

To: SRMSNET@LISTSERV.UMD.EDU
From: Ben Earnhart <Benjamin-Earnhart@uiowa.edu>
Subject: respect for data
Cc:
Bcc: Kelly Richardson <kkrichar@blue.weeg.uiowa.edu>
Attached:

Dear list --

I work for a Sociology department, and among other things, support people doing secondary data analysis. I am getting tired of seeing generations of graduate students making the same mistakes, and am thinking of putting together some sort of a handout that lists some of the major dos and don'ts of working with data -- help them start out with good habits. This it's a bit off-topic, but I hope that it's interesting and/or relevant enough to list members -- the best methods themselves aren't any good if you have bad data. If anybody knows of any good resources along these lines, I'd love to hear about them, and if you could, please have a look at my tentative list of dos and don'ts below, and add to it if you have any pet peeves or common mistakes people make. Most of the things below are often taken for granted, but it's amazing how often I see the same easily avoidable problems come up.

Thanks for any reactions, feedback, and/or contributions.

Ben

1) Use syntax. It's OK to use point-and-click to create syntax in programs such as SPSS that allow it, but always run the syntax, and save it. You should also read the manual to understand what the syntax means. It may seem quicker to rename a variable by point-and-click, or do a simple re-code this way, but if you have to do it over again (and again, and again) it's **not** quicker. In addition, point-and-click manipulation of data would require that you are always right and have a 100% perfect memory. Only your advisor and departmental secretary have these abilities.

2) Save your log files and output you generate, as well as the syntax used to generate it. If you ever have to say "I don't know how I got these numbers" to yourself (or worse, your advisor), you've got a big, big problem.

3) Use comments liberally in your syntax. You should have a comment at the beginning of each syntax file explaining when you wrote it, what it's supposed to do, and what files are related; each time you switch to a new basic task within a given syntax file, you should make a comment. When doing particularly complicated manipulations, you may even want to put a comment for each line of code. Refer to specific pages of the codebook whenever possible so that you're never stuck having to say to yourself (or worse, to your advisor) "I don't know why I did that."

4) Have a naming convention for your files. You normally want the data, syntax, and log files to have the same name, or similar enough you know at a glance what goes with what.

- 5) Related to the naming convention, when doing things in multiple steps, it's generally best to have things in smaller files, so if you goof up, you don't endanger all your work. Generally each step might have the same name but a different number on the end (recodes1, recodes2, recodes3, etc.) so you can easily backtrack. Also, this can greatly speed up your jobs -- it is **not** necessary to read the data in from ASCII and do 100 pages of recodes every time you do an analysis.
- 6) Analysis and data manipulation should not occur in the same step -- at the end of the data manipulation phase, save to a system file, then read it in for analysis. Not only will this speed up your jobs, it will prevent you from inadvertently doing analysis on data that has changed.
- 7) When working with subsets of a large file (for example, the cumulative GSS, which is repeated cross-section but most people only use a single year for a given project, or Census data for which most people only care about a single level of analysis), discard as many cases and variables as early as you can and save as a separate file. There is no need to load up 200,000 cases and 5000 variables every time to work with 50 variables on 1000 cases.
- 8) Keep a journal/diary of what files you worked with and what they did -- update it each day you work. That way, if you need to backtrack, you can get the whole project at a glance.
- 9) Use a directory structure to stay organized. With large projects, files multiply fast, and it can get confusing unless you use sub-directories. It's good to have a journal/diary for each sub-directory, though if the one at the top level is complete enough, it's OK to just have the one.
- 10) Network drives are slower than local drives -- getting a faster processor will not help if it's starving for data to process. If you're working with large datasets, the bottleneck will be the network -- don't complain the computer is too slow when it's the data access that is slowing things down. So copy your data over locally, then when you're finished, copy things you want to save back to the network drive. Of course, if you're running the job remotely on a server, this is no longer true -- then what you need to speed up your jobs is a bigger monitor and better headphones. A more comfortable chair might help as well.
- 11) Back things up! Syntax files and output are very small, there is **no** excuse for not backing them up to multiple locations. This will not only save you from losses due to things like hard drive failures, but also from inadvertently over-writing an important file. Back up systematically, so you know what version of a file you are working with. A handy way to do this is to create compressed (zipped) files with the day's date as the filename so you know what is what.
- 12) As a caveat to the principle of backing up often, do **not** go crazy with backing up large data files -- filling up a shared drive with numerous identical or nearly identical copies of the same data is absurd and can irritate your computer techs to the point where they start imposing disk space quotas -- and this will make you rather unpopular when other users find out why quotas are being imposed. If you practice good habits with your syntax, you can quickly and easily replicate the data with the syntax if needed. So keep a copy of the original data safe, and your few most recent versions, but don't keep many versions around unless you anticipate going back to them often.
- 13) Never recode into the same variable name if it is at all possible to avoid (with the possible

exception of setting missing values) -- when a variable means one thing at one stage of a project, and another thing at another stage, it can really mess things up. If you do recode into the same variable name, at least change the variable label to reflect the change. And when creating new variables, attach variable labels unless the meaning is obvious enough from the variable name; however, note that the only test for whether it is "obvious enough" is if your advisor agrees that the name adequately reflects the variable label you attached. With dummy variables, consider putting what a value of 1 means in the variable label so you (and perhaps more importantly, others) know at a glance what direction it is coded.

14) RTFM/RTFC (Read The [beeping] Manual/Read the [beeping] Codebook). Especially be aware of skip patterns, and never assume a variable means what you think it does based on just the name and label. Never assume a syntax command will do exactly what you expect, either, and be aware of switches/options for modifying the behavior of a command. This will not only help you to avoid outright mistakes, but can let you get things done with much less effort. It will also save you from great embarrassment for those times when you do need to get help from others -- at least try to use your available resources.

15) Be careful about missing value codes. Sometimes the reason a variable is missing can be very, very important -- be aware what the different missing value codes in your data mean. This will help you to avoid pregnant men and people who pay for the privilege of going to work (e.g. have negative hourly wages).

16) After constructing a new variable, always run a check on it to make sure it did what you thought it did; cross-tabs against the original variables can be a good way to do this. Pay attention to both the meaning, and the Ns. For a series of dummy variables that are supposed to be mutually exclusive and completely exhaustive, summing them should add up to 1; this check on your coding is probably the only time a variable with a variance of zero is a good thing. Do occasional sanity checks with descriptives on all the variables -- this will help you weed out those pregnant men and bored rich people early on. Correlation matrices can be a good sanity check as well, though they can contaminate how you think about the relationships in the data, so probably should be avoided in the data construction phase.

17) When you're using a method that requires listwise deletion (which is "evil," but that's another story), consider the issue early on, and be prepared for how to deal with your shrinking N from the beginning. Sure, each variable might have only 2% missing, but by the time a final dataset is constructed for analysis, it's easy to lose a large proportion of cases just through random missingness. Watch your N! Before beginning analyses, determine your actual usable sample - do **not** just start regressing away and watch the N shrink as you add variables to the models.

18) Run descriptives before every analysis. As well as helping you to interpret the output from the main analysis, this can act as a sanity check -- mins and maxes are good for catching missing value codes that slipped through, and can help diagnose problems with your N. In addition, a variable with a variance of 0 is going to not be very useful in your analysis.

19) When doing commands that you are not sure are right and using huge datasets, use options to only access a portion of the file. Do not run it on the full N until you have reason to believe you got your syntax right -- waiting for 5 minutes each run to get the same error over and over is just plain silly (though it does give you a chance to browse the web and complain that you need a

faster computer).

20) Build in checks on mathematically impossible transformations. When diagnosing an error, consider what it is you are trying to do, and read the log. "Division by 0 is impossible" you may have forgotten about since high school math, but it's still just as impossible today as it was then, and your stats package will be happy to remind you of this fact, *if* you read the log. Square roots of negatives... well, while not technically impossible, you better have a heck of a theory to explain why you can and will do that, and program your own software to do it. And no matter how good your theory is, logarithms of negative numbers are problematic as well.

Wow. This list got a *lot* longer than I'd expected. If you made it this far, congratulations for making it through, and thanks for reading it all! Hmm... it got a bit patronizing, will want to make the tone more professional in later drafts, but hopefully you were at least mildly entertained if you made it this far. I did not mean to imply that graduate students at Iowa are dumb (some of them are incredibly bright), just that it's frustrating to see the same problems over and over, and want to help people avoid some common errors. -- Ben

*=====

*Ben Earnhart

*Computer Consultant and ICPSR Data Assistant

*Department of Sociology and College of Liberal Arts

*University of Iowa

*(319) 335-2887

*bearnhar@blue.weeg.uiowa.edu

*=====;