Practical Data Issues Lab

The purpose of today's lab is to gain experience with practical issues of data handling. Toward that end, you will practice constructing and evaluating scales, reshaping datasets, imputing missing values,

Scale Construction

1. Read and complete activities from Sønderskov, Chapter 7 and Appendix F.

Dataset Aggregation

2. Once again, open the **Election05Ch6** dataset. Your task here is to aggregate the data from this file by county (*amt*). Recall that the Danish local governments were reformed in 2007 from 270 *kommuner* in 13 *amter* (pre-reform) to 98 municipalities in five regions (post-reform). These survey data record respondents' location by amt (**v8**). Use the following to create a new variable called **county**: `decode v8, gen(county)`

3. Now, we want to know the expected election results by county, so we'll focus on the variables **partytd1**, **partytd2**, etc. Reduce the dataset to just **county** and these variables. Note the number of variables and the number of observations in the dataset.

4. We want to create a dataset that consists of the proportion of votes for each party in each county. In essence, this is a dataset of the summary statistics of the **partytd*** variables, by **county**. Run a `summ` command by **county** to see our intended result. We'll use this to judge whether the `collapse` command, which we'll use next, was successful.

5. Now, it's time to aggregate using the `collapse` command. The `collapse` has many options, but in simple terms it looks like this: `collapse (statistic) varlist, by(groupvar)`, where `statistic` is the statistic we want to aggregate or summarize, `varlist` is a list of variables to aggregate (anything not in this list will be dropped from the dataset), and `groupvar` is a variable that will be aggregated by. The resulting dataset should have a number of observations equivalent to the number of unique values of `groupvar`. Aggregate the proportion of the sample in each county that voted for each party: `collapse (mean) partytd*, by(county)`. Because these variables are 0/1 indicators, the mean is equal to the proportion of 1's in the dataset.

6. Note the number of variables and number of observations. And use summary statistics (e.g., `tab`, `summ`, etc.) to compare the values in the aggregated dataset to the summary statistics from the original data. Were we successful in aggregating the data? Save the aggregated dataset as **SurveyAggregated.dta**.

Merging Datasets

7. Now try merging two datasets to see a useful application of the `collapse` command when combined with the `merge` command. Open the **Election2005Results.dta** dataset (available on Blackboard). This dataset contains the official 2005 election results (numbers of votes) for seven major parties, by post-reform municipality.[1]

8. Examine the number of observations in the dataset. What do you notice? If we want to merge these data with the survey data to compare the survey results with the official results, what do we have to do?

9. We need to collapse these results to the pre-reform county level. To do this, we need to know what county each current municipality used to be in. On Blackboard, you'll find the **Municipalities.dta** dataset, which maps the 270 former municipalities to the pre-reform counties and the 98 current municipalities.[2] Use the following code to map each post-reform municipality to one county:

```
* tag one post-reform municipality at random
egen tmp = tag(municipality)
* drop all untagged municipalities
drop if tmp == 0
drop tmp prereform
```

   The data should be two variables: post-reform municipality and pre-reform county.

10. Using these data, perform a 1:1 merge of the **Election2005Results.dta** into this dataset:
`merge 1:1 municipality using Election2005Results.dta`

11. Was the merge successful? If it was, use `collapse` to aggregate this dataset by county. Remember to specify `sum` as the statistic to aggregate (rather than `mean`, as we used above). Was the collapse successful? How many observations are there? At this point, you may want to create a **total** variable that sums the party vote totals and then convert the party vote counts into percentages.

---

[1] These data be retrieved from: `http://www.statbank.dk/statbank55/SelectVarVal/Define.asp?Maintable=VALGK3&PLanguage=1`

[2] With a few exceptions, each current municipality is made up of former municipalities that were exclusive to one county. The post-reform Egedal Kommune, for example, is made up pre-reform municipalities from both Københavns Amt and Frederiksborg Amt. We'll ignore this and assume each post-reform municipality comes from only one county.

12. Now, use the `merge` command to merge **SurveyAggregated.dta** into this dataset, by county. Was the merge successful?

13. This exercise was hopefully relatively easy, though perhaps tedious. It would be much more challenging, however, if variables in each dataset had different names or if the values for variables with the same names (e.g., **county**) had different values (e.g., due to spelling errors, etc.). You could rename the variables and try this again. If you want further practice, you may want to read and complete activities from Sønderskov, Appendix B. See also examples from `help merge`.

Dataset Reshaping

To practice reshaping, we will use the Quality of Government Time-Series dataset available at: `http://www.qogdata.pol.gu.se/data/qog_bas_ts_jan15.dta`.

14. Open the dataset in Stata. Note the size of the dataset. How many variables are there and how many observations are there? Use `tab cname` to see how many countries are included in the dataset and how many observations there are for each country. Is the data in **wide** or **long** format?

15. Given that we're going to be doing a lot to the dataset, it's useful to learn the `snapshot save` and `snapshot restore` commands. These keep a record of your dataset and allow you to return to the restore point later on. Run each of the following one at a time to get a sense of how this works:

```
snapshot save, label("original")
keep cname
describe
snapshot list
snapshot restore 1
```

You can create multiple snapshots (and restore them by their number), making this very useful for going back and forth between versions of a dataset.

16. Back to reshaping. To reshape data, we use the `reshape` command. To go from wide to long, the command starts with `reshape long` and the reverse is `reshape wide`. If you just try typing `reshape wide`, though, you'll get an error. You need to specify what variables to reshape and how. Let's start by reducing our dataset to just one substantive variable and then reshape:

```
keep ccodealp cname year p_polity
reshape wide p_polity, i(cname) j(year)
```

17. Now describe the data. What's the data structure look like? How many observations are there and what do they represent? How many variables are there and what do they represent? Are these data in wide or long format?

18. Restore to our original dataset. Now let's try reshaping with multiple variables. We'll use just the fractionalization variables from Alesina et al. which have the prefix `al_`.

```
keep ccodealp cname year al_*
reshape wide al_*, i(cname) j(year)
```

19. Again, describe the data. What's the data structure look like? How many observations are there and what do they represent? How many variables are there and what do they represent? Are these data in wide or long format?

20. At this point, Stata now recognizes the structure of the data, so you can toggle back and forth between wide and long format simply by calling `reshape long` and `reshape wide`. Try this for yourself and then restore to our original data.

21. Try to reshape from long to wide, again, this time specifying additional variables in addition to `al_*`. This can be a sequence of full variable names or patterns (like `bdm_*`). You can also preserve additional "constant" variables in the `i()` option, for example: `i(cname ccode ccodealp)`. These variables will be preserved in wide format.

22. Stata makes a lot of assumptions about how the reshaping should be done. The most important is that variable names in wide format need to have consistent year names. We can disrupt this and observe the consequences:

```
snapshot restore 1
tab year if year > 2000, summarize(al_ethnic)
keep ccodealp cname year al_*
reshape wide al_*, i(cname) j(year)
snapshot save, label("wide")
rename al_ethnic2005 al_ethnic5
reshape long
tab year if year > 2000, summarize(al_ethnic)
```

23. Note how Stata lost the 2005 data because the variable was not named in the same way as all the others. This then has important consequences for how we create variables in wide format. Return to wide format and then create three variables as follows:

```
gen newvar = al_ethnic2000 - al_religion2000
gen diff2000 = al_ethnic2000 - al_religion2000
gen diff2010 = al_ethnic2010 - al_religion2010
```

Then, we're going to manually reshape from wide to long, as if we'd just opened these data for the first time. To do this, we have to specify variable name "stubs" (i.e., the name of all of the time-variant variables without the year portion of the variable name). The syntax is similar to going from long to wide, because we specify time-invariant variables in `i()` and we specify the name of a new variable that serves as an indicator for year `j(year)`. Run the reshape and then see how `newvar` and the `diff` variables come out in the resulting dataset.

```
reshape long al_ethnic al_language al_religion diff, i(cname) j(year)
tab newvar
tab diff
tab diff year
```

Note how `newvar` is assumed to be time-invariant even though we didn't manually specify it `i()`. Every country-year observation now has the same value of `newvar`. The `diff2000` and `diff2010` variables were handled differently, though. Because they had a year in their name (like all the other time-variant variables), they were assumed to be time-variant, as well. If we do not specify `diff` in the `varlist` portion of the `reshape` command, then these variables are assumed to be time-invariant (just like `newvar` was by default). Knowing how to convert data from wide to long and vice versa is an invaluable skill to have, but the process can produce all kinds of errors. When you encounter errors, read the error messages, use the help files, and examine your data to see what might be causing the problem(s).

Missing Data Handling

For this part of the lab, we will use data from the General Social Survey, a nationally representative survey of the United States population. These data, in Stata (.dta) format, are available to you on Blackboard. These data are used in an article we'll revisit later in the course.[3]

24. Open the **GSS2002.dta** dataset from Blackboard. Let's start by recoding our key independent and outcome variables, which have to do with voluntary activities and one's type of employment (public vs. private sector):

```
* outcomes
recode volchrty 1=1 2=0.8 3=0.6 4=0.4 5=0.2 6=0, gen(vol)
recode givchrty 1=1 2=0.8 3=0.6 4=0.4 5=0.2 6=0, gen(char)
recode givblood 1=1 2=0.8 3=0.6 4=0.4 5=0.2 6=0, gen(blood)
* key regressor
recode wrkgovt 2=0, gen(pubemp)
* covariates
* income is ‘rincom98‘
* sex is in ‘sex‘
* race is in ‘race‘
* marital status is ‘marital‘
```

[3]Houston, David J. 2005. "'Walking the Walk' of Public Service Motivation: Public Employees and Charitable Gifts of Time, Blood, and Money." *Journal of Public Administration Research and Theory* 16: 67–86. doi:10.1093/jopart/mui028.

25. Remember: there is missing data in some of our covariates, which we'll want to address. In the GSS data, missing values are coded in three ways:

    - `.d`: don't know
    - `.n`: no answer
    - `.i`: not asked

    The multiple imputation algorithms in Stata only play well with traditional missing values (coded as `.`) so, for the sake of consistency, it is best to recode those early on before doing imputation of any kind.[4] Here's some example code:

```
recode rincom98 (.d=.) (.n=.) (.i=.), gen(mincome)
recode educ (.d=.) (.n=.) (.i=.), gen(meduc)
recode madeg (.d=.) (.n=.) (.i=.), gen(mmoredu)
recode padeg (.d=.) (.n=.) (.i=.), gen(mfaredu)
recode age (.d=.) (.n=.) (.i=.), gen(mage)
```

26. As we saw in lecture, it is easy to unconsciously discard a lot of missing data simply because Stata's default is to rely on available case analysis. Compare the following two estimates of the effects of being a public employee:

```
reg vol pubemp
reg vol pubemp mincome
```

    Simply by adding income as a covariate, we discard about 400 observations from our data and obtain a (slightly) different marginal effect of public employment. Is that change due to confounding (i.e., that income confounded the effect of employment status?) or due to nonresponse bias (i.e., that people with reported incomes differ from those who do not report incomes?).

27. If data are MCAR (missing completely at random), then our estimates in any analysis are unbiased but may be inefficient. To see this, randomly exclude some observations from our regression (by randomly recoding some values of a variable to missing) to see that it has no significant effect on the results:

```
* sort by a randomly generated variable
gen random = runiform()
sort random
* generate an MCAR version of ‘pubemp‘
gen pubempmcar = pubemp
recode pubempmcar (*=.) in 1/300
* original results
reg vol pubemp
* MCAR results
reg vol pubempmcar
```

[4] Of course, it may be the case in a particular situation that you want to handle different kind of missingness in different ways, perhaps treating "don't know" responses as a specific value, but recoding "no answer", etc.

The sample size in the second model (with MCAR missingness) is smaller, the standard errors are larger, and the estimated effects are comparable. We can rely on almost any imputation technique to increase our sample size and thus reduce variance in our estimates.

28. If data are MAR (missing at random/conditionally ignorable), however, we should rely on multiple imputation, which will both produce a more complete dataset and should eliminate any nonresponse bias due to observed variables. We cannot test whether data are MCAR but we can test whether missingness is plausibly MAR/ignorable by regressing an indicator variable for missingness in a variable on other observed (ideally fully observed) variables. To do this, generate a new variable to represent missingness and run a regression of the new variable on other variables in the dataset.[5] You can repeat this process for any variable that has missingness. If any of the variables are related to the missingness indicator, the data are — at least partially — ignorable.

    You can try this first on the MCAR missingness we just introduced into `pubempmcar`:

    ```
    recode pubempmcar (.=1) (else=0), gen(ismiss)
    tab ismiss
    reg ismiss sex i.race pubemp age
    ```

    In this example, we know that the data are MCAR so any significant effects here are spurious. But in general we can't know if data are MCAR, so this is our best test of whether we should develop an imputation model that accounts missingness due to observed values. It may still be the case that they are also missing due to unobserved variables, but this is something we cannot test.

Descriptive Statistics

Before we actually do any imputation, we need some understanding of where the missing data are in our dataset. This section walks through some simple diagnostics.

29. The simplest way to assess missingness in a variable is simply to tabulate it using the `tab` command and the option `, miss`. Of course, you can only tabulate one variable at a time, but it will give you the basic information you need.

30. Another option that allows you to quickly count missing values in variables is available in an add-on package called `nmissing`. You can install this using `ssc install nmissing`. Then, you can just type `nmissing` to see counts of missing values in all variables in the dataset. By specifying a list of variables you can limit the results to just the specified variables. Try this out.

31. The `nmissing` command only shows us raw counts of missingness, but that doesn't help us to understand *where* the missing values are in the data. For example, it's

[5] Generally, you would use a logistic regression to do this, which accommodates 0/1 dependent variables, but we haven't covered those in this course yet.

possible that a given observation has many missing values while other observations have few or no missing values. This can emerge for example, if a survey respondent terminates a survey interview before the end of the survey, or if a country stops reporting a particular measure (e.g., inflation) during years of dictatorship. To see how missingness is distributed across variables for a particular observation, we want to see *patterns* of missingness. We'll need another add-on package to do this. It is called `mvpatterns` and you can install it by using `findit mvpatterns`.

32. To use `mvpatterns`, simply call `mvpatterns varlist`, where `varlist` includes the name of one or more variables. The output will include two tables. The first provides counts of missing values in variables (a la `nmissing`). The second part of the output provides quite a bit of additional information. Here's an example:

```
Patterns of missing values
  +------------------------+
  | _pattern   _mv   _freq |
  |------------------------|
  |     ++..     2    2075 |
  |     ++++     0     659 |
  |     +++.     1      21 |
  |     ....     4       7 |
  |     ++.+     1       2 |
  |------------------------|
  |     ...+     3       1 |
  +------------------------+
```

The first column shows *patterns* of missingness, with a `+` indicating a present value and `.` indicating a missing value. So, if four variables are included in the varlist (as in the above example), the first row with pattern `++..` indicates that there are 2075 observations with missing values in the third and fourth variables. The second row indicates that there are 659 observations with no missing values. In the second-to-last row of the table you see there is also one observation with one missing value in a different pattern of missingness compared to row 3, which also shows observations with one missing value. This output thus lets us know not only how much missingness there is, but how it is distributed across observations and variables. Try it for yourself.

33. We can apply this to an example of missingness we've already encountered: when we want to construct a scale where individual items contain missingness. The GSS data contain seven questions about empathy. Items 2, 4, and 5 are "reverse coded," so let's start by flipping them and then seeing what kind of scale we can build from them.

```
* flip reverse coded items
recode empathy2 empathy4 empathy5 (1=5) (2=4) (3=3) (4=2) (5=1)
cor empathy*
egen empscale = rowmean(empathy*)
```

Try using `mvpatterns empathy* empscale` to see how missing values are handled when we construct a simple scale. The `rowmean()` function ignores missing values even, for example, if six out of seven items are missing for a respondent. It may be the case that we're uncomfortable with using the scale when there are too many missing values in the original items. We could, therefore selectively drop values when there are, e.g., more than 3 missing values in the original scale:

```
egen empmiss = rowmiss(empathy*)
replace empscale=. if empmiss>3
```

You can make a decision for yourself about how best to scale these variables, given the missing values.

Single Imputation

34. Now that you've seen where your missing data are, you can address them with simple single imputation techniques (though these are not really recommended since it is easy to do multiple imputation). But, we'll walk through them because sometimes if you have just a few missing values, it is easy and relatively unproblematic to fill in a few missing values via some kind of single imputation process.

35. Single imputation is basically just recoding, so you already know how to do it. For the simplest methods of zero and mean imputation, you simply are using the `recode` command and/or `gen` command. It's good practice when doing single imputation (as in any recoding) to generate new post-imputation variables rather than overwrite the original variables.

36. A simple example of when it is appropriate to use single imputation is when you know, due to a value on one variable, what the value on another variable should be. For example, `browser1` in the GSS data reports whether the respondents knows what an internet browser is. If they say no (value 2), we can reasonably certain they do not use the internet. As such, responses to other questions about internet use, such as `musicget` or `litauth` should be zero. Try out zero imputation using: `recode musicget (*=0) if browser1==2, gen(impmusicget)`

    You can imagine other situations where you would impute a value other than 0 or possibly do a *sensitivity analysis* and compare your results when imputing all values bottom-coded or top-coded. The procedure is identical.

37. Another common single imputation is mean imputation. Recall that mean imputation has the nice property of not affecting the mean of a variable. This is nice if our interest is in estimating a population mean, but imputing the mean will affect bivariate or multivariate results. For example, with the empathy data, if we replace missing values in `empathy1` with the mean of `empathy1`, it will have no effect on the mean of the variable or correlations:

```
egen emp1imp = mean(empathy1)
replace emp1imp = empathy1 if !mi(empathy1)
```

```
summ empathy1 emp1imp
tab empathy1 emp1imp, miss
cor empathy1 emp1imp empathy1
```

Note how the mean of the variable is unaffected, as are its correlations with other variables, but we artificially deflate its variance.

38. Random value imputation, which preserves the variance of a variable after imputation, is a bit more challenging. While the code to do this is relatively easily, it does require a few lines of code. First, we need to sort the data, then tabulate the variable with missingness to figure out how many missing values there are in the variable, and finally replace the missing values with random draws from the observed values. Here's an example using our recoded income variable:

    ```
    sort mincome
    quietly tab mincome
    gen tincome = mincome[round((r(N)-1)*uniform(),1)+1]
    summ mincome tincome
    tab mincome, miss
    tab tincome, miss
    ```

    Note how the mean and variance differ slightly between the two variables. Random imputation preserves the mean and variance of the original variable only *in expectation*, not in any particular realization. It also changes the correlations between variables:
    `cor mincome tincome empscale`. This is because it does not take into account the relationships between variables that might determine the pattern of missingness.

39. Hot deck imputation is a classic multivariate single imputation method that does take account of possible multivariate patterns of missingness, which dates to when most analysis was performed with punchcards and tabulating machines. We can emulate the hot deck imputation process in Stata by sorting our data according to a list of variables and then manually imputing the values. This never a recommended technique (in part because it is time consuming and moreso because the process is prone to human error) but it is valuable to try it out to get a sense of how it works and how much easier research has gotten in the last thirty years. All you have to do is sort your dataset by a specified list of variables, then use the `edit` command to open the data editor. An example set of commands is:

    ```
    sort varlist
    edit missingvar varlist
    ```

    You can then follow the "last observation carried forward" rule, starting from the top of the dataset whenever there are missing values. You might, for example try:

    ```
    gen hotdeck = rincom98
    summ hotdeck
    tab hotdeck
    ```

```
sort gender race age
edit hotdeck gender race age
summ hotdeck
tab hotdeck
```

40. A final single imputation technique that we can consider is *regression imputation*, which is a form of within-sample prediction. In regression imputation, we predict a partially missing variable based on a regression model. We then use that estimated model to calculate predicted values for all observations. For observations where the variable is missing, we use the imputed values and we use the observed values for all other observations. We can repeat this for multiple variables that contain missingness. Here's an example using generic variable labels:

    ```
    quietly reg missvar varlist
    predict fitted
    gen missvarimputed = missvar
    replace missvarimputed = fitted if missvar==.
    ```

    In many applications of regression imputation we would also add an additional amount of randomness to the imputations (such as a draw from the Normal distribution with mean zero and a variance equal to the variance of the residuals from the imputation model). We won't do this here, but it's important to keep in mind that the above form of deterministic imputation understates uncertainty about the missing values.

Multiple Imputation

41. Single imputation using regression is a logical segue to multiple imputation, which uses the same basic logic of building an imputation model but instead of imputing values once, we construct multiple imputed datasets, estimate our desired test statistic(s), and then average across the results from each imputed dataset. We'll focus on the Giving Blood (**blood**) outcome in the GSS data.

42. In Stata, before we can run any imputation we need to do two steps. One involves calling `mi set wide`, which tells Stata that we're going to multiply impute data in wide format. (Note: You can also set other formats instead of `wide`; see `help mi set`.) Then we need to `mi register` the variables that we're going to use in the imputation. Stata expects us to register three types of variables:

    - `imputed` variables: Those that have missingness that you want to impute.
    - `regular` variables: Those that are complete (i.e., have no missing values) or at least should not be imputed.
    - `passive` variables: These are not used in imputation and consist of things like transformations of other variables in the set of regular variables. And you don't have to register them, or you could just include them in the list of regular variables.

Now, try registering some variables as imputed and regular using the `mi register` command:

```
mi register imputed mincome meduc mmoredu mfaredu mage
mi register regular sex race marital poplog
```

43. At this point, the `mi describe` command will supply some basic information about what variables you've registered, including how many missing values there are in the registered imputed variables.

44. Because multiple imputation relies on (pseudo-)random number generation (RNG) as part of its algorithm, you may want to use the `set seed` command to specify a seed for Stata's (RNG) algorithm. This way you can reproduce the exact same results each time you run your code. If you leave this out, running the same code may produce different results each time because Stata is starting from a different random seed. You can specify it with something like: `set seed 1234`, or whatever your favorite number is.

45. Now it's time to impute using the `mi impute` command. Here we specify the variables we want to impute as a function of our "regular" variables from earlier. `mi impute` supports a number of different imputation algorithms and they're worth exploring. Each basically imputes the missing values as some form of regression model. Several are design for imputing missing values in a single variable, while a few (including the one we'll focus on, `mvn`) can impute missing values in multiple variables simultaneously.[6] Try this out using our variables:

    ```
    mi impute mvn mincome meduc mmoredu mfaredu mage = sex i.race i.marital,
    add(5)
    ```

    The `, add(5)` tells Stata to generate five new imputed datasets and hold onto them. If we've already run some imputations, this will add to the imputed datasets already in memory. You can see these imputations as new variables added to your existing dataset. You can also overwrite previous results using a `, replace` option.

    The output of this command describes details of the MI algorithm and the results of the imputation process. Specifically, a table will report how many observations in each variable are incomplete (i.e., missing) and how many are imputed. Ideally these numbers will match, meaning that all missing values were imputed.

46. Very Important Note: These examples use a very simple imputation model. This is probably not appropriate. As much effort should be put into generating a credible imputation model as is put into generating a credible causal regression model. You already have all the tools you need to build credible regression models, so we don't dwell on it here, but you may want to think seriously about the missingness generation problem and how best to model it (e.g., what variables, what functional form, interactions, etc.).

[6] A common alternative to the `mvn` (multivariate Normal) algorithm, is called MICE and is available in Stata using the command `mi impute chained`. This is quite a bit more complicated than `mvn`, though it may be appropriate if the values you are imputing are categorical, or at least not Normally distributed.

47. Once you have created an imputed dataset, it is now possible to analyze the results using `mi estimate`.[7] We need to use this command in order for Stata to look in the imputed datasets and appropriately aggregate the separate results. We can still run any standard estimation commands on our original data during this time. Let's use the Houston example, first by seeing our original (unimputed) results and then the results from the imputation:

```
reg blood i.pubemp sex race mincome meduc mage
mi estimate: reg blood i.pubemp sex race mincome meduc mage
```

48. Stata, by default, aggregates the multiple imputation results. You can, however, see the results from individual imputed datasets by using the `mi xeq` command. For example, we can see the results for our original data using the `mi xeq 0:` prefix and results for individual imputed datasets by replacing 0 with any number between 1 and the number of imputations in memory. You can see that number using `mi describe` or by simply looking at the bottom of your variable list. Try it out:

```
* these two should match:
reg blood i.pubemp sex race mincome meduc mage
mi xeq 0: reg blood i.pubemp sex race mincome meduc mage

* these are the results from individual imputed datasets
forvalues i = 1/5 {
mi xeq `i': reg blood i.pubemp sex race mincome meduc mage
}
```

To get a handle on what's going on here, note that Stata is using the generated variables (at the bottom of your varlist) to produce the estimates from each imputation. We can generate some new variables based upon those variables and compare the results to see that they are identical:

```
gen tmpi = _1_mincome
gen tmpe = _1_meduc
gen tmpa = _1_mage
reg blood i.pubemp sex race tmpi tmpe tmpa
* compare to imputation dataset 1
mi xeq 1: reg blood i.pubemp sex race mincome meduc mage
```

[7]Note: `mi estimate` supports most estimation commands, but not all. See `help mi estimate` for a complete list.