Community Resource: Egenmore, by command, return lists, and system variables.
Beksahn Jang
Feb 22nd, 2016
SOC561

Egenmore

Egenmore is a package in Stata that extends the capabilities of the egen command. A good description is available at:
http://funwithstata.blogspot.com/?view=sidebar
To download egenmore, simply type:

ssc install egenmore

The challenge in reviewing the egenmore package is covering the breadth of its options. Here I cover a few egen options not covered in the above link.

wordof(), **word**(#)

This option of egen returns the #th word (from word(#) of the string variable specified in wordof(). It is easily applicable to our auto data in stata (auto.dta). For instance, if we want to create a variable the captures the name of the car company we will need to create a variable that takes the first word in the *make* variable. This can be done with the following syntax:

egen firstword=wordof(make), word(1)
list firstword make in 1/15

	firstw~d	make
1.	AMC	AMC Concord
2.	AMC	AMC Pacer
3.	AMC	AMC Spirit
4.	Buick	Buick Century
5.	Buick	Buick Electra
6.	Buick	Buick LeSabre
7.	Buick	Buick Opel
8.	Buick	Buick Regal
9.	Buick	Buick Riviera
10.	Buick	Buick Skylark
11.	Cad.	Cad. Deville
12.	Cad.	Cad. Eldorado
13.	Cad.	Cad. Seville
14.	Chev.	Chev. Chevette
15.	Chev.	Chev. Impala

As we can see, the new variable *firstword* has taken the first word from the make variable, providing us with a string variable for the name of the car company.

Rsum2(varlist)

This option is very much like egen's rsum() function except it has the extra options ", allmiss" and ", anymiss." It creates a row sum of the variables in *varlist*,

but if the option "allmiss" is specified then if all the variables in *varlist* are missing it will be set to missing instead of 0. If the "anymiss" option is selected the row sum for any observation for which any variable in varlist is missing is set to missing. Here I use the publicly available 1993 GSS data:
(http://www.thearda.com/Archive/Files/Descriptions/GSS1993.asp)
First, I am going to generate a new variable for education that recodes values of 98 as missing. I will also generate a new variable for *childs* that recodes 9 as missing.

clonevar educ2=educ
replace educ2=. if educ==98
clonevar childs2=childs
replace childs2=. if childs==9
egen checkmissing4=rsum2(educ2 childs2), anymiss
list checkmissing4 educ2 childs2 if checkmissing4==.

	checkm~4	educ2	childs2
91.	.	3 years	.
129.	.	6 years	.
496.	.	.	Six
509.	.	.	None
668.	.	11th gra	.
1225.	.	.	Two
1279.	.	.	None
1339.	.	12th gra	.
1343.	.	9th grad	.

We can see that checkmissing4 is equal to "." when either educ2 or childs2 is missing. Having specified the "anymiss" option, this is what we would hope to see.

first(varname) [, **by**(byvarlist)]
This command returns the first non-missing value of varname. What comes first will depend on what basis the observations are sorted. This can be useful if we want to compare the lowest prices for foreign and domestic cars in the auto dataset. We can do this with the following commands:
First, we sort the data by price

sort price

Next we generate our variable

egen compareprice=first(price), by(foreign)
tab compareprice foreign

```
. tab foreign compareprice
```

Car type	compareprice 3291	3748	Total
Domestic	52	0	52
Foreign	0	22	22
Total	52	22	74

(*note: there are no missing observations in the auto dataset)
As we can see here, the lowest price for the domestic car is roughly $450 less than the lowest price for the foreign car. The data was sorted from lowest to highest price and the first nonmissing value was selected from each subgroup of foreign, essentially grabbing the lowest value of price within the subgroups of the foreign and domestic cars.

outside(varname) [, by(byvarlist) factor(#)]
This option is useful when looking for extreme values (outliers) within a variable's distribution. Specifically, it calculates outside values as any values more than factor(#) times the IQR from the nearer quartile. It has a default factor of 1.5 if no factor option is specified.
Example: again, let's use the auto dataset. We want to create a variable that identifies outliers for the distribution of miles per gallon (*mpg*). We won't specify a factor given that the default factor value is most commonly used to calculate the value of outliers.

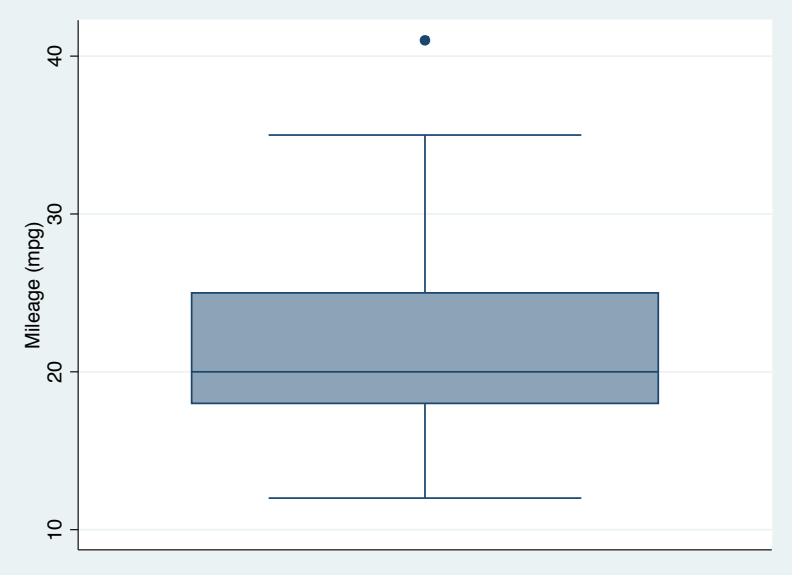
egen extrmpg=outside(mpg)
tab extrmpg, missing

```
. tab extrmpg, missing
```

extrmpg	Freq.	Percent	Cum.
41	1	1.35	1.35
.	73	98.65	100.00
Total	74	100.00	

From the tabulation we can see that there was only one value of mpg that was identified as an outlier. It also coded values that were not outliers as missing values. A quick check with a boxplot shows the same results:

graph box mpg



Return lists in stata:

Return lists are results stored following the use of a command, such as the running of a regression model. See the following website for a review of return lists: http://funwithstata.blogspot.com/2014/02/return-lists-in-stata.html?view=sidebar . While I have no intention of reproducing the work done in the linked blog, I think it is important to mention a few key items:

There are three types of return lists of interest: r-class, e-class, and s-class (n-class excluded).

R-class are return results for general commands and are stored in r()

S-class are for parsing commands and are stored in s()

*note: s-class is used solely by programmers, so it will not see much attention here.

E-class are for estimation commands and are stored in e()

Each form of stored results thus corresponds with a type of command that is run before any r(), e(), or s() commands/variables can be run or used. General commands include things like *summarize* and *describe.* Estimation commands include items such as regression models.

Command types and types of results stored

You will notice that, depending upon which command is run, different r(), s(), and e() variables will be accessible. For example, using the auto dataset (auto.dta) we can look at the stored data available after running a *summarize* and *describe* command:

summarize rep78
return list

This gives us:

scalars:

r(N) = 69

r(sum_w) = 69
r(mean) = 3.405797101449275
r(Var) = .9799658994032396
r(sd) = .9899322701090412
r(min) = 1
r(max)= 5
r(sum)= 235

describe rep78
return list

This gives us:
scalars:
r(changed) = 1
r(width) = 51
r(k) = 14
r(N) = 74

The commands preceding *return list* produce different data stored in memory, so only certain r() variables are available depending on the command that was run most recently. Once the command *describe rep78* is run after *summarize rep78*, many of the r() variables are wiped from memory, such as r(sum_w), r(mean), r(Var), etc.
Example:
Using auto.dta I ran the following commands

summarize rep78
describe rep78

I attempted to generate a new variable:

gen newvar=r(mean)

Interestingly enough, although the command ran, it only produced missing values:

(74 missing values generated)

It is therefore important to keep in mind what r() variables are available conditioned on the command most recently run. It is also important to remember which variable (such as rep78, make, mpg, price, etc) was specified in the most recently used command.

One important thing to note is that commands of one class type do not replace the memory created by commands of another class type. That is, a command that produces r-class stored results will not interfere with e-class stored results. R-class stored results will be replaced when another r-class type command is run. As such, you can run a summarize command followed by a regression and still have the r() variables from the summarize command available.

Applications of r()

R() variables can be used for many things. *Summarize price, detail* with *return list* provides the following r() variables:
scalars:
r(N) = 74
r(sum_w) = 74
r(mean) = 6165.256756756757
r(Var) = 8699525.97426879

r(sd) = 2949.495884768919
r(skewness) = 1.653433511704859
r(kurtosis) = 4.819187528464004
r(sum)= 456229
r(min) = 3291
r(max)= 15906
r(p1) = 3291
r(p5) = 3748
r(p10) = 3895
r(p25) = 4195
r(p50) = 5006.5
r(p75) = 6342
r(p90) = 11385
r(p95) = 13466
r(p99) = 15906

We can use these r() variables to do basic calculations. For example:

*display r(N)*r(mean)*
*assert r(N)*r(mean)==r(sum)*
assert r(sd)^2==r(Var)

We see that the assertions are true, as expected.
If we want to look only at those values above the median value we can choose to keep only those values greater than r(p50):

keep if price>r(p50)

(37 observations deleted)

sort price
list price in 1/10

	price
1.	5,079
2.	5,104
3.	5,172
4.	5,189
5.	5,222
6.	5,379
7.	5,397
8.	5,705
9.	5,719
10.	5,788

We can see that these values are all above the median value of 5006.5.
We could also show how much more expensive each car is than the cheapest car:

generate comparecheap=(price-r(p1))
sort price

list comparecheap in 1/10

comparechea p	Freq.	Percent	Cum.
0	1	10.00	10.00
8	1	10.00	20.00
376	1	10.00	30.00
457	1	10.00	40.00
507	1	10.00	50.00
508	1	10.00	60.00
538	1	10.00	70.00
604	1	10.00	80.00
664	1	10.00	90.00
693	1	10.00	100.00
Total	10	100.00	

As we would expect, the first value is zero.

System variables:

System variables, also referred to as "_variables," are built-in variables in Stata. For a brief review of the definitions of system variables, refer to http://www.stata.com/help.cgi?_variables or simply type "help _variables" in stata. I will not review their definitions here.

Using _n and _N for counting:

Of the system variables, _n and _N are likely the most useful. They are frequently used in combination with *by* and *bysort*. Used together, they can be used for counting within subgroups.

Using the auto dataset (*sysuse auto*), we can run the following commands:

bysort mpg: generate subgrpcount=_n
bysort mpg: generate subgrptotal=_N
list subgrptotal subgrpcount in 1/15

	subgrp~t	subgrp~l
1.	1	2
2.	2	2
3.	1	6
4.	2	6
5.	3	6
6.	4	6
7.	5	6
8.	6	6
9.	1	2
10.	2	2
11.	1	4
12.	2	4
13.	3	4
14.	4	4
15.	1	4

Since we sorted by mpg before we generated the variable *subgrpcount, subgrpcount* will now give us the number of the observation within the subgroup on which it was sorted, whereas subgrptotal will tell us the total size of each subgroup.

Using _n and _N to identify duplicates:

_n and _N are also useful for identifying duplicate values in variables and in subgroups. I provide an in-depth example of this in a section discussing the *by* command. Below I discuss some simpler methods of identifying duplicates.

To identify duplicate values for a single variable using _n we can run the following command:

list if anyvar == anyvar[_n + 1]

Anyvar can be any variable for which you would like to know if that variable has any duplicate values. ID is most often used in this case. If the list does not return anything then no two values of *anyvar* are the same.

Using _N to identify duplicates in a small dataset with 3 variables:

Sort var1 var2 var3

By var1 var2 var3: generate subtotal=_N

*note: system variables cannot be used directly in a command (such as describe _N), so to capture its value you can generate a variable equal to it (e.g. subtotal=_N).

List if subtotal>1

This one is very much like the example in the *by* command section. It is asking the data to run a list if any subgroup has a total number of observations greater than 1. If that is not the case then it will list those observations. Note: this is more suitable for smaller datasets with few variables.

Additional resources:

http://funwithstata.blogspot.com/2014/02/system-variables-in-stata.html?view=sidebar

http://www.stata.com/help.cgi?_variables

The *by* command

The *by* command allows you to run stata commands on groups of variables. You can think of it as temporarily dividing the dataset into multiple mini-datasets and then running the specified command on those subgroups. The subgroups are defined by the variables listed following *by*.
This is the basic syntax:

by varlist: stata_command

For example, using the auto dataset (auto.dta) in stata, you could run the following command:

by foreign: summarize weight

The following output:

-> foreign = Domestic

Variable	Obs	Mean	Std. Dev.	Min	Max
weight	52	3317.115	695.3637	1800	4840

-> foreign = Foreign

Variable	Obs	Mean	Std. Dev.	Min	Max
weight	22	2315.909	433.0035	1760	3420

See http://funwithstata.blogspot.com/2014/02/all-about-by.html?view=sidebar for some additional examples using *by()*.

Similar to the *by* command is the *bysort* command which also sorts the data according to the variables listed following *bysort*. The *bysort* command is useful because often it is necessary to sort the data before a by command will run.
For example, using auto.dta:

by foreign (make): summarize rep78

This returns an error:

Not sorted
r(5)

You can resolve this by sorting and then running a by command:

sort foreign make
by foreign (make): summarize rep78

OR
You could just run a *bysort* command:

bysort foreign make: summarize rep78

Using *by* and *bysort* with *_n* and *_N*

The *by* and *bysort* command used in conjunction with _n and _N are often useful in resolving problems with multilevel data (nested data). Specifically, *by*, *bysort, _n,* and *_N* are useful in the steps prior to merging data. One of the key challenges before merging data is to identify the unique identifiers that create unique identifiers for each observation. The id variable is a unique identifier for most datasets. However, in nested data there can be many observations for each respondent id.

Let us suppose that we have data on individual respondents' (*id*) grades for each class (*class*) they took every semester (*semester*) at all the universities (*institution*) they attended in the last two years. To merge the data requires that we match observations one to one. However, id, class, institution, and semester each by themselves cannot uniquely identify the data. Using the bysort command, however, we can identify a combination of these variables that uniquely identifies each observation:

bysort id institution semester class: assert _n==_N

If the assertion is **not** false then we have identified the variables that uniquely identify each observation. Why is this the case?

We have to keep in mind that for the last observation in a subgroup, _n is == to _N. This is also true when there is *only one observation in each subgroup*. As such, the fact that _n==_N implies that each subgroup has a size of 1 and no larger. Once each subgroup has a size of 1 we can say that the combination of sorted variables listed after the *by* command together uniquely identify every observation. Alternatively, you can test this with the following command:

bysort id institution semester class: assert _n==1

Lastly, it is important to mention that when using the *bysort* command with multiple variables, the variables are sorted giving the first variable in the list priority and then the second in the list, etc. This means that "*bysort id institution semester class*" will first sort the observations according to id values and then within each value of id it will sort according to institutions, and then for semester and class etc. You will want to be careful about the order in which you sort the variables.

Additional resources:

http://funwithstata.blogspot.com/2014/02/all-about-by.html?view=sidebar

http://www.cpc.unc.edu/research/tools/data_analysis/statatutorial/misc/by_detail

Stata *help by()*:

http://www.stata.com/help.cgi?by