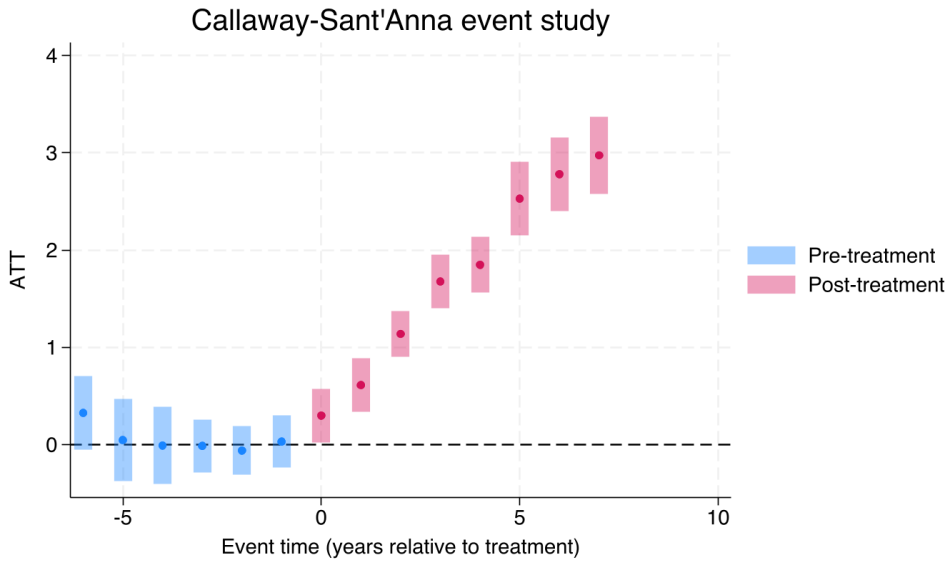


Figure 10.1: `csdid_plot`: the dynamic effect, built only from clean comparisons.

10.4 Why not just two-way FE?

```
generate byte treated_now = (g>0 & year>=g)
reghdfe y treated_now, absorb(id year) vce(cluster id)
```

Where `csdid` aggregates clean group-time effects $ATT(g,t)$ (the overall ATT above), this naive regression forces everything onto a single post-onset dummy:

$$y_{it} = \alpha_i + \gamma_t + \beta^{\text{TWFE}}\,\texttt{treated\_now}_{it} + \varepsilon_{it}.$$

On the *same* data, that single β^{TWFE} returns ≈ 1.01 while `csdid` returns ≈ 1.50 (the true average). The gap is the staggered-timing bias: TWFE quietly uses already-treated units as controls.

```
           y | Coefficient  std. err.      t    P>|t|
 treated_now |   1.008754   .0730061    13.82   0.000     <- 1.01: biased DOWN vs the true ~1.5
```

Gotcha when treatment timing is staggered *and* effects grow over time, distrust the naive `reghdfe`/`xtreg` DiD — not `csdid`. This script exists to make that bias visible.

Chapter 11

Regression discontinuity: 10_rdd_regression_discontinuity.do

The workflow. Simulate a sharp-RD dataset, estimate the jump with `rdrobust`, inspect the bandwidth, draw the canonical plot, and test for manipulation — all recovering a planted jump of 2.0.

11.1 The sharp design

```
generate x = runiform(-1,1) // running variable; cutoff at 0
generate byte d = (x >= 0) // sharp treatment: d = 1[x >= 0]
generate y = 3 + 1.5*x + 0.5*x^2 + 2.0*d + rnormal(0,0.5) // true jump = 2.0
```

`x` is the running variable, treatment switches on deterministically at the cutoff `0` (`d`), and the outcome has a built-in $+2.0$ step there — the effect the estimator must recover. (The uniform `x` makes the density continuous at the cutoff, so the manipulation test should not reject.)

11.2 The estimate

```
rdrobust y x, c(0)
```

`rdrobust` fits a local-linear regression on each side of the cutoff `c(0)`, inside a data-chosen MSE-optimal bandwidth, and reports a bias-corrected robust confidence interval (the Calonico–Cattaneo–Titiunik method). It returns $\hat{\tau} = 1.938$ with a 95% CI of $[1.770,\ 2.082]$ — on the planted 2.0.

Under the hood it fits one line each side of the cutoff and differences them there:

$$\begin{aligned} \text{below } (\mathtt{x} < 0):&\ \ E[\mathtt{y} \mid \mathtt{x}] = \alpha_- + \beta_- \mathtt{x}, \\ \text{above } (\mathtt{x} \geq 0):&\ \ E[\mathtt{y} \mid \mathtt{x}] = \alpha_+ + \beta_+ \mathtt{x}, \end{aligned}$$

so the reported $\hat{\tau} = \alpha_+ - \alpha_-$ is exactly the jump at $\mathtt{x} = 0$.

`rdrobust` prints the bandwidth, kernel, and the bias-corrected estimate in one table:

```
Sharp RD estimates using local polynomial regression.

      Cutoff c = 0 | Left of c  Right of c       Number of obs =   3000
     Number of obs |      1556        1444        Kernel = Triangular
       BW est. (h) |     0.374       0.374        BW type = mserd
--------------------------------------------------------------------------------
                    | Point         | Robust Inference
                    | Estimate      | z-stat   P>|z|     [95% Conf. Interval]
--------------------+-----------------------------------------------------------
         RD Effect |  1.9378       | 24.24    0.000      1.7701     2.0815
--------------------------------------------------------------------------------
```

11.3 Bandwidth and the picture

```
rdbwselect y x, c(0) all // the data-driven bandwidth choices
rdplot y x, c(0) genvars hide // binned means + a polynomial fit each side
```

`rdbwselect` reports the optimal bandwidths; `rdplot` draws the canonical RD picture (here regenerated in colour, with the estimated jump annotated).

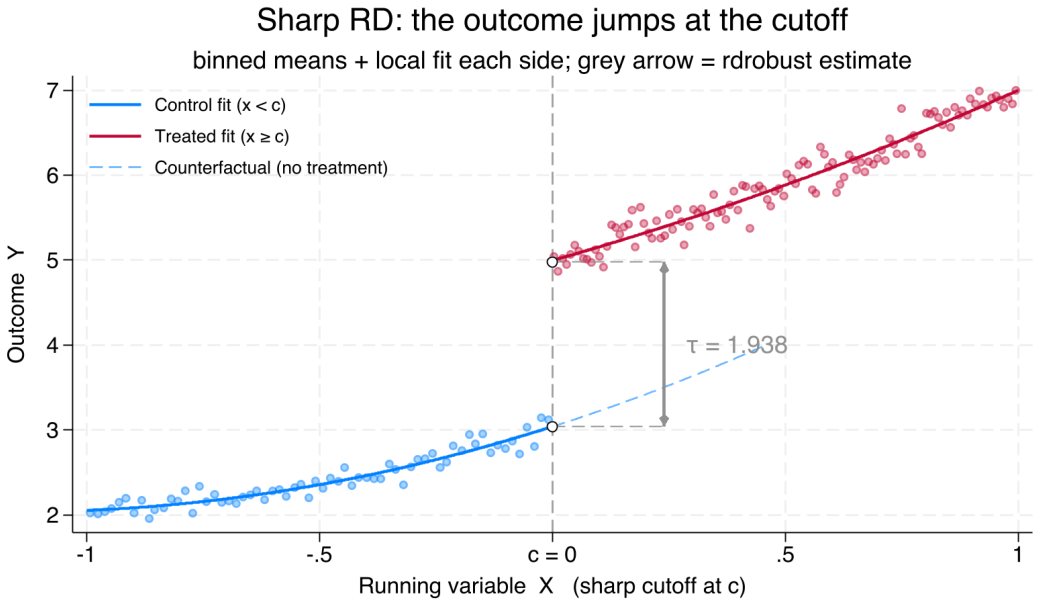


Figure 11.1: The RD plot: binned means and a fit each side; the grey arrow is the `rdrobust` jump.

11.4 The manipulation test

```
rddensity x, c(0)
```

`rddensity` runs the manipulation/density test: is the running variable's density continuous at the cutoff, or do units pile up just past it? Here $p = 0.95$ — no bunching, so the design is clean.

```
RD Manipulation test using local polynomial density estimation.

            Method |      T          P>|T|
            Robust |    0.0594      0.9527     <- p = 0.95: fail to reject -> no manipulation
```

Gotcha RD's credibility rests on units *not* being able to sort across the cutoff. Always run `rddensity`; a rejection is a red flag that the design is broken.

Chapter 12

Synthetic control:
11_synthetic_control.do

The workflow. For a single treated unit, build a "synthetic" version from a weighted average of donors that matches its pre-treatment path, read the effect as the post-treatment gap, and do inference by placebo.

12.1 Fit the synthetic control

```
synth y y(5) y(10) y(15) y(20) u1 u2, ///
    trunit(1) trperiod(21) keep("output/synth_results.dta", replace)
```

`synth` chooses non-negative weights on the donor units so the weighted average tracks the treated unit before treatment. `trunit(1)`=the treated unit, `trperiod(21)`=when treatment starts; the listed variables (lagged outcomes `y(5)...y(20)` and predictors `u1 u2`) are what the match is built on. `keep()` saves the treated and synthetic paths. In symbols, `synth` picks weights $w_j \geq 0$ with $\sum_j w_j = 1$; the synthetic unit is the weighted donor average $\sum_j w_j \, \mathrm{y}_{jt}$, and the estimated effect is the post-period gap

$$\hat{\tau}_t = \mathrm{y}_{1t} - \sum_j w_j \, \mathrm{y}_{jt}, \qquad t \geq 21.$$

12.2 Inference by placebo

```
synth_runner y y(5) y(10) y(15) y(20) u1 u2, ///
    trunit(1) trperiod(21) gen_vars
```

With only one treated unit there are no conventional standard errors, so `synth_runner` re-runs the whole procedure pretending *each donor* was the treated one, building a distribution of "placebo" gaps. If the real unit's gap is extreme against that distribution, the effect is significant. `gen_vars` stores each unit's gap for plotting. `synth_runner` reports each post-period effect with its placebo p-value — all ≈ 3 (the planted effect), all significant against the donor distribution:

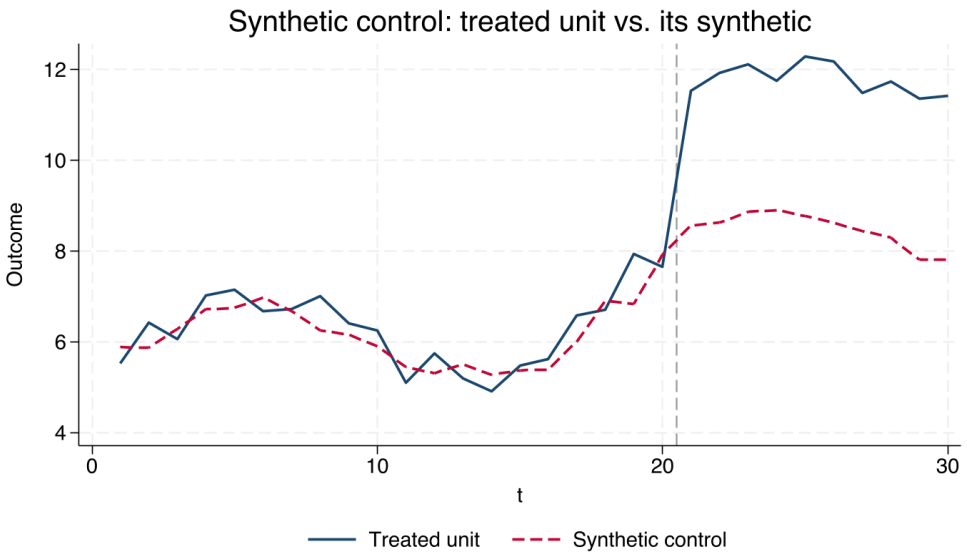


Figure 12.1: `synth`: the treated unit vs. its synthetic twin — they track closely before treatment, then diverge by the planted $+3$.

```
             | estimates      pvals
-------------+--------------------
          c1 |  2.960709          0
          c2 |  3.308788          0
          c3 |  3.250508          0
          c4 |  2.848906          0
          c5 |   3.51174          0
          c6 |   3.54869          0
```

R / pandas there is no `statsmodels/feols` one-liner for this — synthetic control is a Stata (and R `Synth`) speciality. Knowing it signals modern applied-micro fluency.

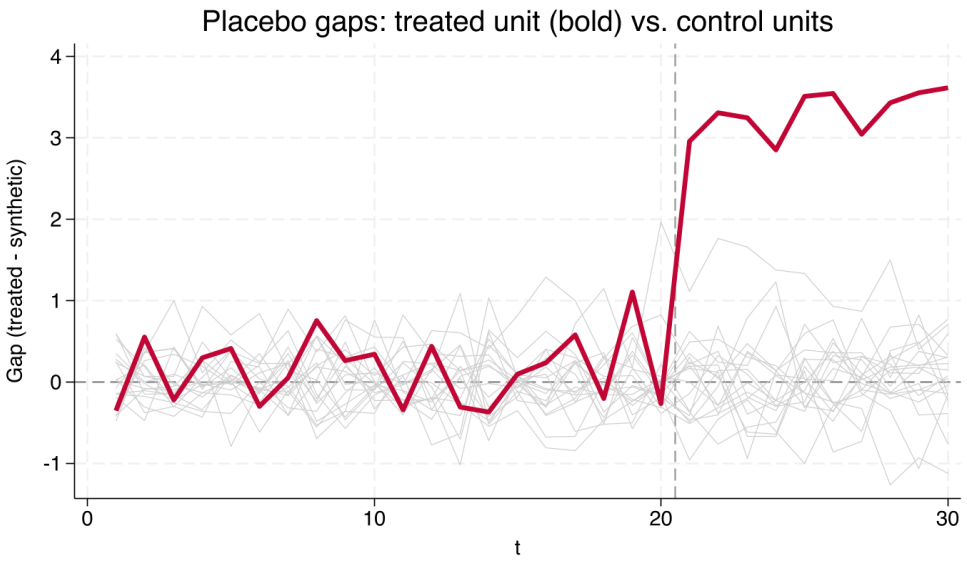


Figure 12.2: Placebo gaps: the treated unit (bold) stands out from the donor placebos (grey).

Chapter 13

Dynamic panel GMM:
12_dynamic_panel_gmm.do

The workflow. A lagged dependent variable sitting next to a fixed effect is endogenous — the classic dynamic-panel problem. Show that pooled OLS and within FE bracket the true persistence from above and below, then recover it with Arellano–Bond and Blundell–Bond GMM. True $\rho = 0.6$.

13.1 Building the dynamic panel

```
generate y = .
replace y = alpha + x + eps if t==1                  // starting value
forvalues s = 2/20 {
    replace y = 0.6*L.y + x + alpha + eps if t==`s'  // y_it = 0.6*y_{t-1} + x +
    alpha + e
}
```

The series is built recursively so $y_{it} = \rho\, y_{i,t-1} + \beta\, \mathrm{x}_{it} + \alpha_i + \varepsilon_{it}$ with $\rho = 0.6$. The trouble: $y_{i,t-1}$ contains the unit effect α_i, so the lag is correlated with the error the moment ρ is estimated.

13.2 The bias: OLS too high, FE too low

```
regress y L.y x, vce(cluster id)    // pooled OLS
xtreg   y L.y x, fe vce(cluster id) // within / fixed effects
```

Neither standard estimator works here:

- `regress` (pooled OLS) *overstates* ρ — the omitted α_i loads onto the lag. Here $\hat{\rho} \approx 0.80$.
- `xtreg, fe` *understates* it — demeaning correlates the lag with the error (the Nickell bias), severe when T is small. Here $\hat{\rho} \approx 0.52$.

The truth (0.6) is bracketed between them.

Gotcha the Nickell bias is $O(1/T)$ — it shrinks as the panel lengthens, but with the short panels typical in finance (a handful of years) it is large. Don't trust FE for a dynamic panel.

13.3 Arellano–Bond difference GMM

```
xtabond y x, lags(1) vce(robust) // difference, instrument with lagged levels
estat abond                      // AR(1) should reject, AR(2) should NOT
```

`xtabond` first-differences away α_i, then instruments $\Delta y_{i,t-1}$ with deeper *levels* $y_{i,t-2}, y_{i,t-3}, \ldots$ — valid because they predict the lag yet are uncorrelated with the differenced error. It returns $\hat{\rho} \approx 0.58$, back inside the bracket. `estat abond` is the crucial check:

```
Arellano-Bond test for zero autocorrelation in first-differenced errors
  Order      z      Prob > z
      1   -17.12     0.0000     <- AR(1): expected (differencing induces it)
      2    -0.94     0.3490     <- AR(2): does NOT reject -> instruments valid
```

Gotcha AR(2) is the test that matters. If it *rejects*, the lagged-level instruments are themselves correlated with the error and the estimates are invalid — use deeper lags or rethink.

13.4 Blundell–Bond system GMM

```
xtdpdsys y x, lags(1) vce(robust) // adds a levels equation -> system GMM
```

`xtdpdsys` augments difference GMM with a *levels* equation, instrumented by lagged *differences*. This *system GMM* is more efficient and much better behaved when ρ is close to one (where the deep-lag instruments in differences turn weak). Here it also returns ≈ 0.58.

R / pandas Roodman's `xtabond2` (SSC) is the community standard — with instrument `collapse` and finite-sample corrections — but the built-in `xtabond`/`xtdpdsys` need no install.

Where to go next

Every command in the portfolio is now decoded. A few habits carry the rest of the way:

- `help` **is the first resort.** Typing `help regress` (or any command) opens Stata's excellent built-in documentation, with the full syntax diagram and examples.
- **Retype, don't copy.** Fluency comes from typing, not reading. Open each `.do` file, run it block by block, and change things to see what breaks.
- **One comma, then options.** When a command won't run, 90% of the time it's the `i./c.` prefix, a missing `tsset`/`xtset`, or a stray second comma.
- **For the *econometrics*** behind each method — when to use it, the assumptions, the identifying logic — see the companion *reference* and its Appendix B. This guide taught the *syntax*; that one teaches the *method*.

To do this...	**...reach for**
load / inspect / save	`sysuse`, `describe`, `summarize`, `save`
new variables	`generate`, `replace`, `egen`
join / reshape / aggregate	`merge`, `reshape`, `collapse`
OLS + a real table	`regress , vce(robust)`, `margins`, `esttab`
panel / DiD	`xtset`, `reghdfe`, `coefplot`
time series	`tsset`, `L./D.`, `newey`
instrumental variables	`ivregress`, `ivreghdfe`, `estat firststage`
binary outcomes	`logit/probit`, `margins , dydx(*)`
staggered DiD	`csdid`
regression discontinuity	`rdrobust`, `rddensity`
synthetic control	`synth`, `synth_runner`

Now open Stata, and type.