vis theory

adam okulicz-kozaryn

adam.okulicz.kozaryn@gmail.com

this version: Wednesday 19th April, 2023 16:52

outline

theory (Tufte)

examples

strategy / howto graph it / galleries

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

why vis? why bother? whats the big picture?

- understanding your data is fundamental/critical for research/insight
- vis is the best way to understand data (humans; AI aside)
- data are numbers, usually many and in a matrix
  - vis allows humans to comprehend those many numbers
  - if you look at numbers you will be slower in understanding
- pictures are not less "scientific" than numbers!
- best journals mostly do vis: nature, science, etc

Know Your Data!

- simply cant use it well if you dont know it well
  - (not just data; the field: theory, lit, method, etc)
  - this is where you beat IT folks (MS/PhD just in IT)
- again, be prepared to invest a lot of time into your data
  - use data that you're passionate about
  - or that can make $ (now or in future career)
  - or ideally both!
- and this is where vis comes in–best way to get to know your data!
- but don't forget to think about it! don't be mindless!
  - ask questions, be investigative, be critical
- double check, cross check, give to others to check

outline

theory (Tufte)

examples

strategy / howto graph it / galleries

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

make it interesting!

- the greatest value of a picture is when it forces us to notice what we never expected to see
  - REFS:
- `http://www.edwardtufte.com/`
- Kosslyn "Clear and to The Point" `http://www.amazon.com/Clear-Point-Psychological-Principles-Presentations/dp/0195320697`

be simple: avoid clutter

- everything should be made as simple as possible
- avoid padding: present only data needed for a specific purpose
- avoid clutter: eg single graph must only present the data that are highly related and must be compared
- put data into appendix if it is not very relevant but may be useful
  - people looking for extra info will find it
  - people interested in the main story will not get distraced

avoid visual clutter

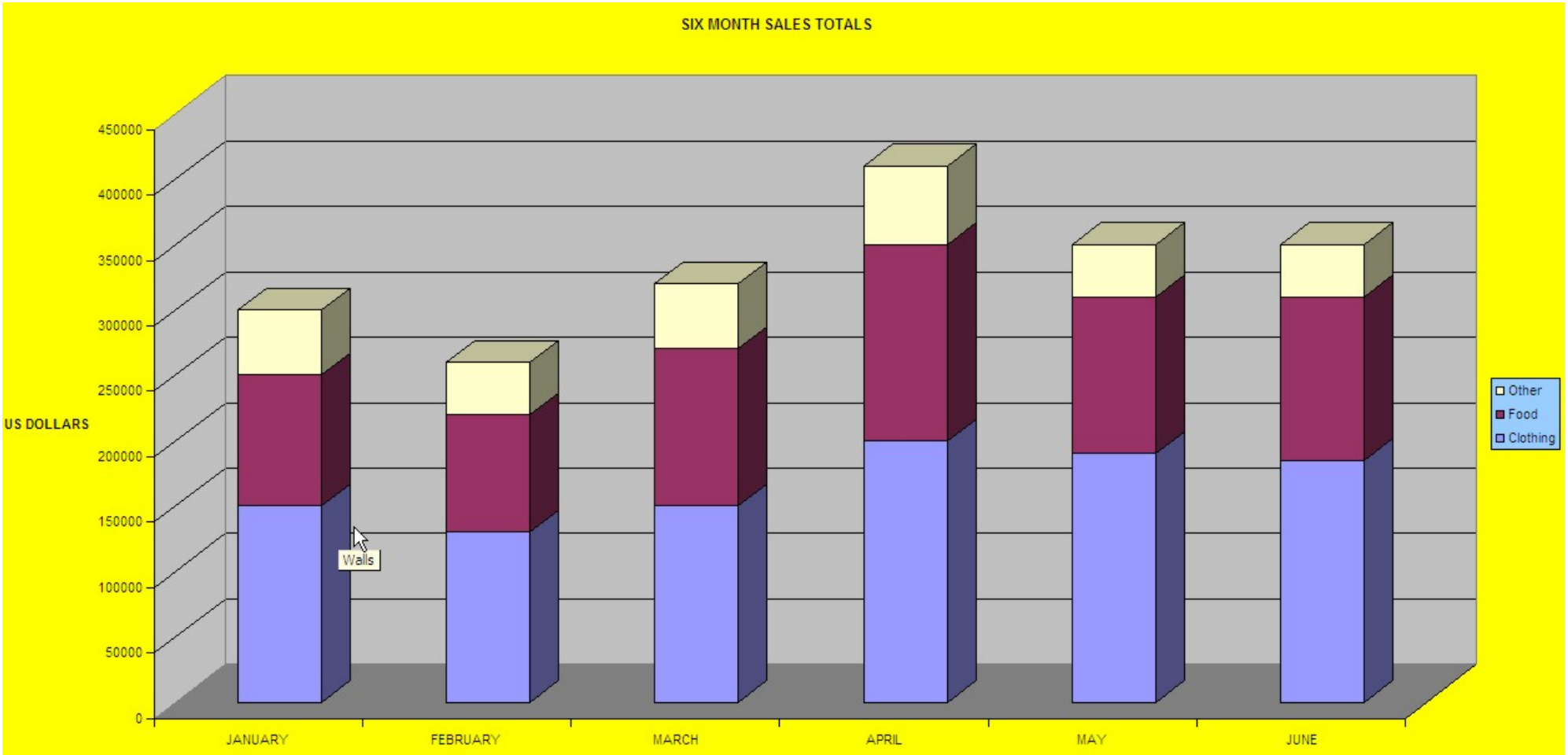
- all parts/attr of vis must mean sth (convey info)
  - shades
  - colors
  - decoration
  - etc
- everything must convey info

chartjunk

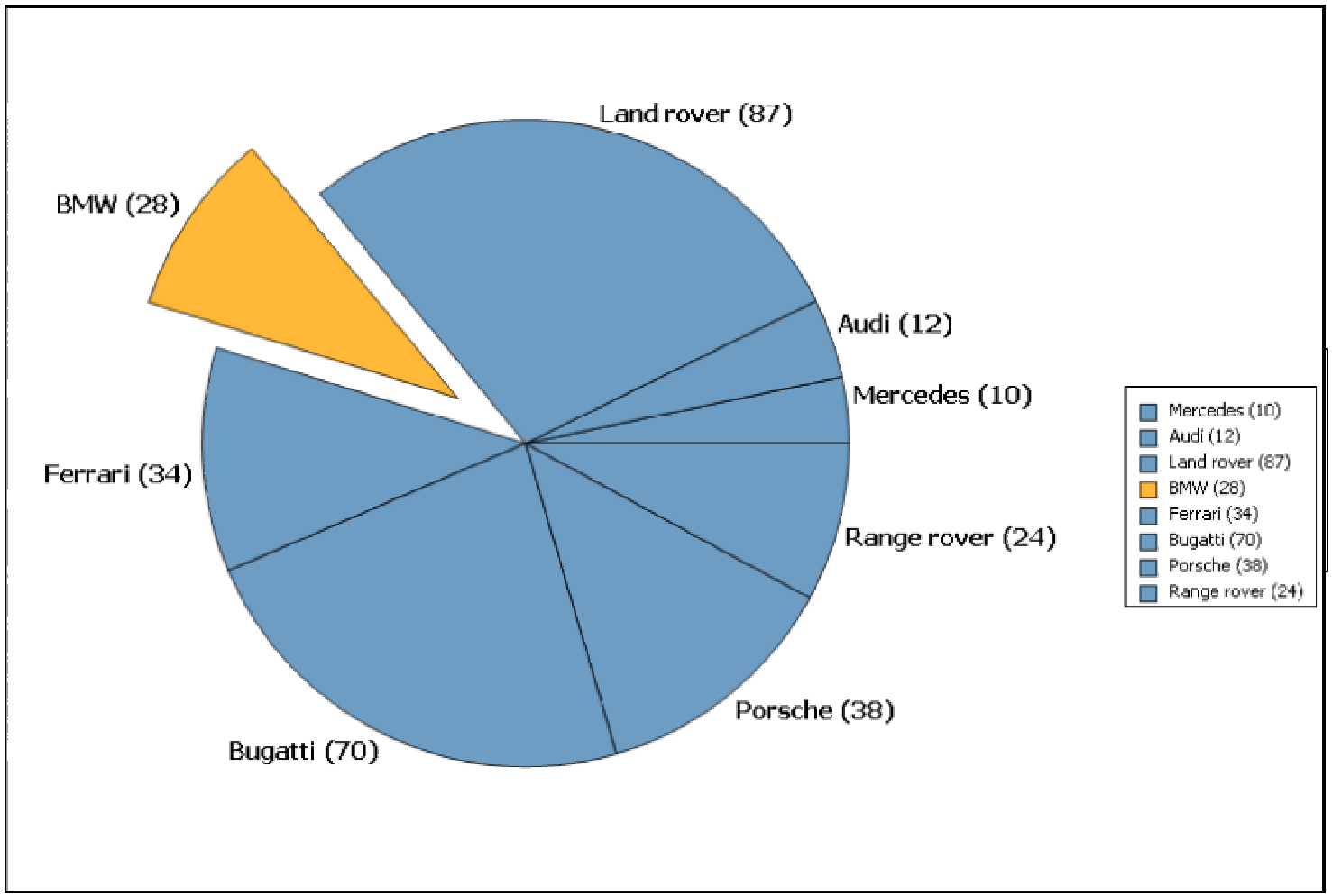
The interior decoration of graphics generates a lot of ink that does not tell the viewer anything new. The purpose of decoration varies to make the graphic appear more scientific and precise, to enliven the display, to give the designer an opportunity to exercise artistic skills. Regardless of its cause, it is all non-data-ink or redundant data-ink, and it is often chartjunk.

Edward Tufte "The Visual Display of Quantitative Information"

chartjunk/business graphs

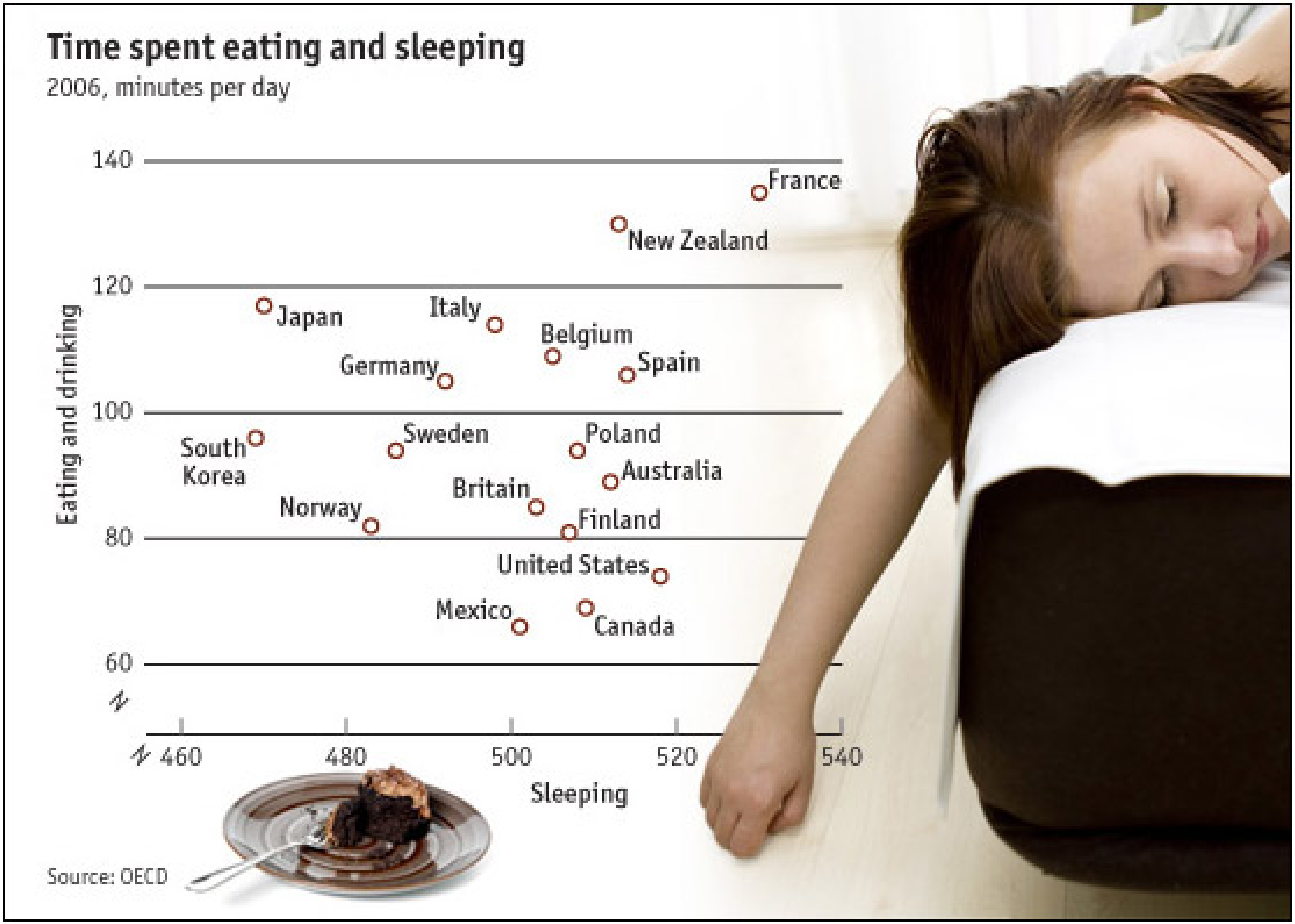


chartjunk


Exploded Pie Slice Chart
Land rover (87)
BMW (28)
Audi (12)
Mercedes (10)
Ferrari (34)
Range rover (24)
Porsche (38)
Bugatti (70)
Mercedes (10)
Audi (12)
Land rover (87)
BMW (28)
Ferrari (34)
Bugatti (70)
Porsche (38)
Range rover (24)


Extra leisure time enjoyed by men compared with women
2006*, minutes per day
0
20
40
60
80
Italy
Poland
Mexico
Spain
Belgium
United States
France
South Korea
Britain
Turkey
Australia
Finland
Canada
Germany
Sweden
Japan
New Zealand
Norway

not chartjunk (the economist)



balance!

- colors (eg either use toned down or stark contrasts)
- fonts: titles, notes, labels, etc should be proportional
- thickness of lines
- and everything else
- in general: rather use less ink than more
- note: my classic 90s mpl theme tends to be heavier on ink
  - but then I do make sure to keep it barebone/simple

one v several vis

- usually to tell a story, may need several vis
- say to show problems in SJ:
  - low educ, poverty, crime, etc
  - but can also show a summary, eg an index
  - to show change: one or two; calc chng var
  - perc chng: 100*(pop10-pop00)/pop00

good practices

- use graphs as much as possible, ditch all tables
- BUT avoid graph padding and within-graph data padding
- be as simple as possible (never use chart junk)
- the fewer graphs the better (like nature, science)
  - but to have few awesome ones to share with the world, first have to have dozens in notebook (so clearly mark the main ones v the auxiliary/robustness checks ones) (for class presentation mostly focus on main ones, the story)
  - otherwhise your vis won't be robust/bullet-proof/thought-through
- display measures of uncertainty, typically 95%CI

think about it/meaning: the 'so what?' question!!!

- ok, you've got the vis... now think about it
- what does it mean? interpret substantively!
  - (beyond technical correctness; lack of mistakes)
  - as you look at it, ask yourself the 'so what?' question
  - if not happy with the answer:
  - drop it, comment ot out, or leave it as it is, and
  - keep going, and produce a better vis
- I grade substantive meaning, too
- in fact the idea, the meaning, the contribution to the knowledge is most important!

outline

theory (Tufte)

examples

strategy / howto graph it / galleries

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

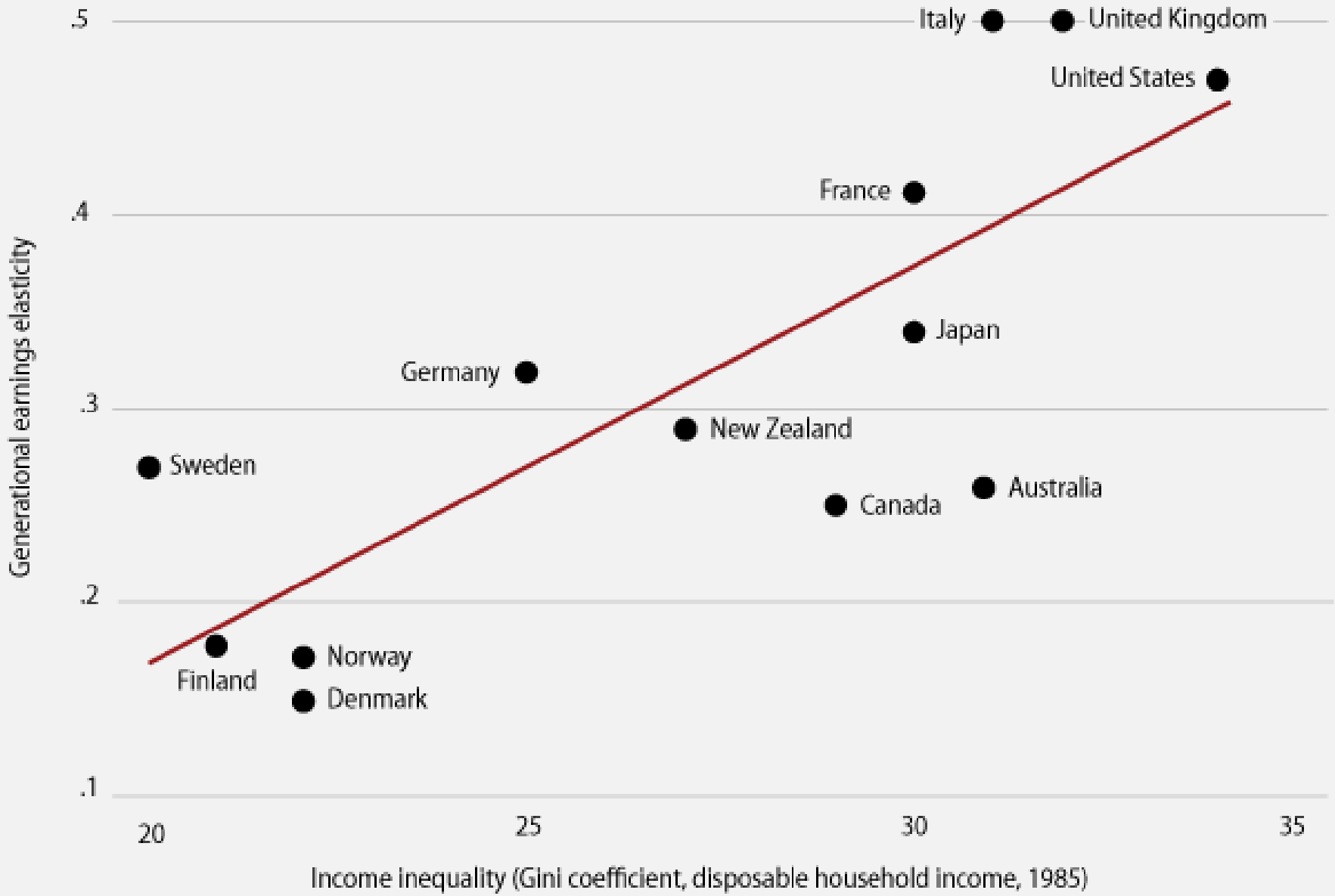
exemplary examples

- time to produce new knowledge
- below examples to inspire–all simple but produce great insight–what counts most is an idea!
- [to find out more about vis/research just goog vis title]

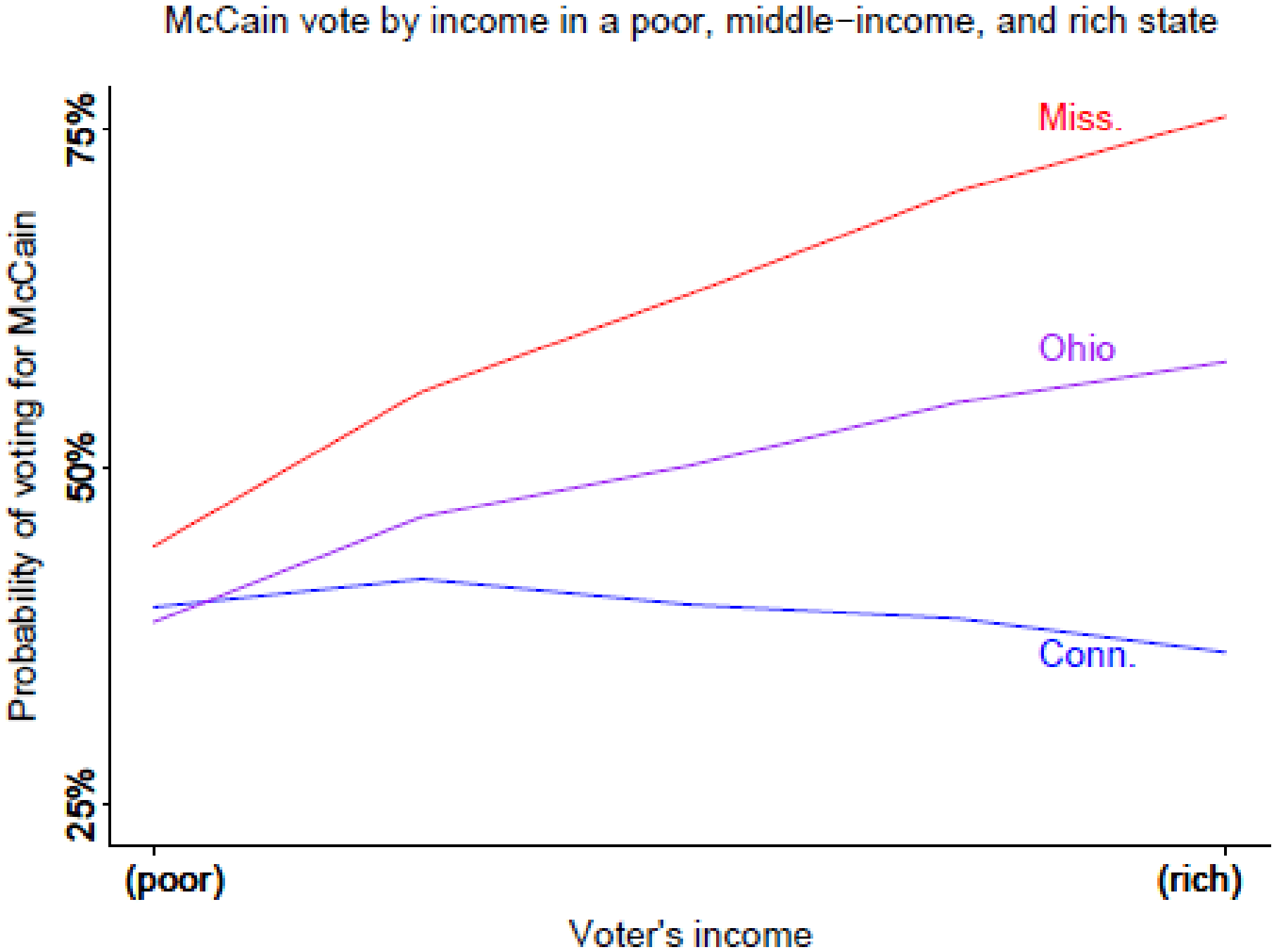
FIGURE 1

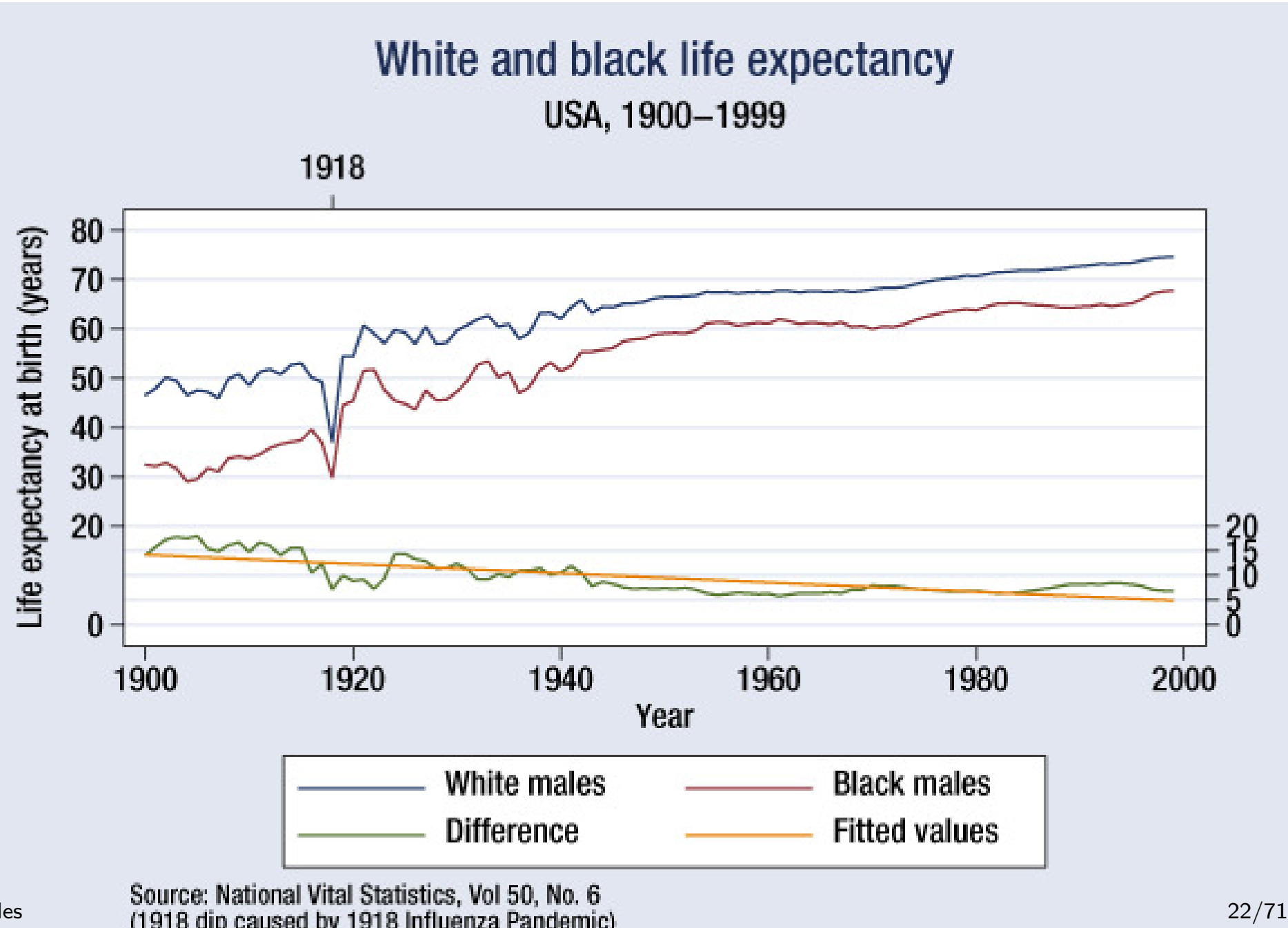
The Great Gatsby Curve

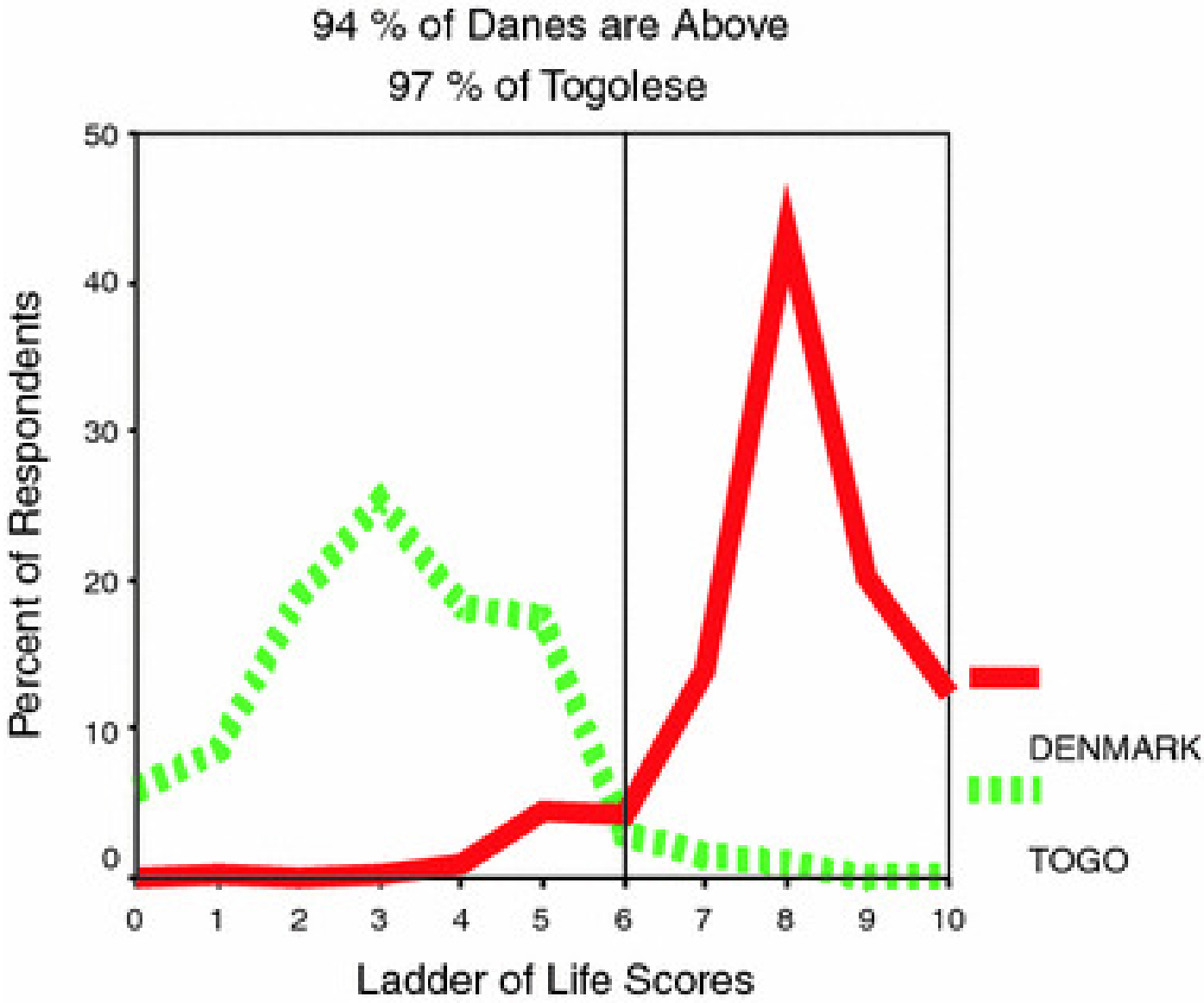
More inequality is associated with less mobility across the generations

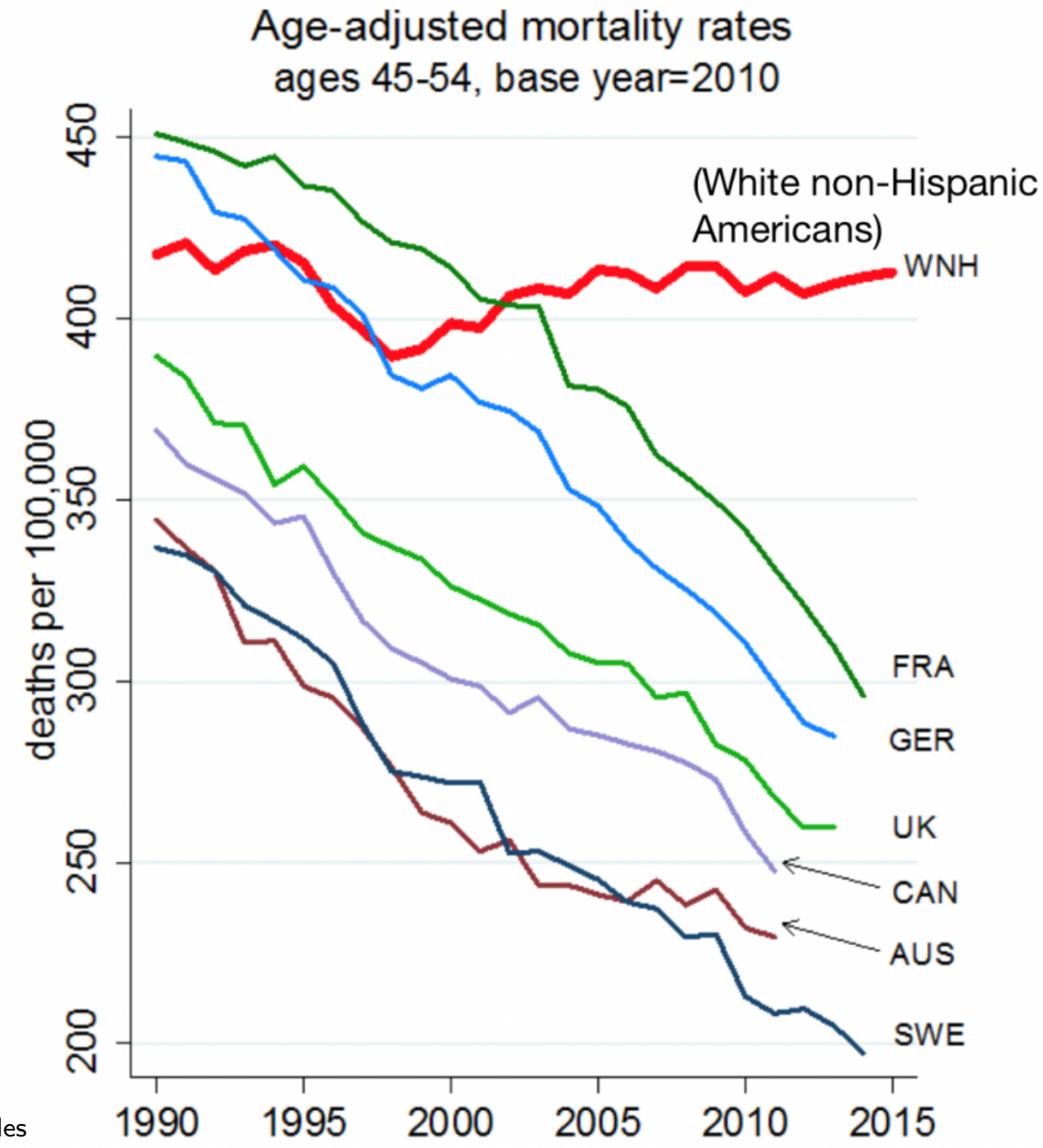


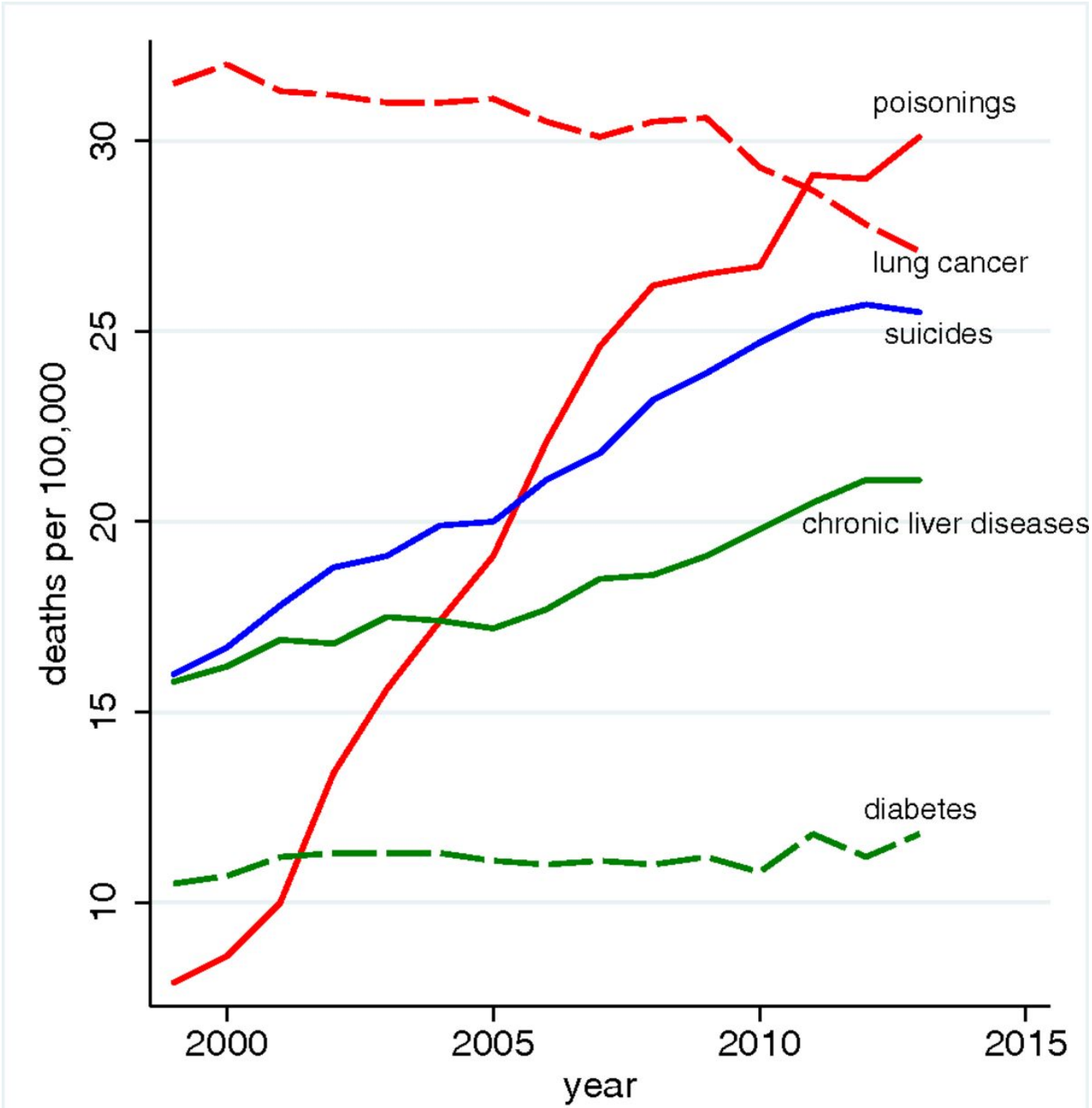
Source: Miles Corak, "Inequality from Generation to Generation: The United States in Comparison," In Robert Rycroft, ed. The Economics

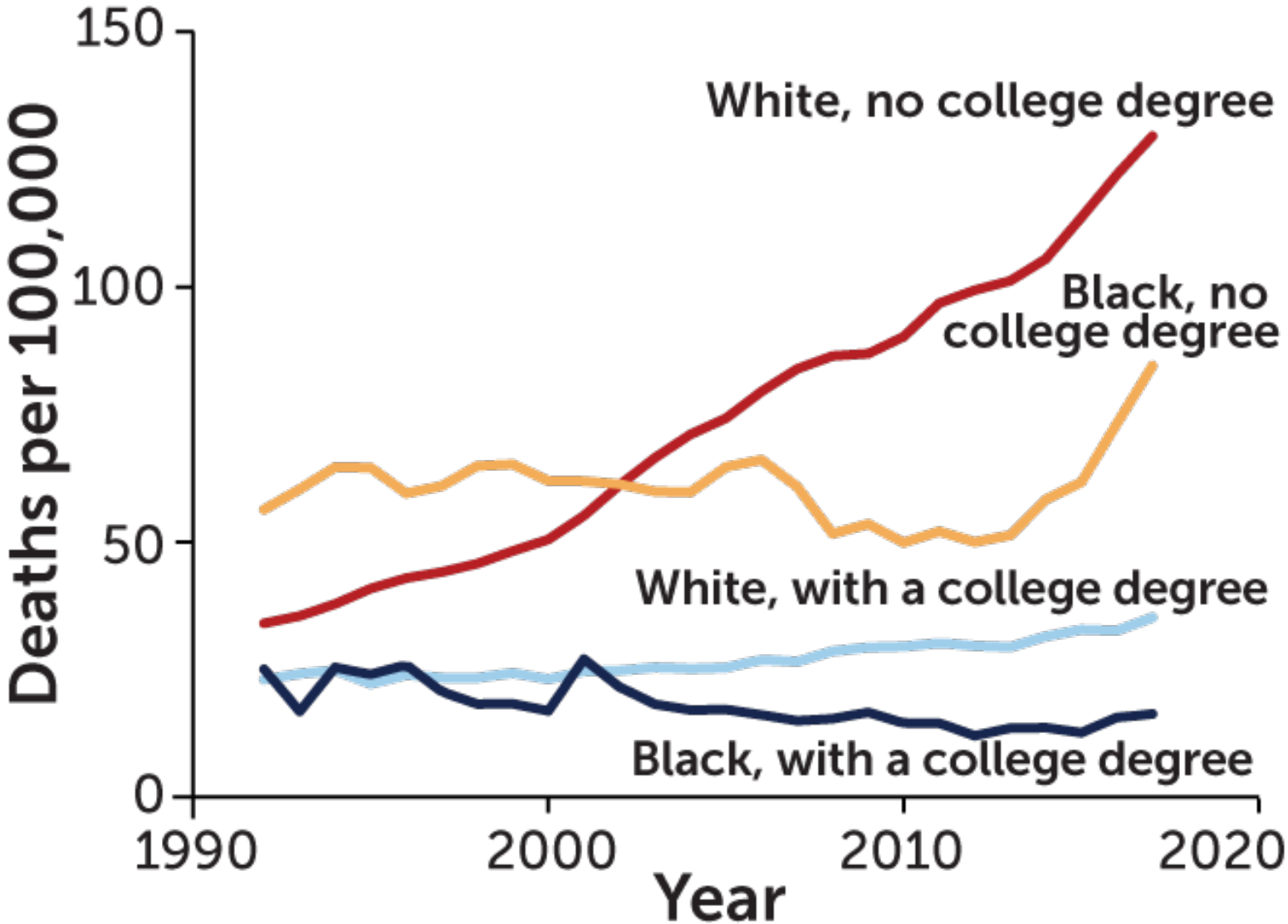

McCain vote by income in a poor, middle−income, and rich state
Probability of voting for McCain
75%
50%
25%
Miss.
Ohio
Conn.
(poor)
(rich)
Voter's income


White and black life expectancy
USA, 1900–1999
1918
Life expectancy at birth (years)
80
70
60
50
40
30
20
0
20
15
10
5
0
1900
1920
1940
1960
1980
2000
Year
White males
Black males
Difference
Fitted values
Source: National Vital Statistics, Vol 50, No. 6
(1918 dip caused by 1918 Influenza Pandemic)

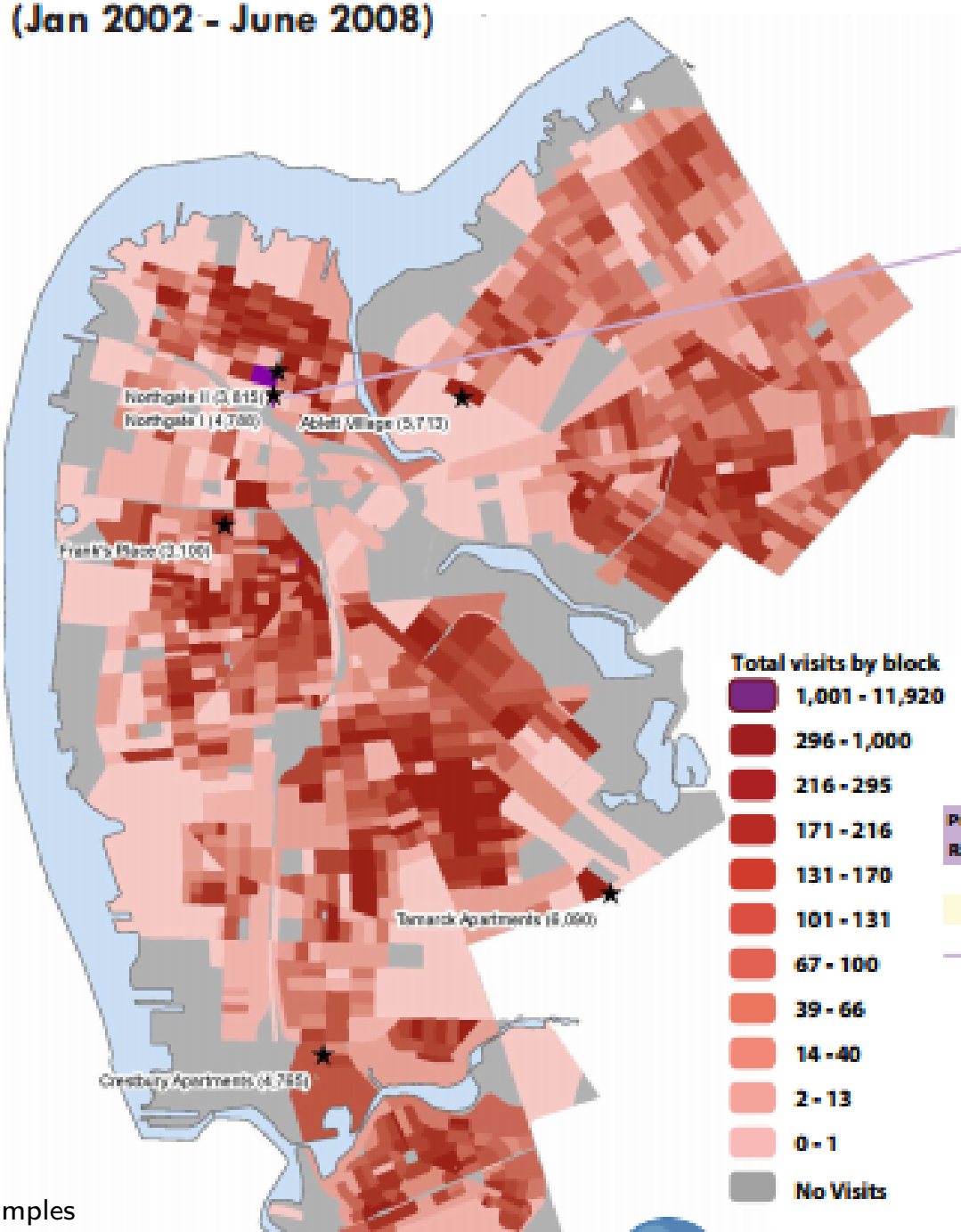

94 % of Danes are Above
97 % of Togolese
Percent of Respondents
50
40
30
20
10
0
0
1
2
3
4
5
6
7
8
9
10
Ladder of Life Scores
DENMARK
TOGO


Age-adjusted mortality rates
ages 45-54, base year=2010
deaths per 100,000
450
400
350
300
250
200
1990
1995
2000
2005
2010
2015
(White non-Hispanic
Americans)
WNH
FRA
GER
UK
CAN
AUS
SWE


poisonings
lung cancer
suicides
chronic liver diseases
diabetes
deaths per 100,000
10
15
20
25
30
2000
2005
2010
2015
year


150
100
50
0
Deaths per 100,000
White, no college degree
Black, no college degree
White, with a college degree
Black, with a college degree
1990
2000
2010
2020
Year

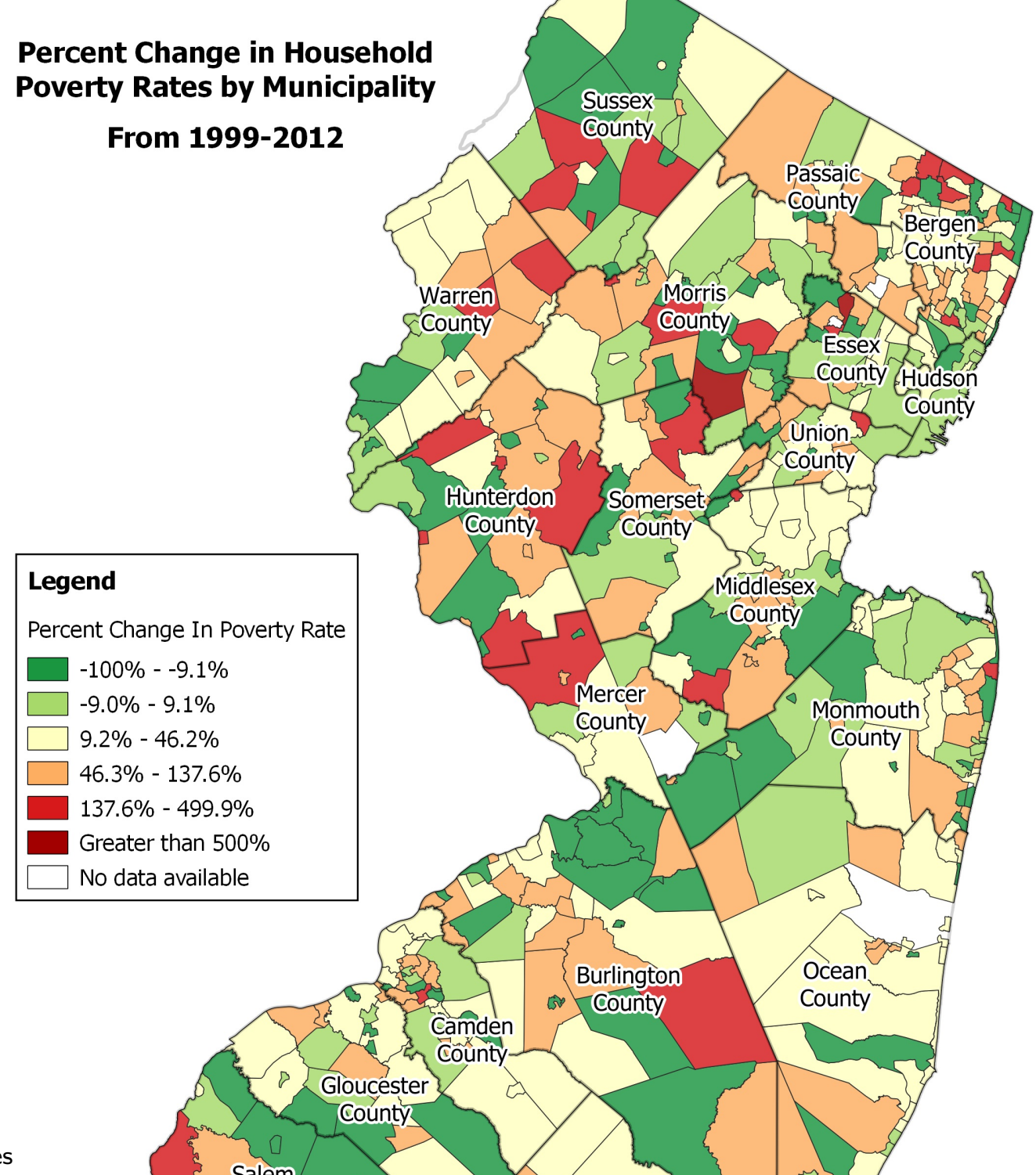
Inpatient and Emergency Room Visits in Camden, NJ
(Jan 2002 - June 2008)



Northgate I Public Housing

	Visits	Patients	Charges	Receipts	Collected
Cooper	3,172	749	$42,144,897	$4,994,658	12%
Lourdes	811	337	$7,848,899	$1,028,611	13%
Virtua	805	333	$1,742,487	$345,092	20%
2005	838	378	$10,834,428	$1,268,373	12%
2006	738	355	$8,867,995	$880,549	13%
2007	790	389	$7,979,262	$900,181	11%
ED	3882	978	$6,150,592	$864,019	14%
Inpatient	906	408	$45,584,781	$5,504,342	12%
Total	4,788	1,070	$51,735,374	$6,368,361	12%

Primary Diagnosis		
Rank	ED	Inpatient
1	abdominal pain (789.0)	live birth (V3X.0)
2	acute URI NOS (465.9)	chest pain (786.5)
3	chest pain (786.5)	congestive heart failure NOS (428.0)


Percent Change in Household Poverty Rates by Municipality
From 1999-2012
Sussex County
Passaic County
Bergen County
Warren County
Morris County
Essex County
Hudson County
Union County
Hunterdon County
Somerset County
Middlesex County
Mercer County
Monmouth County
Burlington County
Ocean County
Camden County
Gloucester County
Salem
Legend
Percent Change In Poverty Rate
-100% - -9.1%
-9.0% - 9.1%
9.2% - 46.2%
46.3% - 137.6%
137.6% - 499.9%
Greater than 500%
No data available

outline

theory (Tufte)

examples

strategy / howto graph it / galleries

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

strategy

- sometimes complicated and/or lengthy code esp for fine-tuning/customization
- and ton of graph options, impossible to memorize
- way easier to figure it out with
  - examples/galleries (syllabus sec 'galleries')
  - or examples i gave you in ipynb
- in colab hover over function it will popup syntax/options
- google it! copy from elsewhere
  - yes do copy-paste from me, internet, chatGPT, etc
  - but then adjust it, clean it up, streamline, simplify
  - and focus on story telling, your story

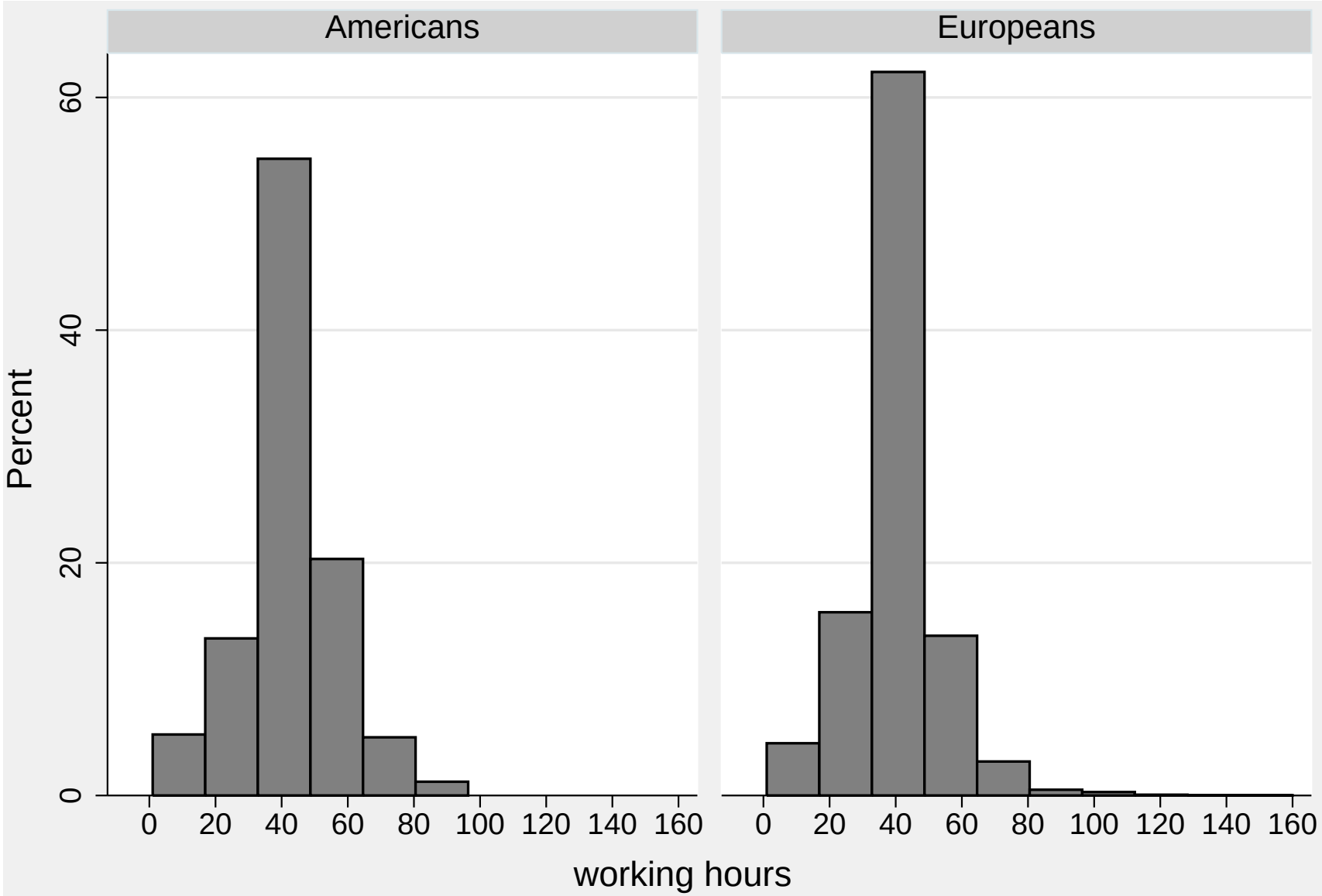
howto graph it?

- again, look at the galleries to get inspired!
  - those from syllabus and my ipynb are broken down by cat distribution / hist; ranking / bar charts, etc
- but also consider lev of measurment:
  - continuous
  - distribution
  - categorical, continuous, summary statistics
- lets break the discussion here in this way

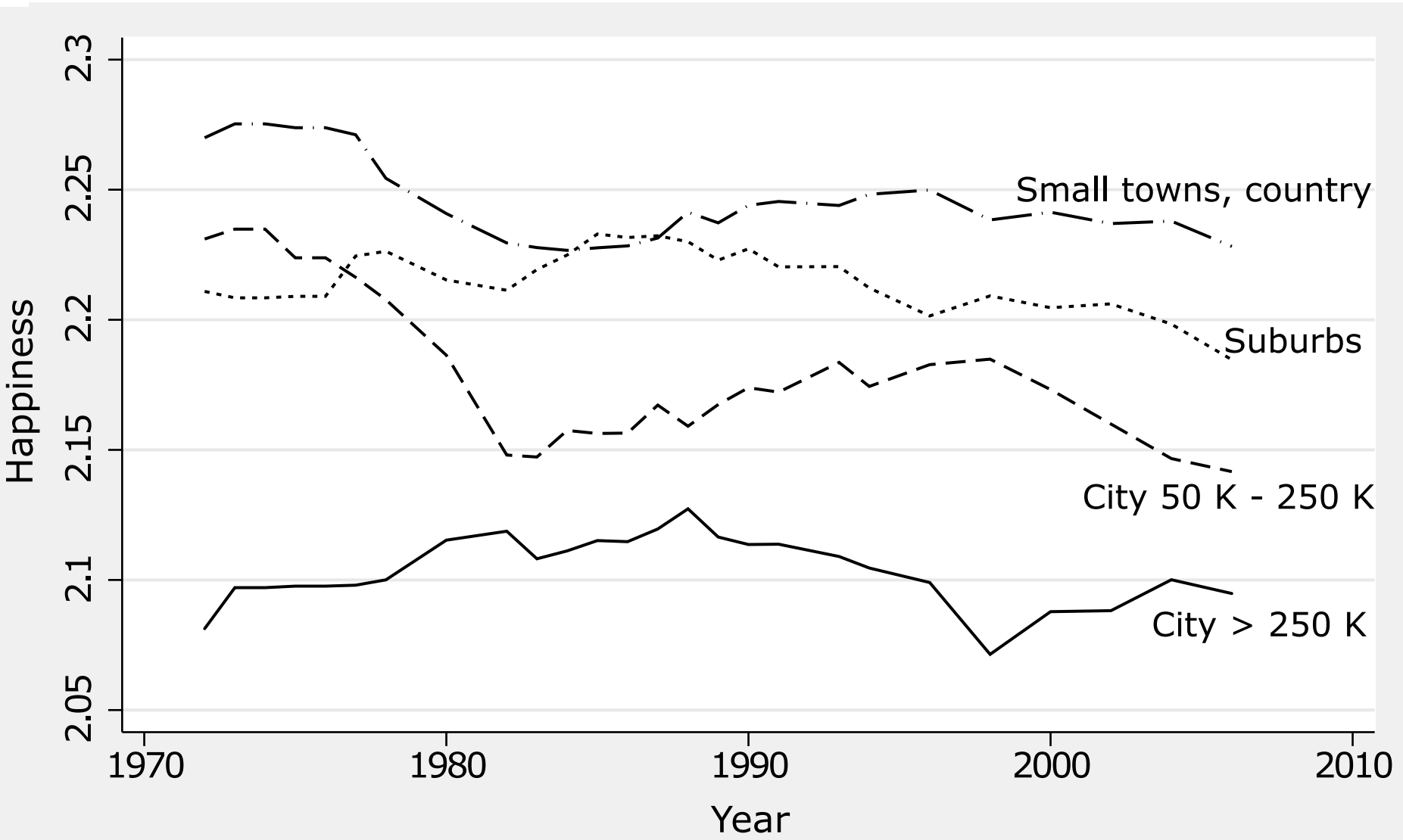
continuous vars

- continuous vars: scatterplots
- time series: lineplots
- combine a bunch of plots in matrix

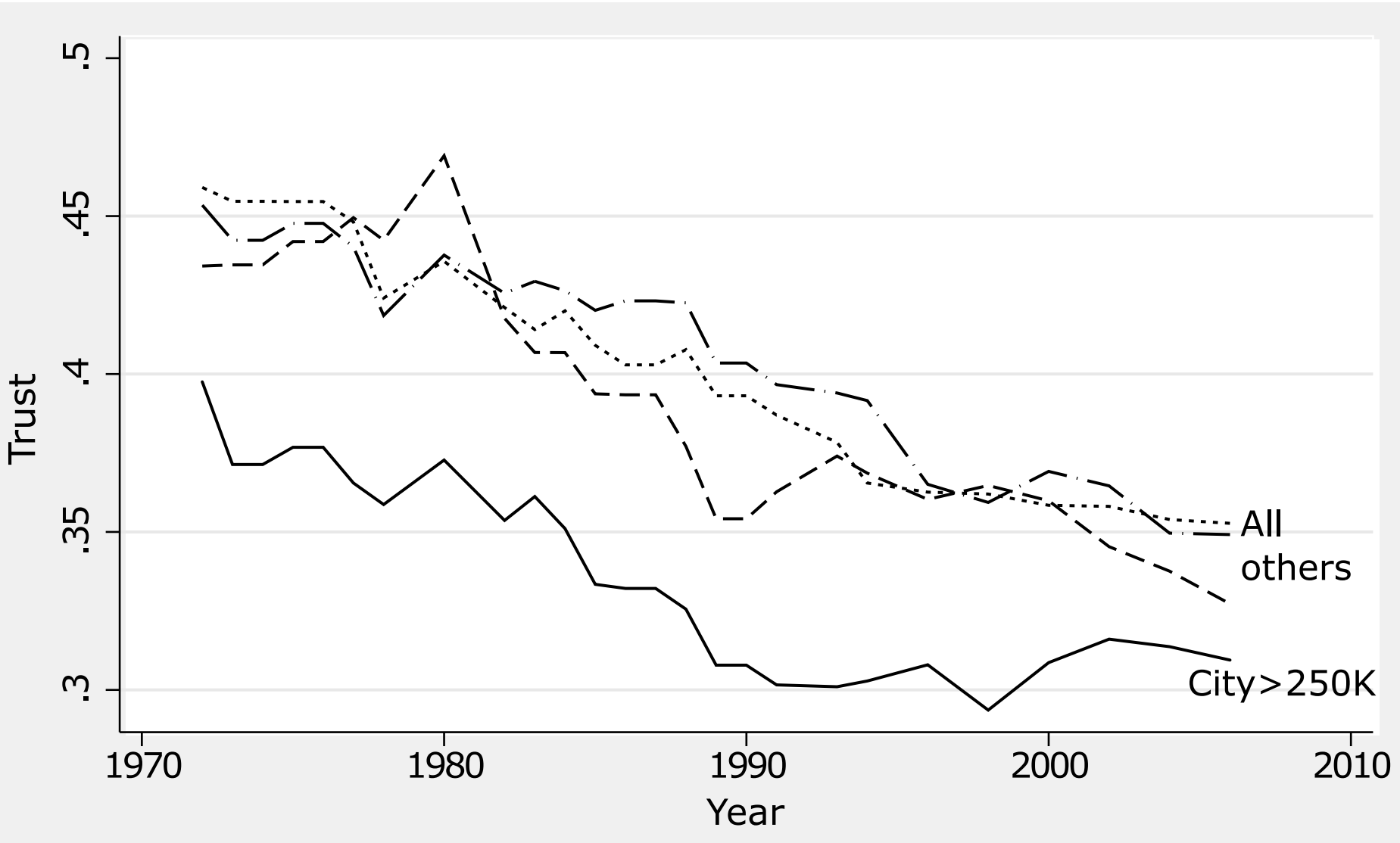
dist (subplots)



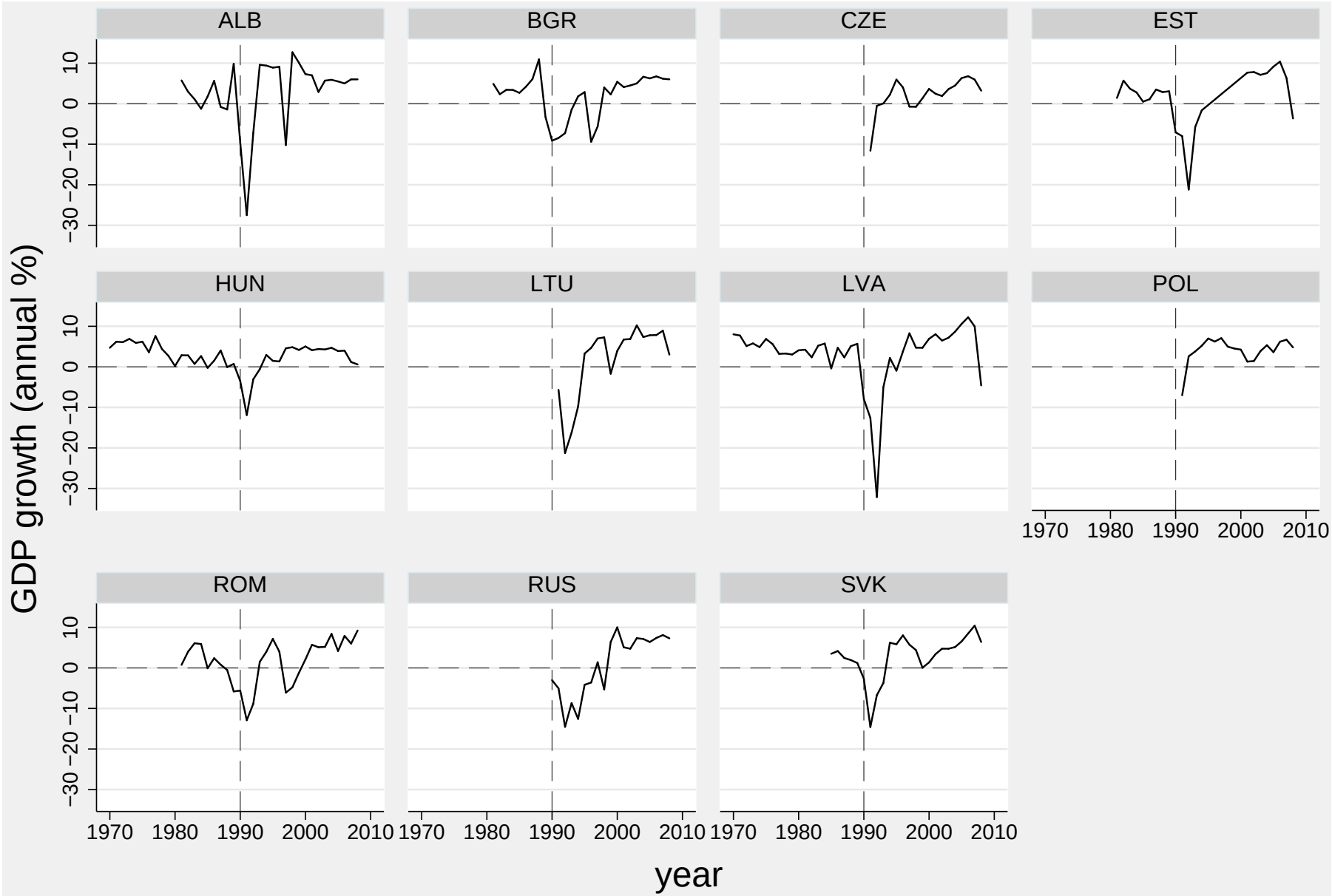
line


Happiness
2.05
2.1
2.15
2.2
2.25
2.3
1970
1980
1990
2000
2010
Year
Small towns, country
Suburbs
City 50 K - 250 K
City > 250 K

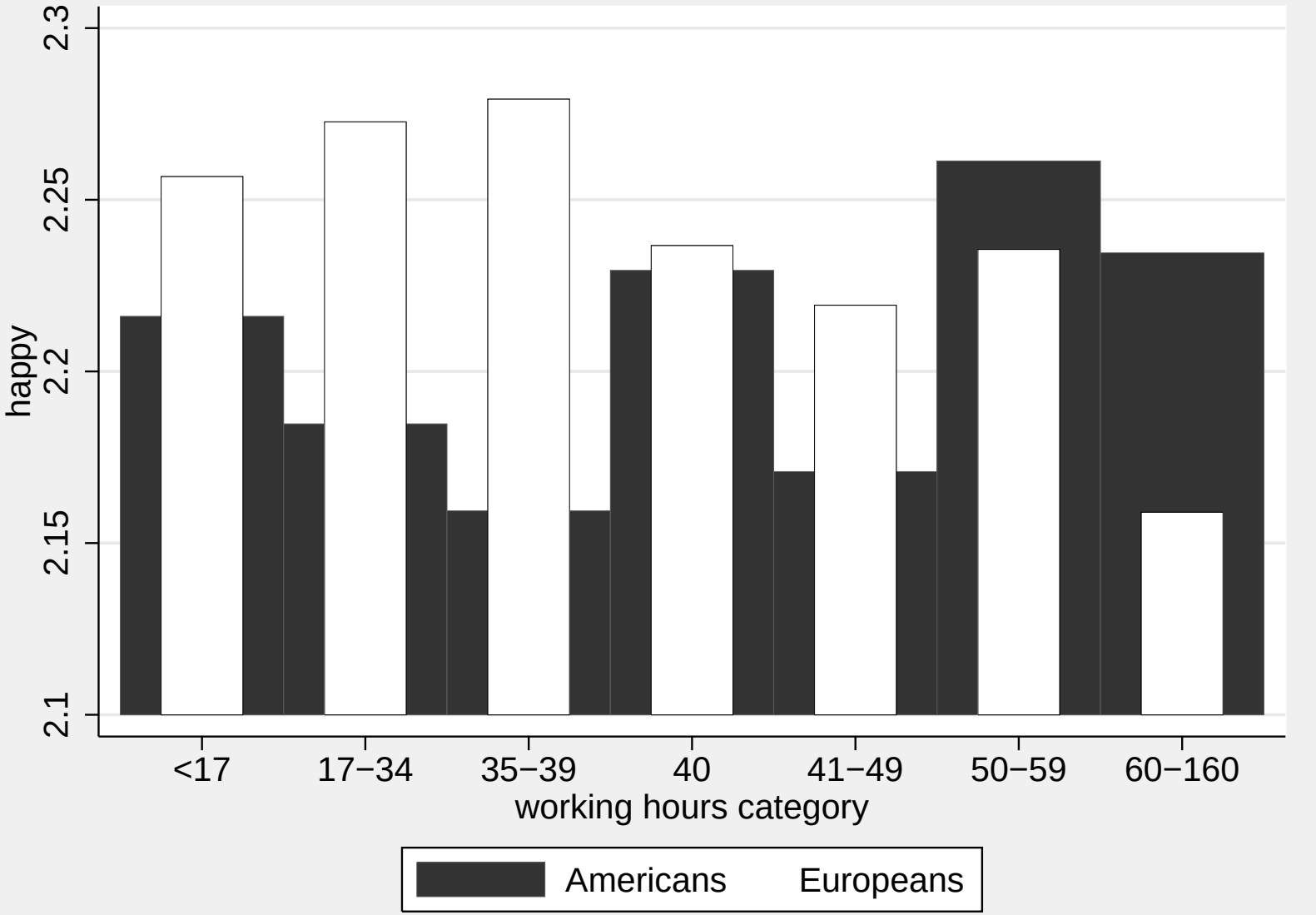
line


Trust
.3
.35
.4
.45
.5
1970
1980
1990
2000
2010
Year
All
others
City>250K

line (mpl: groupby+loop)



categorical, continuous, sum stat (bar overlay)



outline

theory (Tufte)

examples

strategy / howto graph it / galleries

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

Wilson put it well

- print out Box 1 from these 2 art
  - hang it at your office, home, and elsewhere
- `https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745`
- `https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510`
- and Know Your Data (2nd slide)
- and variations on these, and other general rules follow

simplicity, cleanliness, and organization!

- keep it as simple as possible
  - especially if overwhelmed or struggling
- say retain only 5vars and 25obs
  - much easier to understand such data
- simplicity transpancy clarity:
  - use fancy code: eg loops iff they simplify
- have chunks of code only once
- code it all from raw to final (replication principle)
- organize: sections, comments, and logical order (eg rewrite, move code around)

be fast/efficient

- the fancier the code, the more time/effort to write it
- don't do fancy things unless they save time in the long run
- it's all about managing complexity
- automate as much as you can
- simplify and be clear
- have general modules (sections or separate files)
  - that can be reused for different projects
- be lazy: don't reinvent the wheel–google often

outline

accuracy / correctness

- ◇ it's fundamental and obvious: vis bad if wrong
- ○ double check
- ○ especially at the beginning do not assume things
- ○ double/triple check the whole code once finished
- ○ go public, present, give it to others
- ● its human to err, there will be mistakes

1.write; 2.rewrite/reorganize; 3.optimize/improve

- 1. dump it, do "free writing" with code, too (i often come up with some idea out of sudden, and just write it down)
  - start simple and keep on adding things
- 2. rewrite/reorganize your code
- 3. optimize/improve
  - dont optimize too early, first make it work
  - (there is often a tendency to over optimize, ie spending days on small chunk of code that does not really matter that much)
- then can rewrite/reorganize again

rewrite and reorganize

- efficiency: few lines of code do many things
  - do more in fewer lines, drop unnecessary things
- reorganize and rewrite!
  - just like your papers: you print them out
  - and move paragraphs and words around
  - and you simplify and strike out unnecessary words
- do the same with code! drop everything you can!
  - code should be "tight"
  - as few lines as possible to perform given task
- use wrappers on more laborsome functions, google it, stack-overflow, etc
  - instead of mpl: seaborn, pandas, etc

efficiency: on the other hand

- but you also want to be extensive in a way
- in a good way...
- like with free writing, so with code
  - do “free writing”
- be expressive and dump your ideas into notebook
- just be organized so that you know what is going on!
- yes, by all means, be efficient–drop unnecessary things
- but do not drop things that may be useful
  - say in the future or other projects
  - may comment them out, move to LATER sec, etc

optimize/improve/get fancy

- think how to optimize
- related to efficiency: do more in fewer lines
  - but here instead of dropping unnecessary stuff
  - we get fancy: loop, list comprehension, your own function
- but first: dont reinvent the wheel, search for existing functions
  - only then optimize yourself, eg loop, write your own function
  - eg I hate my kludge with scatterplot labels, mistake prone, but looks like nobody wrote a function/wrapper to have it easy–will have to do it myself

simplicity: different, often opposite, from optimization

- ◇ people don't realize this!
- be as simple as possible in writing the code (papers, too)
- ◇ the more code you have (always try to get less) and the more complicated (optimized) it is:
  - ○ the more likely you have mistakes
  - ○ and the more difficult it is to find them
- do not complicate (optimize) your code for the sake of fanciness
  - ○ yes simpler is better

standardize

- ◇ standardize− >fewer mistakes (eg make fewer decisions, like a template, on autopilot)
- ○ standardize− >code more transparent, easier to find weird stuff, errors
- like have a template for some vis: say always hist for all key vars; scatterplot matrix for all continuous vars, etc
- and then have the creative part, vis for specific project

modularity

- break large tasks into small (manageable) blocks/components
  - (like in dissertation–don't overwhelm yourself doing everything at once)
- the components are like sections in a paper, step-by-step
- it is easy then to reuse these components
- so have separate ipynb for different tasks; and have sec and subsec within ipynb

automation (closely related to standardization)

- everything should be coded
- no copy-paste, point-and-click, etc
- dont use excel for anything!
  - ◇ automate as much as possible!
  - ◇ practical reason: faster! (in the long run)
  - ◇ technical reason: computers *never* make mistakes
  - ◇ eg pull automatically from database, upload to github etc

document

- have text fields in ipynb and # comments in code fields
- meaningful commit messages in git
- may have changelog (version, date and explain what changed, eg: 0.1 dumped raw ideas; 0.2 loaded X data; 0.21 loaded Y, Z data)
- difficult to overestimate importance of documentation
- note: typically, i underdocument, too

singularity

- have only one chunk of code and one file in one place, ideally in git
- as projects grow, get complex and maybe branch off, do have parents-children, and possibly branches
- elaborate: eg i do a lot with gss, draw flow-chart
- at first it was a first paper
- then i write 2nd paper, and realize most of dat man is the same
- so i create one root/parent with common code for all gss papers

portability

- import libs at the beginning, the fewer the better
  - i overdid, i need to cut down
- you could get version of key libs, ie pandas and mpl and save info
- but just 2 libs, can easily trace it back if sth doesnt run in couple years
- but do always save raw data! likely in couple years dataset may disappear or change
- also keep in mind magics (%matplotlib) for running in different environments eg spyder

tradeoffs: life is difficult

- ◇ simplicity is sometimes inversely (positively) related to efficiency (amount of code/being verbose) (as you cut stuff down, it may take longer to figure it out (parsimounious is sometimes complicated)
- ○ simplicity usually inveresely related to optimization (eg loops)
- ◇ simplicity often inversely related to automation (eg complicated code with if else)
- ◇ so make choices, the more serious you are about coding, the more work you do:
- ○ the more you should care for automation, efficiency, and optimization
- ○ the more automation/efficiency/optimization actually

tradeoffs: life is difficult

- ◇ so make choices, the more serious you are about coding, the more work you do:
- ◦ the more you should care for automation, efficiency, and optimization
- ◦ the more automation/efficiency/optimization actually simplifies
- • like Py v excel: excel simpler for simple tasks
- ◦ but Py is simpler for complicated tasks
- • ideally stick with one soft [maybe i'm quitting stata]

outline

theory (Tufte)

examples

strategy / howto graph it / galleries

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

more principles

- from 2 books about general programming (classics and free!)
  - `http://catb.org/esr/writings/taoup/`
  - `http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-1.html`

clarity

- "design for transparency and discoverability"
  - write clean code [eg split 1 fn over many l for readability]
  - avoid fancy code
  - fancy code is buggier
  - clarity is better than cleverness
- eg:
  - group logical chunks together
  - more than twice nested loops gets confusing
  - if your code is mostly loops, consider functions

modularity

- "write simple parts that are cleanly connected"
- "controlling complexity is the essence of computer programming"
  - debugging dominates development
- eg:
  - better many small loops that each does one thing, than one huge (>100 lines) loop that does everything
  - clear sections of one file
  - or many files instead of one file without sections

modularity

- code should be organized logically not chronologically
  - do free writing, but then reorganize
  - like with papers, code should be rewritten, eg:
  - no data management in data vis part
  - move rename, replace, etc earlier

composition

- "design programs to be connected to other programs"
- notebook or its sec will produce output for another notebook or sec
- eg: you clean up data in one file to make data ready for another one to vis
  - or just have one big file
- but the workflow needs to be logically organized

optimization (fancier, fewer lines)

- yes, but "get it working before optimizing" !
- eg:
  - first make mpl hist for one var, make it working
  - and then deploy it for 10 vars with a loop

extensibility

- "design for the future because it will be sooner than you think"
  - you will reuse your code in the near future
  - so write it clean
  - have sections, etc
  - use lots of comments
  - reorganize, rewrite
  - optimize

silence

- "when a program has nothing surprising to say, it should say nothing"
- drop unnecessary code
  - if you think it may be useful in the future comment it out
- do not generate unnecessary output, do not lose your reader in unnecessary clutter
  - if the output has nothing useful to say it should be dropped
  - (or commented out)

automation (again)

- "rule of generation: avoid hand-hacking"
- because humans make mistakes and computers don't, computers should replace humans wherever possible
- automate anything that you can
- but stay human, focus on fun creative part, eg vis
- dont automate everything; eg dont crank out bunch of vis mindlessly

save time: reuse (copy-paste), don't reinvent the wheel

- if someone has already solved a problem once, reuse it!
- it is very unlikely you are doing something completely new
  - eg google 'student data analysis python'
- if anything, the problem is that people do not share their code
- usually all you need to do is to adjust somebody else's code or your old code
  - its like doing lit rev, but with code
  - and with data too, eg google scholar psid biking to find out how people use biking var in psid

save time: reuse, don't reinvent the wheel

- ask people for code:
  - your supervisor
  - journal article authors
  - your colleagues, friends, etc
- share your code
  - you may want to protect some parts of it
  - (critical, innovative research ideas, etc)
  - but share as much as possible
- acknowledge others' work

defensive programing

- "people are dumb-make program bullet-proof"
  - you will find negative income, age over 200
- think of likely possibilities/instances; especially if you suspect some specific problems
- thats also why its so important to interpret critically your vis, if something looks funny or unlikely, maybe there's a mistake

outline

theory (Tufte)

examples

strategy / howto graph it / galleries

key scientific computing rules to get your started

elaboration, details, in other words; general rules

and yet one more variation on general rules

the zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.

- There should be one– and preferably only one –obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!