Title

stata.com

permute — Permutation tests

Description

`permute` performs permutation tests using Monte Carlo permutations or by enumeration of all possible distinct permutations. A single variable is chosen to be permuted, and the permutation distribution is estimated (or in the case of enumeration, fully determined) for specified statistics returned by a Stata command or a user-written program.

Quick start

Estimate p-values for a permutation test of the coefficient of `x` in a linear regression, permuting values of the outcome `y`

```
permute y _b[x]: regress y x
```

Test for `r(mystat)` returned by program `myprog`, permuting values of `y`

```
permute y r(mystat): myprog
```

Same as above, but increase the number of permutations from the default of 100 to 10,000

```
permute y r(mystat), reps(10000): myprog
```

Same as above, but display a dot for every 100 permutations instead of every permutation

```
permute y r(mystat), reps(10000) dots(100): myprog
```

Same as above, but set the random-number seed for reproducibility, and save the permuted statistics in `myfile.dta`

```
permute y r(mystat), reps(10000) dots(100) rseed(1) saving(myfile):  ///
     myprog
```

Test for `r(mystat1)` and `r(mystat2)`, naming the statistics `stat1` and `stat2`, respectively

```
permute y stat1=r(mystat1) stat2=r(mystat2), reps(10000) rseed(1):   ///
     myprog
```

Perform permutations within strata defined by `svar`

```
permute y stat=r(mystat), reps(10000) rseed(1) strata(svar): myprog
```

Enumerate the full permutation distribution and display a dot for every 1,000 permutations

```
permute y stat=r(mystat), enumerate dots(1000): myprog
```

Menu

Statistics > Resampling > Permutation tests

Syntax

Perform permutation test

```
permute permvar exp_list [ , options ] : command
```

Report saved results

```
permute [ varlist ] [ using filename ] [ , display_options ]
```

options	Description
Main	
`reps(#)`	perform # Monte Carlo permutations; default is `reps(100)`
`enumerate`	compute all possible distinct permutations
Options	
`rseed(#)`	set random-number seed to #
`strata(varlist)`	permute within strata
`saving(filename, ...)`	save results to *filename* with options for saving in double precision and saving results to *filename* every # permutations
Reporting	
`standardize`	standardize test statistic using permutation distribution mean and variance
`level(#)`	set confidence level; default is `level(95)`
`title(text)`	use *text* as title for permutation results
`dots(#)`	display dots every # permutations
`nodots`	suppress permutation dots
`nowarning`	do not warn when `e(sample)` is not set
`noisily`	display any output from *command*
`trace`	trace *command*
`verbose`	display full table legend
`noheader`	suppress table header
`nolegend`	suppress table legend
Advanced	
`nodrop`	do not drop observations
`reject(exp)`	specify criterion for invalid results
`eps(#)`	numerical tolerance; seldom used

`collect` is allowed; see [U] **11.1.10 Prefix commands**.
weights are allowed in *command*.

display_options	Description
standardize	standardize test statistic using permutation distribution mean and variance
level(#)	set confidence level; default is level(95)
title(*text*)	use *text* as title for results
verbose	display full table legend
noheader	suppress table header
nolegend	suppress table legend
eps(#)	numerical tolerance; seldom used

exp_list contains	(*name*: *elist*)
	elist
	eexp
elist contains	*newvar* = (*exp*)
	(*exp*)
eexp is	*specname*
	[*eqno*]*specname*
specname is	_b
	_b[]
	_se
	_se[]
eqno is	##
	name

exp is a standard Stata expression; see **[U] 13 Functions and expressions**.

Distinguish between [], which are to be typed, and [], which indicate optional arguments.

Options

Main

reps(#) specifies the number of Monte Carlo permutations to perform. The default is reps(100).

The default of 100 permutations is chosen for convenience. In real-world applications, you will most likely need to use more permutations. permute reports the Monte Carlo error, which you can use to evaluate whether the specified number of permutations provides sufficient precision for the reported p-value estimates.

enumerate specifies that all possible distinct permutations be computed. This gives the full permutation distribution and p-values with no error. reps() and rseed() cannot be specified with enumerate.

The number of all possible distinct permutations is typically extremely large. Only for some small problems will it be practical to fully enumerate the permutation distribution. If all the values of *permvar* are unique, the number of possible permutations is $N!$, where N is the number of observations. When *permvar* has a lot of repeated values (when, for example, it is a 0/1 variable), the number of possible distinct permutations can be considerably less than $N!$ and may make enumeration feasible.

Before beginning the enumeration, `permute` will calculate and display the number of distinct permutations, allowing you to press *Break* when you see that the number of permutations is so large as to make computation time impossibly long.

Options

`rseed(#)` sets the random-number seed. Specifying this option is equivalent to typing the following command prior to calling `permute`:

```
. set seed #
```

`strata(`*varlist*`)` specifies that the permutations be performed within each stratum defined by the values of *varlist*.

`saving(`*filename*`[ , double every(#) replace ])` creates a Stata data file (`.dta` file) consisting of variables for each statistic in *exp_list* containing the results for each permutation.

`double` specifies that the results for each permutation be saved as `double`s, meaning 8-byte reals. By default, they are saved as `float`s, meaning 4-byte reals.

`every(#)` specifies that results be written to disk every #th permutation. `every()` should be specified only in conjunction with `saving()` when *command* takes a long time for each permutation. This will allow recovery of partial results should some other software crash your computer. See [P] **postfile**.

`replace` specifies that *filename* be overwritten if it exists.

Reporting

`standardize` specifies that the observed value of each test statistic be standardized. That is, the observed test statistic T_{obs} is standardized by calculating $[T_{\text{obs}} - \text{mean}(T)]/\sqrt{\text{Var}(T)}$, where $\text{mean}(T)$ and $\text{Var}(T)$ are the mean and variance of the permutation distribution of T. Standardized test statistics are useful for seeing roughly how extreme (and in what direction) the observed test statistic is.

`level(#)` specifies the confidence level, as a percentage, for confidence intervals. The default is `level(95)` or as set by `set level`; see [R] **level**.

`title(`*text*`)` specifies a title to be displayed above the table of permutation results.

`dots(#)` and `nodots` specify whether to display permutation dots. By default, one dot character is displayed for each successful permutation. An "x" is displayed if *command* returns an error or if any value in *exp_list* is missing. You can also control whether dots are displayed using `set dots`; see [R] **set**.

`dots(#)` displays dots every # permutations. `dots(0)` is a synonym for `nodots`.

`nodots` suppresses display of the permutation dots.

`nowarning` suppresses the printing of a warning message when *command* does not set `e(sample)`.

`noisily` requests that any output from *command* be displayed. This option implies the `nodots` option.

`trace` causes a trace of the execution of *command* to be displayed. This option implies the `noisily` option.

`verbose` requests that the full table legend be displayed when multiple coefficients or standard errors are specified using the `_b` or `_se` notation.

`noheader` suppresses display of the table header. This option implies the `nolegend` option.

`nolegend` suppresses display of the table legend. The table legend identifies the rows of the table with the expressions they represent.

Advanced

`nodrop` prevents `permute` from dropping observations outside the `if` and `in` qualifiers. `nodrop` will also cause `permute` to ignore the contents of `e(sample)` if it exists as a result of running *command*. By default, `permute` temporarily drops out-of-sample observations.

`reject(`*exp*`)` specifies an expression that indicates when results should be rejected. When *exp* is true, the resulting values are reset to missing values.

`eps(#)` specifies the numerical tolerance for testing $T \leq T_{\text{obs}}$ and $T \geq T_{\text{obs}}$, where T is the test statistic and T_{obs} is its observed value. These are considered true if, respectively, $T \leq T_{\text{obs}} + \#$ or $T \geq T_{\text{obs}} - \#$. The default is `eps(1e-7)`. You will not have to specify `eps()` under normal circumstances.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

Permutation tests are based on the idea of scrambling—that is, permuting—the order of a variable in all possible ways, calculating the value of a test statistic for each permutation, and taking this set of values of the statistic as its distribution.

For instance, consider the correlation of two variables, $\text{corr}(\mathbf{x}, \mathbf{y})$, where $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$. We hold the order of $\mathbf{x}$ fixed and permute the order of $\mathbf{y}$ in all possible ways. For each permutation $\mathbf{y}^*$, we calculate $T^* = \text{corr}(\mathbf{x}, \mathbf{y}^*)$. The set of T^* gives the permutation distribution for the correlation. This permutation distribution is the distribution of the correlation under the null hypothesis that the ordering of the elements of $\mathbf{y}$ are independent of the ordering of the elements of $\mathbf{x}$, conditional on the observed values of $\mathbf{x}$ and $\mathbf{y}$.

Aside: Actually, the null hypothesis does not require independence. A weaker assumption of exchangeability is sufficient. If $\mathbf{x}$ and $\mathbf{y}$ are observed values of the random variates $\mathbf{X} = (X_1, \ldots, X_n)$ and $\mathbf{Y} = (Y_1, \ldots, Y_n)$, then the joint distribution $f(\mathbf{X}, \mathbf{Y})$ is called exchangeable when it is invariant to the orderings of $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$.

The p-value for the permutation test is the proportion of permutations that produce a test statistic T^* as extreme or more extreme than the test statistic T_{obs} computed using the observed data.

`permute` estimates p-values for permutation tests using Monte Carlo permutations or by enumerating all possible permutations when the `enumerate` option is specified. To do Monte Carlo permutations, you type

```
. permute permvar exp_list, reps(#): command
```

The values in the variable *permvar* are randomly permuted # times, each time executing *command* and collecting the associated values from the expressions in *exp_list*.

command defines the statistical command to be executed. Most Stata commands and user-written programs can be used with `permute`, as long as they follow standard Stata syntax; see **[U] 11 Language syntax**. *exp_list* specifies the statistics to be collected from the execution of *command*. Despite the fact that `permute` works with most Stata commands, that does not mean the resulting permutation test is a sensible test. See, for example, Good (2006).

To enumerate all possible distinct permutations, you type

```
. permute permvar exp_list, enumerate: command
```

`permute` may be used for replaying results, but this feature is appropriate only when a dataset generated by `permute` is currently in memory or is identified by the `using` specification.

Monte Carlo permutation tests

We first demonstrate how to apply the `permute` prefix by testing for a difference in the distribution of a variable across two groups. Here we perform the test using Monte Carlo permutations. In example 3, we do the same test using complete enumeration.

▷ Example 1: Wilcoxon rank-sum test

Let's consider calculating the p-value for the Wilcoxon rank-sum test performed by `ranksum`. Suppose that we collected data from some experiment: `y` is some measure we took on 17 individuals, and `group` identifies the group to which an individual belongs.

```
. use https://www.stata-press.com/data/r18/permute2
. list, sepby(group)
```

	group	y
1.	1	6
2.	1	11
3.	1	20
4.	1	2
5.	1	9
6.	1	5
7.	0	2
8.	0	1
9.	0	6
10.	0	0
11.	0	2
12.	0	3
13.	0	3
14.	0	12
15.	0	4
16.	0	1
17.	0	5

We analyze the data using `ranksum`:

```
. ranksum y, by(group)

Two-sample Wilcoxon rank-sum (Mann-Whitney) test

       group |       Obs    Rank sum    Expected
-------------+---------------------------------
           0 |        11          79          99
           1 |         6          74          54
-------------+---------------------------------
    Combined |        17         153         153

Unadjusted variance       99.00
Adjustment for ties       -0.97
                        --------
Adjusted variance         98.03

H0: y(group==0) = y(group==1)
         z = -2.020
Prob > |z| = 0.0434
Exact prob = 0.0436
```

The test gives an approximate p-value of 0.0434 and an exact p-value of 0.0436.

Let's try to reproduce these results using `permute`. The test statistic T for the Wilcoxon rank-sum test is the sum of the ranks for the first group, which is 79, and is stored as `r(sum_obs)`. We specify `reps(10000)` to do 10,000 Monte Carlo permutations and `dots(100)` to display a dot every 100th permutation. We set the random-number seed so that we can duplicate our results.

```
. set seed 1234

. permute group r(sum_obs), reps(10000) dots(100): ranksum y, by(group)
(running ranksum on estimation sample)

warning: ranksum does not set e(sample), so no observations will be excluded
         from the permutations because of missing values or other reasons. To
         exclude observations, press Break, save the data, drop any
         observations that are to be excluded, and rerun permute.

Permutations (10,000): .........1,000.........2,000.........3,000.........4,000
> .........5,000.........6,000.........7,000.........8,000.........9,000.......
> ..10,000 done

Monte Carlo permutation results                 Number of observations =     17
Permutation variable: group                     Number of permutations = 10,000

      Command: ranksum y, by(group)
        _pm_1: r(sum_obs)

                                                              Monte Carlo error
           T |    T(obs)       Test       c       n       p      SE(p)   [95% CI(p)]
-------------+-----------------------------------------------------------------------
       _pm_1 |       79      lower     223   10000   .0223     .0015   .0195   .0254
                            upper    9817   10000   .9817     .0013   .9789   .9842
                        two-sided                   .0446     .0021   .0406   .0486

Notes: For lower one-sided test, c = #{T <= T(obs)} and p = p_lower = c/n.
       For upper one-sided test, c = #{T >= T(obs)} and p = p_upper = c/n.
       For two-sided test, p = 2*min(p_lower, p_upper); SE and CI approximate.
```

The lengthy message about `e(sample)` is worth noting. If there were missing values in the data, we might want to drop those observations before running `permute`. To suppress the message in future runs, use the `nowarning` option.

The two-sided p-value obtained by this Monte Carlo procedure is 0.0446, which is close to the exact p-value of 0.0436 computed by `ranksum`. See the next section for a description of how two-sided

p-values are calculated when performing permutation tests. See example 2 for a test that requires a one-sided p-value.

Note that we typed

```
. permute group ...
```

rather than

```
. permute y ...
```

We permuted the 0/1 variable `group`, which defines the groups, rather than the outcome variable `y`. For a statistic dependent on only two variables, it obviously does not matter which one we permute in terms of the theory of the test, but it does matter in terms of the efficiency of how `permute` does the computation. `permute` uses a different random shuffling algorithm for 0/1 (or dichotomous) variables than it does with other variables. See *Efficiency considerations for Monte Carlo permutations* below for details.

`permute` reports standard errors and confidence intervals for p-values because, as with any other Monte Carlo procedure, they are approximations to the true exact p-values. These statistics are useful to assess the precision of the computed p-values. If you need more precision, specify more permutations in the `reps()` option. See *Methods and formulas* for a description of how the standard errors and confidence intervals are calculated.

The confidence interval for the Monte Carlo two-sided p-value in this example is $[0.0406, 0.0486]$. If we want to increase the precision of the p-value, we could run `permute` again with more random permutations to narrow the confidence interval. The total number of possible distinct permutations of `group`, however, is not extremely large, and we can perform the permutation test using enumeration. See example 3, where we do just that.

◁

Two-sided p-values from permutation tests

In the above example, we used the two-sided p-value for our hypothesis testing. For permutation distributions, two-sided p-values require some explanation about how they are calculated.

`permute` calculates the two-sided p-value as $p = 2\min(p_{\text{lower}}, p_{\text{upper}})$, where p_{lower} is the lower one-sided p-value and p_{upper} the upper one-sided p-value. (More precisely, $p = \min[1, 2\min(p_{\text{lower}}, p_{\text{upper}})]$ is used because obviously p-values must be bounded by 1.)

In general, the p-value is defined as the probability under the null hypothesis of obtaining a value of the test statistic T equal to or more extreme than the value T_{obs} that was actually observed. For one-sided p-values, what is "more extreme" is clear. For lower one-sided p-values, it is the probability that $T \leq T_{\text{obs}}$, and for upper one-sided p-values, it is the probability that $T \geq T_{\text{obs}}$. When T has a symmetric distribution, the two-sided p-value is typically defined as the probability that $|T| \geq |T_{\text{obs}}|$. Permutation distributions, however, are not in general symmetric.

Under a permutation-based null hypothesis, the domain of T consists of all the possible permutations of the underlying data used to calculate T. The domain is discrete and finite, and hence the permutation distribution of T is discrete and finite. These finite distributions are symmetric only in certain cases. For instance, with our example of the Wilcoxon rank-sum test, if the data consist of untied ranks, the distribution is symmetric. When there are ties in the ranks, however, the distribution is in most cases not symmetric.

When distributions are asymmetric, what values of T are "more extreme" than T_{obs}? Suppose T_{obs} is below the mean of the distribution. Clearly, the lower-tail values $T \leq T_{\text{obs}}$ are more extreme. But what values of T from the upper tail are more extreme?

For asymmetric distributions, the rationale for using $p = 2\ \min(p_{\text{lower}}, p_{\text{upper}})$ for two-sided tests is the following: It takes the smallest one-sided p-value and doubles it. Comparing this two-sided p-value against a significance level of, say, 0.05 is equivalent to comparing the smallest one-sided p-value against a level of 0.025. It essentially turns the two-sided test into a one-sided test with the significance level cut in half. So this definition conveniently sidesteps the need to define what values of T from the opposite tail from T_{obs} are more extreme! Also, it is appropriate for both symmetric and asymmetric distributions.

One-sided permutation test

In some cases, we will want to perform a permutation test based on a one-sided p-value.

▷ Example 2: Permutation tests with ANOVA

Consider some fictional data from an experimental randomized complete-block design in which there are 5 subjects each receiving 10 different treatments. We want to test whether any of the treatments have an effect different from the effects of the other treatments.

Let's load the data and list the data for the first two subjects:

```
. use https://www.stata-press.com/data/r18/permute1, clear
. sort subject treatment
. list subject treatment y in 1/20, abbrev(10)
```

	subject	treatment	y
1.	1	1	4.407557
2.	1	2	4.280349
3.	1	3	4.418574
4.	1	4	4.075359
5.	1	5	3.899775
6.	1	6	5.533271
7.	1	7	5.142111
8.	1	8	5.791124
9.	1	9	4.504411
10.	1	10	4.896333
11.	2	1	5.693386
12.	2	2	4.508785
13.	2	3	5.10376
14.	2	4	5.753985
15.	2	5	5.092277
16.	2	6	4.496496
17.	2	7	6.339948
18.	2	8	4.820389
19.	2	9	5.686253
20.	2	10	6.951727

These data may be analyzed using anova.

```
. anova y treatment subject
```

```
                  Number of obs =         50    R-squared     =  0.3544
                  Root MSE      =    .914159    Adj R-squared =  0.1213
```

Source	Partial SS	df	MS	F	Prob>F
Model	16.518219	13	1.2706322	1.52	0.1574
treatment	13.022671	9	1.4469634	1.73	0.1174
subject	3.4955481	4	.87388703	1.05	0.3973
Residual	30.08475	36	.83568751		
Total	46.602969	49	.951081		

anova gives a p-value of 0.1174 for the treatment effect. This p-value is calculated with the assumption of normality for the distribution of the outcome conditional on the means of each treatment and subject effect.

Suppose we do not want to assume normality. The treatments were assigned in a random order to each of the subjects. A null hypothesis of no treatment effect means that the observed values of y and their order were determined by factors other than the treatments. The treatments were essentially labels that had nothing to do with the outcomes, and any other ordering of the labels would be a possible occurrence. That is, we imagine running the experiment multiple times, each with a different ordering of the treatments, but each time, we get the same observed values of y. This is the permutation-based formulation of the null hypothesis.

What about the subjects? Each subject gets each of the 10 treatments, so clearly we must permute the treatments within each subject independently of the permutations for the other subjects. We can do this using the strata() option with permute.

If we type ereturn list after anova, we see that the F statistic for treatment is stored in e(F_1). This is our test statistic for our permutation test.

We save the dataset containing all the permutations of the test statistic using the saving() option. Specifying the test statistic as F_treatment=e(F_1) labels the test statistic as F_treatment in the output and is also the name of the variable containing the test statistic in permanova.dta, the dataset created by saving(). We set the seed for the random-number generator and also specify the nodots option to suppress the dots in the output.

```
. permute treatment F_treatment=e(F_1), reps(10000) strata(subject)
> saving(permanova) rseed(1234) nodots: anova y treatment subject
Monte Carlo permutation results                 Number of observations =     50
Permutation variable: treatment                 Number of strata       =      5
                                                Number of permutations = 10,000

      Command: anova y treatment subject
  F_treatment: e(F_1)
```

						Monte Carlo error		
T	T(obs)	Test	c	n	p	SE(p)	[95%	CI(p)]
F_treatment	1.731465	lower	8788	10000	.8788	.0033	.8722	.8851
		upper	1212	10000	.1212	.0033	.1149	.1278
		two-sided			.2424	.0043	.2340	.2508

```
Notes: For lower one-sided test, c = #{T <= T(obs)} and p = p_lower = c/n.
       For upper one-sided test, c = #{T >= T(obs)} and p = p_upper = c/n.
       For two-sided test, p = 2*min(p_lower, p_upper); SE and CI approximate.
```

Our test statistic is an F statistic, so we are interested in the number of permutations that have a larger (more extreme) statistic than the 1.73 we obtained with our original data. Therefore, we want the upper one-sided p-value, which is 0.1212. This value is close to the p-value given by `anova` of 0.1174 for the treatment effect.

◁

For an additional example of a permutation test, with an application in epidemiology, see Hayes and Moulton (2017, 237–241).

Enumeration

When the number of observations, N, in a dataset is small, it may be possible to enumerate all possible permutations and obtain p-values without the error involved in computing Monte Carlo p-values.

When `permute` does an enumeration, not only does N matter, but the number of distinct values of the permutation variable matters as well. `permute` does the enumeration by computing only permutations that give different arrangements of the permutation variable; that is, it does not compute any duplicate permutations. So the number of distinct values (and their multiplicity) of the permutation variable determines the number of permutations enumerated and so determines whether enumeration is feasible. See *Efficiency considerations for enumeration* below for details.

▷ Example 3: Wilcoxon rank-sum test using enumeration

Here we repeat example 1, but this time we do it by enumerating all possible permutations. We load the data:

```
. use https://www.stata-press.com/data/r18/permute2
```

The data consist of an outcome y grouped by the variable group. If we tabulate group

```
. tabulate group
```

Group	Freq.	Percent	Cum.
0	11	64.71	64.71
1	6	35.29	100.00
Total	17	100.00	

we see that group consists of 11 zeros and 6 ones. Hence, there are only $\binom{17}{6} = 12{,}376$ possible distinct permutations of group.

In example 1, we used ranksum to compute the test statistic, but each time ranksum is called, it computes the ranks of y. It is unnecessary to recompute the ranks for each permutation. It is better to compute the ranks just once at the outset. We can do this using the rank() function of egen:

```
. egen r = rank(y)
```

The test statistic is the sum of the ranks for either one of the groups. The sum can be computed efficiently using summarize with an if restriction and the meanonly option.

```
. permute group r(sum), enumerate nodrop nowarning dots(100): summarize r
> if group == 1, meanonly
(running summarize on estimation sample)
(enumerating all 12,376 possible permutations)
Permutations (12,376): .........1,000.........2,000.........3,000.........4,000
> .........5,000.........6,000.........7,000.........8,000.........9,000.......
> ..10,000.........11,000.........12,000.... done
Enumeration permutation results
Number of observations =     17
Number of permutations = 12,376
Permutation variable: group
      Command: summarize r if group == 1, meanonly
        _pm_1: r(sum)
```

T	T(obs)	Test	c	n	p
_pm_1	74	lower	12142	12376	.9811
		upper	270	12376	.0218
		two-sided			.0436

```
Notes: For lower one-sided test, c = #{T <= T(obs)} and p = p_lower = c/n.
       For upper one-sided test, c = #{T >= T(obs)} and p = p_upper = c/n.
       For two-sided test, p = 2*min(p_lower, p_upper).
```

Note that it is necessary to specify the nodrop option. Otherwise, permute would drop all observations not satisfying if group == 1 before doing the permutations, and that would not give us what we want.

permute with the enumerate option gave a two-sided p-value of 0.0436, which is the same as the exact p-value reported by ranksum, as it should be.

With group as the variable being permuted, the number of distinct permutations is quite small. If, however, we attempt to do the enumeration for all the distinct permutations of r:

```
. permute r r(sum), enumerate nodrop nowarning dots(100):
> summarize r if group == 1, meanonly
(running summarize on estimation sample)
(enumerating all 3.71e+12 possible permutations)
Permutations (3,705,077,376,000): .........1,000.........2,000.........3,000
> .........4,000.........--Break--
r(1);
```

Each permutation takes about 0.1 millisecond on our computer. Thus, the enumeration will take $0.1 \times 3.71 \times 10^{12}/(365 \times 24 \times 60 \times 60 \times 1000) \approx 12$ years, so we pressed *Break*.

◁

Efficiency considerations for Monte Carlo permutations

Suppose you want to perform a randomization two-sample t test, which is like the two-sample t test that assumes normality, only the randomization test is based on permuting the variable that defines the samples. It is the same as the Wilcoxon rank-sum test, except the observed values of the outcome, rather than their ranks, are used for the test statistic.

So say we have a variable x with continuous outcomes for two groups defined by the variable group, with values 0 or 1. The randomization two-sample t test could be done using Monte Carlo permutations by typing

```
. permute x r(mu_1), reps(10000): ttest x, by(group)
```

Or by typing

```
. permute group r(mu_1), reps(10000): ttest x, by(group)
```

In the first case, x was permuted, and in the second, group. Both are valid ways to do Monte Carlo permutations. permute is smart, however, and treats a 0/1 variable differently from how it treats a variable with lots of distinct values.

Suppose there are fewer 1s than 0s in group. Rather than randomly permuting all the 0s and 1s, permute randomly shuffles the 1s into the 0s. If there are only a few 1s relative to the number of 0s, this method is much faster than permuting all the values. Hence, if you are doing a permutation test that involves two variables, pick the one with the fewest distinct values to be the *permvar*.

When the strata() option is specified, permute uses special code for the case in which the *permvar* is dichotomous (with, say, values y_0 and y_1) and each stratum contains a single observation equal to y_1 and all other observations in the stratum equal to y_0 (and y_0 and y_1 do not flip or change in value across strata). If you are doing a stratified permutation test and you have such a variable whose permutations will give the test you want, be sure to make it the *permvar*.

For all types of data, Monte Carlo permutations of the *permvar* are computed quickly. If the command calculating the test statistic for each permutation is not fast, it is unlikely you will notice the greater speed of permuting a dichotomous variable. If, however, you are using a fast command such as regress, you likely will notice the greater speed.

Efficiency considerations for enumeration

Doing an enumeration of all possible distinct permutations using the `enumerate` option, however, is a different story. Here the selection of the *permvar* is crucial and typically determines whether it is feasible to do the enumeration.

Consider performing the randomization t test we described earlier using enumeration:

```
. permute group r(mu_1), enumerate: ttest x, by(group)
```

Suppose there are 20 observations, 10 with `group` $= 0$ and 10 with `group` $= 1$. Typing `permute group` ... will enumerate all $\binom{20}{10} = 184{,}756$ possible distinct permutations of `group`. (When we say "distinct permutations", we mean that duplicate permutations are not computed.)

If, however, we type

```
. permute x r(mu_1), enumerate: ttest x, by(group)
```

`permute x` ... will attempt to enumerate all possible permutations of `x`. If all the values of `x` are unique, there are $20! \approx 2.4 \times 10^{18}$ possible permutations of `x`, which is much larger than 184,756.

Hence, a dichotomous variable or a variable with few distinct values should always be chosen as the *permvar* rather than another variable with many distinct values whenever possible. (To be precise, both the number of distinct values and their multiplicities determine the number of permutations. See *Methods and formulas*.)

See example 3 for an example using enumeration.

Stored results

`permute` stores the following in `r()`:

Scalars

`r(N)`	number of observations for *command*
`r(n_reps)`	number of permutations performed
`r(k_exp)`	number of standard expressions
`r(k_eexp)`	number of `_b` and `_se` expressions
`r(n_strata)`	number of strata, if `strata()` specified
`r(level)`	confidence level

Macros

`r(cmd)`	`permute`
`r(command)`	*command* following colon
`r(permvar)`	permutation variable
`r(enumerate)`	`"enumerate"`, if `enumerate` specified
`r(title)`	title in output
`r(rngstate)`	random-number state used for Monte Carlo permutations
`r(exp#)`	#th expression
`r(strata)`	strata variable, if `strata()` specified
`r(missing)`	`"missing"` when one or more expressions equal missing value

Matrices

`r(b)`	observed statistics
`r(b_std)`	standardized observed statistics, if `standardize` specified
`r(n)`	number of nonmissing results
`r(c_lower)`	counts for lower one-sided p-values
`r(c_upper)`	counts for upper one-sided p-values
`r(p_lower)`	lower one-sided p-values
`r(p_upper)`	upper one-sided p-values
`r(p_twosided)`	two-sided p-values
`r(se_p_lower)`	Monte Carlo standard errors of lower one-sided p-values
`r(se_p_upper)`	Monte Carlo standard errors of upper one-sided p-values

r(se_p_twosided)	Monte Carlo standard errors of two-sided p-values
r(ci_p_lower)	Monte Carlo confidence intervals of lower one-sided p-values
r(ci_p_upper)	Monte Carlo confidence intervals of upper one-sided p-values
r(ci_p_twosided)	Monte Carlo confidence intervals of two-sided p-values

Methods and formulas

One-sided p-values are based on counts of the test statistic T calculated for each permutation that are more extreme than the observed value T_{obs}. The lower one-sided p-value uses the count $c = \#\{T \leq T_{\text{obs}}\}$, and the upper one-sided p-value uses $c = \#\{T \geq T_{\text{obs}}\}$.

The counts from Monte Carlo permutations are assumed to have a binomial distribution. Standard errors and confidence intervals are computed using `cii proportions` n c, where n is the number of permutations that yielded nonmissing results and c is the count. The confidence intervals are exact binomial confidence intervals. See *Methods and formulas* in [R] **ci**.

`permute` calculates the two-sided p-value as $p = \min[1, 2\min(p_{\text{lower}}, p_{\text{upper}})]$, where p_{lower} is the lower one-sided p-value and p_{upper} the upper one-sided p-value. Because the definition of the two-sided p-value does not yield a simple formula for the standard error or confidence interval for Monte Carlo permutations, the following ad hoc procedure is used. If p_{lower} is the minimum one-sided p-value, its count c_{lower} is doubled. If p_{upper} is the minimum one-sided p-value, its count c_{upper} is doubled. More precisely, the value $c_2 = \min[n, 2\min(c_{\text{lower}}, c_{\text{upper}})]$ is used, and its distribution is assumed to be approximately binomial. Standard errors and confidence intervals are computed using `cii proportions` n c_2`, wald`. The confidence intervals produced are asymptotic binomial confidence intervals.

When `enumerate` is specified, the p-values have no error.

Suppose there are N observations and the variable being permuted contains K distinct values, each with multiplicity n_k, $k = 1, \ldots, K$. The total number of distinct permutations is

$$\frac{N!}{n_1!\, n_2! \cdots n_K!}$$

This is the number of permutations computed when `enumerate` is specified.

When `standardize` is specified, instead of displaying the observed test statistic T_{obs}, the standardized statistic

$$\frac{T_{\text{obs}} - \text{mean}(T)}{\sqrt{\text{Var}(T)}}$$

is displayed where $\text{mean}(T)$ and $\text{Var}(T)$ are the mean and variance of the permutation distribution of T:

$$\text{mean}(T) = \frac{1}{n}\sum_{i=1}^{n} T_i$$

$$\text{Var}(T) = \frac{1}{n}\sum_{i=1}^{n} \{T_i - \text{mean}(T)\}^2$$

n is the number of permutations, and T_i is the test statistic calculated for the ith permutation.

References

Ängquist, L. 2010. Stata tip 92: Manual implementation of permutations and bootstraps. *Stata Journal* 10: 686–688.

Gallis, J. A., F. Li, H. Yu, and E. L. Turner. 2018. cvcrand and cptest: Commands for efficient design and analysis of cluster randomized trials using constrained randomization and permutation tests. *Stata Journal* 18: 357–378.

Good, P. I. 2006. *Resampling Methods: A Practical Guide to Data Analysis*. 3rd ed. Boston: Birkhäuser.

Hayes, R. J., and L. H. Moulton. 2017. *Cluster Randomised Trials*. 2nd ed. Boca Raton, FL: CRC Press.

Kaiser, J. 2007. An exact and a Monte Carlo proposal to the Fisher–Pitman permutation tests for paired replicates and for independent samples. *Stata Journal* 7: 402–412.

Kaiser, J., and M. G. Lacy. 2009. A general-purpose method for two-group randomization tests. *Stata Journal* 9: 70–85.

Also see

[R] **bootstrap** — Bootstrap sampling and estimation

[R] **jackknife** — Jackknife estimation

[R] **simulate** — Monte Carlo simulations