mgc Documentation

Release alpha

Sambit Panda

Dec 07, 2019

Contents

`mgc` (pronounced "Magic") is an open-source software package for independence and k-sample testing.

CHAPTER 1

Motivation

With the increase in the amount of data in many fields, a method to consistently and efficiently decipher relationships within high dimensional data sets is important. Because many modern datasets are multivariate, univariate independence tests are not applicable. While many multivariate independence tests have R packages available, the interfaces are inconsistent and most are not available in Python. `mgc` is an extensive Python library that includes many state of the art multivariate independence testing procedures using a common interface. The package is easy-to-use and is flexible enough to enable future extensions.

CHAPTER 2

Python

Python is a powerful programming language that allows concise expressions of network algorithms. Python has a vibrant and growing ecosystem of packages that mgc uses to provide more features such as numerical linear algebra and plotting. In order to make the most out of `mgc` you will want to know how to write basic programs in Python. Among the many guides to Python, we recommend the Python documentation.

CHAPTER 3

Free software

`mgc` is free software; you can redistribute it and/or modify it under the terms of the *MIT*. We welcome contributions. Join us on GitHub.

CHAPTER 4

History

`mgc` is a rebranding of `mgcpy`, which was founded in September 2018. The original version was designed and written by Satish Palaniappan, Sambit Panda Junhao Xiong, Sandhya Ramachandran, and Ronak Mehtra. This new version was written by Sambit Panda.

CHAPTER 5

Documentation

5.1 Install

Below we assume you have the default Python environment already configured on your computer and you intend to install `mgc` inside of it. If you want to create and work with Python virtual environments, please follow instructions on venv and virtual environments. We also highly recommend conda. For instructions to install this, please look at conda.

First, make sure you have the latest version of `pip` (the Python package manager) installed. If you do not, refer to the Pip documentation and install `pip` first.

5.1.1 Install from Github

You can manually download `mgc` by cloning the git repo master version and running the `setup.py` file. That is, unzip the compressed package folder and run the following from the top-level source directory using the Terminal:

```
$ git clone https://github.com/neurodata/mgc
$ cd mgc
$ python3 setup.py install
```

5.1.2 Python package dependencies

mgc requires the following packages:

- numba
- numpy
- scipy
- scikit-learn

5.1.3 Hardware requirements

`mgc` package requires only a standard computer with enough RAM to support the in-memory operations.

5.1.4 OS Requirements

This package is supported for all major operating systems. The following versions of operating systems was tested on Travis CI:

- **Linux:** Ubuntu Xenial 16.04
- **Windows:** Windows Server, version 1803

5.1.5 Testing

`mgc` uses the Python `pytest` testing package. If you don't already have that package installed, follow the directions on the pytest homepage.

5.2 Reference

5.2.1 Independence

Distance Correlation (Dcorr)

class `mgc.independence.`**Dcorr**(*compute_distance=<function euclidean>*)

Class for calculating the Dcorr test statistic and p-value.

Dcorr is a measure of dependence between two paired random matrices of not necessarily equal dimensions. The coefficient is 0 if and only if the matrices are independent. It is an example of an energy distance.

Parameters **compute_distance** : callable(), optional (default: euclidean)

A function that computes the distance among the samples within each data matrix. Set to *None* if *x* and *y* are already distance matrices. To call a custom function, either create the distance matrix before-hand or create a function of the form `compute_distance(x)` where *x* is the data matrix for which pairwise distances are calculated.

See also:

Hsic Hilbert-Schmidt independence criterion test statistic and p-value.

HHG Heller Heller Gorfine test statistic and p-value.

Notes

The statistic can be derived as follows:

Let x and y be (n, p) samples of random variables X and Y. Let D^x be the $n \times n$ distance matrix of x and D^y be the $n \times n$ be the distance matrix of y. The distance covariance is,

$$\mathrm{Dcov}_n(x, y) = \frac{1}{n^2}\mathrm{tr}(D^x H D^y H)$$

where $\mathrm{tr}(\cdot)$ is the trace operator and H is defined as $H = I - (1/n)J$ where I is the identity matrix and J is a matrix of ones. The normalized version of this covariance is Dcorr[1] and is

$$\mathrm{Dcorr}_n(x, y) = \frac{\mathrm{Dcov}_n(x, y)}{\sqrt{\mathrm{Dcov}_n(x, x)\mathrm{Dcov}_n(y, y)}}$$

This version of distance correlation is defined using the following centering process where $\mathbb{1}(\cdot)$ is the indicator function:

$$C^x_{ij} = \left[D^x_{ij} - \frac{1}{n-2}\sum_{t=1}^{n} D^x_{it} - \frac{1}{n-2}\sum_{s=1}^{n} D^x_{sj} + \frac{1}{(n-1)(n-2)}\sum_{s,t=1}^{n} D^x_{st} \right] \mathbb{1}_{i \neq j}$$

and similarly for C^y. Then, this unbiased Dcorr is,

$$\mathrm{UDcov}_n(x, y) = \frac{1}{n(n-3)}\mathrm{tr}(C^x C^y)$$

The normalized version of this covariance[2] is

$$\mathrm{UDcorr}_n(x, y) = \frac{\mathrm{UDcov}_n(x, y)}{\sqrt{\mathrm{UDcov}_n(x, x)\mathrm{UDcov}_n(y, y)}}$$

References

test (*x*, *y*, *reps=1000*, *workers=1*, *random_state=None*)

Calculates the Dcorr test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples. That is, the shapes must be *(n, p)* and *(n, q)* where *n* is the number of samples and *p* and *q* are the number of dimensions. Alternatively, *x* and *y* can be distance matrices, where the shapes must both be *(n, n)*.

reps : int, optional (default: 1000)

The number of replications used to estimate the null distribution when using the permutation test used to calculate the p-value.

workers : int, optional (default: 1)

The number of cores to parallelize the p-value computation over. Supply -1 to use all cores available to the Process.

random_state : int or np.random.RandomState instance, (default: None)

If already a RandomState instance, use it. If seed is an int, return a new RandomState instance seeded with seed. If None, use np.random.RandomState.

Returns **stat** : float

The computed Dcorr statistic.

pvalue : float

The computed Dcorr p-value.

[1] Székely, G. J., Rizzo, M. L., & Bakirov, N. K. (2007). Measuring and testing dependence by correlation of distances. *The annals of statistics*, 35(6), 2769-2794.

[2] Székely, G. J., & Rizzo, M. L. (2014). Partial distance correlation with methods for dissimilarities. *The Annals of Statistics*, 42(6), 2382-2412.

Examples

```
>>> import numpy as np
>>> from mgc.independence import Dcorr
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = Dcorr().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

The number of replications can give p-values with higher confidence (greater alpha levels).

```
>>> import numpy as np
>>> from mgc.independence import Dcorr
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = Dcorr().test(x, y, reps=10000)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

In addition, the inputs can be distance matrices. Using this is the, same as before, except the `compute_distance` parameter must be set to `None`.

```
>>> import numpy as np
>>> from mgc.independence import Dcorr
>>> x = np.ones((10, 10)) - np.identity(10)
>>> y = 2 * x
>>> dcorr = Dcorr(compute_distance=None)
>>> stat, pvalue = dcorr.test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'0.0, 1.00'
```

Hilbert Schmidt Independence Criterion (Hsic)

class mgc.independence.**Hsic**(*compute_kernel=<function gaussian>*)

Class for calculating the Hsic test statistic and p-value.

Hsic is a kernel based independence test and is a way to measure multivariate nonlinear associations given a specified kernel[3]. The default choice is the Gaussian kernel, which uses the median distance as the bandwidth, which is a characteristic kernel that guarantees that Hsic is a consistent test[3][4].

Parameters **compute_kernel** : callable(), optional (default: rbf kernel)

A function that computes the similarity among the samples within each data matrix. Set to *None* if *x* and *y* are already similarity matrices. To call a custom function, either create the distance matrix before-hand or create a function of the form `compute_kernel(x)` where *x* is the data matrix for which pairwise similarties are calculated.

See also:

Dcorr Distance correlation test statistic and p-value.

HHG Heller Heller Gorfine test statistic and p-value.

[3] Gretton, A., Fukumizu, K., Teo, C. H., Song, L., Schölkopf, B., & Smola, A. J. (2008). A kernel statistical test of independence. In *Advances in neural information processing systems* (pp. 585-592).

[4] Gretton, A., & GyĂśrfi, L. (2010). Consistent nonparametric tests of independence. *Journal of Machine Learning Research*, 11(Apr), 1391-1423.

Notes

The statistic can be derived as follows[3]:

Let x and y be (n, p) samples of random variables X and Y. Let K^x be the $n \times n$ kernel similarity matrix of x and D^y be the $n \times n$ be the kernel similarity matrix of y. The Hsic statistic is,

$$\text{Hsic}_n(x, y) = \frac{1}{n^2}\text{tr}(K^x H K^y H)$$

where $\text{tr}(\cdot)$ is the trace operator and H is defined as $H = I - (1/n)J$ where I is the identity matrix and J is a matrix of ones. The normalized version of Hsic[1] and is

$$\text{Hsic}_n(x, y) = \frac{\text{Hsic}_n(x, y)}{\sqrt{\text{Hsic}_n(x, x)\text{Hsic}_n(y, y)}}$$

This version of Hsic is defined using the following centering process where $\mathbb{1}(\cdot)$ is the indicator function:

$$C^x_{ij} = \left[D^x_{ij} - \frac{1}{n-2}\sum_{t=1}^{n} D^x_{it} - \frac{1}{n-2}\sum_{s=1}^{n} D^x_{sj} + \frac{1}{(n-1)(n-2)}\sum_{s,t=1}^{n} D^x_{st}\right]\mathbb{1}_{i \neq j}$$

and similarly for C^y. Then, this unbiased Dcorr is,

$$\text{UHsic}_n(x, y) = \frac{1}{n(n-3)}\text{tr}(C^x C^y)$$

The normalized version of this covariance[2] is

$$\text{UHsic}_n(x, y) = \frac{\text{UHsic}_n(x, y)}{\sqrt{\text{UHsic}_n(x, x)\text{UHsic}_n(y, y)}}$$

References

test (*x*, *y*, *reps=1000*, *workers=1*, *random_state=None*)

Calculates the Hsic test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples. That is, the shapes must be *(n, p)* and *(n, q)* where *n* is the number of samples and *p* and *q* are the number of dimensions. Alternatively, *x* and *y* can be distance matrices, where the shapes must both be *(n, n)*.

reps : int, optional (default: 1000)

The number of replications used to estimate the null distribution when using the permutation test used to calculate the p-value.

workers : int, optional (default: 1)

The number of cores to parallelize the p-value computation over. Supply -1 to use all cores available to the Process.

random_state : int or np.random.RandomState instance, (default: None)

If already a RandomState instance, use it. If seed is an int, return a new RandomState instance seeded with seed. If None, use np.random.RandomState.

Returns **stat** : float

The computed Hsic statistic.

pvalue : float

The computed Hsic p-value.

Examples

```
>>> import numpy as np
>>> from mgc.independence import Hsic
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = Hsic().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

The number of replications can give p-values with higher confidence (greater alpha levels).

```
>>> import numpy as np
>>> from mgc.independence import Hsic
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = Hsic().test(x, y, reps=10000)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

In addition, the inputs can be distance matrices. Using this is the, same as before, except the `compute_kernel` parameter must be set to `None`.

```
>>> import numpy as np
>>> from mgc.independence import Hsic
>>> x = np.ones((10, 10)) - np.identity(10)
>>> y = 2 * x
>>> hsic = Hsic(compute_kernel=None)
>>> stat, pvalue = hsic.test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'0.0, 1.00'
```

Heller Heller Gorfine (HHG)

class mgc.independence.**HHG**(*compute_distance=<function euclidean>*)

Class for calculating the HHG test statistic and p-value.

This is a powerful test for independence based on calculating pairwise euclidean distances and associations between these distance matrices. The test statistic is a function of ranks of these distances, and is consistent against similar tests[5]. It can also operate on multiple dimensions[5].

Parameters **compute_distance** : callable(), optional (default: euclidean)

A function that computes the distance among the samples within each data matrix. Set to *None* if *x* and *y* are already distance matrices. To call a custom function, either create the distance matrix before-hand or create a function of the form `compute_distance(x)` where *x* is the data matrix for which pairwise distances are calculated.

See also:

Dcorr Distance correlation test statistic and p-value.

Hsic Hilbert-Schmidt independence criterion test statistic and p-value.

[5] Heller, R., Heller, Y., & Gorfine, M. (2012). A consistent multivariate test of association based on ranks of distances. *Biometrika*, 100(2), 503-510.

Notes

The statistic can be derived as follows[5]:

Let x and y be (n, p) samples of random variables X and Y. For every sample $j \neq i$, calculate the pairwise distances in x and y and denote this as $d_x(x_i, x_j)$ and $d_y(y_i, y_j)$. The indicator function is denoted as $\mathbb{1}\{\cdot\}$. The cross-classification between these two random variables can be calculated as

$$A_{11} = \sum_{k=1, k\neq i,j}^{n} \mathbb{1}\{d_x(x_i, x_k) \leq d_x(x_i, x_j)\}\mathbb{1}\{d_y(y_i, y_k) \leq d_y(y_i, y_j)\}$$

and A_{12}, A_{21}, and A_{22} are defined similarly. This is organized within the following table:

	$d_x(x_i, \cdot) \leq d_x(x_i, x_j)$	$d_x(x_i, \cdot) \leq d_x(x_i, x_j)$	
$d_x(x_i, \cdot) \leq d_x(x_i, x_j)$	$A_{11}(i, j)$	$A_{12}(i, j)$	$A_{1\cdot}(i, j)$
$d_x(x_i, \cdot) > d_x(x_i, x_j)$	$A_{21}(i, j)$	$A_{22}(i, j)$	$A_{2\cdot}(i, j)$
	$A_{\cdot 1}(i, j)$	$A_{\cdot 2}(i, j)$	$n - 2$

Here, $A_{\cdot 1}$ and $A_{\cdot 2}$ are the column sums, $A_{1\cdot}$ and $A_{2\cdot}$ are the row sums, and $n - 2$ is the number of degrees of freedom. From this table, we can calculate the Pearson's chi squared test statistic using,

$$S(i, j) = \frac{(n-2)(A_{12}A_{21} - A_{11}A_{22})^2}{A_{1\cdot}A_{2\cdot}A_{\cdot 1}A_{\cdot 2}}$$

and the HHG test statistic is then,

$$\text{HHG}_n(x, y) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} S(i, j)$$

References

test(*x*, *y*, *reps=1000*, *workers=1*, *random_state=None*)

Calculates the HHG test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples. That is, the shapes must be *(n, p)* and *(n, q)* where *n* is the number of samples and *p* and *q* are the number of dimensions. Alternatively, *x* and *y* can be distance matrices, where the shapes must both be *(n, n)*.

reps : int, optional (default: 1000)

The number of replications used to estimate the null distribution when using the permutation test used to calculate the p-value.

workers : int, optional (default: 1)

The number of cores to parallelize the p-value computation over. Supply -1 to use all cores available to the Process.

random_state : int or np.random.RandomState instance, (default: None)

If already a RandomState instance, use it. If seed is an int, return a new RandomState instance seeded with seed. If None, use np.random.RandomState.

Returns **stat** : float

The computed HHG statistic.

pvalue : float

The computed HHG p-value.

Examples

```
>>> import numpy as np
>>> from mgc.independence import HHG
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = HHG().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'160.0, 0.00'
```

The number of replications can give p-values with higher confidence (greater alpha levels).

```
>>> import numpy as np
>>> from mgc.independence import HHG
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = HHG().test(x, y, reps=10000)
>>> '%.1f, %.2f' % (stat, pvalue)
'160.0, 0.00'
```

In addition, the inputs can be distance matrices. Using this is the, same as before, except the `compute_distance` parameter must be set to `None`.

```
>>> import numpy as np
>>> from mgc.independence import HHG
>>> x = np.ones((10, 10)) - np.identity(10)
>>> y = 2 * x
>>> hhg = HHG(compute_distance=None)
>>> stat, pvalue = hhg.test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'0.0, 1.00'
```

Cannonical Correlation Analysis (CCA)

class mgc.independence.**CCA**

Class for calculating the CCA test statistic and p-value.

This test can be thought of inferring information from cross-covariance matrices[6]. It has been thought that virtually all parametric tests of significance can be treated as a special case of CCA[7]. The method was first introduced by Harold Hotelling in 1936[8].

See also:

Pearson Pearson product-moment correlation test statistic and p-value.

RV RV test statistic and p-value.

[6] Härdle, W. K., & Simar, L. (2015). Canonical correlation analysis. In *Applied multivariate statistical analysis* (pp. 443-454). Springer, Berlin, Heidelberg.

[7] Knapp, T. R. (1978). Canonical correlation analysis: A general parametric significance-testing system. *Psychological Bulletin*, 85(2), 410.

[8] Hotelling, H. (1992). Relations between two sets of variates. In *Breakthroughs in statistics* (pp. 162-190). Springer, New York, NY.

Notes

The statistic can be derived as follows[9]:

Let x and y be *:math:'(n, p)* samples of random variables X and Y. We can center x and y and then calculate the sample covariance matrix $\hat{\Sigma}_{xy} = x^T y$ and the variance matrices for x and y are defined similarly. Then, the CCA test statistic is found by calculating vectors $a \in \mathbb{R}^p$ and $b \in \mathbb{R}^q$ that maximize

$$\text{CCA}_n(x, y) = \max_{a \in \mathbb{R}^p, b \in \mathbb{R}^q} \frac{a^T \hat{\Sigma}_{xy} b}{\sqrt{a^T \hat{\Sigma}_{xx} a} \sqrt{b^T \hat{\Sigma}_{yy} b}}$$

References

test (*x*, *y*, *reps=1000*, *workers=1*, *random_state=None*)

Calculates the CCA test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples and dimensions. That is, the shapes must be *(n, p)* where *n* is the number of samples and *p* is the number of dimensions.

reps : int, optional (default: 1000)

The number of replications used to estimate the null distribution when using the permutation test used to calculate the p-value.

workers : int, optional (default: 1)

The number of cores to parallelize the p-value computation over. Supply -1 to use all cores available to the Process.

random_state : int or np.random.RandomState instance, (default: None)

If already a RandomState instance, use it. If seed is an int, return a new RandomState instance seeded with seed. If None, use np.random.RandomState.

Returns **stat** : float

The computed CCA statistic.

pvalue : float

The computed CCA p-value.

Examples

```
>>> import numpy as np
>>> from mgc.independence import CCA
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = CCA().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

The number of replications can give p-values with higher confidence (greater alpha levels).

[9] Hardoon, D. R., Szedmak, S., & Shawe-Taylor, J. (2004). Canonical correlation analysis: An overview with application to learning methods. *Neural computation*, 16(12), 2639-2664.

```
>>> import numpy as np
>>> from mgc.independence import CCA
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = CCA().test(x, y, reps=10000)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

RV

class mgc.independence.**RV**

Class for calculating the RV test statistic and p-value.

RV is the multivariate generalization of the squared Pearson correlation coefficient[10]. The RV coefficient can be thought to be closely related to principal component analysis (PCA), canonical correlation analysis (CCA), multivariate regression, and statistical classification[10].

See also:

Pearson Pearson product-moment correlation test statistic and p-value.

CCA CCA test statistic and p-value.

Notes

The statistic can be derived as follows[10][11]:

Let x and y be (n, p) samples of random variables X and Y. We can center x and y and then calculate the sample covariance matrix $\hat{\Sigma}_{xy} = x^T y$ and the variance matrices for x and y are defined similarly. Then, the RV test statistic is found by calculating

$$\mathrm{RV}_n(x, y) = \frac{\mathrm{tr}\left(\hat{\Sigma}_{xy}\hat{\Sigma}_{yx}\right)}{\mathrm{tr}\left(\hat{\Sigma}_{xx}^2\right)\mathrm{tr}\left(\hat{\Sigma}_{yy}^2\right)}$$

where $\mathrm{tr}(\cdot)$ is the trace operator.

References

test(*x*, *y*, *reps=1000*, *workers=1*, *random_state=None*)

Calculates the RV test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples and dimensions. That is, the shapes must be *(n, p)* where *n* is the number of samples and *p* is the number of dimensions.

reps : int, optional (default: 1000)

The number of replications used to estimate the null distribution when using the permutation test used to calculate the p-value.

[10] Robert, P., & Escoufier, Y. (1976). A unifying tool for linear multivariate statistical methods: the RV-coefficient. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 25(3), 257-265.

[11] Escoufier, Y. (1973). Le traitement des variables vectorielles. *Biometrics*, 751-760.

workers : int, optional (default: 1)

The number of cores to parallelize the p-value computation over. Supply -1 to use all cores available to the Process.

random_state : int or np.random.RandomState instance, (default: None)

If already a RandomState instance, use it. If seed is an int, return a new RandomState instance seeded with seed. If None, use np.random.RandomState.

Returns **stat** : float

The computed RV statistic.

pvalue : float

The computed RV p-value.

Examples

```
>>> import numpy as np
>>> from mgc.independence import RV
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = RV().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

The number of replications can give p-values with higher confidence (greater alpha levels).

```
>>> import numpy as np
>>> from mgc.independence import RV
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = RV().test(x, y, reps=10000)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

Pearson

class mgc.independence.**Pearson**

Class for calculating the Pearson test statistic and p-value.

Pearson product-moment correlation coefficient is a measure of the linear correlation between two random variables[12]. It has a value between +1 and -1 where 1 is the total positive linear correlation, 0 is not linear correlation, and -1 is total negative correlation.

See also:

RV RV test statistic and p-value.

CCA CCA test statistic and p-value.

Spearman Spearman's rho test statistic and p-value.

Kendall Kendall's tau test statistic and p-value.

[12] Pearson, K. (1895). VII. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58(347-352), 240-242.

Notes

This class is a wrapper of scipy.stats.pearsonr. The statistic can be derived as follows[12]:

Let x and y be $(n, 1)$ samples of random variables X and Y. Let $\hat{\text{cov}}(x, y)$ is the sample covariance, and $\hat{\sigma}_x$ and $\hat{\sigma}_y$ are the sample variances for x and y. Then, the Pearson's correlation coefficient is,

$$\text{Pearson}_n(x, y) = \frac{\hat{\text{cov}}(x, y)}{\hat{\sigma}_x \hat{\sigma}_y}$$

References

test (x, y)

Calculates the Pearson test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples and dimensions. That is, the shapes must be *(n, 1)* where *n* is the number of samples.

Returns **stat** : float

The computed Pearson statistic.

pvalue : float

The computed Pearson p-value.

Examples

```
>>> import numpy as np
>>> from mgc.independence import Pearson
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = Pearson().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

Kendall's tau

class mgc.independence.**Kendall**

Class for calculating the Kendall's τ test statistic and p-value.

Kendall's τ coefficient is a statistic to meassure ordinal associations between two quantities. The Kendall's τ correlation between high when variables similar rank relative to other observations[13]. Both this and the closely related Spearman's ρ coefficient are special cases of a general correlation coefficient.

See also:

Pearson Pearson product-moment correlation test statistic and p-value.

Spearman Spearman's rho test statistic and p-value.

[13] Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika*, 30(1/2), 81-93.

Notes

This class is a wrapper of scipy.stats.kendalltau. The statistic can be derived as follows[13]:

Let x and y be $(n, 1)$ samples of random variables X and Y. Define (x_i, y_i) and (x_j, y_j) as concordant if the ranks agree: $x_i > x_j$ and $y_i > y_j$ or *x_i > x_j* and $y_i < y_j$. They are discordant if the ranks disagree: $x_i > x_j$ and $y_i < y_j$ or $x_i < x_j$ and $y_i > y_j$. If $x_i > x_j$ and $y_i < y_j$, the pair is said to be tied. Let n_c and n_d be the number of concordant and discordant pairs respectively and $n_0 = n(n-1)/2$. In the case of no ties, the test statistic is defined as

$$\text{Kendall}_n(x, y) = \frac{n_c - n_d}{n_0}$$

Further, define $n_1 = \sum_i \frac{t_i(t_i-1)}{2}$, $n_2 = \sum_j \frac{u_j(u_j-1)}{2}$, t_i be the number of tied values in the i be the number of tied values in the :math:'j'th group. Then, the statistic is[14],

$$\text{Kendall}_n(x, y) = \frac{n_c - n_d}{\sqrt{(n_0 - n_1)(n_0 - n_2)}}$$

References

test (*x*, *y*)

Calculates the Kendall's τ test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples and dimensions. That is, the shapes must be *(n, 1)* where *n* is the number of samples.

Returns **stat** : float

The computed Kendall's tau statistic.

pvalue : float

The computed Kendall's tau p-value.

Examples

```
>>> import numpy as np
>>> from mgc.independence import Kendall
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = Kendall().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

Spearman's rho

class mgc.independence.**Spearman**

Class for calculating the Spearman's ρ test statistic and p-value.

Spearman's ρ coefficient is a nonparametric measure or rank correlation between two variables. It is equivalent to the Pearson's correlation with ranks.

See also:

[14] Agresti, A. (2010). *Analysis of ordinal categorical data* (Vol. 656). John Wiley & Sons.

Pearson Pearson product-moment correlation test statistic and p-value.

Kendall Kendall's tau test statistic and p-value.

Notes

This class is a wrapper of scipy.stats.spearmanr. The statistic can be derived as follows[15]:

Let x and y be $(n, 1)$ samples of random variables X and Y. Let rg_x and rg_y are the n raw scores. Let $\hat{\text{cov}}(rg_x, rg_y)$ is the sample covariance, and $\hat{\sigma}_{rg_x}$ and $\hat{\sigma}_{rg_x}$ are the sample variances of the rank variables. Then, the Spearman's ρ coefficient is,

$$\text{Spearman}_n(x, y) = \frac{\hat{\text{cov}}(rg_x, rg_y)}{\hat{\sigma}_{rg_x} \hat{\sigma}_{rg_y}}$$

References

test (*x*, *y*)

Calculates the Spearman's ρ test statistic and p-value.

Parameters **x, y** : ndarray

Input data matrices. *x* and *y* must have the same number of samples and dimensions. That is, the shapes must be *(n, 1)* where *n* is the number of samples.

Returns **stat** : float

The computed Spearman's rho statistic.

pvalue : float

The computed Spearman's rho p-value.

Examples

```
>>> import numpy as np
>>> from mgc.independence import Spearman
>>> x = np.arange(7)
>>> y = x
>>> stat, pvalue = Spearman().test(x, y)
>>> '%.1f, %.2f' % (stat, pvalue)
'1.0, 0.00'
```

5.2.2 *K*-Sample

Non-parametric *K*-Sample Test

class mgc.ksample.**KSample**(*indep_test*, *compute_distance=<function euclidean>*)

Class for calculating the *k*-sample test statistic and p-value.

A *k*-sample test tests equality in distribution among groups. Groups can be of different sizes, but generally have the same dimensionality. There are not many non-parametric *k*-sample tests, but this version cleverly leverages the power of some of the implemented independence tests to test this equality of distribution.

[15] Myers, J. L., Well, A. D., & Lorch Jr, R. F. (2013). *Research design and statistical analysis*. Routledge.

Parameters **indep_test** : {"CCA", "Dcorr", "HHG", "RV", "Hsic"}

A string corresponding to the desired independence test from `mgc.independence`.

compute_distance : callable(), optional (default: euclidean)

A function that computes the distance among the samples within each data matrix. Set to *None* if *x* and *y* are already distance matrices. To call a custom function, either create the distance matrix before-hand or create a function of the form `compute_distance(x)` where *x* is the data matrix for which pairwise distances are calculated.

Notes

The ideas behind this can be found in an upcoming paper:

The *k*-sample testing problem can be thought of as a generalization of the two sample testing problem. Define $\{u_i \overset{iid}{\sim} F_U,\ i = 1, ..., n\}$ and $\{v_j \overset{iid}{\sim} F_V,\ j = 1, ..., m\}$ as two groups of samples deriving from different distributions with the same dimensionality. Then, problem that we are testing is thus,

$$H_0 : F_U = F_V$$
$$H_A : F_U \neq F_V$$

The closely related independence testing problem can be generalized similarly: Given a set of paired data $\{(x_i, y_i) \overset{iid}{\sim} F_{XY},\ i = 1, ..., N\}$, the problem that we are testing is,

$$H_0 : F_{XY} = F_X F_Y$$
$$H_A : F_{XY} \neq F_X F_Y$$

By manipulating the inputs of the *k*-sample test, we can create concatenated versions of the inputs and another label matrix which are necessarily paired. Then, any nonparametric test can be performed on this data.

test (**args*, *reps=1000*, *workers=1*, *random_state=None*)

Calculates the *k*-sample test statistic and p-value.

Parameters ***args** : ndarrays

Variable length input data matrices. All inputs must have the same number of samples. That is, the shapes must be *(n, p)* and *(m, p)* where *n* and *m* are the number of samples and *p* are the number of dimensions. Alternatively, inputs can be distance matrices, where the shapes must all be *(n, n)*.

reps : int, optional (default: 1000)

The number of replications used to estimate the null distribution when using the permutation test used to calculate the p-value.

workers : int, optional (default: 1)

The number of cores to parallelize the p-value computation over. Supply -1 to use all cores available to the Process.

random_state : int or np.random.RandomState instance, optional

If already a RandomState instance, use it. If seed is an int, return a new RandomState instance seeded with seed. If None, use np.random.RandomState. Default is None.

Returns **stat** : float

The computed *k*-Sample statistic.

pvalue : float

The computed *k*-Sample p-value.

Examples

```
>>> import numpy as np
>>> from mgc.ksample import KSample
>>> x = np.arange(7)
>>> y = x
>>> z = np.arange(10)
>>> stat, pvalue = KSample("Dcorr").test(x, y)
>>> '%.3f, %.1f' % (stat, pvalue)
'-0.136, 1.0'
```

The number of replications can give p-values with higher confidence (greater alpha levels).

```
>>> import numpy as np
>>> from mgc.ksample import KSample
>>> x = np.arange(7)
>>> y = x
>>> z = np.ones(7)
>>> stat, pvalue = KSample("Dcorr").test(x, y, z, reps=10000)
>>> '%.3f, %.1f' % (stat, pvalue)
'0.224, 0.0'
```

5.2.3 Simulations

Independence Simulations

mgc.sims.**linear**(*n*, *p*, *noise=False*, *low=-1*, *high=1*)

Simulates univariate or multivariate linear data.

Parameters **n** : int

The number of samples desired by the simulation.

p : int

The number of dimensions desired by the simulation.

noise : float, (default: 1)

The noise amplitude of the simulation.

low : float, (default: -1)

The lower limit of the uniform distribution simulated from.

high : float, (default: -1)

The upper limit of the uniform distribution simulated from.

Returns **x, y** : ndarray

Simulated data matrices. *x* and *y* have shapes *(n, p)* and *(n, 1)* where *n* is the number of samples and *p* is the number of dimensions.

Notes

Linear $(X, Y) \in \mathbb{R}^p \times \mathbb{R}$:

$$X \sim \mathcal{U}(-1, 1)^p$$
$$Y = w^T X + \kappa\epsilon$$

Examples

```
>>> from mgc.sims import linear
>>> x, y = linear(100, 2)
>>> print(x.shape, y.shape)
(100, 2) (100, 1)
```

mgc.sims.**exponential**(*n*, *p*, *noise=False*, *low=0*, *high=3*)
Simulates univariate or multivariate exponential data.

Parameters **n** : int

The number of samples desired by the simulation.

p : int

The number of dimensions desired by the simulation.

noise : float, (default: 10)

The noise amplitude of the simulation.

low : float, (default: 0)

The lower limit of the uniform distribution simulated from.

high : float, (default: 3)

The upper limit of the uniform distribution simulated from.

Returns **x, y** : ndarray

Simulated data matrices. *x* and *y* have shapes *(n, p)* and *(n, 1)* where *n* is the number of samples and *p* is the number of dimensions.

Notes

Exponential $(X, Y) \in \mathbb{R}^p \times \mathbb{R}$:

$$X \sim \mathcal{U}(0, 3)^p$$
$$Y = \exp(w^T X) + 10\kappa\epsilon$$

Examples

```
>>> from mgc.sims import exponential
>>> x, y = exponential(100, 2)
>>> print(x.shape, y.shape)
(100, 2) (100, 1)
```

mgc.sims.**cubic**(*n, p, noise=False, low=-1, high=1, cubs=[-12, 48, 128], scale=0.3333333333333333*)
Simulates univariate or multivariate cubic data.

Parameters **n** : int

The number of samples desired by the simulation.

p : int

The number of dimensions desired by the simulation.

noise : float, (default: 80)

The noise amplitude of the simulation.

low : float, (default: -1)

The lower limit of the uniform distribution simulated from.

high : float, (default: -1)

The upper limit of the uniform distribution simulated from.

cubs : list of ints (default: [-12, 48, 128])

Coefficients of the cubic function where each value corresponds to the order of the cubic polynomial.

scale : float (default: 1/3)

Scaling center of the cubic.

Returns **x, y** : ndarray

Simulated data matrices. *x* and *y* have shapes *(n, p)* and *(n, 1)* where *n* is the number of samples and *p* is the number of dimensions.

Notes

Cubic $(X, Y) \in \mathbb{R}^p \times \mathbb{R}$:

$$X \sim \mathcal{U}(-1, 1)^p$$
$$Y = 128\left(w^T X - \frac{1}{3}\right)^3 + 48\left(w^T X - \frac{1}{3}\right)^2 - 12\left(w^T X - \frac{1}{3}\right) + 80\kappa\epsilon$$

Examples

```
>>> from mgc.sims import cubic
>>> x, y = cubic(100, 2)
>>> print(x.shape, y.shape)
(100, 2) (100, 1)
```

mgc.sims.**spiral**(*n*, *p*, *noise=False*, *low=0*, *high=5*)

Simulates univariate or multivariate spiral data.

Parameters **n** : int

The number of samples desired by the simulation.

p : int

The number of dimensions desired by the simulation.

noise : int, (default: 0.4)

The noise amplitude of the simulation.

low : float, (default: 0)

The lower limit of the uniform distribution simulated from.

high : float, (default: 5)

The upper limit of the uniform distribution simulated from.

Returns **x, y** : ndarray

Simulated data matrices. *x* and *y* have shapes *(n, p)* and *(n, 1)* where *n* is the number of samples and *p* is the number of dimensions.

Notes

Spiral $(X, Y) \in \mathbb{R}^p \times \mathbb{R}$: For $U \sim \mathcal{U}(0, 5)$, $\epsilon \sim \mathcal{N}(0, 1)$

$$
\begin{aligned}
X_{|d|} &= U \sin(\pi U) \cos^d(\pi U) \text{ for } d = 1, ..., p-1 \\
X_{|p|} &= U \cos^p(\pi U) \\
Y &= U \sin(\pi U) + 0.4p\epsilon
\end{aligned}
$$

Examples

```
>>> from mgc.sims import spiral
>>> x, y = spiral(100, 2)
>>> print(x.shape, y.shape)
(100, 2) (100, 1)
```

5.3 License

mgc is distributed with a MIT license.

```
MIT License

Copyright (c) 2019 Sambit Panda

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 6

Indices and tables

- genindex
- search

Index