`permute`—a Python package for permutation tests and confidence sets

by

Kenneth Jarrod Millman

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Arts

in

Biostatistics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sandrine Dudoit, Chair
Professor Philip B. Stark
Professor Nicholas P. Jewell

Spring 2015

`permute`—a Python package for permutation tests and confidence sets

Contents

Chapter 1

Introduction

`permute` is a Python package for permutation tests and confidence sets.[1] Philip B. Stark, Kellie Ottoboni, Stéfan van der Walt, and I developed this package over the last year with most of the work occurring during the last few months. In this report, I briefly explain the purpose of the package (§ 1), our development practices (§ 2), the current functionality (§ 3), and our short- and long-term plans (§ 4).

1.1 Permutation tests

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 [6] and further developed by E. J. G. Pitman [16, 15]. After the introduction of the bootstrap, the ideas were extended in the 1980's by J. Romano [17, 18].

Permutation tests were developed to test hypotheses for which relabeling the observed data was justified by exchangeability[2] 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.

[1] `http://statlab.github.io/permute`

[2] 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.

More generally, permutation tests are possible whenever the null distribution of the data is invariant under the action of some group (see Appendix A 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.

Example 1. 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).

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 p-value from the t-distribution. If the p-value 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).

Example 2. Given $n = 10$ 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 $\{\epsilon_i\}_{i=1}^{10}$ 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 $\{y_i\}_{i=1}^{10}$ are exchangeable since they are just shifted versions of the exchangeable $\{\epsilon_i\}_{i=1}^{10}$. Thus every permutation of the $\{y_i\}_{i=1}^{10}$ 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 normalized by its standard error 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 p-value is the ratio of the count of the *as extreme* or *more extreme* test statistics to the total number of such test statistics. For $n = 10$ you might in principle enumerate all 10! 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 test statistic has a t-distribution with $n-2$ degrees of freedom. If this additional assumption holds, then we can read the p-value 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.

In summary, the general procedure for computing a p-value using permutation testing is to formulate the null hypothesis based on the experimental design and question of interest and then determine whether, under the null hypothesis, the probability distribution of the data is invariant under the action of some group. If so, then given the orbit of the observed data, every element of that orbit is conditionally equally likely; with that knowledge, one can either calculate the null conditional probability distribution of any test statistic or simulate it to any desired level of accuracy.

A big advantage of the permutation approach is that it produces (essentially) exact tests with weaker assumptions about the generating process than parametric tests require. Those weaker assumptions may follow directly from the experimental design. Thus permutation testing allows us to focus on the exact question of interest, rather than a similar question that is amenable to analysis. Additionally, justifying these claims does not require asymptotic theory; we get exact results for small samples. A potential disadvantage is the computational cost; however, with increasing computational resources and improved algorithms, this limitation is less of a concern. Moreover, permutation tests are easier for most people to understand than parametric tests are.

1.2 Confidence sets

Given $X \sim P_\theta$ for some $\theta \in \Theta$,[3] recall that a random set $C(X)$ is a $1-\alpha$ confidence set for θ if

$$P_\theta(\theta \in C(X)) \geq 1-\alpha$$

for all $\theta \in \Theta$. For every $\theta_0 \in \Theta$, let $A(\theta_0)$ be the acceptance region for a test at level α of the null hypothesis $H_{\theta_0} : \theta = \theta_0$ versus the alternative that θ_0 is not the true θ. Denote the family of all such null hypotheses $\mathcal{H} = \{H_{\theta_0} \mid \theta_0 \in \Theta\}$. Thus we have a family of significance-level α tests $\{A(\theta_0) \mid \theta_0 \in \Theta\}$ such that for each $H_{\theta_0} \in \mathcal{H}$,

$$P_{\theta_0}(X \notin A(\theta_0)) \leq \alpha.$$

Define $C(x) \equiv \{\theta \in \Theta \mid x \in A(\theta)\}$. By construction, $\theta \in C(X)$ exactly when $X \in A(\theta)$ and so

$$P_\theta(\theta \in C(X)) = P_\theta(X \in A(\theta)) \geq 1-\alpha.$$

This result allows us to construct a $1-\alpha$ confidence set dual to the family of tests with acceptance regions $A(\theta_0)$.[4] Despite the duality between tests and confidence sets, computing the permutation confidence set often requires additional assumptions in order to construct a family of permutation tests with acceptance regions $A(\theta_0)$ for all $\theta_0 \in \Theta$.

For instance in Example 1 above, we could form an acceptance region for a level α permutation test for the null hypothesis that the treatment had no effect by stipulating that the p-value under the null given the observed data is greater than α. However, using only the original assumptions, it is unclear how you would form acceptance regions for a level α permutation test of any other null hypotheses.

[3]In general, we allow any arbitrary set Θ to index the family of distributions. In the following example, we will take $\Theta = \mathbf{R}$.

[4]Conversely, we can construct families of tests from confidence sets.

Example 1 (Continued)**.** To illustrate how to form $1 - \alpha$ confidence sets from a family of level α permutation tests, we further assume that the effect of treatment in the paired clone experiment is explained by a *shift model.* The shift model assumes that the treatment adds an unknown constant real number d to the control response of the cell. Let (t_i, c_i) denote the observed growth rates of the ith cloned pair where t_i corresponds to the treated clone and c_i the control. Then the shift model is that for each treated clone the effect of treatment is equivalent to the hypothetical growth rate it would have had without treatment t_i^c plus the constant treatment shift d (i.e., $t_i = t_i^c + d$).

Let $H_{d'}$ denote the null hypothesis that $d = d'$ for any $d' \in \mathbf{R}$. Now we can form a permutation test of $H_{d'}$ versus the alternative that $d \neq d'$ using a procedure analogous to the one we used to test the null hypothesis that treatment had no effect. For each fixed value of d', consider the hypothetical pretreatment cloned pair (a_i, b_i) where $a_i \equiv t_i - d'$ and $b_i \equiv c_i$. If $H_{d'}$ were true, then the data for the ith pair would just as likely have been $(b_i + d', a_i)$ as (t_i, c_i), and all pairs are independent. That lets us find the p-value of the hypothesis $d = d'$, on the assumption that the effect of treatment is a simple shift, equal for each clone pair. Inverting the tests to find the range of values of d' for which we would not reject $d = d'$ gives a confidence interval for d.

By assuming a shift model we are able to invert the permutation test to form confidence sets. However, assuming a constant additive effect of treatment across all cell pairs may be unwarranted. It is likely that individual variability among cloned pairs would correspond to variability in treatment effect. For instance, if the treatment involves administering a nutritional supplement, then there is a possibility that the different cloned pairs will have differing abilities to metabolize it.

1.3 Python

Python is a high-level, general purpose programming language, which has become increasingly popular for scientific computing [10, 13]. Unlike some high-level languages used in scientific computing, Python was not specifically designed for scientific applications. However, it quickly attracted interest among scientists and engineers. Initially, it was employed primarily as a “glue” language to couple together compiled binaries for scientific applications written in C or Fortran [4].

As more scientists and engineers began using Python, they started developing third party libraries to provide additional functionality for scientific and numeric computing. In particular, NumPy[5], SciPy[6], and matplotlib[7] provide a core foundation on which other scientific Python packages (such as `permute`) build [12, 19, 7]. NumPy provides the basic n-dimensional array data structure and a small number of basic functions (e.g., linear algebra, Fourier transforms) to compute on this data structure. SciPy adds additional general routines

[5]`http://numpy.org`

[6]`http://scipy.org`

[7]`http://matplotlib.org`

on top of this core functionality necessary for scientific computing including basic statistics and optimization. Complementing these data structures and algorithms, matplotlib provides publication quality 2D plotting.

While it is beyond the scope of this report to explore these packages in more detail, I note that the pseudo-random number generator (PRNG) provided by NumPy is the Mersenne Twister. The Mersenne Twister is an efficient PRNG with a sufficiently large period for most statistical simulations.[8] In addition to providing a high-quality PRNG, NumPy implements Knuth shuffling—an efficient (and simple) algorithm for uniformly generating permutations of sequences.

While Python is similar to R in many respects and is widely used in scientific and numerical computing, it lacks R's extensive support for statistical applications. Recently, however, as an increasing number of data scientists have embraced Python as their primary programming language they have developed several new Python packages including Pandas, statsmodels, and scikit-learn. These new packages have greatly enhanced Python as a language for statistical computing. Our intention is to help accelerate this trend by supplying a high-quality, rigorously tested, and statistically sound package for a large variety of permutation tests and confidence sets. As the package matures, we anticipate contributing generic functionality upstream to the packages we depend on.

1.4 Putting it all together

Our aim is for `permute` to increase the use of permutation methods as well as the usefulness of Python for statistical data analysis. While we wish to see an increased use of permutation methods, our aim is to not only make these methods easier to apply, but also to help researchers apply these methods correctly. To this end, we intend to provide many worked examples illustrating the thought process and not just the mechanical application of these methods. Further our goal is to provide tools to prototype and implement custom tests such that researchers can easily develop permutation tests appropriate for their experiments. As suggested by our examples, experimental design needs to be carefully considered in designing permutation tests. We would like to provide the tools that will enable scientists to conduct tests that correspond to their designs. Finally, we aim to provide Python tools to statisticians as well as better statistical tools to Python users.

[8]It is the default PRNG in R as well. While sufficient for most statistical simulations, for other applications such as cryptography it may be insufficient.

Chapter 2

Development practices

As this is a new software project, we invested significant effort in setting up a development infrastructure to ensure our work is tracked, thoroughly and continually tested, and incrementally improved and documented. To this end, we have adopted best practices for software development used by many successful open source projects [11].

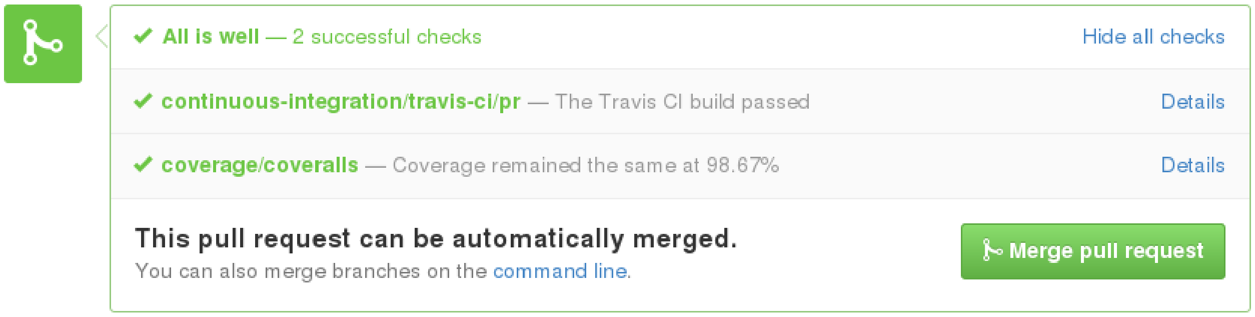
2.1 Version control and code review

We use Git[1] as our version control system (VCS) and GitHub[2] as the public hosting service for our official `upstream` repository (`https://github.com/statlab/permute`). Each developer has their own copy, or fork, of the `upstream` repository. We each work on our own repositories and use the `upstream` repository as our coordination or integration repository.

Git allows us to track and manage how our code changes over time as well as review all new functionality before merging it into the `upstream` repository. To get new code integrated in the `upstream` repository, we use GitHub's *pull request* mechanism. This enables us to review code before integrating it. In the following section, I describe how we automate our testing to generate reports for all pull requests. This way we can reduce the risk that changes to our code break existing functionality. Once a pull request is reviewed and accepted, it is merged into the `upstream` repository.

Requiring all new code to undergo review provides several benefits. Code review increases the quality and consistency of our codebase. It helps maintain a high level of test coverage (see below). Moreover, it also helps keep the development team aware of the work other team members are doing. While we are currently a small team and we meet regularly, having the code review system in place will make it easier for new people to contribute as well as capturing our design discussions and decisions for future reference.

[1]`http://git-scm.com`
[2]`https://github.com`


✓ All is well — 2 successful checks Hide all checks

✓ continuous-integration/travis-ci/pr — The Travis CI build passed Details

✓ coverage/coveralls — Coverage remained the same at 98.67% Details

This pull request can be automatically merged.
You can also merge branches on the command line.

Figure 1: Pull request and continuous integration.

Each pull request triggers an automated system to run the full test suite on the updated codebase. This means that when you go to review a pull request you can immediately see whether the change breaks any of the tests as well as whether the new code decreases the overall test coverage. For example, the above report indicates that the associated pull request does not break existing code and does not change our test coverage.

2.2 Testing and continuous integration

We use the `nose` testing framework for automating our testing procedures.[3] This is the standard testing framework used by the core packages in the scientific Python ecosystem. Automating the tests allows us to monitor a proxy for code correctness when making changes as well as simplifying the code review process for new code. Without automated testing, we would have to manually test all the code every time a change is proposed. The `nose` testing framework simplifies test creation, discovery, and running. It has an extensive set of plugins to add functionality for coverage reporting, test annotation, profiling, as well as inspecting and testing documentation.

Our goal is to test every line of code. For example, not only do we want to test every function in our package, but if a specific function has internal logic we want to test each possible execution path through the function. Having tested each line of code increases our confidence in our codebase, but more importantly provides us some measure of assurance that changes we make do not break existing code. It also increases our confidence that new code works, which reduces the friction of accepting contributions. Currently over 98% of `permute`'s lines of code get executed at least once by our test system.

We have configured Travis CI[4] and `coveralls`[5] to be automatically triggered whenever a commit is made to a pull request or the upstream master (see Figure 1). These systems then run the full test suite using different versions of our dependencies (e.g., Python 2.7 and 3.4) every time a new commit is made to a repository or pull request.

[3]`https://nose.readthedocs.org`
[4]`https://travis-ci.org`
[5]`https://coveralls.io`

2.3 Documentation

We use Sphinx[6] as our documentation system and already have good developer documentation and the foundation for high-quality user documentation. Sphinx is the standard documentation system for Python projects and is used by the core scientific Python packages. We use Python docstrings and follow the NumPy docstring standard[7] to document all the modules and functions in `permute`. Using Sphinx and some NumPy extensions, we have a system for autogenerating the project documentation (as HTML or PDF) using the docstrings as well as stand-alone text written in a light-weight markdown-like language, called reStructuredText[8]. This system enables us to easily embed references, figures, code that is auto-run during documentation generation, as well as mathematics using LaTeX.

2.4 Release management

Our development workflow ensures that the official `upstream` repository is always stable and ready for use. This means anyone can get our official upstream master at any point, install it and start using it. We also make official releases available as source tarballs as well as Python built-packages[9] uploaded to the Python Package Index, or PyPI,[10] with release announcements posted to our mailing list. To install the latest release of `permute` and its dependencies, type the following command from a shell prompt (assuming you have Python and a recent version of pip):

```
    $ pip install permute
```

[6]`http://sphinx-doc.org`

[7]`https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt`

[8]`http://docutils.sourceforge.net/rst.html`

[9]Presently our code is pure Python, but we release Python wheels. Wheels are the new standard built-package format for Python.

[10]PyPI is the Python equivalent of The Comprehensive R Archive Network (CRAN).

Chapter 3

Current functionality

While the scope of our package is large, we have initially focused on a few features based on our current collaborations. In particular, we have implemented built-in datasets, functions for data cleaning and quality assurance, core functionality like various types of permutations and confidence intervals, some stratified tests and confidence sets, and a novel stratified permutation test for multirater interrater reliability.[1]

3.1 Data

To simplify testing, for use in our documentation, and to provide users with example data, we provide built-in datasets. At this point, the majority of the datasets come from the examples used in “Permutation Tests for Complex Data: Theory, Applications and Software” [14]. These datasets are discussed in more details in § 4.1. We also have a data set, which we use as an example for the stratified multirater interrater reliability test (§ 3.6).

3.2 Utility functions

We provide utility functions that are useful in multiple specific permutation tests as well as functions useful for ensuring our unit tests are repeatable when they call functions that rely on a PRNG. This includes core functions for permuting various data structures in different ways. Currently, we have functions for permuting the rows of a matrix in-place, permuting conditions within each group, and permuting the elements of a (binary) incidence matrix keeping the row and column sums constant.

[1]In Appendix B, I have included a couple worked examples to give the reader a sense of how they would use `permute` in practice. For additional documentation and examples, please see the project website: `http://statlab.github.io/permute`.

3.3 Quality assurance

As we added datasets, we also incrementally added tools to simplify data cleaning or quality assurance (QA). Currently, we provide two helper functions: one for reporting duplicate rows in a matrix in various formats and another for reporting duplicate consecutive rows in a matrix in various formats. Rather than having a list of planned QA functionality, we have been opportunistic in only implementing functionality as needed by the datasets included in our `data` module.

3.4 Core tools

The `core` module contains classic (non-stratified) permutation tests and confidence sets. For instance we have a function for computing a one- or two-sided, two-sample permutation test for the equality of two means. We also have functions for computing confidence intervals for a binomial. While one of our collaborators, Anne Boring from OFCE-Sciences Po in Paris, was visiting to work on a study evaluating bias in student evaluations of teachers [2], we added code to simulate permutation p-values for a Spearman correlation coefficient.

3.5 Stratified tests

Some experimental designs have natural groupings. In these situations, it often makes sense to estimate effects within groups and then combine the within-group estimates to get an overall estimate. To do this you carry out the permutation test within each group and then aggregate test statistics across groups. This helps control for group-level effects.

The stratified tests module was the first code we developed and began by adapting some functions from an IPython notebook written by Philip demonstrating how to perform a stratified permutation test using the sum of the differences in means between two or more conditions in each group (stratum) as the test statistic. More recently we added a function to simulate permutation p-values for a stratified Spearman correlation test, which we implemented during Anne Boring's visit.

3.6 Interrater reliability

The multirater interrater reliability module was motivated by a problem presented to us by Naomi Stark. She was interested in analyzing rater reliability in a dataset collected during the following experiment. Several different raters watched several videos. Each video was paused at regular time intervals and each rater was asked to indicating whether or not a particular thing (e.g., a person in the video was seated) occurred during the video segment they just viewed.[2] The question posed to us was to determine whether these responses (or ratings) were reliable among raters.

For each video and rater, we have a vector of 0s and 1s indicating whether the event occurred or not during each time interval. Under the null hypothesis, the elements of this vector are exchangeable. So the permutation distribution is derived from permuting the elements of each rater's rating vector independently. For our test statistic, we count the number of concordant pairs of assignments for each video (i.e., stratum) and normalize this number to range between 0 and 1. We then calculate the p-value for each video. Then we compute an overall p-value by combining the results across videos (i.e., strata) using the nonparametric combination of tests (NPC) described in Pesarin and Salmaso.

In 1960, Cohen [3] introduced the kappa statistic for measuring the agreement between two raters correcting for chance. Since then a number of extensions have been proposed, but we were unable to find one that provided a stratified test for multiple raters. We developed our own based on a proposal by Philip Stark. The complete details of our method are included in the documentation and are implemented with tests in our code. A descriptive manuscript will soon follow.

[2]In fact, they were asked whether several different things occurred in the video segment, but we consider each thing separately. So for the purpose of this report, we will assume that there is only one thing of interest that is either present or absent during each video segment.

Chapter 4

Next steps

While the development of `permute` will continue to be guided by the scientific projects we engage with, there are a few things already planned.

4.1 "Permutation Tests for Complex Data" Examples

On the companion website[1] to "Permutation Tests for Complex Data: Theory, Applications and Software" by F. Pesarin and L. Salmaso [14], the authors provide R functions for implementing the application examples presented in their book. They've also made their datasets available for download (as well as some Matlab and SAS code). In preparation for the second edition, we will be implementing these examples in Python as part of our package.

As mentioned above, these datasets are already included in our package. We have also created a GitHub repository,[2] where we have made some tentative improvements to the code. In order to create a test suite to ensure we are able to independently produce the results given by the R implementation, we plan to develop it into an R package. Since the existing R code was released for the purpose of illustration, rather than as an attempt by the authors to provide an R library for reuse, we do not currently plan to submit the package to CRAN. However, it will be publicly available on GitHub and will be easily installable (we will need to make it easy to install for the purposes of our automated testing anyway).

In addition to using the examples and R code as part of our testing suite, we will also include a detailed discussion of each example from the book in our user documentation. While our intention is not to further develop and improve the R package, we will fix bugs and refactor the code as necessary to ensure that we can validate our results for the examples using `permute` with an independent implementation in R. As part of our documentation we will include snippets of the R code, in the hopes that it might serve users of `permute` who are more familiar with R as they work through the examples in Python.

[1] `http://www.wiley.com/legacy/wileychi/pesarin/material.html`
[2] `https://github.com/statlab/permuter`

4.2 Missing features and new projects

Now that we have our development infrastructure in place and some experience in designing the API, it will be easier to add new permutation tests (e.g., for slope in linear regression, for independence, or for symmetry). Even without motivating scientific collaborations, we will begin adding these tests (and others) and their associated confidence sets as it will help us refine our naming and call signature conventions.

Our motivating scientific collaborations have so far involved small- to moderate-sized datasets. We will soon begin working with larger datasets. In particular, we will begin using our software for problems from genomics and functional neuroimaging. Both of these problem domains already use permutation tests and the size of the datasets will help us better understand the performance and scalability issues.

4.3 Design decisions

Thus far we have limited our core dependencies to NumPy and SciPy. As we add new functionality, we will be tempted to leverage additional external packages. For pure Python packages, this will cause little difficulty. However, many of the packages we will be interested in are partially implemented in C, C++, or Fortran. Depending on packages that require compilation will increase the complexity of installation for some users.

While choosing what packages to depend on will require some thought, a more difficult set of questions will arise as we attempt to finalize our application programming interface (API). We've already spent a lot of effort unifying our call signatures and naming conventions. However, we have begun discussing more generic structures for specifying test statistics. For instance, here is the call signature of our two sample permutation test for the equality of two means:

```
def two_sample(x, y, reps=10**5, stat='mean', alternative='greater',
               keep_dist=False, interval=False, level=0.95, seed=None)
```

Currently, you can specify the test statistic by setting `stat` to `"mean"` or `"t"` depending on whether you want to use the difference in the means or the two-sample t-statistic. The function checks which string is passed in and then chooses the correct function for the test statistic. Ideally, we would like to allow users to pass a function in directly rather than a string. If we go this route it may make sense to consider creating test statistics classes that we can instantiate with the test statistic function. Another possibility would be to see if we can create something like R's formula for specifying our permutation models.

Finally, as we start implementing more tests and applying our package to larger datasets, we will need to take performance more seriously. While we have been able to stick to a pure Python implementation so far, we will eventual need to write C extensions or use an

NumPy-aware optimizing static compiler for Python such Cython[3] or Numba[4] to improve computation time. As permutation tests are inherently parallelizable, we will also need to consider providing parallel processing support.

[3]http://cython.org
[4]http://numba.pydata.org

Bibliography

[1] Mark A Armstrong. *Groups and Symmetry.* Springer, 1997.

[2] Anne Boring, Kellie Ottoboni, and Philip B Stark. "Student Evaluations of Teaching (Mostly) Do Not Measure Teaching Effectiveness." Forthcoming.

[3] Jacob Cohen. "A coefficient of agreement for nominal scales." In: *Educational and Psychological Measurement* 20.1 (1960), pp. 37–46.

[4] Paul F Dubois. "Python: batteries included." In: *Computing in Science & Engineering* 9.3 (2007), pp. 7–9.

[5] David S Dummit and Richard M Foote. *Abstract Algebra.* Wiley, 2003.

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

[7] John D Hunter. "Matplotlib: A 2D graphics environment." In: *Computing in science and engineering* 9.3 (2007), pp. 90–95.

[8] Serge Lang. *Undergraduate Algebra. (Undergraduate Texts in Mathematics).* Springer, 2005.

[9] Lillian MacNell, Adam Driscoll, and Andrea N Hunt. "What's in a Name: Exposing Gender Bias in Student Ratings of Teaching." In: *Innovative Higher Education* (2014), pp. 1–13.

[10] K Jarrod Millman and Michael Aivazis. "Python for scientists and engineers." In: *Computing in Science & Engineering* 13.2 (2011), pp. 9–12.

[11] K Jarrod Millman and Fernando Pérez. "Developing open source scientific practice." In: *Implementing Reproducible Research.* Ed. by V. Stodden, F. Leisch, and R. D. Peng. Chapman and Hall/CRC, 2014, pp. 149–183.

[12] Travis E Oliphant. "Python for scientific computing." In: *Computing in Science & Engineering* 9.3 (2007), pp. 10–20.

[13] Fernando Pérez, Brian E Granger, and John D Hunter. "Python: an ecosystem for scientific computing." In: *Computing in Science & Engineering* 13.2 (2011), pp. 13–21.

[14] Fortunato Pesarin and Luigi Salmaso. *Permutation tests for complex data: theory, applications and software.* John Wiley & Sons, 2010.

[15] Edwin JG Pitman. “Significance tests which may be applied to samples from any populations: III. The analysis of variance test.” In: *Biometrika* (1938), pp. 322–335.

[16] Edwin JG Pitman. “Significance tests which may be applied to samples from any populations (Parts I and II).” In: *Supplement to the Journal of the Royal Statistical Society* 4.1 (1937), 119–130, 225–232.

[17] Joseph P Romano. “A bootstrap revival of some nonparametric distance tests.” In: *Journal of the American Statistical Association* 83.403 (1988), pp. 698–708.

[18] Joseph P Romano. “Bootstrap and randomization tests of some nonparametric hypotheses.” In: *The Annals of Statistics* 17.1 (1989), pp. 141–159.

[19] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. “The NumPy array: a structure for efficient numerical computation.” In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

Appendix A

Groups, actions, and orbits

Groups are a basic object of study in abstract algebra and are useful in many areas outside of mathematics [1, 5, 8]. They are intimately related to the notion of symmetry.

Definition 1. A *group* $(G, \cdot)$ is a set G and a binary operation $\cdot : G \times G \to G$ called (group) *multiplication* such that

1. $ab \in G$ for all $a, b \in G$ (i.e., G is *closed* under multiplication),
2. for all $a, b, c \in G$, $(ab)c = a(bc)$ (i.e., multiplication is *associative*),
3. there exists an *identity* element $e \in G$ such that $eg = ge = g$ for all $g \in G$, and
4. for each $g \in G$ there exists an *inverse* element $g^{-1} \in G$ such that $g^{-1}g = gg^{-1} = e$.

Given a group $(G, \cdot)$, a subset H of G is called a subgroup if $(H, \cdot)$ is a group under the multiplication of G. The integers under addition $(\mathbf{Z}, +)$ is a group with identity element 0 and where the inverse of z is $-z$. The even integers are a subgroup of $(\mathbf{Z}, +)$. The rational numbers excluding 0 under multiplication $(\mathbf{Q} \setminus \{0\}, \cdot)$ is a group with identity element 1 and where the inverse of p/q is q/p. In both cases, it is easy to verify that the group properties are met. Here is a more complicated example that we will use in the sequel.

Example 1. A permutation of an arbitrary set $\mathcal{X}$ is a bijection from $\mathcal{X}$ to itself. The set $S_{\mathcal{X}}$ of all permutations of $\mathcal{X}$ under function composition $(S_{\mathcal{X}}, \circ)$ is a group. The associativity of permutations follows from the associativity of function composition. The bijection taking every element of $\mathcal{X}$ to itself is the identity permutation. Since a permutation is a bijective function, the inverse of every permutation exists and operates from both left and right. If $\mathcal{X} = \{1, 2, \ldots, n\}$, then $S_{\mathcal{X}}$ is the symmetry group of order n and is usually denoted S_n.

Definition 2. Let G and H be groups. A *homeomorphism* $\varphi : G \to H$ takes multiplication from G to H such that $\varphi(xy) = \varphi(x)\varphi(y)$ for all $x, y \in G$.

Note that the multiplication on G and the multiplication on H may be different operations and in general $\varphi(x)y$ will be undefined. To make that explicit, let $(G, \star)$ and $(H, \diamond)$ be groups then we write the condition for a function $\varphi : G \to H$ to be a homeomorphism as $\varphi(x \star y) = \varphi(x) \diamond \varphi(y)$. Since the context will make clear the group multiplication being used, we will not usually distinguish the different operators. A homeomorphism takes the identity of G to the identity of H since $\varphi(g) = \varphi(ge) = \varphi(g)\varphi(e)$ for all $g \in G$. It takes inverses to inverses since $\varphi(e) = \varphi(g^{-1}g) = \varphi(g^{-1})\varphi(g)$ for all $g \in G$. If φ is a bijection, then it is called an isomorphism.

Definition 3. An *action* of a group G on a set $\mathcal{X}$ is a homeomorphism $\varphi : G \to S_{\mathcal{X}}$.

We say *an* action since a group G can act on $\mathcal{X}$ in more than one way. Recalling the definition of a homeomorphism, this means that $\varphi(xy) = \varphi(x)\varphi(y)$ for all $x, y \in G$ where $\varphi(xy), \varphi(x), \varphi(y) \in S_{\mathcal{X}}$. Also notice that $\varphi(x)\varphi(y)$ represents composition of the permutations $\varphi(x)$ and $\varphi(y)$. Moreover, $\varphi(e)$ is the identity permutation in $S_{\mathcal{X}}$. It is standard to write gx instead of $\varphi(g)(x)$ since it is more compact. Note that gx is not "multiplying" an element g of G with an element x of $\mathcal{X}$, but represents the group action of g on x, which can be thought of as the image of x under the permutation of $\mathcal{X}$ induced by g. Each $g \in G$ corresponds to a permutation of $\mathcal{X}$ defined by the homeomorphism.

Definition 4. The *orbit* $G(x) \equiv \{gx \mid g \in G\}$ of any x in $\mathcal{X}$ is the set of all images of the particular point x under all elements of G.

Example 2. The additive group $(\mathbf{Z}, +)$ acts on the real line $\mathbf{R}$ by translation. That is, for every integer z and any real number x the group action is given by $x \mapsto z + x$. To show that this a homeomorphism $\varphi : (\mathbf{Z}, +) \to (S_{\mathbf{R}}, \circ)$, note that given any two integers m and n and any real number x the following holds $(m + n) + x = m + (n + x)$. The orbit of π (i.e., 3.14152...) under this group action is

$$\{z\pi \mid z \in \mathbf{Z}\} = \{z + \pi \mid z \in \mathbf{Z}\} = \{\ldots, 2.14152..., 3.14152..., 4.14152..., \ldots\}.$$

In a similar fashion, the orbit of any real number x under this action of $(\mathbf{Z}, +)$ on $\mathbf{R}$ creates and equivalence relation under the group structure of $(\mathbf{Z}, +)$ such that $x \sim y$ with y in $\mathbf{R}$ exactly when there exists a z in $\mathbf{Z}$ such that $y = z + x$.

The set of all orbits as x ranges over $\mathcal{X}$ is a partition of $\mathcal{X}$ into equivalence classes. The relevance for permutation tests is as follows: Suppose that $\mathcal{X}$ is the outcome space of some experiment, and we will make an observation X drawn from P. Let $(G, \cdot)$ be a group that acts on $\mathcal{X}$. Suppose that under the null hypothesis, all elements of $\mathcal{X}$ in any given orbit under G are equally likely. Then if we observe $X = x$, it is just as likely that we would have observed gx for any $g \in G$: conditional on the orbit Gx, all elements of Gx are equally likely. This then determines the (conditional) distribution of X: every element of Gx is equally likely given $GX = Gx$—even though we do not know $P(X \in Gx)$. The uniform conditional distribution lets us calculate the conditional null distribution of any test statistic.

Theorem 1. *Given a probability space $(\mathcal{X}, \Sigma, \mu)$ and a group $(G, \cdot)$ acting on $\mathcal{X}$ such that Σ is closed under the action of G, let $X \sim P$ for some unknown P dominated by μ. Consider testing the hypothesis that P is invariant under G*

$$H_0 : P(S \subset \mathcal{X}) = P(GS \subset \mathcal{X}) \textit{ for all } S \in \Sigma.$$

Suppose we have a family of tests with (conditional) level no greater than α conditional on the orbit of the observed data

$$P(\textit{reject } H_0 \mid GX = Gx \parallel H_0) \leq \alpha,$$

where the notation "$\parallel H_0$" means "assuming H_0 is true." Then the resulting overall test has unconditional level no greater than α

$$P(\textit{reject } H_0 \parallel H_0) \leq \alpha.$$

Proof. By the law of total probability, we have

$$\begin{aligned}
P(\text{reject } H_0 \parallel H_0) &= \int_{\mathcal{X}} P(\text{reject } H_0 \mid GX = Gx \parallel H_0) P(GX = Gx \parallel H_0) \, \mathrm{d}\mu(x) \\
&\leq \int_{\mathcal{X}} \alpha P(GX = Gx \parallel H_0) \, \mathrm{d}\mu(x) \\
&= \alpha \int_{\mathcal{X}} P(GX = Gx \parallel H_0) \, \mathrm{d}\mu(x) \\
&= \alpha.
\end{aligned}$$

□

Appendix B

Software demonstration

For the following two worked examples, we assume that the following imports have been made.

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

You will also need to have `permute` version `0.1a1` installed.

```
>>> import permute
>>> permute.__version__
'0.1a1'
```

Note that this is an *alpha* release, so we are making no claims regarding the stability of our package structure or call signatures. We do not have plans to break our API, but we are allowing ourselves to continue refining our API as we gain experience using the package and continue adding new functionality. However, the following example sessions will give you an idea of what working with `permute` is like.

Two sample permutation test

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 [9]. 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 the professor used a male name versus a female name.

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

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

For the two-sample t-test, the null hypothesis is that the reported/perceived teacher gender has no effect on teacher ratings. The alternative hypothesis is that teacher ratings differ by reported/perceived teacher 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.

```
>>> from permute.data import macnell2014
>>> ratings = macnell2014()
>>> maleid = ratings.overall[ratings.taidgender==1]
>>> femaleid = ratings.overall[ratings.taidgender==0]
>>> df = len(maleid) + len(femaleid) - 2
>>> t, p = stats.ttest_ind(maleid, femaleid)
>>> print 'Test statistic:', np.round(t, 5)
Test statistic: 1.54059
>>> print 'P-value (two-sided):', np.round(p, 5)
P-value (two-sided): 0.1311
```

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

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 reported/perceived teacher gender has no effect whatsoever on ratings—as if "male" and "female" are randomly assigned labels. The alternative hypothesis is that reported/perceived teacher gender has some effect on ratings. The only assumption we need to make is that randomization is fair and independent across units. This can be verified directly from the experimental design.

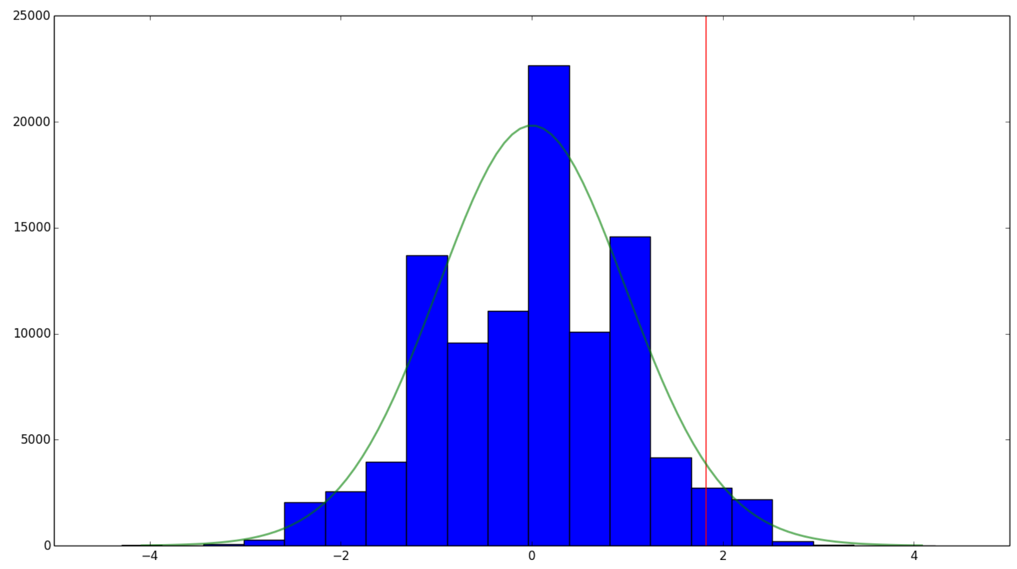


Figure 2: Permutation Null Distribution.

```
>>> from permute.core import two_sample
>>> p, t, ci, dist = two_sample(maleid, femaleid, stat='t',
...                                 interval='two-sided', keep_dist=True)
>>> print 'Test statistic:', np.round(t, 5)
Test statistic: 1.82159
>>> print 'P-value (two-sided):', np.round(p, 5)
P-value (two-sided): 0.04436
>>> print '95% Confidence Interval for the P-value', np.round(ci, 5)
95% Confidence Interval for the P-value [ 0.04309  0.04565]
```

Not only is the use of this p-value justified (since our assumptions are met), but its value is below the cut-off for significance commonly used. Since the permutation test also returns the approximately exact distribution of the test statistic, let's compare the actual distribution with the t-distribution.

```
>>> n, bins, patches = plt.hist(dist, 20, histtype='bar')
>>> plt.axvline(x = t, color = 'red')
>>> x = np.linspace(stats.t.ppf(0.0001, df),
                    stats.t.ppf(0.9999, df), 100)
>>> plt.plot(x, stats.t.pdf(x, df)*(100000/2.0), lw=2, alpha=0.6, label='t pdf')
>>> plt.show()
```

In Figure 2, you can see the simulated distribution of the test statistic under the null with the t-distribution superimposed on it.

Stratified Spearman correlation permutation test

To illustrate the use of one of our stratified tests, I will use fake data similar to that used in [2]; we are unable to use the actual data due to data privacy laws in France.

The scientific question of interest is now "Do students' evaluations of teachers measure teaching effectiveness?" To answer this question, we were able use final exam performance as a proxy for value added by teacher—students have different teachers but take the same final exam. Our null hypothesis is that there is no association between a student's final exam performance and the rating they give their professor. The alternative hypothesis is that there is a positive association between a student's final exam performance and the rating they give their professor.

There are 8 professors, each with multiple student ratings. One might think that a certain professor gets consistently high reviews because of something (e.g., gender, attractiveness, likeability) unrelated to teaching effectiveness. Stratifying by professor allows us to assess teaching effectiveness for each individual student. The test statistic we use within groups is the Spearman correlation.

```
>>> from permute.stratified import sim_corr
>>> evals = np.recfromcsv("SET2.csv")
>>> rho, plower, pupper, pboth, sim = sim_corr(x=evals.rating, y=evals.final,
...                                            group=evals.prof_id)
>>> print 'Test statistic:', np.round(rho, 5)
Test statistic: 0.94787
>>> print 'One-sided (upper) P-value:', np.round(pupper, 5)
One-sided (upper) P-value: 0.18
```

Finally, I plot the simulated distribution of the test statistics under the null conditioned on the observed data in Figure 3.

```
>>> n, bins, patches = plt.hist(sim, 40, histtype='bar')
>>> plt.axvline(x=rho, color='red')
>>> plt.show()
```

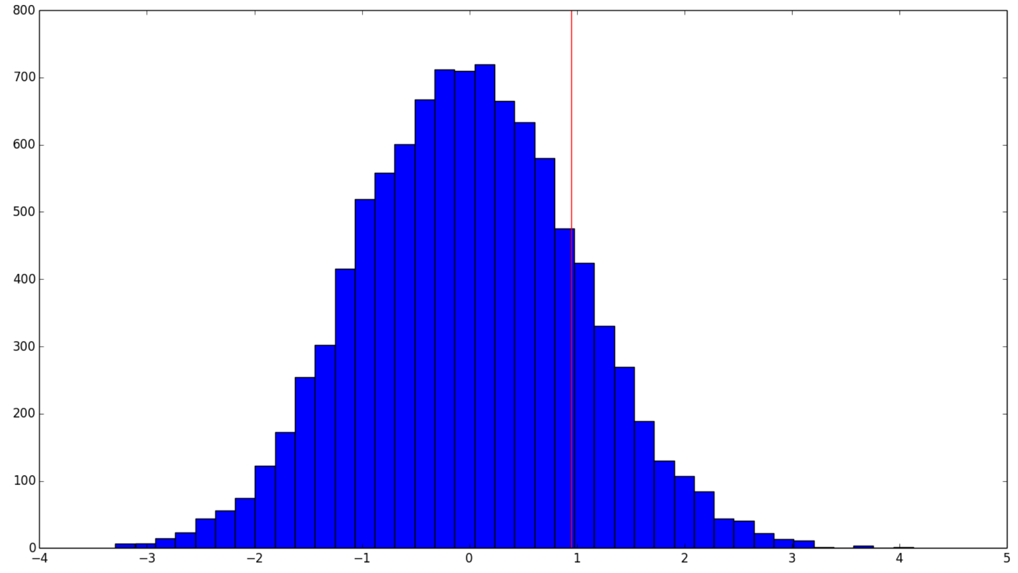


Figure 3: Permutation Null Distribution.