Organizing Data for Economic Research

Part 2: Handling Data

Brendan M. Price*
UC Davis

November 14, 2019

This presentation includes figures based on joint work with John Coglianese of the Federal Reserve Board. The views expressed in this presentation are those of the authors and do not necessarily represent the views or policies of the Board of Governors of the Federal Reserve System or its staff.

This lecture

Roadmap:

1. Coding practices
2. Data cleaning
3. Data validation
4. Data exploration

A bit more Stata-centric than last time

Recommended reading, redux

Principles, practices, and object lessons:

"Code & Data for the Social Sciences: A Practitioner's Guide"
— Matthew Gentzkow and Jesse M. Shapiro

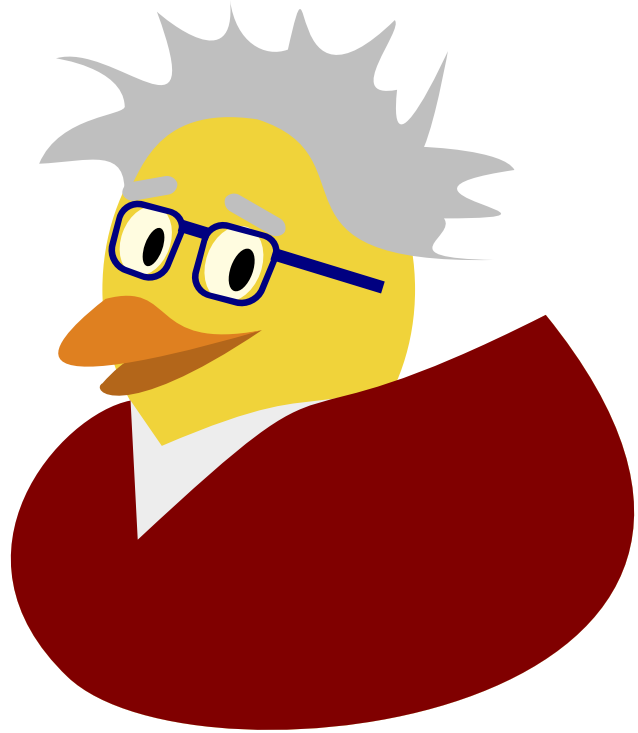
Filling in many of the details:

The Workflow of Data Analysis Using Stata
— J. Scott Long

Nuts and bolts of metrics commands:

Microeconometrics Using Stata
— A. Colin Cameron and Pravin K. Trivedi

coding practices



Writing good code*

1. The better you know the language, the more choices you have.
2. Balance brevity, readability, and computational efficiency. There are real tradeoffs, but stick to the Pareto frontier.
3. Comment heavily, but keep comments up to date.
4. Employ visual cues to clarify logical structure.
5. Automation pays for itself sooner than you expect.
6. Looping is usually better than repetition, but not always.
7. Store parameter values in macros: do not hard-code them.
8. Use consistent naming and typographical conventions.
9. Seek creative solutions, but don't reinvent the wheel.
10. Don't leave code in disrepair unless you'll be back tomorrow.

*This does not come close to being exhaustive. Coding is a lifelong art.

Learn the lingo

The more commands you know, the more versatile you'll be

- Read `help` files often
- Read other people's code
- Read `Statalist` posts
- Talk to classmates/colleagues

(Periodically) skim the Stata reference manuals

- Data management (direct download)
- Programming (direct download)
- Functions (direct download)
- Graphics (direct download)

Google liberally

A cursory Google search turned up this

Data Processing with Stata 15 Cheat Sheet

For more info see Stata's reference manual (stata.com)

Useful Shortcuts

F2 — keyboard buttons describe data

Ctrl + 9 open a new .do file

Ctrl + 8 open the data editor

Ctrl + D highlight text in .do file, then ctrl + d executes it in the command line

clear delete data in memory

At Command Prompt

PgUp PgDn scroll through previous commands

Tab autocompletes variable name after typing part

cls clear the console (where results are displayed)

Set up

pwd
print current (working) directory

cd "C:\Program Files (x86)\Stata13"
change working directory

dir
display filenames in working directory

dir *.dta
List all Stata data in working directory

capture log close — underlined parts are shortcuts – use "capture" or "cap"
close the log on any existing do files

log using "myDoFile.txt", **replace**
create a new log file to record your work and results

search mdesc
find the package mdesc to install

ssc install mdesc — packages contain extra commands that expand Stata's toolkit
install the package mdesc; needs to be done once

Import Data

sysuse auto, clear
load system data (Auto data) — for many examples, we use the auto dataset.

use "yourStataFile.dta", **clear**
load a dataset from the current directory

import excel "yourSpreadsheet.xlsx", **/*** — frequently used commands are highlighted in yellow
***/ sheet**("Sheet1") **cellrange**(A2:H11) **firstrow**
import an Excel spreadsheet

import delimited "yourFile.csv", **/***
***/ rowrange**(2:11) **colrange**(1:8) **varnames**(2)
import a .csv file

webuse set "https://github.com/GeoCenter/StataTraining/raw/master/Day2/Data"
webuse "wb_indicators_long"
set web-based directory and load data from the web

Basic Syntax

All Stata commands have the same format (syntax):

[by varlist1**:]** apply the *command* across each unique combination of variables in *varlist1*

command function: what are you going to **do** to *varlists*?

[varlist2] column to apply *command* to

[=*exp*] save output as a new variable

[*if exp*] condition: only apply the function *if* something is true

[in range] apply to specific rows

[weight] apply weights

[using filename] pull data from a file (if not loaded)

[,*options*] special options for *command*

bysort rep78 : summarize price if foreign == 0 & price <= 9000, detail

In this example, we want a *detailed* summary with stats like kurtosis, plus mean and median

To find out more about any command – like what options it takes – type **help** *command*

Basic Data Operations

Arithmetic

\+ add (numbers) combine (strings)

\- subtract

* multiply

/ divide

^ raise to a power

Logic

== tests if something is equal
= assigns a value to a variable

& and

! or ~ not

| or

== equal

!= or ~= not equal

< less than

<= less than or equal to

> greater than

>= greater or equal to

if foreign != 1 & price >= 10000

make	foreign	price
Chevy Colt	0	3,984
Buick Riviera	0	10,372
Honda Civic	1	4,499
Volvo 260	1	11,995

if foreign != 1 | price >= 10000

make	foreign	price
Chevy Colt	0	3,984
Buick Riviera	0	10,372
Honda Civic	1	4,499
Volvo 260	1	11,995

Explore Data

View Data Organization

describe make price
display variable type, format, and any value/variable labels

count
count if price > 5000
number of rows (observations)
Can be combined with logic

ds, has(type string)
lookfor "in."
search for variable types, variable name, or variable label

isid mpg
check if mpg uniquely identifies the data

See Data Distribution

codebook make price
overview of variable type, stats, number of missing/unique values

summarize make price mpg
print summary statistics (mean, stdev, min, max) for variables

inspect mpg
show histogram of data, number of missing or zero observations

histogram mpg, **frequency**
plot a histogram of the distribution of a variable

Browse Observations within the Data

browse or **Ctrl** + **8**
open the data editor

Missing values are treated as the largest positive number. To exclude missing values, ask whether the value is less than ".".

list make price **if** price > 10000 & !**missing**(price) **clist** ... (compact form)
list the make and price for observations with price > $10,000

display price[4]
display the 4th observation in price; only works on single values

gsort price mpg (ascending) **gsort** –price –mpg (descending)
sort in order, first by price then miles per gallon

duplicates report
finds all duplicate values in each variable

assert price**!=.**
verify truth of claim

levelsof rep78
display the unique values for rep78

Change Data Types

Stata has 6 data types, and data can also be missing:

no data **missing** | true/false **byte** | words **string** | numbers **int long float double**

To convert between numbers & strings:

1 **gen** foreignString = **string**(foreign) "1"
tostring foreign, **gen**(foreignString) "1"
decode foreign , **gen**(foreignString) "foreign"

1 **gen** foreignNumeric = **real**(foreignString) "1"
destring foreignString, **gen**(foreignNumeric) "1"
encode foreignString, **gen**(foreignNumeric) "foreign"

recast double mpg
generic way to convert between types

Summarize Data

include missing values — create binary variable for every rep78 value in a new variable, repairRecord

tabulate rep78, **mi gen**(repairRecord)
one-way table: number of rows with each value of rep78

tabulate rep78 foreign, **mi**
two-way table: cross-tabulate number of observations for each combination of rep78 and foreign

bysort rep78**: tabulate** foreign
for each value of rep78, apply the command tabulate foreign

tabstat price weight mpg, **by**(foreign) **stat**(mean sd n)
create compact table of summary statistics — displays stats formats numbers for all data

table foreign, **contents**(mean price sd price) f(%9.2fc) **row**
create a flexible table of summary statistics

collapse (mean) price (max) mpg, **by**(foreign) – replaces data
calculate mean price & max mpg by car type (foreign)

Create New Variables

generate mpgSq = mpg^2 **gen byte** lowPr = price < 4000
create a new variable. Useful also for creating binary variables based on a condition (**generate byte**)

generate id = _n **bysort** rep78**: gen** repairIdx = _n
_n creates a running index of observations in a group

generate totRows = _N **bysort** rep78**: gen** repairTot = _N
_N creates a running count of the total observations per group

pctile mpgQuartile = mpg, **nq** = 4
create quartiles of the mpg data

egen meanPrice = **mean**(price), **by**(foreign) see **help egen** for more options
calculate mean price for each group in foreign

Tim Essam (tessam@usaid.gov) • Laura Hughes (lhughes@usaid.gov) follow us @StataRGIS and @flaneuseks

inspired by RStudio's awesome Cheat Sheets (rstudio.com/resources/cheatsheets)

geocenter.github.io/StataTraining

updated June 2016

Search term: "useful stata commands". Find more "Stata cheat sheets" here.

Some useful Stata commands

Master the workhorse commands

`append`, `assert`, `bysort`, `capture`, `collapse`, `compress`, `confirm`, `decode`/`encode`, `egen`, `expand`, `fillin`, `format`, `joinby`, `label`, `levelsof`, `local`, `merge`, `preserve`/`restore`, `reshape`, `reshape`, `tabstat`, `tempfile`/`tempvar`, `twoway`, `xpose`

Discover commands you may not already know

`adopath`, `clonevar`, `coefplot`, `distinct`, `duplicates`, `estimates`, `estout`/`esttab`, `export`/`import`, `file write`/`file read`, `findit`, `gsort`, `labmask`, `lookfor`, `nlcom`, `notes`, `postfile`/`post`, `recast`, `sample`, `savesome`, `shell`, `spmap`, `ssc`, `timer`, `unab`

Humdrum commands can have unexpected functionality

- `use [varlist] if` can load a subset of vars./obs.
- `describe using` can list variables without loading a dataset
- `display` can do subtle string transformations
- `rename` can batch-rename in surprisingly versatile ways

`ssc install gtools`: lightning-fast versions of key commands*

*Note of thanks to Sergio Correia for `ftools` and Mauricio Cáceres Bravo for `gtools`.

Special notation and hidden gems

Develop proficiency in all of the following:

- System variables (`_n` and `_N`): `help _n`
- Factor variables: `help fvvarlist`
- Time-series operators: `help tsvarlist`
- Date/time variables: `help datetime`
- String functions: `help string_functions`
- Extended macro functions: `help extended_fcn`
- Regular expressions: `regexm()`/`regexs()`/`regexr()`
- Return codes: `return`, `creturn`, `ereturn`
- Matrix commands: `help matrix`, `help mata`
- Graphical templates: `help scheme`

Many of these are easy to overlook

Be brief

Strunk and White, *The Elements of Style*:

> *Omit needless words. Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.*

Concise code is:

- Easier to read
- Easier to debug
- Easier to maintain
- Easier to recycle

...but not overly brief

1. This is good:

```
gen sep_type = .
replace sep_type = 1 if sep == 1 & ump == 1
replace sep_type = 2 if sep == 1 & nlf == 1
```

2. This is more compact, but it's worse:

```
gen sep_type = 1 if sep == 1 & ump == 1
replace sep_type = 2 if sep == 1 & nlf == 1
```

3. The one-line solution is inadvisable:

```
gen sep_type = ump + 2 * nlf if sep == 1
```

4. This is a bit better, but I'd still go with door #1:

```
gen sep_type = (1 * ump) + (2 * nlf) if sep == 1
```

Documentation plays multiple roles

Part 1: a sound file structure offers meta-documentation

Within scripts, documentation serves three roles:

1. Delineate blocks of related code
2. Explain what the code is doing
3. Flag outstanding issues

Given rationale #1, it's okay to state the obvious:

```
* Restrict to the period 1984-2013
keep if inrange(year(dofm(tm)), 1984, 2013)

* Restrict to prime-age observations
keep if inrange(age, 25, 54)
```

Syntax highlighting and documentation are complements

Employ visual cues

Start each script with an explanatory header

```
*-------------------------------------
* Project: "Jabberwocky"
* Author: Lewis Carroll
*
* Description: Survival analysis.
* Last updated: 1871
*-------------------------------------
```

Use comments to delineate sections of code

```
*-------------------------------------
* Check Jabberwock vital signs
*-------------------------------------
```

Indent subordinate code (loops, if-thens, etc.)

```
if jabberwock == "slain" {
     disp "O frabjous day! Callooh! Callay!"
}
```

Parallel structure aids legibility

Misaligned code is hard to read & error-prone

```
gen emp_share = 100 * emp/(emp + unemp + nilf)
gen unemp_share = 10 * unemp/(emp + unemp + nilf)
gen nilf_share = 100 * nilf/(emp + unemp + nilf)
```

Helps to give parallel variables names of equal length

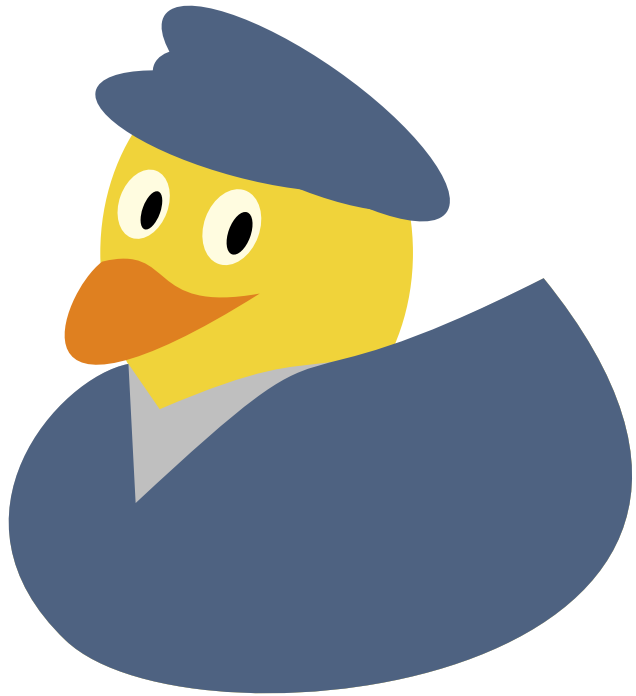
```
gen emp_share = 100 * emp/(emp + ump + nlf)
gen ump_share = 100 * ump/(emp + ump + nlf)
gen nlf_share = 100 * nlf/(emp + ump + nlf)
```

A similar trick makes it easy to see what regressors are included:*

```
reg ‘yvar’ ‘summer’                          ‘if’ ‘wt’, ‘vce’
reg ‘yvar’ ‘summer’ ‘married’                ‘if’ ‘wt’, ‘vce’
reg ‘yvar’ ‘summer’           ‘anykids’ ‘if’ ‘wt’, ‘vce’
reg ‘yvar’ ‘summer’ ‘married’ ‘anykids’ ‘if’ ‘wt’, ‘vce’
```

*Thanks to Monica Rodriguez-Guevara for showing me this trick.

data cleaning



Data cleaning in a nutshell

Here's the gameplan:

1. Store pristine source data in `$projdir/dta/src`.
2. Write code to import source data into Stata.
3. [Weeks of painstaking labor.]
4. Save "clean" `.dta` files.
5. Impose sample restrictions, process further.
6. Save estimation samples.

In some cases, might want to jump right to sample files

Importing data

Wrong approach: manual conversion from source to `.dta`

- Interactive conversion using `StatTransfer`
- Copy-and-paste using Stata's `edit` command (yikes)

Right approach: codified conversion from within a `.do` file

- Creates a paper trail
- No need to save a redundant `.dta` version of original data
- Often need to download updated extracts or similar files

One of these usually suffices: `import`, `insheet`, `infix`

If Stata can't handle it, try using `shell` or `python` code

Unique identifiers

As soon as you load a source file, identify unique identifiers

- Useful command: ``isid [varlist]''
- Code crashes if false, continues if true

Every `.dta` file should have a unique identifier

- Database lingo: primary key
- May consist of multiple variables (e.g., person $\times$ year)
- Gentzkow & Shapiro have a nice discussion

Watch out for numerical issues

- Never store ID variables as `float` or `double`
- Instead: `int`, `long`, or `string`

Eliminate redundancies

Raw data files often contain redundant information

- County-level file: extra rows for state-wide totals
- Excel spreadsheet: column K is average of C and D

Verify redundancies, then eliminate them

- Make sure the data are internally consistent
- Drop anything you can recompute later

Common scenario: raw data *not* internally consistent

- Document whatever inconsistencies you discover
- Deal with inconsistencies on a case-by-case basis

Keep what you need

Raw data files contain lots of clutter

- Variables you'll never use
- Messy or gratuitous value labels
- Early years with fatally incomplete data

Keep only what you (1) need and (2) will take the time to clean

- If you're never going to need it, just drop it
- If you might need it later, either clean it or drop it
- Can always come back later

Main principle: only retain what you can "vouch for"

- Otherwise: liable to use messy variable without knowing it

Know your storage types

Where possible, store data in numeric format

- Strings take up way more space
- Example: store education as `byte` w/values 1, 2, 3, 4
- Attach value label: "<HS", "HSG", "SMC", "CLG"
- Use extended macro functions to extract labels as needed

Understand the available storage types:

- `byte`: for indicator variables and other small integers
- `int`: for mid-sized integers
- `long`: for very large integers
- `float`: for non-integer numeric values
- `double`: only if you need the extra precision

Efficient storage types drastically reduce file size

Declare if you dare

I explicitly declare `byte`, `int`, & `long` variables

```
gen byte month = month(td)
gen int year = year(td)
egen long hhid = group(su_id hh_add)
```

Advantages:

- Economizes on storage space
- Speeds execution
- Nebulous fear of floating-point issues

Warning: invalid values get silently set to missing!

```
gen byte ruhroh = 150
assert missing(ruhroh)
```

If unsure: leave unspecified, then `compress`

Checklist for newly cleaned .dta files

1. Confirm that the data have a unique identifier.
2. Harmonize over time, fix errors, standardize missings ...
3. Give all variables brief but intelligible names.
4. Explicitly declare storage types, use `compress`, or both.
5. Store strings as numeric codes with accompanying value labels.
6. Store date/time variables in `datetime` format.
7. If relevant: `tsset` or `xtset`
8. Label every variable. No exceptions!
9. Put the variables in a sensible `order`, with IDs on top.
10. Sort the data by the unique identifier. (Speeds merges.)

data validation

Put your data through their paces

It is incredibly easy to make mistakes in empirical work

- Underlying data are flawed
- Underlying data are not what you think
- Introduce bugs in the course of cleaning

Solution: systematic data validation

- Invariably reveals errors & false assumptions
- If fatal: find better data, ditch project
- If fixable: fix!

Many other substantive benefits:

- Better equipped to answer seminar questions
- Often stumble on new ideas or ID strategies

Goal: give your estimation sample a clean bill of health

Set tripwires

Force your code to crash until you work out the kinks

Tremendously important: Stata's `assert` command

- Basic idea: code proceeds if true, breaks if false

Several related commands:

- `isid` (faster `gtools` version: `gisid`)
- `confirm [new file|numeric|string|variable|...]`
- `merge` (has its own `assert()` option)
- `fillin [varlist]` followed by `assert _fillin == 0`
- `assertnested` for variables that are logically nested
- `cf` for comparing two datasets
- `checksum` for verifying file integrity

Pair with `capture` for error-handling

Check logical identities

Valid data satisfy a host of logical restrictions:

- Additive identities (`assert pop_t == pop_m + pop_f`)
- Sub-additive identities (`assert earnings <= totinc`)
- Range restrictions (`assert wage >= 0 if !mi(wage)`)
- Time-invariance (`bys id (tm): assert sex == sex[1]`)

Subject your code to many such tests

- Assertions can point the way towards necessary adjustments
- Assertions can document known facts about/issues with data

Keep these assertions turned on: don't comment out!

- Common situation: add new years of data $\implies$ code breaks
- Assertions can reveal that old assumptions no longer hold

Sweat the small stuff

Temptation: ignoring "small" bugs

- Issue only applies to a few observations
- Results make sense despite known bug

Most errors are canaries in coal mines

- Misunderstanding the data structure
- Misunderstanding a Stata command

"Small" bugs can easily get amplified

Do not rest easy until you get to the bottom of them

Look at the data

Key step in data validation: looking at the actual data

- Look at some individual observations (`list in 1/10`)
- Look at summary statistics, correlations, ranges, etc.
- Plot the data over time

Some useful descriptive commands:

`browse` (if using GUI: but servers $\succ$ GUI), `codebook`, `collapse`, `compare`, `correlate`, `describe`, `duplicates`, `histogram`, `inspect`, `list`, `regress`, `summarize`, `tabulate`, `tabstat`, `twoway`

Questions to keep in mind:

- Are summary statistics plausible?
- Are cross-tabs/correlations plausible?
- Are time trends plausible? continuous?

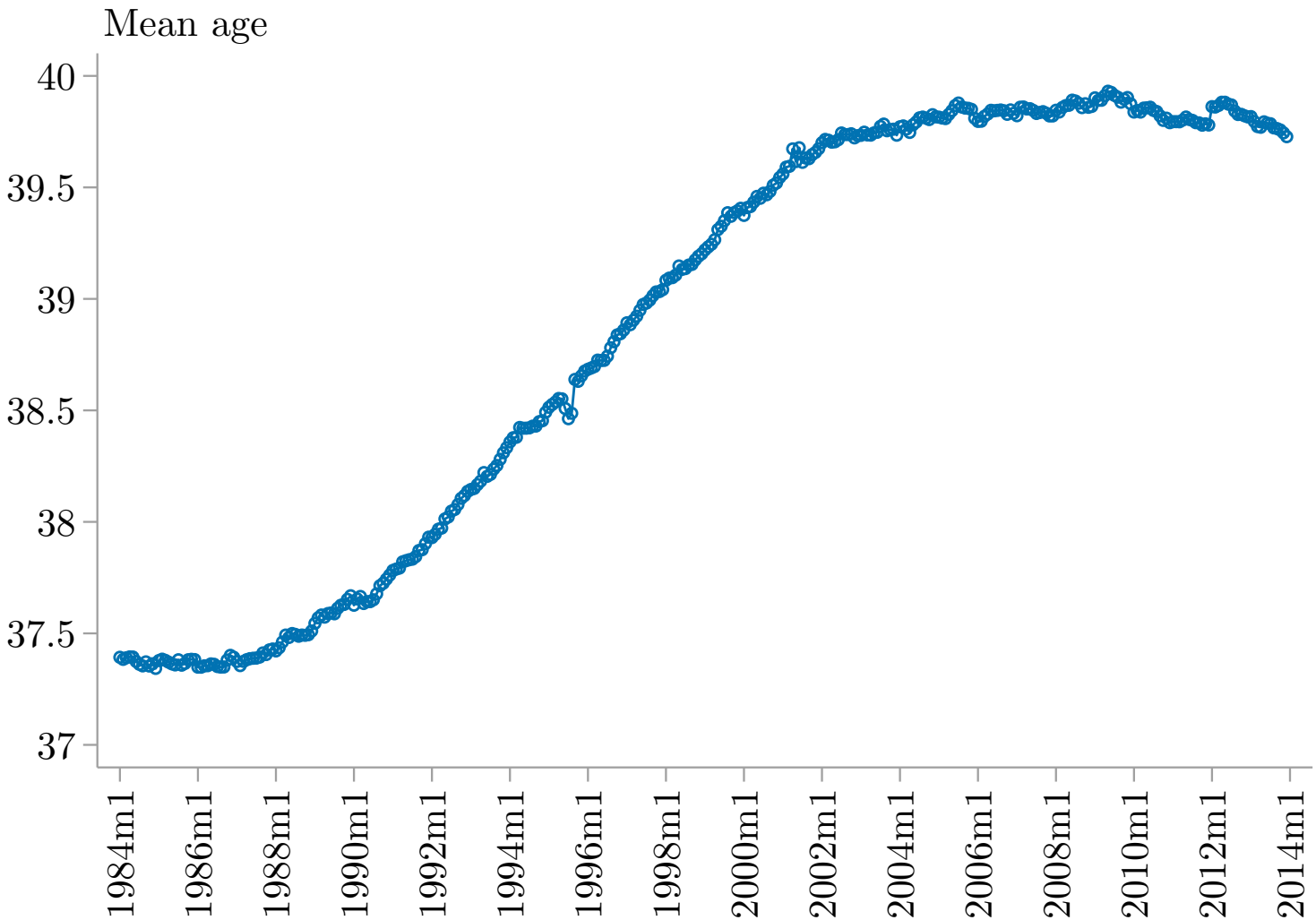
Visualize the data

Once the data seem reasonably clean, visualize them

- Sensible plots provide reassurance
- Implausible plots help pinpoint problems

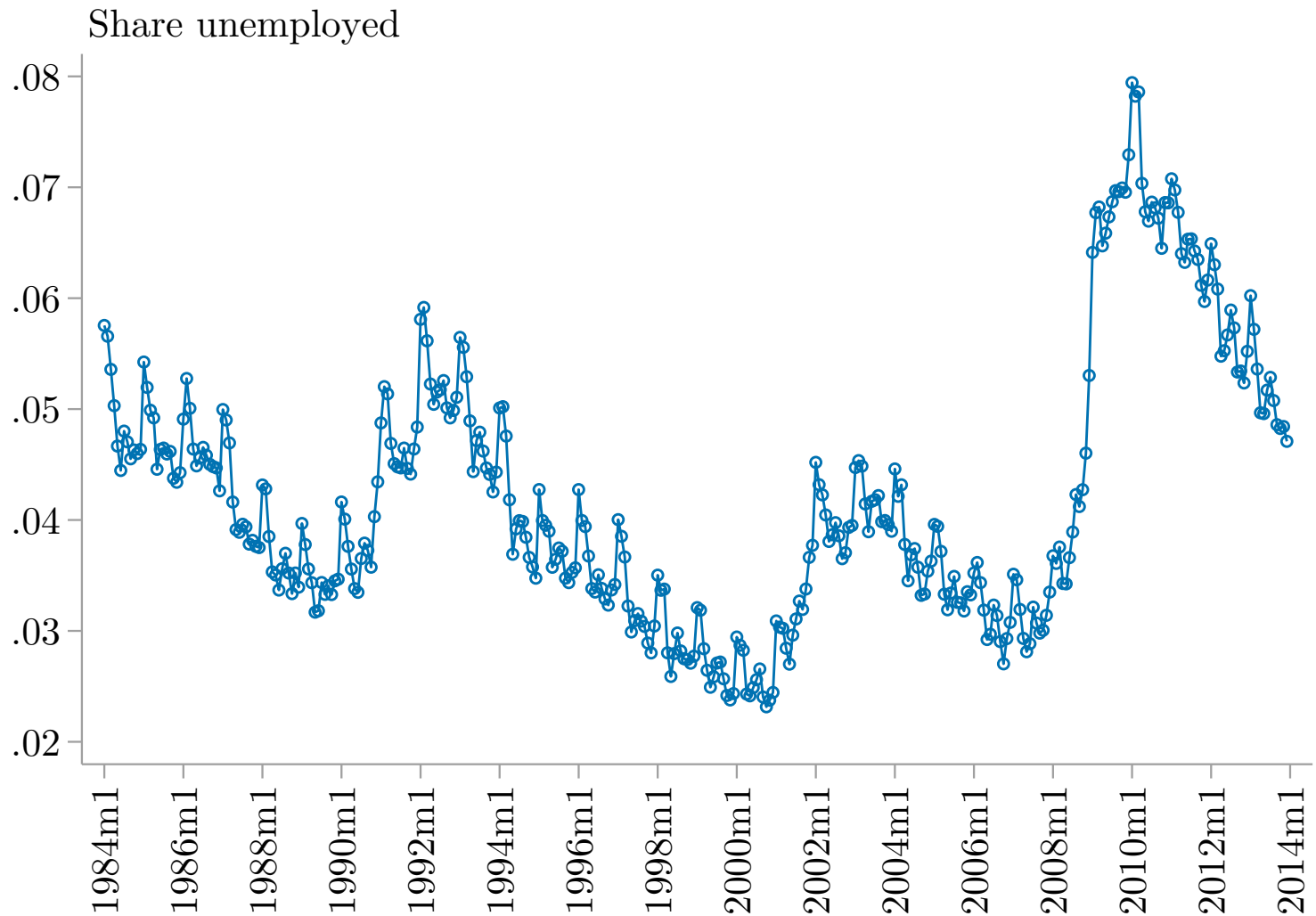
Here are a few examples

Smooth series should vary smoothly



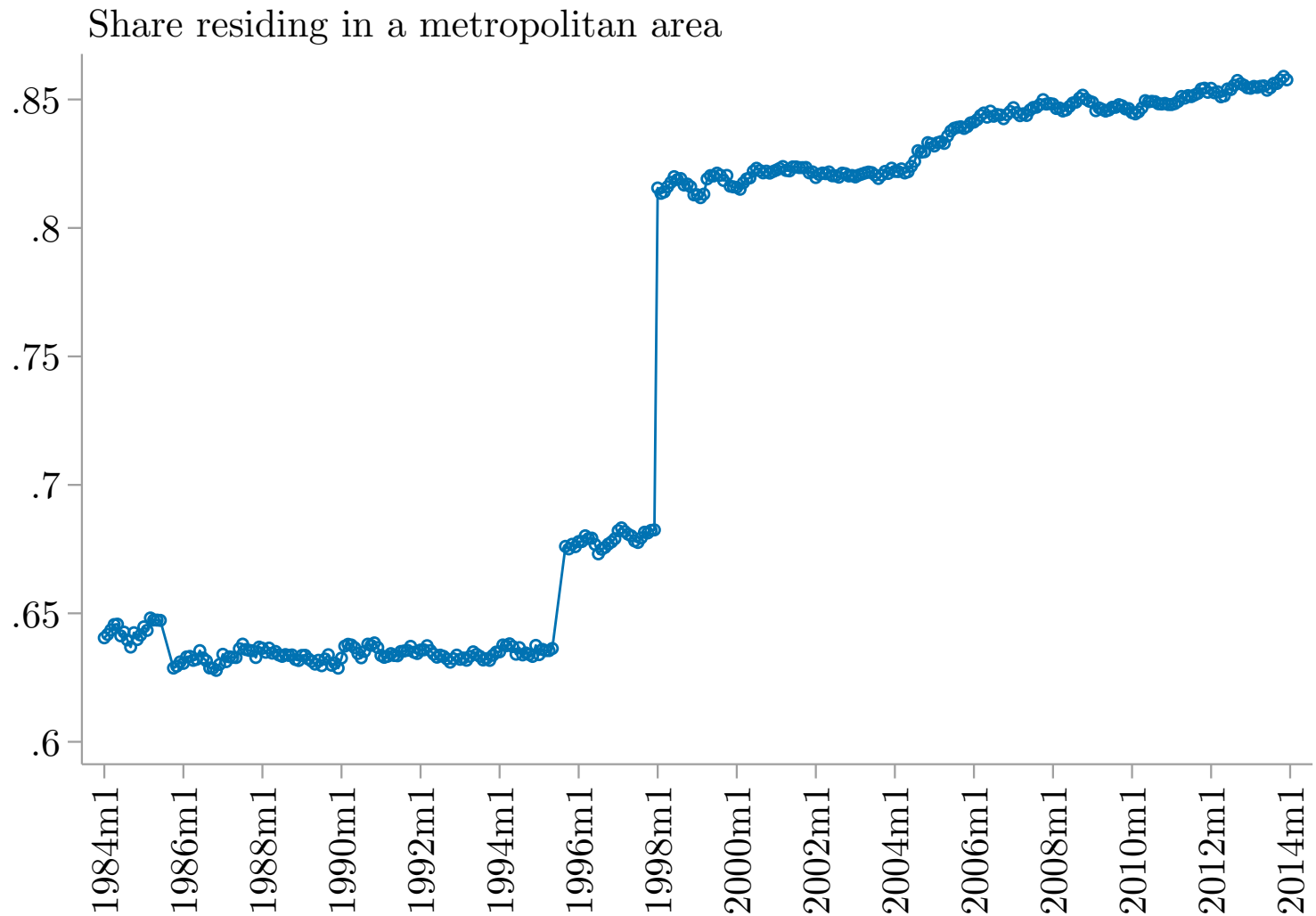
Source: IPUMS CPS data on prime-age US workers.

Cyclical series should track the cycle



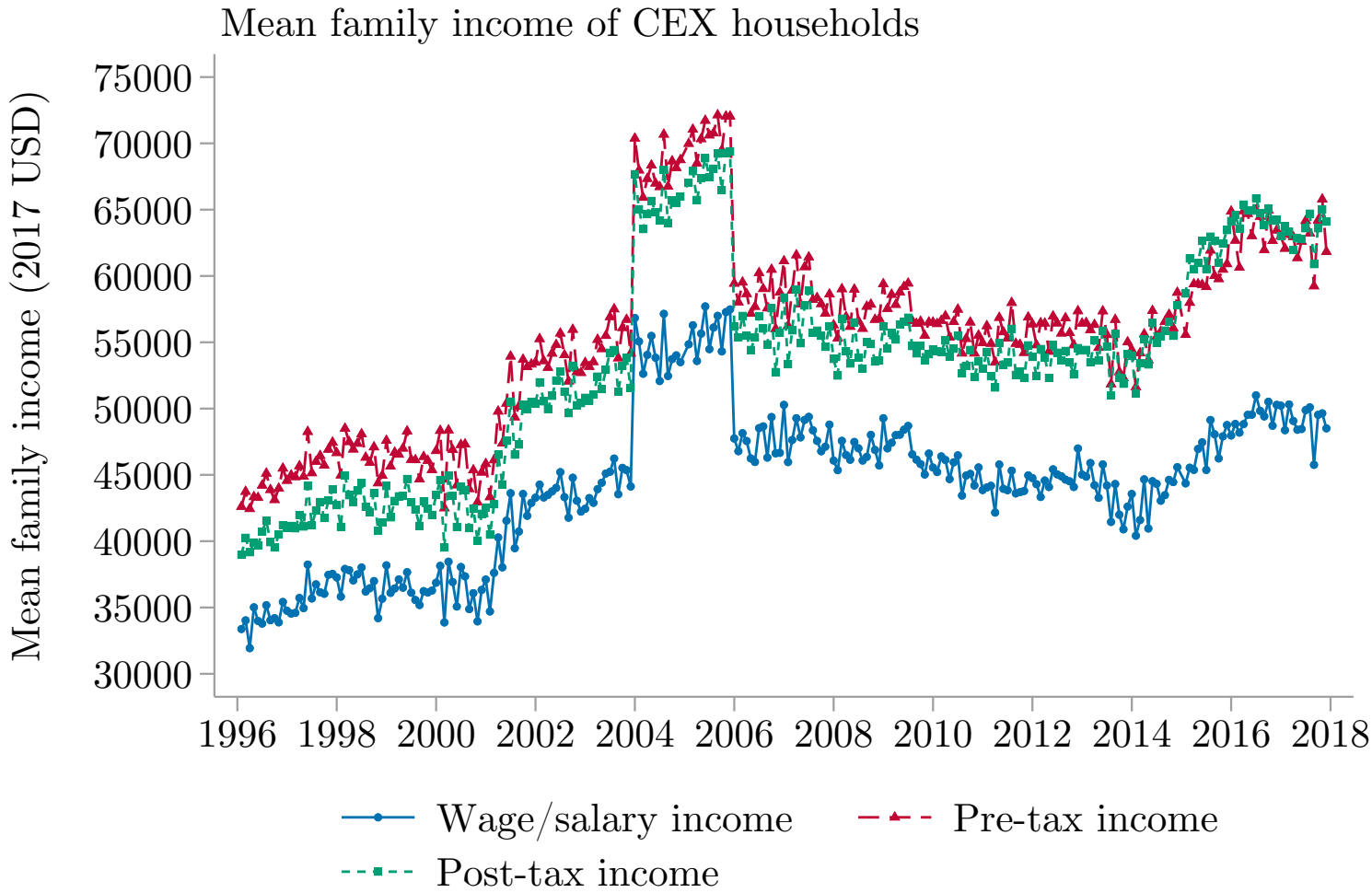
Source: IPUMS CPS data on prime-age US workers.

Plotting data often reveals data seams



Source: IPUMS CPS data on prime-age US workers.

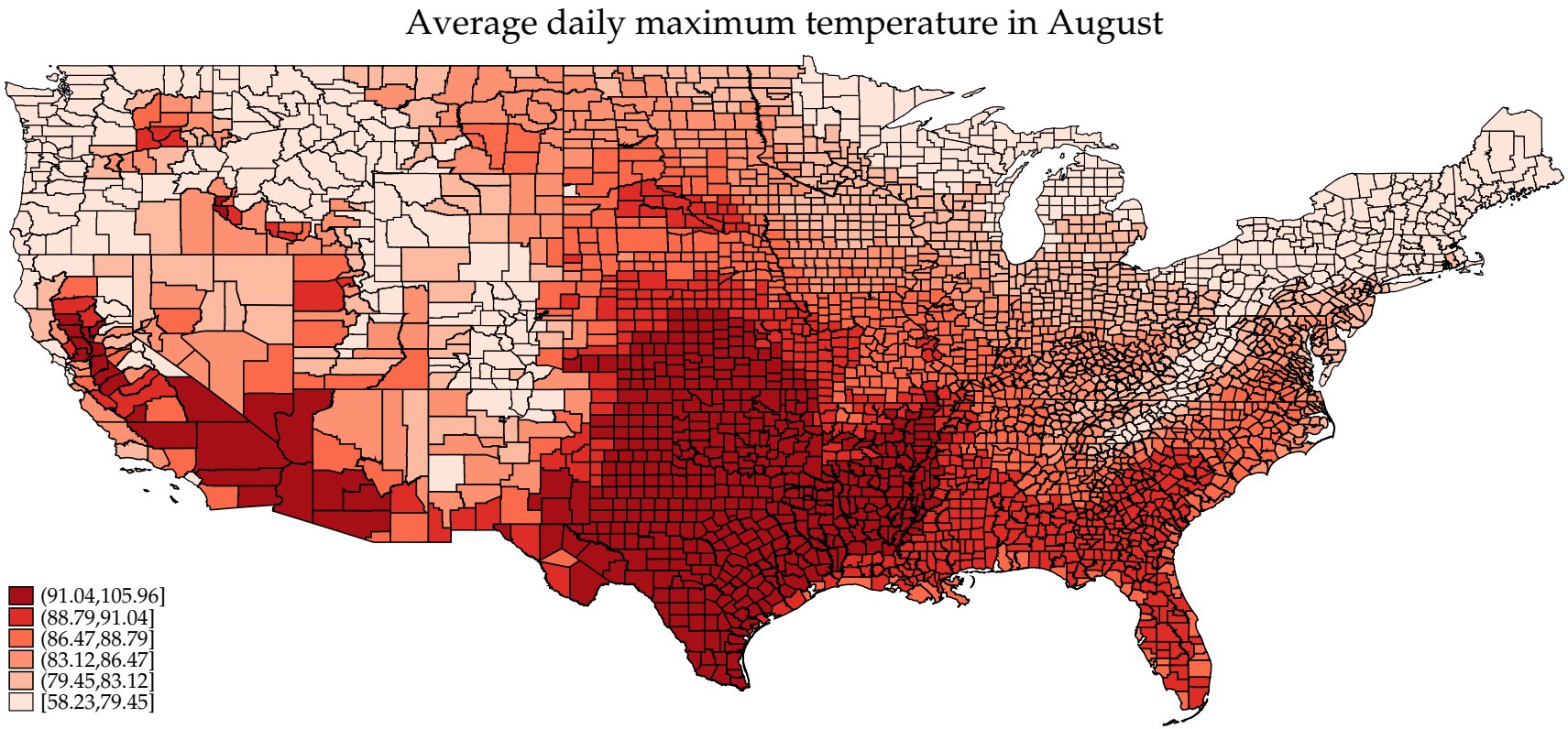
This imputation didn't work out so well



Note: only imputed income is available in 2004-2005.

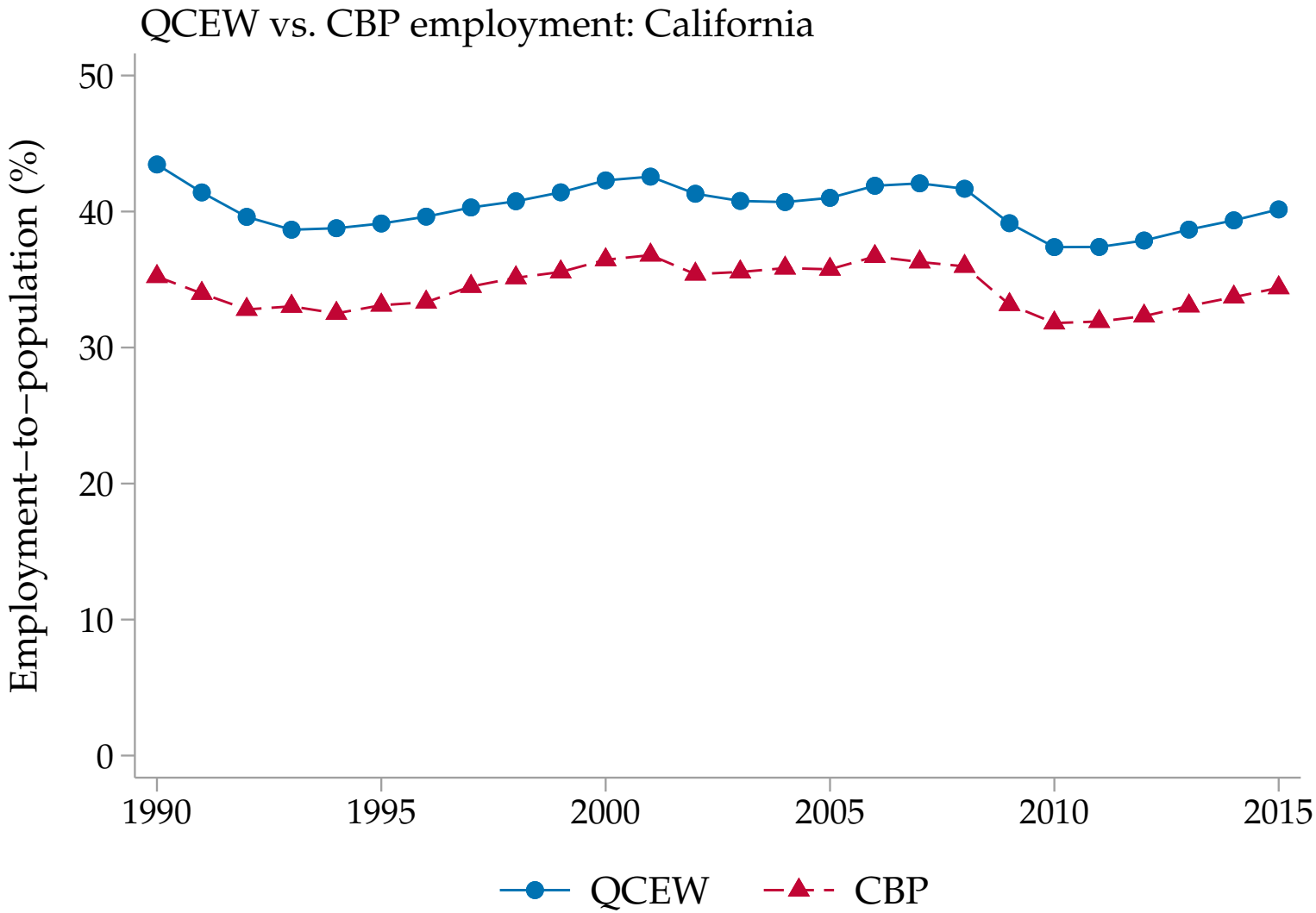
Source: Consumer Expenditure Survey.

Hot places should, in fact, be hot



Source: North American Land Data Assimilation System, 1979–2011.

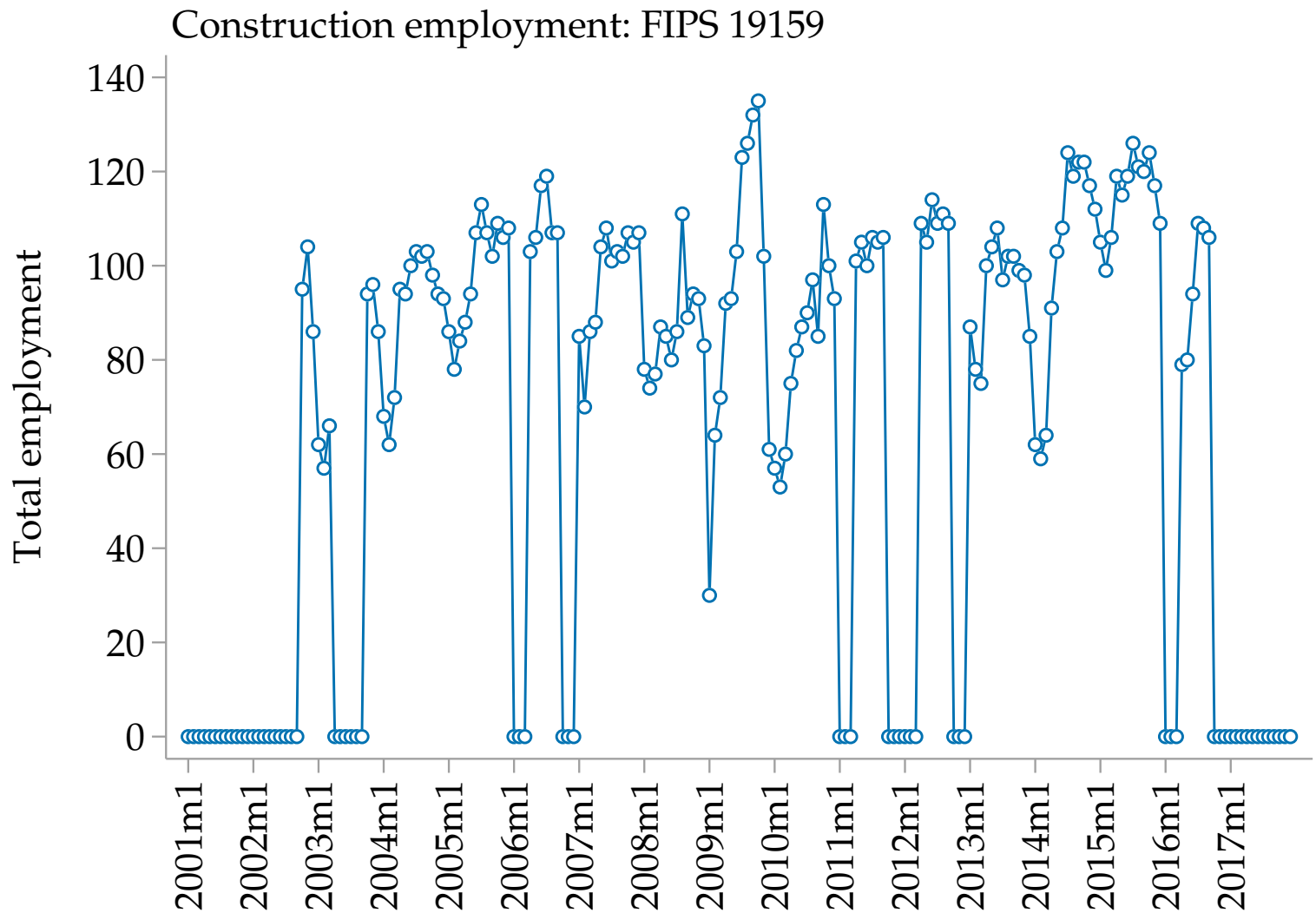
Validate one dataset against another



Source: Quarterly Census of Employment and Wages; County Business Patterns.

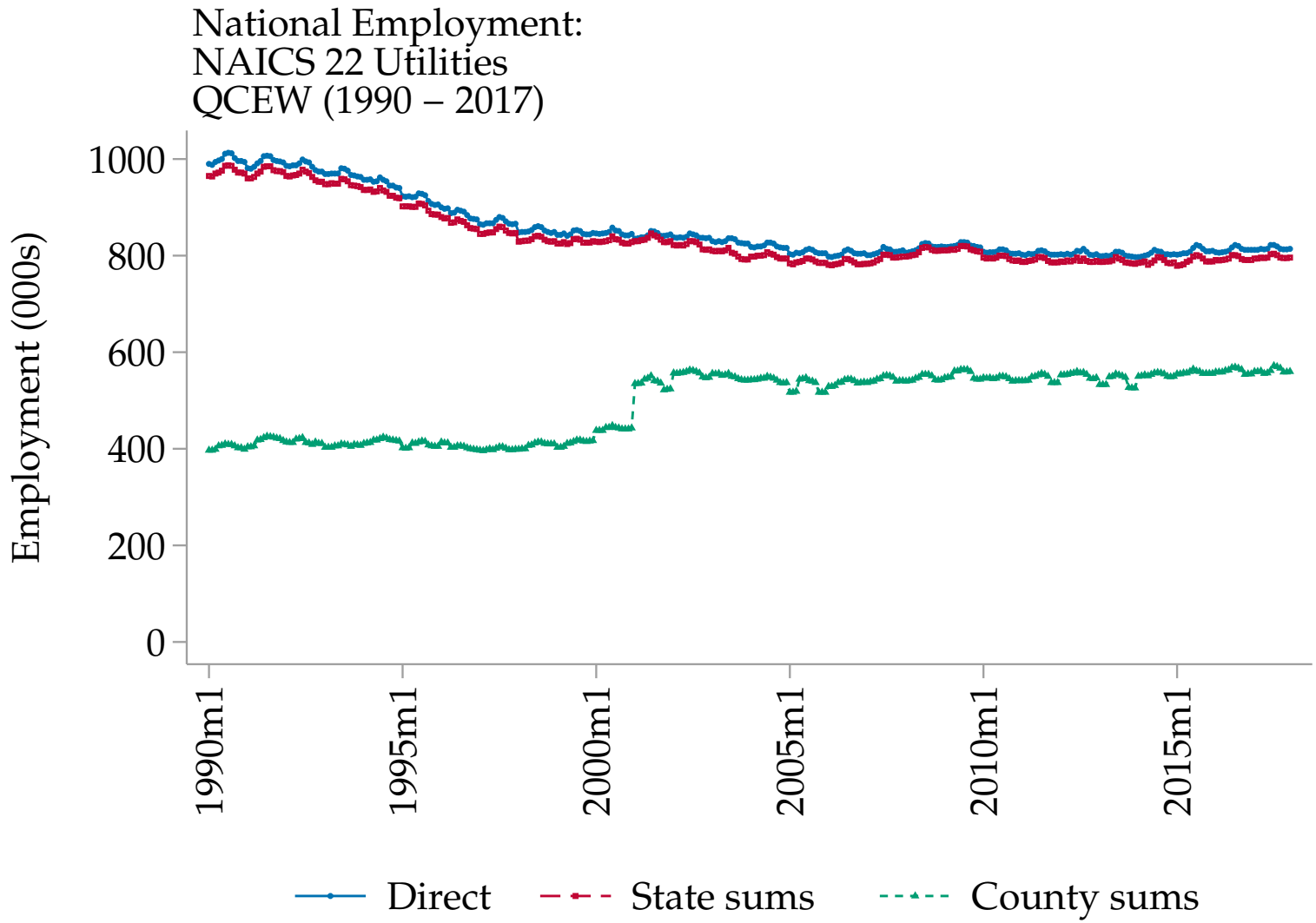
Suppressed employment in the QCEW

Ringgold County, IA, population 5,131 in the 2010 Census



Source: Quarterly Census of Employment and Wages.

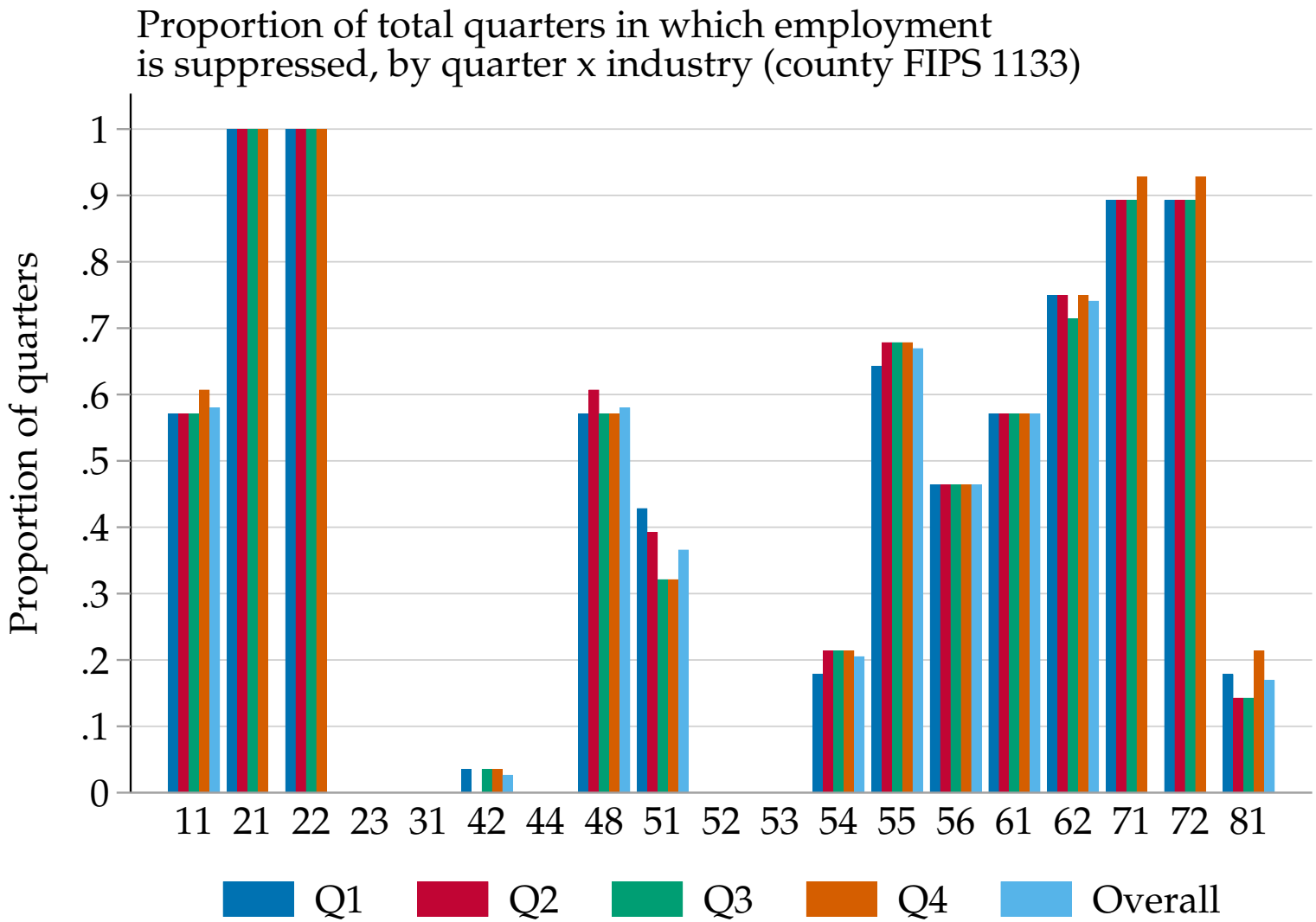
Severe suppressions at the county level



Source: QCEW. Figure created by Justin Wiltshire.

What to plot varies with the question

Here, we're interested in understanding county-level seasonality.



Source: QCEW. Figure created by Justin Wiltshire.

data exploration

Start simple, then add complexity

Good causal work requires good descriptive work

Start by looking at the "raw" (cleaned) data

- Levels + changes over time
- Variation across groups

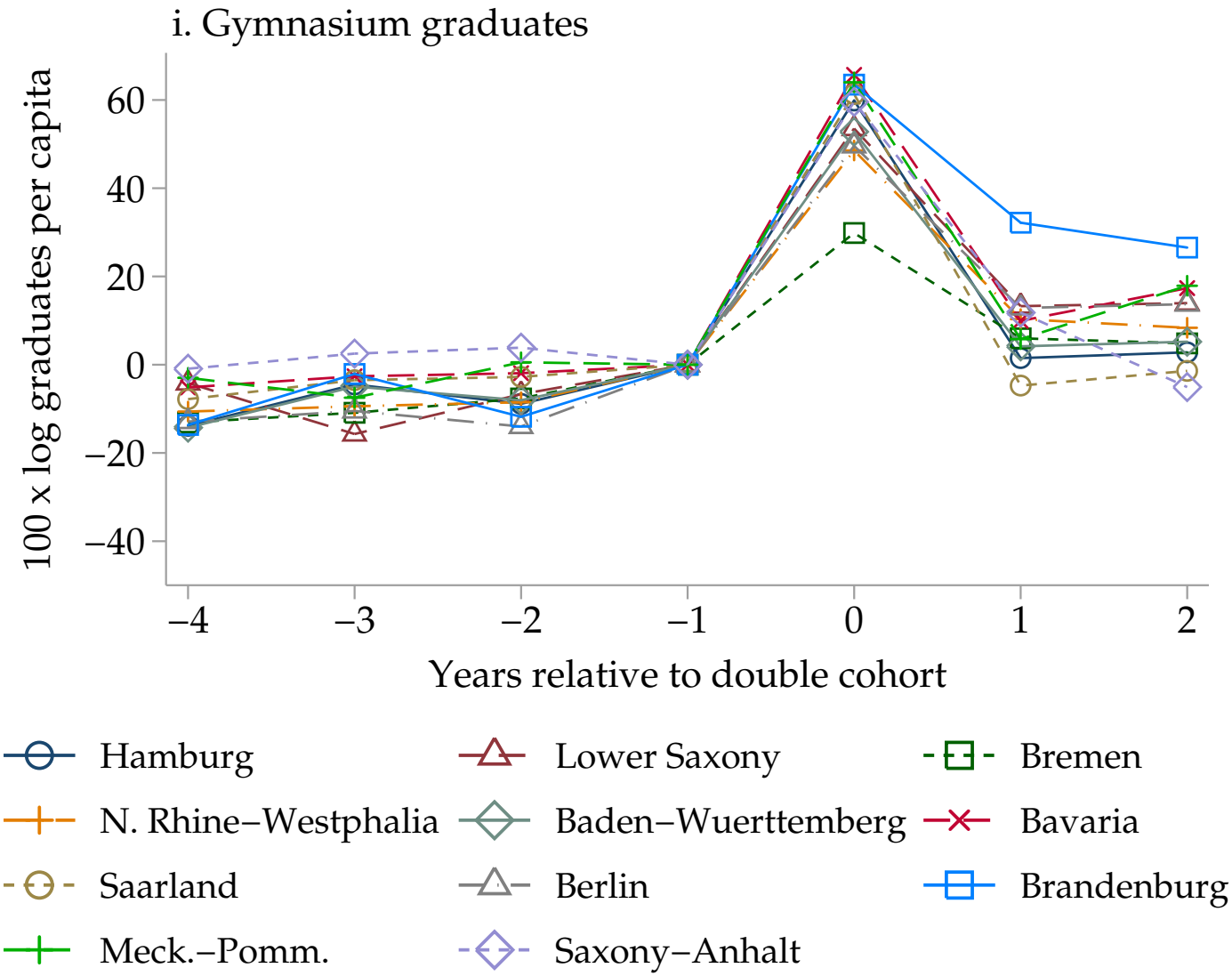
If it's not there in the raw data, folks tend to be skeptical

- Small effects: plausibly masked by background variation
- Big effects: should show up loud and clear in the raw

Simple plots & tabs are harder to screw up

- Ultimately: full-blown regression/IV/event study
- But don't start there

E.g.: look at events 1-by-1 pre-pooling



Source: Price (2017), "Can Local Labor Markets Absorb Crowded Cohorts? Evidence from German High School Reforms". PhD dissertation, chapter 4.

Beware the curse of dimensionality

Early exploration involves many sample splits/spec choices

- Stratify analysis by sex?
- Stratify analysis by education, age, marital status, . . . ?
- Outcome variable: employment or LFP? logs or levels?
- Weight counties by population? by employment? not at all?
- Which control variables to include?

Result: $2 \times 4 \times 2 \times 2 \times 3 \times \ldots =$ too many permutations

- Messy code with 14 nested loops
- Messy file structure with oodles of output
- Hard to get your head around everything

As you introduce new dimensions, collapse some old ones

- Example: if male/female results very similar, pool sexes

Separate estimation from presentation

Runtime varies across tasks by orders of magnitude

- Data prep & estimation: usually slow
- Creating figures & tables: always fast

Use "toggles" to control what's executed

```
* Set toggles
local run_specs = 0
local run_figs  = 1

[later in script]

* Run specifications
if `run_specs' == 1 {
    qui reg `y' `x' `billion_FEs' in `terrifyingly_large_sample', robust
    [save estimates to disk]
}

* Create figures that would make Edward Tufte proud
if `run_figs' == 1 {
    [load saved estimates, plot coefficients]
}
```

Saving estimates to disk

Back in the bad old days:

- Run 4-hour regression task, create figures
- Oops, typo in axis title
- Rerun 4-hour regression task, recreate figures

`estimates save` to the rescue

- Run 4-hour regression task, `estimates save`, create figures
- Six months later: decide you want to tweak something
- `estimates use`, then `estimates store`
- Futz with graphics to your heart's content

Other use cases:

- Want to renormalize the omitted group (easy with `nlcom`)
- Want to run a t-test after the fact

Estimate, save, load, plot

```
*-------------------------------------------------
* Estimate seasonality in unemployment
*-------------------------------------------------

if `run_specs' == 1 {
    qui newey lnemp ib12.month tmspline*, lag(12)
    qui est save ``$projdir/est/show/aggump/stocks.ster'', replace
}

*-------------------------------------------------
* Load estimates into memory
*-------------------------------------------------

* Get list of saved estimates
local flist : dir ``$projdir/est/show/aggump/'' files *

* Load saved estimates into active memory
est clear
foreach f of local flist {
    local fstub `=substr("`f'", 1, length("`f'") - 5)'
    est use ``$projdir/est/show/aggump/`f' ''
    qui est store `fstub'
}

*-------------------------------------------------
* Plot results
*-------------------------------------------------

* Seasonality in unemployment stock
coefplot stocks [lots of graphical options]
```

From estimation to visualization

`estimates save` pairs really nicely with `coefplot`

But that's for next time ...

Next up

Final lecture rescheduled & relocated

- When: Monday, 12/02 from 3:40–5:00pm
- Where: ARE Library Conference Room (4101 SSH)

Topic: "Sharing Your Work"

- Figures vs. tables
- Making pub-quality figures
- Definitely: designing a slide deck
- Possibly: designing a manuscript

until next time

Powered by tikzducks