The Stata Journal (2010)
10, Number 4, pp. 686–688

Stata tip 92: Manual implementation of permutations and bootstraps

Lars Ängquist
Institute of Preventive Medicine
Copenhagen University Hospitals
Copenhagen, Denmark
la@ipm.regionh.dk

In mathematics, a permutation might be seen as a reordering of an ordered set of abstract elements (see, for example, Fraleigh [2002]),[1] whereas in data analysis—when facing empirical data—this concept may correspond to a reordering of an ordered set of observations. Vaguely speaking, in statistics and significance testing, this might be an interesting concept when simulating under a null hypothesis corresponding to, in some sense, a null association or an effect (most often an outcome) of one specific variable with respect to another one. Here one basically keeps the dataset constant except for the values, which are instead randomly permuted, corresponding to the core variable. Because all permutations are generally equally likely (at least if properly dealing with potential confounding) under the null hypothesis of no association, this is a way, through such simulations, of estimating the corresponding null distribution underlying, for instance, related p-values.

For similar reasons, one may apply the bootstrap simulation procedure. Here one does not reorder observations (or in general elements) but rather simulates from the empirical distribution based on this very set. In simulation terminology, the bootstrap and permutation procedures in this sense correspond to a uniformly random selection of values from the empirical distribution with and without replacement, respectively. For more information, see, for example, Manly (2007) for permutations, Davison and Hinkley (1997) for bootstraps, and Robert and Casella (2004) for stochastic simulation, in general.

In Stata, one may—given some assumed framework—use the commands `permute` and `bootstrap` to perform tasks related to permutation-based and bootstrap-based significance tests, respectively. Sometimes however, whether it arises as a need to be more specific or because one simply wants to keep more detailed control over the actual data manipulations, it might be favorable to perform some related manual labor at your computer keyboard. This tip is about the general structure of a solution for such a task.

Permuting: Assume that you have a variable of interest, `permvar`, that you want to permute in the sense noted above. Typing

1. The set of all possible reorderings (permutations) includes the permutation that actually leaves the order intact. This is called the identity permutation.


```
generate id=_n
generate double u=runiform()
sort u
local type: type permvar
generate `type´ upermvar=permvar[id]
```

in Stata will give you an additional column (`upermvar`) of permuted values. In the first command, a new variable, `id`, that corresponds to the current sort order is created.[2] In the second command, a column is generated with values uniformly distributed between 0 and 1. Because the values of `u` were randomly generated, sorting on `u` puts the observations in a random order. The next command saves the variable type of `permvar` in the local macro `type` so that the type can be applied to the new variable in the last command. The last command stores the permutation in the new variable `upermvar`: each new value is a value of `permvar` from a randomly selected observation. (The random selection is controlled by the `id` variable, which was put in a random order by the `sort` command.)

To reduce the risk of tied values with respect to the (inherently discrete) random draws, and moreover to further increase the, so to speak, randomness of the derived values, one might replace the code lines 2–3 with the following:

```
...
generate double u1=runiform()
generate double u2=runiform()
sort u1 u2
...
```

The randomness reference corresponds to the fact that computer-generated random numbers are random only to the extent permitted by the implementation of what is termed pseudorandom numbers (see, for instance, Knuth [1998]). To achieve reproducible results, one might take advantage of this pseudorandomness by explicitly stating a starting point, that is, a seed value, for the deterministic algorithm:

```
set seed 760130
```

The number must be a positive integer. For instance, this command might be used when assuring that different methods give equivalent results or, for example, with respect to estimated variances of certain derived estimates of interest, when comparing methods with respect to efficiency performance.[3]

2. In other words, this construction is based on the observation number indicator `_n`, which equals 1, 2, ..., N through the present observations (rows), where N is the number of observations in the dataset (generally reachable in a similar fashion through `_N` in Stata). Moreover, one approach to retaining a sort order, irrespective of the content of an executed program, is by taking advantage of the `sortpreserve` option (see `help program` or Newson [2004]).
3. You might use your personal birthdate as an easily remembered seed value. This is in fact used in the above case, though I am not revealing which date format I used; see `help dates and times`. I thank Claus Holst for this tip!

Bootstrapping: A related but slightly different variant of the above schedule might be used to derive a bootstrapped variable called `ubootsvar`. It is based on the empirical distribution formed or constituted by the present observations of the original variable `bootsvar`.

```
generate u=ceil(runiform()*_N)
generate ubootsvar=bootsvar[u]
```

Here the uniformly distributed values are not used to decide on a sort order (the underlying index values), but rather to directly constitute index values by making them be part of a uniformly distributed simulation of values on the integers 1, 2, ..., N. To achieve this, the so-called ceiling function, `ceil()`, is used. For more information on `runiform()`, see `help runiform` or Buis (2007)[4] (with respect to its use for simulations); further, `ceil()` and the related `floor()` function are described in Cox (2003).

Moreover, one might implement the above code structures into loops based on, for instance, `foreach` or `forvalues`. Under such circumstances, one might also take advantage of both usage of temporary variables (see `help tempvar`) and the specific matrix-oriented environment of Mata (see `help mata`) though the general structure described here might to some extent serve as a guideline or a template for such cases, as well. Once ready, strap your boots and let the permutation begin!

References

Buis, M. L. 2007. Stata tip 48: Discrete uses for uniform(). *Stata Journal* 7: 434–435.

Cox, N. J. 2003. Stata tip 2: Building with floors and ceilings. *Stata Journal* 3: 446–447.

Davison, A. C., and D. V. Hinkley. 1997. *Bootstrap Methods and Their Application*. Cambridge: Cambridge University Press.

Fraleigh, J. B. 2002. *A First Course in Abstract Algebra*. 7th ed. Reading, MA: Addison–Wesley.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Reading, MA: Addison–Wesley.

Manly, B. F. J. 2007. *Randomization, Bootstrap and Monte Carlo Methods in Biology*. 3rd ed. Boca Raton, FL: Chapman & Hall/CRC.

Newson, R. 2004. Stata tip 5: Ensuring programs preserve dataset sort order. *Stata Journal* 4: 94.

Robert, C. P., and G. Casella. 2004. *Monte Carlo Statistical Methods*. 2nd ed. New York: Springer.

4. The `uniform()` function was improved in Stata 11 and was renamed `runiform()`.