mcpt

Release 0.1.2

Sep 09, 2019

Contents:

`mcpt` is a Python 3 library for calculating p-values through Monte Carlo permutation tests, providing an intuitive, simple, and highly customisable interface to determining statistical significance.

To get started, we recommend you work through Installation, Quickstart, and Functions pages. Also check out the FAQ, which we update regularly. If you have concerns about the software, or feel that there is something that should be more explicit, then we'd love to hear from you – please open an issue on Github and we'll get back in touch ASAP.

If you use `mcpt` in your research, please support us by citing the initial release:

> David J. Skelton. (2019, September 5). mcpt: Monte Carlo permutation tests for Python (Version 0). Zenodo. http://doi.org/10.5281/zenodo.3387528

CHAPTER 1

Installation

`mcpt` is available on PyPI, and can be installed with its dependencies using `pip`

```
pip install mcpt
```

CHAPTER 2

Quickstart

`mcpt` comes with two main functions: `permutation_test` and `correlation_permutation_test`.

2.1 Permutation test

This function takes values for two groups, `X` and `Y`, and tests whether the labels on the members of the group are exchangable under the null hypothesis.

For simplest use, the `permutation_test` function takes four arguments:

- `x` - An iterable of values for members of the 1st group.
- `y` - An iterable of values for members of the 2nd group.
- `side` - The side that we want to test under the null hypothesis
- `f` - The function for calculating the test statistic

```
>>> import mcpt
>>> treatment = [10, 9, 11]
>>> control = [12, 11, 13]
>>> side = "lower"
>>> f = "mean"

>>> result = mcpt.permutation_test(treatment, control, f=f, side=side)
>>> print(result)
Result(lower=0.09815650454064283, upper=0.10305649415095638, confidence=0.99)
```

In the above example, we are evaluating whether the mean of the samples in the `treatment` group is signifcantly lower than the mean of the samples in `control` group.

Unlike an exhaustive permutation test, `mcpt` samples from the distribution of all possible permutations. Thus, the p-value obtained by `mcpt` is *approximate*, and will vary run-to-run; rather than returning the approximate p-value, therefore, `mcpt` returns upper and lower bounds that, with a given confidence, contains the true p-value. The default confidence is 99%; see this link for more information on approximating p-values.

In the above example, there is a 99% probability that the p-value is between 0.098 and 0.103 – thus, at an alpha of 0.05, we cannot reject the null hypothesis.

`f` takes a string value (`"mean"`, `"median"`, or `"stdev"`) – alternatively, a function can be passed (e.g. `"numpy.mean"`).

`side` takes one of three values: `"both"` for a two-sided permutation test, or `"greater"` and `"lower"` for one-sided permutation tests.

For more advanced usage, see *Permutation test*.

2.2 Correlation permutation test

This function takes a set of paired scores, $(x_1, y_1), (x_2, y_2)...(x_i, y_i)$, and tests whether the pairings are exchangable under the null hypothesis.

For simplest use, the `correlaton_permutation_test` function takes four arguments:

- `x` - An iterable of the x values for the pairs.
- `y` - An iterable of the y values for the pairs.
- `side` - The side that we want to test under the null hypothesis
- `f` - The function for calculating the test statistic

```
>>> import mcpt
>>> x = [-2.31, 1.06, 0.76, 1.38, -0.26, 1.29, -1.31, 0.41, -0.67, -0.58]
>>> y = [-1.08, 1.03, 0.90, 0.24, -0.24, 0.76, -0.57, -0.05, -1.28, 1.04]
>>> side = "both"
>>> f = "pearsonr"

>>> result = mcpt.correlation_permutation_test(x, y, f=f, side=side)
>>> print(result)
Result(lower=0.021282451892029475, upper=0.029347445354757373, confidence=0.99)
```

In the above example, we determine that there is a 99% probability that the p-value is between 0.021 and 0.029. If we set an alpha value of 0.05, then it would be reasonable to reject the null hypothesis that the correlation is significantly different from $\rho = 0$.

`f` takes a string value (`"pearsonr"` or `"spearmanr"`) – alternatively, a function can be passed.

`side` takes one of three values: `"both"` for a two-sided permutation test, or `"greater"` and `"lower"` for one-sided permutation tests.

For more advanced usage, see *Correlation permutation test*.

CHAPTER 3

Functions

In this section, more advanced usage of the two main functions will be discussed:

- `mcpt.permutation_test`
- `mcpt.correlation_permation_test`

3.1 Common features

There are a number of advanced uses of the functions discussed here that are common to both functions.

3.1.1 Setting the number of permutations

The number of random permutations to be used is set by specifying the `n` parameter when calling either function. By default this value is `10,000`; increasing this value is one approach to narrowing the range returned (if required).

3.1.2 Multiprocessing

In order to speed up calculations, both functions are configured to take a `cores` parameter. This determines the number of (logical) CPUs to be used for permuting; `multiprocessing.Pool` is used to create a pool of workers, and the permutation calculations farmed across these workers. **Note that the default is to use a single core (i.e., not multiprocessed).**

3.1.3 Seeding

To allow reproducibility of results, `mcpt` also implemented seeding, and can be set by passing the `seed` parameter to either function. By default, `seed=None`, meaning that two runs of `mcpt` will *not* give exactly the same answer unless a seed is explicitly passed. Seeding works in both single core and multi-core versions.

If a seed is given, this is used to seed a random number generator (`random.Random`), which in turn is used to generate random integers in $[0, 1e100)$ to seed the randomisation in each permutation. If no seed is given, then the initial seed is random.

3.1.4 Confidence

A Monte Carlo permutation test has a random factor and, thus, the approximate p-value returned differs from run-to-run. However, based on the number of permutations and the approximate p-value returned, it is possible to calculate a range in which, with a certain confidence, we can say the true p-value lies in. There are a number of ways to achieve this, but `mcpt` uses the Wilson score interval binomial approximation approach. The confidence can be set by passing the `confidence` parameter, where confidence is a float – for example, passing `confidence = 0.999` means that there is a $1/1000$ chance that the true p-value lies outside the interval returned.

3.2 Permutation test

There are two main advanced uses to consider with the `mcpt.permutation_test` function. The first is passing a custom function, and the second is using multi-dimensional inputs.

3.2.1 Using a custom function

Normally, `x` and `y` passed to the function are one dimensional iterables (e.g., a `list`) of values. For this section, we will assume this is the case, but the next section will consider the multi-dimensional situation.

Let's say rather than standard deviation, we want to consider the interquartile range (IQR). We can define a function that can take a list and calculate / return the value of the test statistic (IQR in this case) to use with `mcpt.permutation_test`

```
import numpy as np

def iqr(x):
    q3 = np.quantile(x, .75)
    q1 = np.quantile(x, .25)
    return q3 - q1
```

We can now call `mcpt.permutation_test` with `f=iqr`. In the below example we create `x` and `y` by sampling from a normal distribution with `mean = 0.5, std=0.5` and `mean = 0.5, std=1.5` respectively. We would expect `x` to have a lower IQR than `y`, and can test this as follows:

```
import mcpt

# Generate x
x = [np.random.normal(0.5, 0.5) for _ in range(100)]
# Generate y
y = [np.random.normal(0.5, 1.5) for _ in range(100)]
# Run the permutation test
result = mcpt.permutation_test(x, y, side="lower", f=iqr)
print(result)
```

The result may vary from run-to-run (unseeded), but the result obtained should be something around `Result(lower=0.0, upper=0.0006630497334598373, confidence=0.99)`; unsurprisingly, a statistically significant difference was detected.

Of course, `scipy` has an interquartile range function, and this can be used directly instead of defining a custom function.

```
from scipy import stats as _st
result = mcpt.permutation_test(x, y, side="lower", f=_st.iqr)
print(result)
```

Which gives (subject to randomness) approximately the same answer. Thus, to summarize, `mcpt.permutation_test` gives a flexible interface for hypothesis testing.

3.2.2 Multi-dimensional inputs

It may be the case that, rather than having as single value for the members of your group, you have multiple values. An example use-case for this would be where you have a pair of values $(x_1, y_1), (x_2, y_2)...(x_i, y_i)$ for two groups, *A* and *B*, and you wish to test whether the correlation between *x* and *y* differs between *A* and *B*.

`mcpt.permutation_test` puts no restrictions on what `x` and `y` look like, and so you can define custom functions that are aware of this shape. The below example shows this for considering whether there is a statistically significant difference in the correlation of variables in two groups (and fails to reject the null hypothesis).

```
import mcpt

from scipy.stats import pearsonr

x = [(10, 8), (9, 6), (8, 9), (2, 4), (5, 3), (3, 10), (3, 4), (8,10)]
y = [(1, 9), (10, 4), (2, 3), (2, 8), (5, 9), (7, 2), (5, 5), (1, 4)]

def pearson_correlation(a):
        x_vals = [i[0] for i in a]
        y_vals = [i[1] for i in a]

        return pearsonr(x_vals, y_vals)[0]

result = mcpt.permutation_test(x, y, side="both", f=pearson_correlation)
print(result)
# Result(lower=0.11605000822777765, upper=0.13304820734194409, confidence=0.99)
```

3.3 Correlation permutation test

Similar to `mcpt.permutation_test`, this `mcpt.correlation_permutation_test` can accept a custom function where a custom test statistic is being calculated. We anticipate that these will most often be either `spearmanr` or `pearsonr`. However, other correlation measures exist (e.g., Kendall's tau).

The key difference in the implementation of custom functions for `mcpt.correlation_permutation_test` is that we expect to be able to pass two variables to it – `x` and `y`.

```
import mcpt

from scipy.stats import kendalltau

x = [4.02, 4.52, 4.79, 4.89, 5.27, 5.63, 5.89, 6.08, 6.13, 6.19, 6.47]
y = [4.56, 2.92, 2.71, 3.34, 3.53, 3.47, 3.20, 4.51, 3.76, 3.77, 4.03]

def ktau(x, y):
```

(continues on next page)

(continued from previous page)

```
        tau, _ = kendalltau(x, y)
        return tau

result = mcpt.correlation_permutation_test(x, y, side="both", f=ktau)
print(result)
# Result(lower=0.1594695442150737, upper=0.17876952731842338, confidence=0.99)
```

The above example is from here, where the true value was found to be 0.1646.

CHAPTER 4

FAQ

4.1 How does mcpt compare to the mlxtend implementation?

This is a fair question, as a Google search for "permutation test Python" brings `mlxtend`'s implementation up. I would like to start by saying that `mlxtend` is a great package, which I've used on a number of projects. However, there are a few reasons I would prefer `mcpt` over `mlxtend` for permutation testing.

Firstly, I have a couple of concerns with the implementation of permutation test in `mlxtend`.

1. In the source code for the latest release (`9c044a9` at the time of writing), it appears that the p-value returned is the probability of getting a more extreme ($>$) result by chance. However, p-value should actually be the probability of getting a result at least as extreme ($\geq$).
2. `mlxtend` uses a single function for p-values in both correlations *and* comparing a test statistic in two groups. This is problematic, because the treatment of `x` and `y` is different in the two tests. In the correlation case, only `y` should be permuted to create different pairs. However, `mlxtend` pools `x` and `y` together and randomizes both, meaning that new pairs such as (x_1, x_2) are possible, which should not be the case.
3. Permutation tests from Monte Carlo sampling, due to randomisation, results in approximations of the p-value, which differ from run-to-run. It would be incorrect to state that an approximate p-value of 0.049 from one run is significant with `alpha = 0.05`, as randomness may make this value differ run-to-run. What is better is to return a confidence interval, and conclude significance if $p_{upper} < 0.05$ at a confidence that we're satisfied with (e.g. 99.9%).
4. `mlxtend.evalue.permutation_test` uses combinations for `method='exact'` calculations. This does not work for correlations, because the order matters for correlation (i.e., which `x` is paired with which `y`). For this reason, the result obtained for example 2 in the documentation is actually incorrect.

Test-driven development for `mcpt` means that we test our implementation against a number of use-cases before release.

The second set of reasons I would prefer `mcpt` over `mlxtend` are quality-of-life based.

1. We implement multiprocessing, allowing the use of multiple processors if desired and available.
2. Subjectively, we believe the combination of our documentation and implementation is more intuitive, flexible, and simple.

3. The use of confidence intervals rather than the approximate p-value from randomisation testing is i) easier to justify and ii) more scientifically sound. For example, in a study, you can report:

   > "We used `mcpt` to calculate approximate p-values using a Monte Carlo permutation test. A result was deemed to be significant if the upper bound of a 99.9% confidence interval was < 0.05."

CHAPTER 5

Indices and tables

- genindex
- modindex
- search