vis theory

adam okulicz-kozaryn adam.okulicz.kozaryn@gmail.com

this version: Friday 7th March, 2025 11:45

<u>outline</u>

- theory (Tufte)
- examples
- strategy / howto graph it / galleries
- key sci computing rules [next class]
- 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
- o vis allows humans to comprehend those many numbers
- o 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)
- \circ this is where you beat IT folks (MS/PhD just in IT)
- again, be prepared to invest a lot of time into your data
- sth you're passionate about
- o or can make \$ (now or in future career)
- this is where vis comes in-best way to get to know data
- don't forget to think about it, don't be mindless
- \circ ask questions, be investigative, be critical
- double check, cross check, give to others to check/present

<u>outline</u>

theory (Tufte)

- examples
- strategy / howto graph it / galleries
- key sci computing rules [next class]
- elaboration, details, in other words; general rules
- and yet one more variation on general rules
- the zen of Python

make it interesting and tell a story!

- its all about story telling
- the greatest value of a picture is when it forces us to notice what we never expected to see
- good stories are anomalous: Dog bites man is not news.
 Man bites dog is news.
- REFS:
- http://www.edwardtufte.com/
- http://www.stat.columbia.edu/~gelman/research/published/storytelling.pdf [ideally anomalous, but also immutable]
- 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
- no padding: put only data needed for a specific purpose
- no clutter: eg single graph must only present the data that are highly related and must be compared
- put into appendix if not very relevant but maybe useful
- \circ those looking for extra info will find it
- \circ those into the main story won't get distracted

avoid visual clutter

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



Extra leisure time enjoyed by men compared with women 2006*, minutes per day





not chartiunk (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:
- o low educ, poverty, crime, etc
- o but can also show a summary, eg an index
- and can show 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!
- o (beyond technical correctness; lack of mistakes)
- \circ as you look at it, ask yourself the 'so what?' question
- if not happy with the answer:
- \circ drop it, comment it out, or leave it as it is, and
- \circ 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 sci computing rules [next class]
- 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
- examples to inspire-all simple but great insight
- the idea counts most
- [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 Constration to Constation: The United States in Comparison" In Pohert Dusreft, ed. The Economics

20/68

McCain vote by income in a poor, middle-income, and rich state



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

(1918 din caused by 1918 Influenza Pandemic)

examples











Health Spending and Life Expectancy, 1970-2021



28/68

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



Total	4,788	1,070	\$51,735,374	\$6,368,361	12%
Inpatient	906	406	\$45,584,781	\$5,504,342	12%
ED	3992	978	\$6,150,592	\$864,019	14%
2007	790	369	\$7,979,262	\$901,181	11%
2005	738	355	\$6,857,995	\$881,549	13%
2005	835	370	\$10,834,420	\$1,269,373	12%
Virtua	805	331	\$1,742,467	\$345,092	20%
Lourdes	811	337	\$7,848,809	\$1,028,611	13%

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)		





Northgate II (3, 815) Northgate 1 (4,700) Ablett (Village (5.713)

Frank's Place (2.100)

Crestbury Apertments (4 765)

29/68

12%

examples



30/68

outline

theory (Tufte)

- examples
- strategy / howto graph it / galleries
- key sci computing rules [next class]
- elaboration, details, in other words; general rules
- and yet one more variation on general rules
- the zen of Python

strategy

- sometimes complicated or lengthy code esp for fine-tuning/customization, a ton of options, impossible to memorize
- way easier to figure it out with
- \circ examples/galleries: syll sec 'galleries', esp first 2 Py
- o or examples i gave you in ipynb
- copy from elsewhere
- \circ copy-paste from me, internet/google, chatGPT, etc
- o Gemini! code cell: generate with AI; explain code; fix error
- o but then adjust it, clean it up, streamline, simplify
- \circ and focus on story telling, your story
- google what you study "+ python notebook" strateg::how policing and crime python notebook"

howto graph it?

- again, galleries and my ipynb by cat: distribution / hist; ranking / bar charts, etc
- but also consider lev of measurment:
- o continuous: hist, scatterplot, lineplot
- o categorical: hist, tab, crosstab/heatmap
- bar chart! ranking, summary stats by grp and subgrp

outline

- theory (Tufte)
- examples
- strategy / howto graph it / galleries
- key sci computing rules [next class]
- 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
- \circ 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
- o especially if overwhelmed or struggling
- say retain only 5vars and 25obs
- o much easier to understand such data
- simplicity transpancy clarity:
- o 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)
- o that can be reused for different projects
- be lazy: don't reinvent the wheel-google often

outline

- theory (Tufte)
- examples
- strategy / howto graph it / galleries
- key sci computing rules [next class]
- elaboration, details, in other words; general rules
- and yet one more variation on general rules
- the zen of Python

accuracy / correctness

- it's fundamental and obvious: vis bad if wrongdouble check
- \circ especially at the beginning do not assume things
- \circ double/triple check the whole code once finished
- o 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)
- \circ start simple and keep on adding things
- 2. rewrite/reorganize your code
- 3. optimize/improve
- o 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!
- o just like your papers: you print them out
- and move paragraphs and words around
- \circ and you simplify and strike out unnecessary words
- do the same with code! drop everything you can!
- o code should be "tight"
- o as few lines as possible to perform given task
- use wrappers on more laborsome functions, google it, stack-overflow, etc
- o instead of mpl: seaborn, pandas, etc

elaboration, details, in other words; general rules

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
- o 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
- o say in the future or other projects
- \circ 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
- o 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

- o 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:
- o the more likely you have mistakes
- o 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
- \circ 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

elaborate heramore automation / efficiency / optimization actually 51/68

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
- o 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 sci computing rules [next class]
- 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!)
- O http://catb.org/esr/writings/taoup/
- O http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-1.html

clarity

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

modularity

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

modularity

- code should be organized logically not chronologically
- o do free writing, but then reorganize
- o like with papers, code should be rewritten, eg:
- o no data management in data vis part
- o 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
- o 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:
- o first make mpl hist for one var, make it working
- \circ and then deploy it for 10 vars with a loop

extensibility

- "design for the future because it will be sooner than you think"
- o you will reuse your code in the near future
- o so write it clean
- o have sections, etc
- \circ use lots of comments
- o reorganize, rewrite
- o 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
- o 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
- \circ 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:
- o your supervisor
- o journal article authors
- o your colleagues, friends, etc
- share your code
- o you may want to protect some parts of it
- (critical, innovative research ideas, etc)
- o 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 sci computing rules [next class]
- 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!