Permutation tests and confidence sets

Release 0.2

K. Jarrod Millman, Kellie Ottoboni, and Philip B. Stark

Jun 01, 2021

CONTENTS

CHAPTER

ONE

USER GUIDE

Permutation tests (sometimes referred to as randomization, re-randomization, or exact tests) are a nonparametric approach to statistical significance testing. They were first introduced by R. A. Fisher in 1935 [Fis35] and further developed by E. J. G. Pitman [Pit37, Pit38]. After the introduction of the bootstrap, the ideas were extended in the 1980's by J. Romano [Rom88, Rom89].

Permutation tests were developed to test hypotheses for which relabeling the observed data was justified by exchangeability[1] of the observed random variables. In these situations, the conditional distribution of the test statistic under the null hypothesis is completely determined by the fact that all relabelings of the data are equally likely. That distribution might be calculable in closed form; if not, it can be simulated with arbitrary accuracy by generating relabelings uniformly at random. In contrast to approximate parametric methods or asymptotic methods, the accuracy of the simulation for any finite (re)sample size is known, and can be made arbitrarily small at the expense of computing time.

More generally, permutation tests are possible whenever the null distribution of the data is invariant under the action of some group (see Appendix [app:def] for background). Then, a subset of outcomes is conditionally equally likely, given that the data fall in a particular *orbit* of the group (all potential observations that result from applying elements of the group to the observed value of the data). That makes it possible to determine the conditional distribution of any test statistic, given the orbit of the data. Since the conditional distribution is uniform on the orbit of the original data, the probability of any event is the proportion of possible outcomes that lie in the event. If tests are performed conditionally at level α regardless of the observed data, the resulting overall test has unconditional level α, by the law of total probability.

1.1 Paired permutation tests

To illustrate the paired two-sample permutation test, consider the following randomized, controlled experiment. You suspect a specific treatment will increase the growth rate of a certain type of cell. To test this hypothesis, you clone 100 cells. Now there are 200 cells composed of 100 pairs of identical clones. For each cloned pair you randomly assign one to treatment, with probability 1/2, independently across the 100 pairs. At the end of the treatment, you measure the growth rate for all the cells. The null hypothesis is that treatment has no effect. If that is true, then the assignment of a clone to treatment amounts to an arbitrary label that has nothing to do with the measured response. So, given the responses within each pair (but not the knowledge of which clone in each pair had which response), it would have been just as likely to observe the same *numbers* but with flipped labels within each pair. We could generate new hypothetical datasets from the observed data by assigning the treatment and control labels for all the cloned pairs independently. This yields a total of 2^{100} total datasets (including the observed data and all the hypothetical datasets that you generated), all equally likely to have occurred under the null, conditioning on the observed data (but not the labeling).

[1] A sequence $X_1, X_2, X_3, \ldots, X_n$ of random variables is *exchangeable* if their joint distribution is invariant to permutations of the indices; that is,

$$p(x_1, \ldots, x_n) = p(x_{\pi(1)}, \ldots, x_{\pi(n)})$$

for all permutations π of $\{1, 2, \ldots, n\}$. It is closely related to the notion of *independent and identically-distributed* random variables. Independent and identically-distributed random variables are exchangeable. However, simple random sampling *without* replacement produces an exchangeable, but not independent, sequence of random variables.

The standard parametric approach to this problem is the paired t-test, since the cloned cells are presumably more similar to each other than to another randomly chosen cell (and thus more readily compared). The paired t-test assumes that, if the null hypothesis is true, the differences in response between each pair of clones are independently and identically (iid) normally distributed with mean zero and unknown variance. The test statistic is the mean of the differences between each cloned pair divided by the standard error of these differences. Under these assumptions, the test statistic is distributed as a t-distribution with $n - 1$ degrees of freedom. This means you can calculate the test statistic and then read off the from the t-distribution. If the is below some prespecified critical value α, then you reject the null. If the true generative model for the data is not iid normal, however, the probability of rejecting the null hypothesis can be quite different from α even if treatment has no effect.

A permutation version of the t-test can avoid that vulnerability: one can use the t-statistic as the test statistic, but instead of selecting the critical value on the basis of Student's t-distribution, one uses the distribution of the statistic under the permutation distribution. Of course, other test statistics could be used instead; the test statistic should be sensitive to the nature of the alternative hypothesis, to ensure that the test has power against the alternatives the science suggests are relevant.

Regardless of which test statistic you choose for your permutation test, if the problem size is not too large then you enumerate all equally likely possibilities under the null given the observed data. If the problem is too large to feasibly enumerate, then you use a suitably large, iid random sample from the exact distribution just described, by selecting permutations uniformly at random and applying the test statistic to those permutations. As you increase the number of samples, you will get increasingly better (in probability) approximations of the exact distribution of the test statistic under the null. The null conditional probability of any event can be estimated as the proportion of random permutations for which the event occurs, and the sampling variability of that estimate can be characterized exactly, for instance, using binomial tests (since the distribution of the number of times the event occurs is Binomial with n equal to the number of samples and p the unknown probability to be estimated).

1.1.1 Example

We'll generate some fake data to demonstrate the paired two-sample problem. We'll follow the cloned cells example above. The controls have a response that is distributed uniformly between 0 and 10. There is random variation among the cells, so the difference between responses in a pair is normally distributed with 0 mean. In the first case, suppose we give the treatment group an ineffective treatment, so there is no treatment effect. The treated cell is equally likely to have a response that is larger or smaller than it's clone's response.

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from numpy.random import RandomState
>>> from permute.core import one_sample

>>> prng = RandomState(42)
>>> control = prng.uniform(low = 0, high=10, size=100)
>>> ineffective_treatment = control + prng.normal(loc=0, scale=1, size=100)
>>> (p, diff_means) = one_sample(ineffective_treatment, control, stat='mean', seed=prng)
>>> print("P-value: ", p)
P-value:  0.50726
>>> print("Difference in means:", diff_means)
Difference in means: -0.00108036016736
```

Now, suppose we give a new treatment that has a constant effect that increases the cell's response by 1.

```
>>> good_treatment = control + prng.normal(loc=1, scale=1, size=100)
>>> (p, diff_means) = one_sample(good_treatment, control, stat='mean', seed=prng)
>>> print("P-value: ", p)
P-value:  0.0
```

(continues on next page)

(continued from previous page)

```
>>> print("Difference in means:", diff_means)
Difference in means: 1.08009705107
```

one_sample is written to either take in two arguments and test the difference between pairs as we've done above, or to take in a single argument and test whether that variable is centered around 0. Below, we call *one_sample* in that manner, supplying the difference in response within pairs, and get the same results.

```
>>> paired_differences = good_treatment - control
>>> (p, diff_means) = one_sample(paired_differences, stat='mean', seed = prng)
>>> print("P-value: ", p)
P-value:  0.0
>>> print("Difference in means:", diff_means)
Difference in means: 1.08009705107
```

1.2 Two sample permutation tests

Suppose that we have a completely randomized experiment, where people are assigned to two groups at random. Suppose we have N individuals indexed by $i = 1, \ldots, N$. We assign them at random to one of two groups with a random treatment vector Z: if $Z_i = 1$, then individual i receives treatment (for example, a drug) and if $Z_i = 0$, individual i receives no treatment (a placebo). We'd like to test whether or not the drug has an effect on how often catches a cold. The outcome measure is the number of times somebody gets a cold wihin one year of starting to take the drug (for simplicity, assume that this can be measured perfectly). We can measure the difference in outcomes between the two groups with any statistic we'd like. The statistic will be a function of the treatment vector Z and the outcomes, $Y_i(1)$ being the outcome under treatment and $Y_i(0)$ being the outcome under no treatment. We'll use the difference-in-means test statistic:

$$T(Z, Y(1), Y(0)) = \frac{1}{N_t} \sum_{i:Z_i=1} Y_i(1) - \frac{1}{N_c} \sum_{i:Z_i=0} Y_i(0)$$

Here, N_t is the number of treated individuals and N_c is the number of untreated individuals, so $N_t + N_c = N$. If the test statistic is negative, then we may have evidence that the drug reduces colds. Conversely, if the test statistic is positive, we may believe that the drug actually makes people more vulnerable to getting sick. If we have no a priori belief about what the drug may do, we simply want to know if it has any effect at all. How extreme does the statistic need to be to indicate that there is likely an effect?

If the drug has no effect on colds, then the number of colds that somebody has would be the same whether he or she received the drug or the placebo. This is the *strong null hypothesis*: the drug has no effect on any individual. Under the strong null, we know both potential outcomes for each individual; namely, their number of colds would be the same regardless of which treatment group they were assigned. In mathematical notation, $Y_i(1) = Y_i(0)$ for all i under the strong null.

The random assignment of people to treatment groups ensures that all possible assignments of N_t people to treatment are equally likely. Thus, we can find the null distribution of the test statistic by calculating $T(Z^*, Y(1), Y(0))$ for all possible treatment assignment vectors Z^*. In general, this would not be possible, because for each individual we observe only $Y_i(1)$ or $Y_i(0)$, but not both. However, the strong null hypothesis allows us to impute the missing potential outcome for each individual.

There are $\binom{N}{N_t}$ possible values of Z^*. In practice, this number is often too large to enumerate all possible values of $T(Z^*, Y(1), Y(0))$. Instead, we simulate the distribution by taking a random subset of B of the Z^*. Then, our estimated p-value for the test is

$$P = 2 \times \min\left(\frac{\#\{T(Z^*) <= T(Z)\}}{B}, \frac{\#\{T(Z^*) >= T(Z)\}}{B}\right)$$

1.2.1 Gender bias in student evaluation of teachers

There is growing evidence of gender bias in student evaluations of teaching. To address the question "Do students give higher ratings to male teachers?," an online experiment was done with two professors, one male and one female [MDH14]. Each professor taught two sections. In one section, they used a male name. In the other, they used a female name. The students didn't know the teacher's real gender. We test whether student evaluations of teaching are biased by comparing the ratings when one of the professors used a male name versus a female name.

As an aside, note that we cannot simply pool the ratings for the two professors when they identified as male and when they identified as female. The "treatment" is the gender the instructor reports, but other things affect the ratings students give. For instance, the two instructors may have different teaching styles, thereby introducing differences in the ratings that are unrelated to their identified gender. This is why we choose to focus on one instructor.

Parametric Approach

First let us consider the parametric two-sample t-test. In this case, our test statistic is

$$t = \frac{\text{mean(rating for M-identified) - mean(rating for F-identified)}}{\sqrt{\text{pooled SD of ratings}}}$$

For the two-sample t-test, the null hypothesis is that the reported/perceived instructor gender has no effect on ratings. The alternative hypothesis is that ratings differ by reported/perceived instructor gender. For the two-sample t-test to be valid, we require the following assumptions:

- Ratings are normally distributed. (But they are on a Likert 1-5 scale, which is definitely not normal.)
- Noise is zero-mean and constant variance across raters. (How should we interpret "noise" in this context? Besides constant variance is not plausible: some raters might give a range of scores, other raters might always give 5.)
- Independence between observations. (Students might talk about ratings with their peers in the class, creating dependence.)

Despite the problematic assumptions we are required to make, let's temporarily assume they hold and calculate a "p-value" anyway.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import stats
```

```
>>> from permute.data import macnell2014
>>> ratings = macnell2014()
>>> prof1 = ratings[ratings.tagender==0]
>>> maleid = prof1.overall[prof1.taidgender==1]
>>> femaleid = prof1.overall[prof1.taidgender==0]
>>> df = len(maleid) + len(femaleid) - 2
>>> t, p = stats.ttest_ind(maleid, femaleid)
>>> print('Test statistic:', np.round(t, 5))
Test statistic: 1.32905
>>> print('P-value (two-sided):', np.round(p, 5))
P-value (two-sided): 0.20043
```

Note that the computed "p-value" is above the standard cut-offs for reporting significance in the literature.

Permutation approach

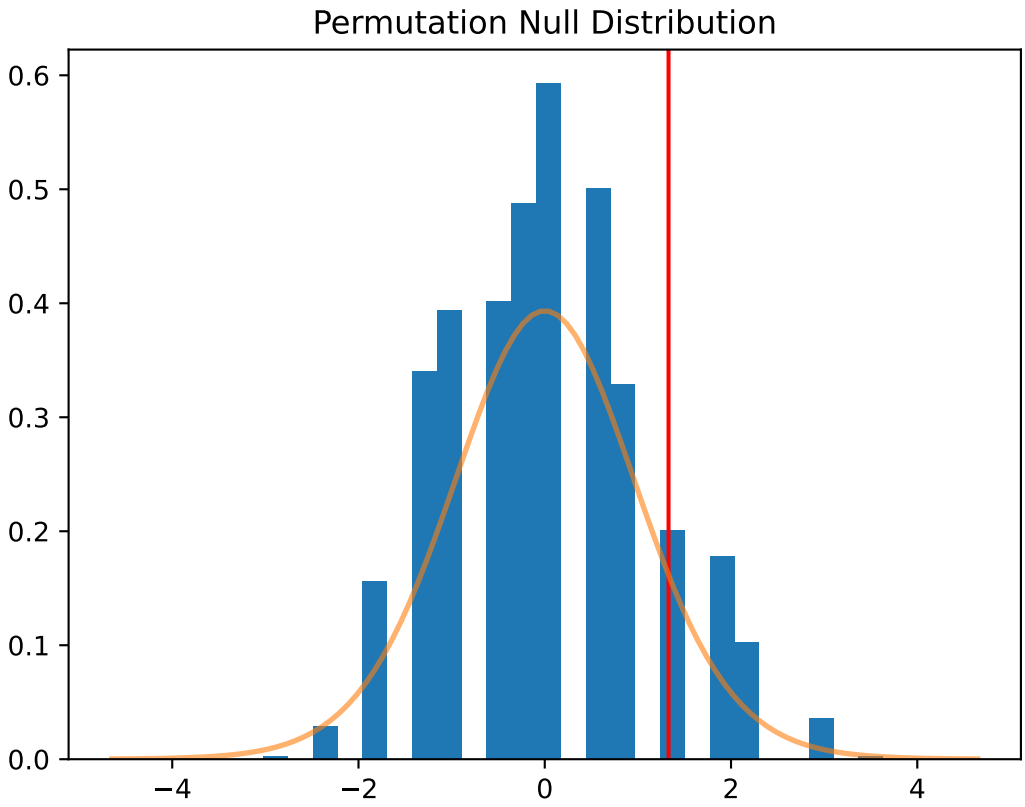
For the permutation test we can use the same test statistic, but we will compute the p-value by randomly sampling the exact distribution of the test statistics. The null hypothesis is that the ratings are uninfluenced by reported gender—any particular student would assign the same rating regardless of instructor gender. The alternative hypothesis is that the ratings differ by instructor gender—some students would assign different ratings depending on reported instructor gender. The only assumption we need to make is that the random assignment of students to instruction sections is fair and independent across individuals. This can be verified directly from the experimental design.

```
>>> from permute.core import two_sample
>>> p, t = two_sample(maleid, femaleid, stat='t', alternative='two-sided', seed=20)
>>> print('Test statistic:', np.round(t, 5))
Test statistic: 1.32905
>>> print('P-value (two-sided):', np.round(p, 5))
P-value (two-sided): 0.27918
```

```
>>> p, t = two_sample(maleid, femaleid, reps=100, stat='t', alternative='two-sided',␣
↪seed=20)
>>> print('P-value (two-sided):', np.round(p, 5))
P-value (two-sided): 0.41584
```

Since the permutation test also returns the approximately exact distribution of the test statistic, let's compare the actual distribution with the t-distribution.

```
>>> p, t, distr = two_sample(maleid, femaleid, stat='t', reps=10000,
...                          alternative='greater', keep_dist=True, seed=55)
>>> n, bins, patches = plt.hist(distr, 25, histtype='bar', density=True)
>>> plt.title('Permutation Null Distribution')
Text(0.5, 1.0, 'Permutation Null Distribution')
>>> plt.axvline(x=t, color='red')
<matplotlib.lines.Line2D object at ...>
>>> x = np.linspace(stats.t.ppf(0.0001, df),
...       stats.t.ppf(0.9999, df), 100)
>>> plt.plot(x, stats.t.pdf(x, df), lw=2, alpha=0.6)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.show()
```



The plot above shows the null distribution generated by 10,000 permutations of the data. The t distribution is superimposed for comparison. The null distribution is much more concentrated around 0 than the t distribution, which has longer tails. Furthermore, it is not perfectly symmetric around zero. This is the source of the difference in p-values between the two tests.

1.2.2 Stratified Spearman correlation permutation test

Some experimental designs have natural groupings. It makes sense to estimate effects within groups, then combine within-group estimates.

To turn this idea into a permutation test, we carry out permutations within groups, then aggregate the test statistics across groups. This helps control for group-level effects.

More on teaching evaluations

We established that one instructor got higher ratings when they used a male name than when they used a female name, but the difference was not significant. Now we may ask, did ratings differ according in this way for either of the two instructors?

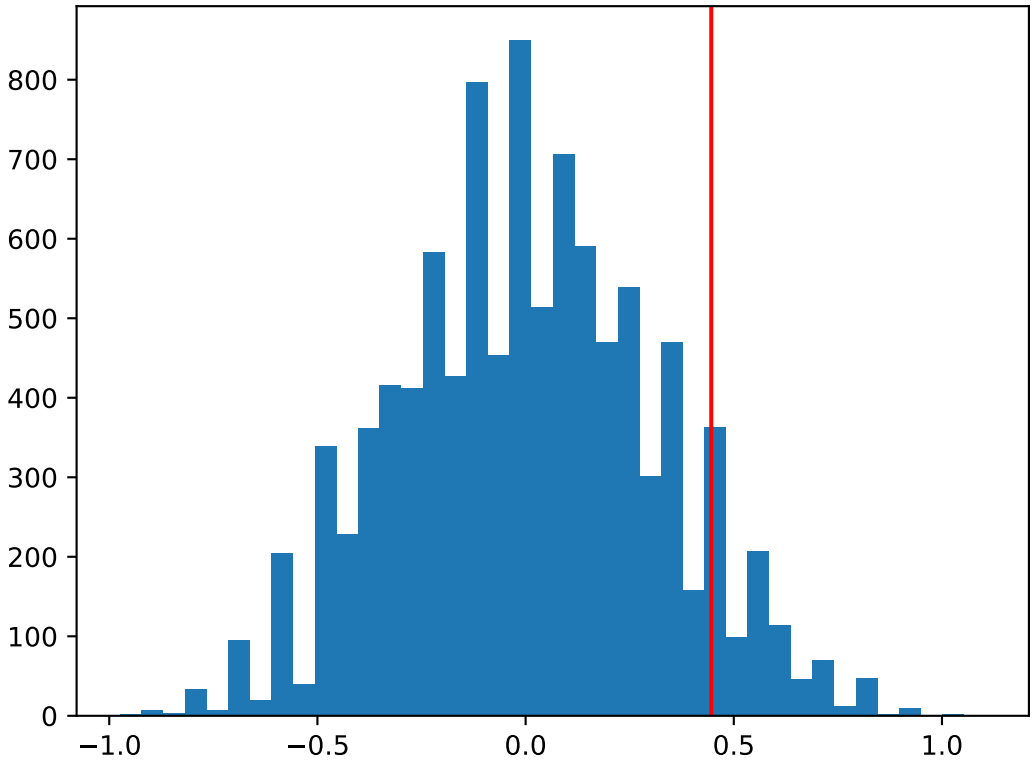
If there is no gender bias in the ratings, then students should give the same rating to the male instructor regardless of the gender he claims to be and students should give the same rating to the female instructor regardless of the gender she claims to be. However, we don't necessarily believe that students would rate the two instructors the same, since there may be some difference in their teaching styles.

Null hypothesis: student by student, the instructor would receive the same rating regardless of reported gender

Alternative hypothesis: there is at least one student who would rate their instructor higher if they identified as male

The test statistic we use within groups is the Spearman correlation. For each instructor, we compute the correlation between their rating and reported gender, then add the absolute values of the correlations for the instructors. Because reported gender is just a binary indicator, the correlation is equivalent to using the mean rating for male-identified instructors as a test statistic.

```
>>> from permute.stratified import sim_corr
>>> p, rho, sim = sim_corr(x=ratings.overall, y=ratings.taidgender, group=ratings.
↪tagender, seed = 25)
>>> n, bins, patches = plt.hist(sim, 40, histtype='bar')
>>> plt.axvline(x=rho, color='red')
<matplotlib.lines.Line2D object at ...>
>>> plt.show()
```



Finally, I plot the simulated distribution of the test statistics under the null conditioned on the observed data in Figure [fig:figure2].

```
>>> print('Test statistic:', np.round(rho, 5))
Test statistic: 0.4459
>>> print('P-value:', np.round(p, 3))
P-value: 0.09
```

At the 10% level, there is a significant difference in ratings between male-identified and female-identified instructors.

We could not have computed this p-value with any common distribution, since the null hypothesis assumes some observations (ratings for a single instructor) are exchangeable but others are not.

1.3 Regression

Given n observations of two scalars (x_i, y_i) for $i = 1, 2, \ldots, n$, consider the simple linear regression model

$$y_i = a + bx_i + \epsilon_i.$$

Assume that $\{\epsilon_i\}_{i=1}^n$ are exchangeable.

You are interested in testing whether the slope of the population regression line is non-zero; hence, your null hypothesis is $b = 0$. If $b = 0$, then the model reduces to $y_i = a + \epsilon_i$ for all i. If this is true, the $\{y_i\}_{i=1}^n$ are exchangeable since they are just shifted versions of the exchangeable $\{\epsilon_i\}_{i=1}^n$. Thus every permutation of the $\{y_i\}_{i=1}^n$ has the same conditional probability regardless of the xs. Hence every pairing (x_i, y_j) for any fixed i and for $j = 1, 2, \ldots, n$ is equally likely.

Using the least squares estimate of the slope as the test statistic, you can find its exact distribution under the null given the observed data by computing the test statistic on all possible pairs formed by permuting the y values, keeping the original order of the x values. From the distribution of the test statistic under the null conditioned on the observed data, the is the ratio of the count of the *as extreme* or *more extreme* test statistics to the total number of such test statistics. You might in principle enumerate all $n!$ equally likely pairings and then compute the exact p-value. For sufficiently large n, enumeration becomes infeasible; in which case, you could approximate the exact p-value using a uniform random sample of the equally likely pairings.

A parametric approach to this problem would begin by imposing additional assumptions on the noise ϵ. For example, if we assume that $\{\epsilon_i\}$ are iid Gaussians with mean zero, then the the least squares estimate of the slope normalized by its standard error has a t-distribution with $n - 2$ degrees of freedom. If this additional assumption holds, then we can read the off a table. Note that, unlike in the permutation test, we were only able to calculate the p-value (even with the additional assumptions) because we happened to be able to derive the distribution of this specific test statistic.

1.3.1 Derivation

Given n observations

$$y_i = a + bx_i + \epsilon_i,$$

the least square solution is

$$\min_{a,b} \sum_{i=1}^{n} (y_i - a - bx_i)^2$$

Taking the partial derivative with respect to a

$$\begin{aligned}
\frac{\partial}{\partial b} \sum_{i=1}^{n} (y_i - a - bx_i)^2 &= -2 \sum_{i=1}^{n} (y_i - a - bx_i) \\
&= -2 \left(\sum_{i=1}^{n} y_i - na - b \sum_{i=1}^{n} x_i \right) \\
&= -2n \left(\bar{y} - a - b\bar{x} \right).
\end{aligned}$$

Setting this to 0 and solving for a yields our estimate $\hat{a}$

$$\hat{a} = \bar{y} - b\bar{x}.$$

Taking the partial derivative with respect to b

$$\begin{aligned}\frac{\partial}{\partial b}\sum_{i=1}^{n}\left(y_i - a - bx_i\right)^2 &= -2\sum_{i=1}^{n}\left(y_i - a - bx_i\right)x_i \\ &= -2\left(\sum_{i=1}^{n} y_i x_i - a\sum_{i=1}^{n} x_i - b\sum_{i=1}^{n} x_i x_i\right) \\ &= -2n\left(\overline{xy} - a\bar{x} - b\overline{xx}\right).\end{aligned}$$

Plugging in $\hat{a}$, setting the result to 0, and solving for b yields

$$\hat{b} = \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{xx} - \bar{x}\bar{x}} = \frac{\mathrm{Cov}(x,y)}{\mathrm{Var}(x)} = \mathrm{Cor}(x,y)\left(\frac{\mathrm{Std}(y)}{\mathrm{Std}(x)}\right).$$

Since $\frac{\mathrm{Std}(y)}{\mathrm{Std}(x)}$ is constant under the permutation of y, we can calculate the p-value using the permutation test of the correlation.

```
>>> from __future__ import print_function
>>> import numpy as np

>>> X = np.array([np.ones(10), np.random.random_integers(1, 4, 10)]).T
>>> beta = np.array([1.2, 2])
>>> epsilon = np.random.normal(0, .15, 10)
>>> y = X.dot(beta) + epsilon

>>> from permute.core import corr
>>> t, pv_left, pv_right, pv_both, dist = corr(X[:, 1], y)
>>> print(t)
0.998692462616
>>> print(pv_both)
0.0007
>>> print(pv_right)
0.0007
>>> print(pv_left)
1.0

>>> t, pv_both, dist = corr(X[:, 1], y)
>>> print(t)
0.103891027265
>>> print(pv_both)
0.765
>>> print(pv_right)
0.3818
>>> print(pv_left)
0.619
```

1.4 Permutation Tests for Complex Data

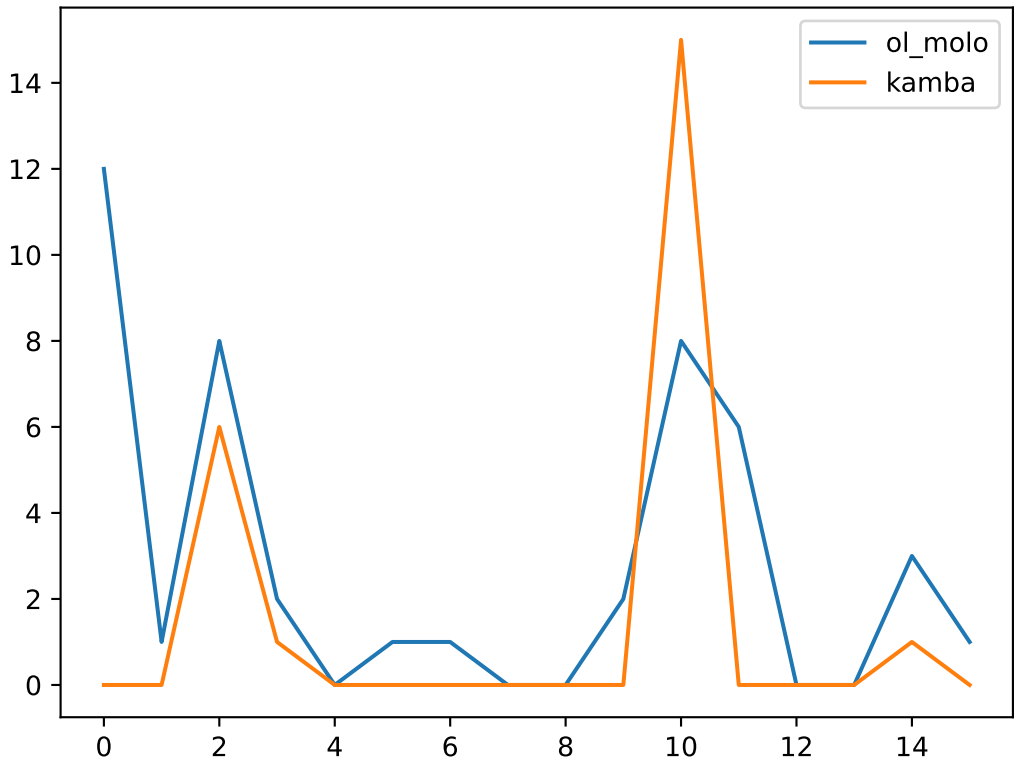
Examples from "Permutation Tests for Complex Data: Theory, Applications and Software" by F. Pesarin and L. Salmaso.

1.4.1 Kenya

The Kenya dataset [CM77] contains 16 observations and two variables in total. It concerns an anthropological study on the "Ol Molo" and "Kamba" populations described above. Table 1 shows the sample frequencies of the 16 phenotypic combinations in the samples selected from the two populations.

Given $X_1, X_2, \ldots, X_n$ and ...

```
>>> from __future__ import print_function
>>> from matplotlib import mlab
>>> import numpy as np
>>> from permute.data import kenya
>>> d = kenya()
>>> for i in range(len(d)):
...     if i == 0:
...         print(d.dtype.names)
...     print(d[i])
('classes', 'ol_molo', 'kamba')
(1, 12, 0)
(2, 1, 0)
(3, 8, 6)
(4, 2, 1)
(5, 0, 0)
(6, 1, 0)
(7, 1, 0)
(8, 0, 0)
(9, 0, 0)
(10, 2, 0)
(11, 8, 15)
(12, 6, 0)
(13, 0, 0)
(14, 0, 0)
(15, 3, 1)
(16, 1, 0)
>>> import matplotlib.pyplot as plt
>>> plt.plot(d['ol_molo'])
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(d['kamba'])
[<matplotlib.lines.Line2D object at ...>]
>>> plt.legend(['ol_molo', 'kamba'])
<matplotlib.legend.Legend object at ...>
>>> plt.show()
```



1.4.2 Chapter 1-4 examples

IPAT Data

This example is shown in Chapter 1.9, page 33-34.

```
>>> import permute.data as data
>>> from permute.core import one_sample, two_sample
>>> from permute.ksample import k_sample, bivariate_k_sample
>>> from permute.utils import get_prng
>>> import numpy as np
```

```
>>> prng = get_prng(2019426)
>>> ipat = data.ipat()
>>> ipat_res = one_sample(ipat.ya, ipat.yb, stat='mean', alternative='greater',␣
↪seed=prng)
>>> print("P-value:", round(ipat_res[0], 4))
P-value: 0.0002
```

Job Satisfaction Data

This example is shown in Chapter 1.10.3, page 41-42.

```
>>> job = data.job()
>>> job_res = two_sample(job.x[job.y == 1], job.x[job.y == 2], stat='mean', reps = 10**5,
↪ alternative='greater', seed=prng)
>>> print("P-value:", round(job_res[0], 4))
P-value: 0.0003
```

Worms Data

This example is shown in Chapter 1.11.12, page 47-48.

```
>>> worms = data.worms()
>>> res = k_sample(worms.x, worms.y, stat='one-way anova', seed=prng)
>>> print("ANOVA p-value:", round(res[0], 4))
ANOVA p-value: 0.0107
```

Testosterone Data

This example is shown in Chapter 2.6.1, page 92-93.

```
>>> testosterone = data.testosterone()
>>> x = np.hstack(testosterone.tolist())
>>> group1 = np.hstack([[i]*5 for i in range(len(testosterone))])
>>> group2 = np.array(list(range(5))*len(testosterone))
>>> print(len(group1), len(group2), len(x))
55 55 55
>>> res = bivariate_k_sample(x, group1, group2, reps=5000, seed=prng)
>>> print("ANOVA p-value:", round(res[0], 4))
ANOVA p-value: 0.0002
```

Massaro-Blair Data

This example is shown in Chapter 4.6, page 240.

```
>>> from permute.npc import npc
>>> mb = data.massaro_blair()
>>> sam1 = mb.y[mb.group == 1]
>>> sam2 = mb.y[mb.group == 2]
>>> first_moment = two_sample(sam1, sam2, alternative='two-sided', reps=5000, keep_
↪dist=True, seed=42)
>>> second_moment = two_sample(sam1**2, sam2**2, alternative='two-sided', reps=5000,␣
↪keep_dist=True, seed=423)
>>> partial_pvalues = np.array([first_moment[0], second_moment[0]])
>>> print("Partial p-values:", round(first_moment[0], 3), round(second_moment[0], 3))
Partial p-values: 0.022 0.01
```

```
>>> npc_distr = np.vstack([first_moment[2], second_moment[2]]).T
>>> global_p = npc(partial_pvalues, np.abs(npc_distr))
>>> print("Global p-value:", round(global_p, 4))
Global p-value: 0.002
```

Fly Data

This example is shown in Chapter 4.6, page 253.

```
fly = data.fly()
vars = fly.dtype.names[1:]
results = {}
for col in vars:
    sam1 = fly[col][fly.group == 0]
    sam2 = fly[col][fly.group == 1]
    if col == 'x7':
        results[str(col)] = two_sample(sam1, sam2, keep_dist=True, seed=prng, plus1=True,
↪ reps=10**4)
    else:
        results[str(col)] = two_sample(sam1, sam2, keep_dist=True, alternative = 'less',␣
↪seed=prng, plus1=True, reps=10**4)
partial_pvalues = np.array(list(map(lambda col: results[col][0], vars)))
print(np.round(partial_pvalues, 3))
[0.027 0.226 0.    0.391 0.    0.413 0.098]

npc_distr = np.array(list(map(lambda col: results[col][2], vars))).T
npc_distr.shape
(10000, 7)
alternatives = ['greater']*6 + ['less']*1
fisher = npc(partial_pvalues, npc_distr, alternatives=alternatives)
liptak = npc(partial_pvalues, npc_distr, alternatives=alternatives, combine = 'liptak')
tippett = npc(partial_pvalues, npc_distr, alternatives=alternatives, combine='tippett')
print("Fisher combined p-value:", fisher)
Fisherer combined p-value: 0.0
print("Liptak combined p-value:", liptak)
Liptak combined p-value: 0.0
print("Tippett combined p-value:", tippett)
Tippett combined p-value: 0.0
```

Post-hoc conditional power analysis

These examples come from Chapter 3.2.1, pages 139-141.

```
# IPAT data
alpha = 0.01
prng = get_prng(78943501)
effect_est = ipat_res[1]
print("Estimated difference in means:", effect_est)
Estimated difference in means: 3.1

z = ipat.ya - ipat.yb - effect_est
```

(continues on next page)

(continued from previous page)

```
simulated_pvalues = np.zeros(1000)
for i in range(1000):
    prng.shuffle(z)
    sim_sam = z.copy() + effect_est
    simulated_pvalues[i] = one_sample(sim_sam, stat='mean', alternative='greater',␣
↪seed=1234, reps=1000)[0]
power = np.mean(simulated_pvalues <= alpha)
print("Estimated power:", power)
Estimated power: 1.0

# Job data
effect_est = job_res[1]
print("Estimated difference in means:", effect_est)
Estimated difference in means: 17.29166666666667

xnorm = job.x
xnorm[job.y == 1] = job.x[job.y == 1] - effect_est
simulated_pvalues = np.zeros(1000)
for i in range(1000):
    prng.shuffle(xnorm)
    sim_sam = xnorm.copy()
    sim_sam[job.y==1] = sim_sam[job.y==1] + effect_est
    simulated_pvalues[i] = two_sample(sim_sam[job.y == 1], sim_sam[job.y == 2], stat=
↪'mean', reps = 10**3, alternative='greater', seed=1234)[0]
power = np.mean(simulated_pvalues <= alpha)
print("Estimated power:", power)
Estimated power: 0.96
```

1.4.3 Westfall-Wolfinger "mult" data

This example comes from Chapter 5.5.

When running the two sample test, use the same initial PRNG seed in order to keep permutations correlated for each test. This is crucial for NPC.

```
>>> import permute.data as data
>>> from permute.core import two_sample
>>> from permute.npc import fwer_minp
>>> import numpy as np
```

```
>>> ww = data.mult()
>>> p1 = two_sample(ww.y1[ww.x == 0], ww.y1[ww.x == 1], alternative="two-sided",␣
↪seed=40929102, keep_dist=True, reps=5000)
>>> p2 = two_sample(ww.y2[ww.x == 0], ww.y2[ww.x == 1], alternative="two-sided",␣
↪seed=40929102, keep_dist=True, reps=5000)
>>> p3 = two_sample(ww.y3[ww.x == 0], ww.y3[ww.x == 1], alternative="two-sided",␣
↪seed=40929102, keep_dist=True, reps=5000)
```

```
>>> pvalues = np.array([p1[0], p2[0], p3[0]])
>>> distr = np.abs(np.vstack([p1[2], p2[2], p3[2]]).T)
>>> pvalues_adj_fisher = fwer_minp(pvalues, distr, combine="fisher")
```

(continues on next page)

(continued from previous page)

```
>>> pvalues_adj_liptak = fwer_minp(pvalues, distr, combine="liptak")
>>> pvalues_adj_tippett = fwer_minp(pvalues, distr, combine="tippett")
```

```
print("Adjusted p-values \nFisher:", pvalues_adj_fisher, "\nLiptak:", pvalues_adj_liptak,
↪ "\nTippett:", pvalues_adj_tippett)
Adjusted p-values
Fisher: [0.08758248 0.04819036 0.0219956 ]
Liptak: [0.08758248 0.0459908  0.02339532]
Tippett: [0.08758248 0.06258748 0.02359528]
```

CHAPTER

TWO

API REFERENCE

2.1 Data sets

Standard test data.

For more information, see

- http://www.wiley.com/legacy/wileychi/pesarin/material.html

permute.data.**kenya**()
The Kenya dataset contains 16 observations and two variables in total. It concerns an anthropological study on the "Ol Molo" and "Kamba" populations.

permute.data.**load**(*f*)
Load a data file located in the data directory.

Parameters

f [string] File name.

Returns

x [array like] Data loaded from permute.data_dir.

2.2 Utility functions

Various utilities and helper functions.

permute.utils.**binom_conf_interval**(*n*, *x*, *cl=0.975*, *alternative='two-sided'*, *p=None*, ***kwargs*)
Compute a confidence interval for a binomial p, the probability of success in each trial.

Parameters

n [int] The number of Bernoulli trials.

x [int] The number of successes.

cl [float in (0, 1)] The desired confidence level.

alternative [{"two-sided", "lower", "upper"}] Indicates the alternative hypothesis.

p [float in (0, 1)] Starting point in search for confidence bounds for probability of success in each trial.

kwargs [dict] Key word arguments

Returns

tuple lower and upper confidence level with coverage (approximately) 1-alpha.

Notes

xtol [float] Tolerance

rtol [float] Tolerance

maxiter [int] Maximum number of iterations.

permute.utils.**binomial_p**(*x*, *n*, *p*, *alternative='greater'*)

Parameters

x [array-like] list of elements consisting of x in {0, 1} where 0 represents a failure and 1 represents a seccuess

p [int] hypothesized number of successes in n trials

n [int] number of trials

alternative [{'greater', 'less', 'two-sided'}] alternative hypothesis to test (default: 'greater')

Returns

——-

float estimated p-value

permute.utils.**get_prng**(*seed=None*)

Turn seed into a cryptorandom instance

Parameters

seed [{None, int, str, RandomState}] If seed is None, return generate a pseudo-random 63-bit seed using np.random and return a new SHA256 instance seeded with it. If seed is a number or str, return a new cryptorandom instance seeded with seed. If seed is already a numpy.random RandomState or SHA256 instance, return it. Otherwise raise ValueError.

Returns

RandomState

permute.utils.**hypergeom_conf_interval**(*n*, *x*, *N*, *cl=0.975*, *alternative='two-sided'*, *G=None*, ***kwargs*)

Confidence interval for a hypergeometric distribution parameter G, the number of good objects in a population in size N, based on the number x of good objects in a simple random sample of size n.

Parameters

n [int] The number of draws without replacement.

x [int] The number of "good" objects in the sample.

N [int] The number of objects in the population.

cl [float in (0, 1)] The desired confidence level.

alternative [{"two-sided", "lower", "upper"}] Indicates the alternative hypothesis.

G [int in [0, N]] Starting point in search for confidence bounds for the hypergeometric parameter G.

kwargs [dict] Key word arguments

Returns

tuple lower and upper confidence level with coverage (at least) 1-alpha.

Notes

xtol [float] Tolerance

rtol [float] Tolerance

maxiter [int] Maximum number of iterations.

permute.utils.**hypergeometric**(*x*, *N*, *n*, *G*, *alternative='greater'*)

Parameters

x [int] number of *good* elements observed in the sample

N [int] population size

n [int] sample size

G [int] hypothesized number of good elements in population

alternative [{'greater', 'less', 'two-sided'}] alternative hypothesis to test (default: 'greater')

Returns

——-

float estimated p-value

permute.utils.**permute**(*x*, *seed=None*)

Permute an array in-place

Parameters

x [array-like] A 1-d array

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

Returns

None Original array is permuted in-place, nothing is returned.

permute.utils.**permute_incidence_fixed_sums**(*incidence*, *k=1*, *seed=None*)

Permute elements of a (binary) incidence matrix, keeping the row and column sums in-tact.

Parameters

incidence [2D ndarray] Incidence matrix to permute.

k [int] The number of successful pairwise swaps to perform.

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

Returns

permuted [2D ndarray] The permuted incidence matrix.

Notes

The row and column sums are kept fixed by always swapping elements two pairs at a time.

permute.utils.**permute_rows**(*m*, *seed=None*)
Permute the rows of a matrix in-place

Parameters

m [array-like] A 2-d array

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

Returns

None Original matrix is permuted in-place, nothing is returned.

permute.utils.**permute_within_groups**(*x*, *group*, *seed=None*)
Permutation of condition within each group.

Parameters

x [array-like] A 1-d array indicating treatment.

group [array-like] A 1-d array indicating group membership

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

Returns

permuted [array-like] The within group permutation of x.

permute.utils.**potential_outcomes**(*x*, *y*, *f*, *finverse*)
Given observations x under treatment and y under control conditions, returns the potential outcomes for units under their unobserved condition under the hypothesis that $x_i = f(y_i)$ for all units.

Parameters

x [array-like] Outcomes under treatment

y [array-like] Outcomes under control

f [function] An invertible function

finverse [function] The inverse function to f.

Returns

potential_outcomes [2D array] The first column contains all potential outcomes under the treatment, the second column contains all potential outcomes under the control.

2.3 Quality assurance

Quality assurance and data cleaning.

permute.qa.**find_consecutive_duplicate_rows**(*x*, *as_string=False*)
Find rows which are duplicated in x

permute.qa.**find_duplicate_rows**(*x*, *as_string=False*)
Find rows which are duplicated in x

Notes

If you load a file, for example *nsgk.csv*, as a 2D array, say *x*, then if you found '16,20,2,8' in the list returned by `find_duplicate_rows(x, as_string=True)` you might do something like:

```
$ grep -n --context=1 '16,20,2,8' nsgk.csv
12512-16,15,2,8
12513:16,20,2,8
12514-16,45,2,8
--
12532-17,17,2,8
12533:16,20,2,8
12534-17,24,2,8
```

http://stackoverflow.com/questions/8560440/removing-duplicate-columns-and-rows-from-a-numpy-2d-array

2.4 Core functions

Core functions.

permute.core.**corr**(*x*, *y*, *alternative='greater'*, *reps=10000*, *seed=None*, *plus1=True*)
Simulate permutation p-value for Pearson correlation coefficient

Parameters

x [array-like]

y [array-like]

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

reps [int]

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

tuple Returns test statistic, p-value, simulated distribution

permute.core.**one_sample**(*x*, *y=None*, *reps=100000*, *stat='mean'*, *alternative='greater'*, *keep_dist=False*, *seed=None*, *plus1=True*)

One-sided or two-sided, one-sample permutation test for the mean, with p-value estimated by simulated random sampling with reps replications.

Alternatively, a permutation test for equality of means of two paired samples.

Tests the hypothesis that x is distributed symmetrically symmetric about 0 (or x and y have the same center) against the alternative that x comes from a population with mean

(a) greater than 0 (greater than that of the population from which y comes), if side = 'greater'

(b) less than 0 (less than that of the population from which y comes), if side = 'less'

(c) different from 0 (different from that of the population from which y comes), if side = 'two-sided'

If `keep_dist`, return the distribution of values of the test statistic; otherwise, return only the number of permutations for which the value of the test statistic and p-value.

Parameters

x [array-like] Sample 1

y [array-like] Sample 2. Must preserve the order of pairs with x. If None, x is taken to be the one sample.

reps [int] number of repetitions

stat [{'mean', 't'}] The test statistic. The statistic is computed based on either z = x or z = x - y, if y is specified.

(a) If stat == 'mean', the test statistic is mean(z).

(b) If stat == 't', the test statistic is the t-statistic– but the p-value is still estimated by the randomization, approximating the permutation distribution.

(c) If stat is a function (a callable object), the test statistic is that function. The function should take a permutation of the data and compute the test function from it. For instance, if the test statistic is the maximum absolute value, $\max_i |z_i|$, the test statistic could be written:

f = lambda u: np.max(abs(u))

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

keep_dist [bool] flag for whether to store and return the array of values of the irr test statistic

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float the estimated p-value

float the test statistic

list The distribution of test statistics. These values are only returned if *keep_dist* == True

permute.core.**spearman_corr**(*x*, *y*, *alternative='greater'*, *reps=10000*, *seed=None*, *plus1=True*)

Simulate permutation p-value for Spearman correlation coefficient

Parameters

x [array-like]

y [array-like]

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

reps [int]

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

tuple Returns test statistic, p-value, simulated distribution

permute.core.**two_sample**(*x*, *y*, *reps=100000*, *stat='mean'*, *alternative='greater'*, *keep_dist=False*, *seed=None*, *plus1=True*)

One-sided or two-sided, two-sample permutation test for equality of two means, with p-value estimated by simulated random sampling with reps replications.

Tests the hypothesis that x and y are a random partition of x,y against the alternative that x comes from a population with mean

(a) greater than that of the population from which y comes, if side = 'greater'

(b) less than that of the population from which y comes, if side = 'less'

(c) different from that of the population from which y comes, if side = 'two-sided'

If `keep_dist`, return the distribution of values of the test statistic; otherwise, return only the number of permutations for which the value of the test statistic and p-value.

Parameters

x [array-like] Sample 1

y [array-like] Sample 2

reps [int] number of repetitions

stat [{'mean', 't'}] The test statistic.

(a) If stat == 'mean', the test statistic is (mean(x) - mean(y)) (equivalently, sum(x), since those are monotonically related)

(b) If stat == 't', the test statistic is the two-sample t-statistic– but the p-value is still estimated by the randomization, approximating the permutation distribution. The t-statistic is computed using scipy.stats.ttest_ind

(c) If stat is a function (a callable object), the test statistic is that function. The function should take two arguments: given a permutation of the pooled data, the first argument is the "new" x and the second argument is the "new" y. For instance, if the test statistic is the Kolmogorov-Smirnov distance between the empirical distributions of the two samples, $\max_t |F_x(t) - F_y(t)|$, the test statistic could be written:

f = lambda u, v: np.max([abs(sum(u<=val)/len(u)-sum(v<=val)/len(v)) for val in np.concatenate([u, v])])

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

keep_dist [bool] flag for whether to store and return the array of values of the irr test statistic

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float the estimated p-value

float the test statistic

list The distribution of test statistics. These values are only returned if *keep_dist* == True

permute.core.**two_sample_conf_int**(*x*, *y*, *cl=0.95*, *alternative='two-sided'*, *seed=None*, *reps=10000*, *stat='mean'*, *shift=None*, *plus1=True*)

One-sided or two-sided confidence interval for the parameter determining the treatment effect. The default is the "shift model", where we are interested in the parameter d such that x is equal in distribution to y + d. In general, if we have some family of invertible functions parameterized by d, we'd like to find d such that x is equal in distribution to f(y, d).

Parameters

x [array-like] Sample 1

y [array-like] Sample 2

cl [float in (0, 1)] The desired confidence level. Default 0.95.

alternative [{"two-sided", "lower", "upper"}] Indicates the alternative hypothesis.

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

reps [int] number of repetitions in two_sample

stat [{'mean', 't'}] The test statistic.

(a) If stat == 'mean', the test statistic is (mean(x) - mean(y)) (equivalently, sum(x), since those are monotonically related)

(b) If stat == 't', the test statistic is the two-sample t-statistic– but the p-value is still estimated by the randomization, approximating the permutation distribution. The t-statistic is computed using scipy.stats.ttest_ind

(c) If stat is a function (a callable object), the test statistic is that function. The function should take two arguments: given a permutation of the pooled data, the first argument is the "new" x and the second argument is the "new" y. For instance, if the test statistic is the Kolmogorov-Smirnov distance between the empirical distributions of the two samples, $\max_t |F_x(t) - F_y(t)|$, the test statistic could be written:

f = lambda u, v: np.max([abs(sum(u<=val)/len(u)-sum(v<=val)/len(v)) for val in np.concatenate([u, v])])

shift [float] The relationship between x and y under the null hypothesis.

(a) If None, the relationship is assumed to be additive (e.g. x = y+d)

(b) A tuple containing the function and its inverse (f, f^{-1}), so $x_i = f(y_i, d)$ and $y_i = f^{-1}(x_i, d)$

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

tuple the estimated confidence limits

Notes

xtol [float] Tolerance in brentq

rtol [float] Tolerance in brentq

maxiter [int] Maximum number of iterations in brentq

permute.core.**two_sample_core**(*potential_outcomes_all*, *nx*, *tst_stat*, *alternative='greater'*, *reps=100000*, *keep_dist=False*, *seed=None*, *plus1=True*)

Main workhorse function for two_sample and two_sample_shift

Parameters

potential_outcomes_all [array-like] 2D array of potential outcomes under treatment (1st column) and control (2nd column). To be passed in from potential_outcomes

nx [int] Size of the treatment group x

reps [int] number of repetitions

tst_stat: function The test statistic

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

keep_dist [bool] flag for whether to store and return the array of values of the test statistic. Default is False.

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float the estimated p-value

float the test statistic

list The distribution of test statistics. These values are only returned if *keep_dist* == True

permute.core.**two_sample_shift**(*x*, *y*, *reps=100000*, *stat='mean'*, *alternative='greater'*, *keep_dist=False*, *seed=None*, *shift=None*, *plus1=True*)

One-sided or two-sided, two-sample permutation test for equality of two means, with p-value estimated by simulated random sampling with reps replications.

Tests the hypothesis that x and y are a random partition of x,y against the alternative that x comes from a population with mean

(a) greater than that of the population from which y comes, if side = 'greater'

(b) less than that of the population from which y comes, if side = 'less'

(c) different from that of the population from which y comes, if side = 'two-sided'

If `keep_dist`, return the distribution of values of the test statistic; otherwise, return only the number of permutations for which the value of the test statistic and p-value.

Parameters

x [array-like] Sample 1

y [array-like] Sample 2

reps [int] number of repetitions

stat [{'mean', 't'}] The test statistic.

(a) If stat == 'mean', the test statistic is (mean(x) - mean(y)) (equivalently, sum(x), since those are monotonically related)

(b) If stat == 't', the test statistic is the two-sample t-statistic– but the p-value is still estimated by the randomization, approximating the permutation distribution. The t-statistic is computed using scipy.stats.ttest_ind

(c) If stat is a function (a callable object), the test statistic is that function. The function should take two arguments: given a permutation of the pooled data, the first argument is the "new" x and the second argument is the "new" y. For instance, if the test statistic is the Kolmogorov-Smirnov distance between the empirical distributions of the two samples, $\max_t |F_x(t) - F_y(t)|$, the test statistic could be written:

f = lambda u, v: np.max([abs(sum(u<=val)/len(u)-sum(v<=val)/len(v)) for val in np.concatenate([u, v])])

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

keep_dist [bool] flag for whether to store and return the array of values of the irr test statistic

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

shift [float] The relationship between x and y under the null hypothesis.

(a) A constant scalar shift in the distribution of y. That is, x is equal in distribution to y + shift.

(b) A tuple containing the function and its inverse (f, f^{-1}), so $x_i = f(y_i)$ and $y_i = f^{-1}(x_i)$

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float the estimated p-value

float the test statistic

list The distribution of test statistics. These values are only returned if *keep_dist* == True

2.5 Stratified testing

Stratified permutation tests.

permute.stratified.**corrcoef**(*x*, *y*, *group*)

Calculates sum of Spearman correlations between x and y, computed separately in each group.

Parameters

x [array-like] Variable 1

y [array-like] Variable 2, of the same length as x

group [array-like] Group memberships, of the same length as x

Returns

float The sum of Spearman correlations

permute.stratified.**sim_corr**(*x*, *y*, *group*, *reps=10000*, *alternative='greater'*, *seed=None*, *plus1=True*)

Simulate permutation p-value of stratified Spearman correlation test.

Parameters

x [array-like] Variable 1

y [array-like] Variable 2, of the same length as x

group [array-like] Group memberships, of the same length as x

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

reps [int] Number of repetitions

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator.

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float the estimated p-value

float the observed test statistic

list the null distribution

permute.stratified.**stratified_permutationtest**(*group*, *condition*, *response*, *alternative='greater'*, *reps=100000*, *testStatistic='mean'*, *seed=None*, *plus1=True*)

Stratified permutation test based on differences in means.

The test statistic is

$$\sum_{g \in \text{groups}} [f(mean(\text{response for cases in group } g \text{ assigned to each condition}))].$$

The function f is the difference if there are two conditions, and the standard deviation if there are more than two conditions.

There should be at least one group and at least two conditions. Under the null hypothesis, all assignments to the two conditions that preserve the number of cases assigned to the conditions are equally likely.

Groups in which all cases are assigned to the same condition are skipped; they do not contribute to the p-value since all randomizations give the same contribution to the difference in means.

Parameters

group [array-like] Group memberships

condition [array-like] Treatment conditions, of the same length as group

response [array-like] Responses, of the same length as group

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

reps [int] Number of repetitions

testStatistic [function] Function to compute test statistic. By default, stratified_permutationtest_mean The test statistic. Either a string or function.

(a) If stat == 'mean', the test statistic is stratified_permutationtest_mean (default).

(b) If stat is a function (a callable object), the test statistic is that function. The function should take a permutation of the data and compute the test function from it. For instance, if the test statistic is the maximum absolute value, $\max_i |z_i|$, the test statistic could be written:

f = lambda u: np.max(abs(u))

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float the estimated p-value

float the observed test statistic

list the null distribution

permute.stratified.**stratified_permutationtest_mean**(*group*, *condition*, *response*, *groups=None*, *conditions=None*)

Calculates variability in sample means between treatment conditions, within groups.

If there are two treatment conditions, the test statistic is the difference in means, aggregated across groups. If there are more than two treatment conditions, the test statistic is the standard deviation of the means, aggregated across groups.

Parameters

group [array-like] Group memberships

condition [array-like] Treatment conditions, of the same length as group

response [array-like] Responses, of the same length as group

groups [array-like] Group labels. By default, it is the unique values of group

conditions [array-like] Condition labels. By default, it is the unique values of condition

Returns

tst [float] The observed test statistic

permute.stratified.**stratified_two_sample**(*group*, *condition*, *response*, *stat='mean'*, *alternative='greater'*, *reps=100000*, *keep_dist=False*, *seed=None*, *plus1=True*)

One-sided or two-sided, two-sample permutation test for equality of two means, with p-value estimated by simulated random sampling with reps replications.

Tests the hypothesis that x and y are a random partition of x,y against the alternative that x comes from a population with mean

(a) greater than that of the population from which y comes, if side = 'greater'

(b) less than that of the population from which y comes, if side = 'less'

(c) different from that of the population from which y comes, if side = 'two-sided'

Permutations are carried out within the given groups. Under the null hypothesis, observations within each group are exchangeable.

If `keep_dist`, return the distribution of values of the test statistic; otherwise, return only the number of permutations for which the value of the test statistic and p-value.

Parameters

group [array-like] Group memberships

condition [array-like] Treatment conditions, of the same length as group

response [array-like] Responses, of the same length as group

stat [{'mean', 't'}] The test statistic.

(a) If stat == 'mean', the test statistic is (mean(x) - mean(y)) (equivalently, sum(x), since those are monotonically related), omitting NaNs, which therefore can be used to code non-responders

(b) If stat == 't', the test statistic is the two-sample t-statistic– but the p-value is still estimated by the randomization, approximating the permutation distribution. The t-statistic is computed using scipy.stats.ttest_ind

(c) If stat == 'mean_within_strata', the test statistic is the difference in means within each stratum, added across strata.

(d) If stat is a function (a callable object), the test statistic is that function. The function should take a permutation of the pooled data and compute the test function from it. For instance, if the test statistic is the Kolmogorov-Smirnov distance between the empirical distributions of the two samples, $max_t|F_x(t) - F_y(t)|$, the test statistic could be written:

f = lambda u: np.max([abs(sum(u[:len(x)]<=v)/len(x)-sum(u[len(x):]<=v)/len(y)) for v in u])

alternative [{'greater', 'less', 'two-sided'}] The alternative hypothesis to test

reps [int] Number of permutations

keep_dist [bool] flag for whether to store and return the array of values of the test statistic

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator.

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float the estimated p-value

float the test statistic

list The distribution of test statistics. These values are only returned if *keep_dist* == True

2.6 Nonparametric combination of tests

class permute.npc.**Experiment**(*group=None*, *response=None*, *covariate=None*, *randomizer=None*)

A class to represent an experiment.

Attributes

group [vector] group assignment for each observation

response [array_like] array of response values for each observation

covariate [array_like] array of covariate values for each observation

randomizer [Randomizer object] randomizer to use when randomizing group assignments. default is unstratified randomization, randomize_group

Methods

Randomizer	
TestFunc	
make_test_array	
randomize	

permute.npc.**check_combfunc_monotonic**(*pvalues*, *combfunc*)

Utility function to check that the combining function is monotonically decreasing in each argument.

Parameters

pvalues [array_like] Array of p-values to combine

combine [function] The combining function to use.

Returns

True if the combining function passed the check, False otherwise.

permute.npc.**fisher**(*pvalues*)

Apply Fisher's combining function

$$-2 \sum_i \log(p_i)$$

Parameters

pvalues [array_like] Array of p-values to combine

Returns

float Fisher's combined test statistic

permute.npc.**fwer_minp**(*pvalues*, *distr*, *combine='fisher'*, *plus1=True*)

Adjust p-values using the permutation "minP" variant of Holm's step-up method.

When considering a closed testing procedure, the adjusted p-value p_i for a given hypothesis H_i is the maximum of all p-values for tests including H_i as a special case (including the p-value for the H_i test itself).

Parameters

pvalues [array_like] Array of p-values to combine

distr [array_like] Array of dimension [B, n] where B is the number of permutations and n is the number of partial hypothesis tests. The i.

combine [{'fisher', 'liptak', 'tippett'} or function] The combining function to use. Default is "fisher". Valid combining functions must take in p-values as their argument and be monotonically decreasing in each p-value.

Returns

array of adjusted p-values

permute.npc.**inverse_n_weight**(*pvalues*, *size*)

Compute the test statistic

$$-\sum_{s=1}^{S} \frac{p_s}{\sqrt{N_s}}$$

Parameters

pvalues [array_like] Array of p-values to combine

size [array_like] The :math:`i`th entry is the sample size used for the :math:`i`th test

Returns

float combined test statistic

permute.npc.**liptak**(*pvalues*)

Apply Liptak's combining function

$$\sum_i \Phi^{-1}(1-p_i)$$

where Φ^{-1} is the inverse CDF of the standard normal distribution.

Parameters

pvalues [array_like] Array of p-values to combine

Returns

float Liptak's combined test statistic

permute.npc.**npc**(*pvalues*, *distr*, *combine='fisher'*, *plus1=True*)

Combines p-values from individual partial test hypotheses H_{0i} against H_{1i}, $i = 1, \ldots, n$ to test the global null hypothesis

$$\cap_{i=1}^{n} H_{0i}$$

against the alternative

$$\cup_{i=1}^{n} H_{1i}$$

using an omnibus test statistic.

Parameters

pvalues [array_like] Array of p-values to combine

distr [array_like] Array of dimension [B, n] where B is the number of permutations and n is the number of partial hypothesis tests. The i.

combine [{'fisher', 'liptak', 'tippett'} or function] The combining function to use. Default is "fisher". Valid combining functions must take in p-values as their argument and be monotonically decreasing in each p-value.

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

float A single p-value for the global test

permute.npc.**randomize_group**(*data*)
Unstratified randomization

Parameters

data [Experiment object]

Returns

Experiment object Experiment object with randomized group assignments

permute.npc.**randomize_in_strata**(*data*)
Stratified randomization where first covariate is the stratum

Parameters

data [Experiment object]

Returns

Experiment object Experiment object with randomized group assignments

permute.npc.**sim_npc**(*data*, *test*, *combine='fisher'*, *in_place=False*, *reps=10000*, *seed=None*)
Combines p-values from individual partial test hypotheses H_{0i} against H_{1i}, $i = 1, \ldots, n$ to test the global null hypothesis

$$\cap_{i=1}^{n} H_{0i}$$

against the alternative

$$\cup_{i=1}^{n} H_{1i}$$

using an omnibus test statistic.

Parameters

data [Experiment object]

test [array_like] Array of functions to compute test statistic to apply to each column in cols

combine [{'fisher', 'liptak', 'tippett'} or function] The combining function to use. Default is "fisher". Valid combining functions must take in p-values as their argument and be monotonically decreasing in each p-value.

in_place [Boolean] whether randomize group in place, default False

reps [int] number of repetitions

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

Returns

array A single p-value for the global test, test statistic values on the original data, partial p-values

permute.npc.**tippett**(*pvalues*)

Apply Tippett's combining function

$$\max_i \{1 - p_i\}$$

Parameters

pvalues [array_like] Array of p-values to combine

Returns

float Tippett's combined test statistic

2.7 Interrater reliability

A stratified permutation test for multi-rater inter-rater reliability.

There are S strata. There are N_s items in stratum s. There are $N = \sum_{s=1}^{S} N_s$ items in all.

There are C non-exclusive categories to which each of the N items might belong; an item might belong to none of the categories. That is, each item might be "labeled" with any of the 2^C subsets of the C labels, including the empty set.

There are R "raters," each of whom labels each of the N items with zero or more elements of C.

Define $L_{s,i,c,r} = 1$, if rater r assigns label c to item i in stratum s; and $L_{s,i,c,r} = 0$ if not.

We observe $\{L_{s,i,c,r}\}$ for $s = 1...S$; $i = 1, ..., N_s$; $c = 1, ..., C$; and $r = 1, ..., R$.

We want to know whether the categorizations are "reliable," in the sense that agreement among the raters is higher than would be expected "by chance." The reliability of each category c is of interest, rather than an overall rating for all C categories.

Fix c, since we are considering only one category at a time.

The null hypothesis for category c is that, for each rater r, and each stratum s, the values $\{L_{s,i,c,r}\}$ are exchangeable; that for each rater r, the values $\{L_{s,i,c,r}\}$ for different strata s are independent; and that the values are independent across raters.

Our test conditions on the sets of labels each rater assigns within each stratum, but not on the items to which those labels are assigned. The null distribution involves permuting the assignments each given rater makes of category c to items within each stratum s, permuting independently across across raters and across strata.

The test statistic within stratum s is

$$\rho_s \equiv \frac{1}{N_s \binom{R}{2}} \sum_{i=1}^{N_s} \sum_{r=1}^{R-1} \sum_{v=r+1}^{R} 1(L_{s,i,r} = L_{s,i,v}) = \frac{1}{N_s R(R-1)} \sum_{i=1}^{N_s} (y_{si}(y_{si} - 1) + (R - y_{si})(R - y_{si} - 1)).$$

That is, within each stratum, we count the number of concordant pairs of assignments. If all R raters agree whether item i in stratum s belongs to category c, that contributes a term $\binom{R}{2}$ to the sum. If only half agree, the term for item i contributes $2\binom{N/2}{2}$ to the sum. The normalization makes perfect agreement within stratum s correspond to $\rho_s = 1$.

To combine the results across strata to get an overall p-value, we could use any of the methods we've discussed, or the NPC (nonparametric combination of test) methods described in Pesarin and Salmaso, based on the p-values in different strata. For instance, Fisher's combination statistic is

$$\lambda = -\sum_{s=1}^{S} w_s \log \hat{p}_s,$$

where the nonnegative weights $\{w_s\}$ are chosen in some sensible manner (e.g., $w_s = N_s^{-1/2}$ would be reasonable).

permute.irr.**compute_ts**(*ratings*)

Compute the test statistic

$$\rho_s \equiv \frac{1}{N_s\binom{R}{2}} \sum_{i=1}^{N_s} \sum_{r=1}^{R-1} \sum_{v=r+1}^{R} 1(L_{s,i,r} = L_{s,i,v}) = \frac{1}{N_s R(R-1)} \sum_{i=1}^{N_s} (y_{si}(y_{si}-1) + (R-y_{si})(R-y_{si}-1)).$$

Parameters

ratings [array_like] Input array of dimension [R, Ns] Each row corresponds to the ratings given by a single rater; columns correspond to items rated.

Returns

rho_s [float] concordance of the ratings, where perfect concordance is 1.0

permute.irr.**simulate_npc_dist**(*perm_distr*, *size*, *obs_ts=None*, *pvalues=None*, *plus1=True*)

Simulates the permutation distribution of the combined NPC test statistic for S matrices of ratings `ratings` corresponding to S strata. The distribution comes from applying `simulate_ts_dist` to each of the S strata.

If obs_ts is not null, computes the reference value of the test statistic before the first permutation. Otherwise, uses the value `obs_ts` for comparison.

If `keep_dist`, return the distribution of values of the test statistic; otherwise, return only the number of permutations for which the value of the irr test statistic is at least as large as `obs_ts`.

Parameters

perm_distr [array_like] Input array of dimension [B, S] Column s is the permutation distribution of ρ_s, for s=1,...,S

size [array_like] Input array of dimension S Each entry corresponds to the number of items, Ns, in the s-th stratum.

obs_ts [array_like] Optional input array of dimension S The s-th entry is ρ_s, the concordance for the s-th stratum. If not input, pvalues must be specified.

pvalues [array_like] Optional input array of dimension S The s-th entry is the p-value corresponding to ρ_s, the concordance for the s-th stratum. If not input, obs_ts must be specified.

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

dict A dictionary containing:

obs_npc [float] observed value of the combined test statistic for the input data, or the input value of `obs_ts` if `obs_ts` was given as input

pvalue [float] A single p-value for the global test. The number of times that obs_npc was at least as extreme as the distribution of combined IRR statistics.

num_perm [int] number of permutations

permute.irr.**simulate_ts_dist**(*ratings*, *obs_ts=None*, *num_perm=10000*, *keep_dist=False*, *seed=None*, *plus1=True*)

Simulates the permutation distribution of the irr test statistic for a matrix of ratings `ratings`

If `obs_ts` is not `None`, computes the reference value of the test statistic before the first permutation. Otherwise, uses the value `obs_ts` for comparison.

If `keep_dist`, return the distribution of values of the test statistic; otherwise, return only the number of permutations for which the value of the irr test statistic is at least as large as `obs_ts`.

Parameters

ratings [array_like] Input array of dimension [R, Ns]

obs_ts [float] if None, `obs_ts` is calculated as the value of the test statistic for the original data

num_perm [int] number of random permutation of the elements of each row of ratings

keep_dist [bool] flag for whether to store and return the array of values of the irr test statistic

seed [RandomState instance or {None, int, RandomState instance}] If None, the pseudorandom number generator is the RandomState instance used by *np.random*; If int, seed is the seed used by the random number generator; If RandomState instance, seed is the pseudorandom number generator

plus1 [bool] flag for whether to add 1 to the numerator and denominator of the p-value based on the empirical permutation distribution. Default is True.

Returns

dict A dictionary containing:

obs_ts [int] observed value of the test statistic for the input data, or the input value of `obs_ts` if `obs_ts` was given as input

geq [int] number of iterations for which the test statistic was greater than or equal to `obs_ts`

num_perm [int] number of permutations

pvalue [float] geq / num_perm

dist [array-like] if `keep_dist`, the array of values of the irr test statistic from the `num_perm` iterations. Otherwise, `None`.

`permute` provides permutation tests and confidence intervals for a variety of nonparametric testing and estimation problems, for a variety of randomization designs.

- Stratified and unstratified tests
- Test statistics in each stratum
- Methods of combining tests across strata
- Nonparametric combinations of tests

2.8 Problems/Methods:

1. The 2-sample problem
2. The *n*-sample problem
3. Tests for the slope in linear regression
4. Tests for quantiles
5. Tests of independence and association: runs tests, permutation association
6. Tests of exchangeability
7. Tests of symmetry: reflection, spherical
8. Permutation ANOVA
9. Goodness of fit tests

2.9 Confidence sets

1. Constant shifts
2. Proportional shifts
3. Monotone shifts

2.10 Links

UC Berkeley's Statistics 240: Nonparametric and Robust Methods.

- 2015 course website
- Philip Stark's lecture notes

"Permutation Tests for Complex Data: Theory, Applications and Software" by Fortunato Pesarin, Luigi Salmaso

- Publisher's website
- Supplementary Material (i.e., code and data)
- NPC test code

"Stochastic Ordering and ANOVA: Theory and Applications with R" by Basso D., Pesarin F., Salmaso L., Solari A.

- R code

BIBLIOGRAPHY

[CM77] C Corrain and F Pesarin Mezzavilla. F. and scardellato, u.(1977). il valore discriminativo di alcuni fattori gm, tra le popolazioni pastorali del kenya. *Atti e Memorie dell'Accademia Patavina di Scienze, Lettere ed Arti, LXXXIX, Parte II: Classe di Scienze Matematiche e Naturali. University of Padua*, pages 55–63, 1977.

[Fis35] Ronald A Fisher. *The design of experiments.* Oliver & Boyd, 1935.

[MDH14] Lillian MacNell, Adam Driscoll, and Andrea N Hunt. What's in a name: exposing gender bias in student ratings of teaching. *Innovative Higher Education*, pages 1–13, 2014.

[Pit37] Edwin JG Pitman. Significance tests which may be applied to samples from any populations (parts i and ii). *Supplement to the Journal of the Royal Statistical Society*, 4(1):119–130; 225–232, 1937.

[Pit38] Edwin JG Pitman. Significance tests which may be applied to samples from any populations: iii. the analysis of variance test. *Biometrika*, pages 322–335, 1938.

[Rom88] Joseph P Romano. A bootstrap revival of some nonparametric distance tests. *Journal of the American Statistical Association*, 83(403):698–708, 1988.

[Rom89] Joseph P Romano. Bootstrap and randomization tests of some nonparametric hypotheses. *The Annals of Statistics*, 17(1):141–159, 1989.

PYTHON MODULE INDEX

p

INDEX

S

T