ecopy Documentation

Release 0.1.2.2

Nathan Lemoine

Sep 01, 2017

Contents

EcoPy contains numerous numerical and statistical techniques for working with and analyzing multivariate data. Although designed with ecologists in mind, many of the functions and features are widely applicable. In general, it focuses on multivariate data analysis, which can be useful in any field, but with particular attention to those methods widely used in ecology. This website contains documentation and examples for all functions.

CHAPTER 1

Version

Current Version - 0.1.2.2

CHAPTER 2

Installing EcoPy

pip install ecopy

Documentation

Base Functions

EcoPy contains several basic functions:

- *wt_mean()*
- *wt_var()*
- *wt_scale()*
- *impute()*
- *spatial_median()*

wt_mean(*x*, *wt=None*)

Calculates as weighted mean. Returns a float.

$$\mu = \frac{\sum x_i w_i}{\sum w_i}$$

Parameters

x: numpy.ndarray or list A vector of input observations

wt: numpy.ndarray or list A vector of weights. If this vector does not sum to 1, this will be transformed internally by dividing each weight by the sum of weights

Example

Weighted mean:

```
import ecopy as ep
print(ep.wt_mean([1,3,5], [1,2,1]))
```

wt_var (*x*, *wt=None*, *bias=0*)
Calculates as weighted variance. Returns a float.

$$\sigma^2 = \frac{\sum w_i (x_i - \mu_w)^2}{\sum w_i}$$

where μ_w is the weighted mean.

Parameters

x: numpy.ndarray or list A vector of input observations

wt: numpy.ndarray or list A vector of weights. If this vector does not sum to 1, this will be transformed internally by dividing each weight by the sum of weights

bias: [0 | 1] Whether or not to calculate unbiased (0) or biased (1) variance. Biased variance is given by the equation above. Unbiased variance is the biased variance multiplied by $\frac{1}{1-\sum w^2}$.

Example

Weighted variance:

```
import ecopy as ep
print(ep.wt_var([1,3,5], [1,2,1]))
```

wt_scale (*x*, *wt=None*, *bias=0*)
Returns a vector of scaled, weighted observations.

$$z = \frac{x - \mu_w}{\sigma_w}$$

where μ_w is the weighted mean and σ_w is weighted standard deviation (the square root of weighted variance).

Parameters

x: numpy.ndarray or list A vector of input observations

wt: numpy.ndarray or list A vector of weights. If this vector does not sum to 1, this will be transformed internally by dividing each weight by the sum of weights

bias: [0 | 1] Whether or not the weighted standard deviation σ_w should be calculated from the biased or unbiased variance, as above

Example

Weighted variance:

```
import ecopy as ep
print(ep.wt_scale([1,3,5], [1,2,1]))
```

impute (*Y*, *method='mice'*, *m=5*, *delta=0.0001*, *niter=100*)
Performs univariate missing data imputation using one of several methods described below. NOTE: This method will not work with categorical or binary data (see TO-DO list). See van Buuren et al. (2006) and/or van Buuren (2012) for descriptions of univariate, monotone, and MICE algorithms.

Parameters

Y: numpy.ndarray or pandas.DataFrame Data matrix containing missing values. Missing values need not be only in one column and can be in all columns

method: ['mean' | 'median' | 'multi_norm' | 'univariate' | 'monotone' | 'mice'] Imputation method to be used. One of the following:

mean: Replaces missing values with the mean of their respective columns. Returns a single numpy.ndarray.

median: Replaces missing values with the median of their respective columns. Returns a single numpy.ndarray.

multi_norm: Approximates the multivariate normal distribution using the fully observed data. Replaces missing values with random draws from this distribution. Returns m numpy.ndarrays.

univariate: Conducts univariate imputation based on posterior draws of Bayesian regression parameters.

monotone: Monotone imputation for longitudinally structured data.

mice: Implements the MICE algorithm for data imputation. Assumes the *univariate* model is the correct model for all columns.

m: integer Number of imputed matrices to return

delta: float [0.0001 - 0.1] Ridge regression parameter to prevent non-invertible matrices.

niter: integer Number of iterations implemented in the MICE algorithm

Example

First, load in the urchin data:

```
import ecopy as ep
import numpy as np
import pandas as pd
data = ep.load_data('urchins')
```

Randomly replace mass and respiration values with NAs:

```
massNA = np.random.randint(0, 24, 5)
respNA = np.random.randint(0, 24, 7)
data.loc[massNA, 'UrchinMass'] = np.nan
data.loc[respNA, 'Respiration'] = np.nan
```

Impute using the MICE algorithm, then convert the returned arrays to dataframes:

```
imputedData = ep.impute(data, 'mice')
imputedFrame = [pd.DataFrame(x, columns=data.columns) for x in imputedData]
```

Alternatively, replace the missing values with the column means:

```
meanImpute = ep.impute(data, 'mean')
```

spatial_median(*X*)

Calculates the spatial median of a multivariate dataset. The spatial median is defined as the multivariate point a that minimizes:

$$E||x - a||$$

where $||x - a||$ is the euclidean distance between the vector x and a. Minimization is achieved by minimization optimization using scipy.optimize.minimize and the 'BFGS' algorithm.

Parameters

X: numpy.ndarray or pandas.DataFrame A matrix of input observations

Example

Calculate the spatial median for a random matrix:

```
import ecopy as ep
from scipy.stats import multivariate_normal

np.random.seed(654321)
cov = np.diag([3.,5.,2.])
data = multivariate_normal.rvs([0,0,0], cov, (100,))
spatialMed = ep.spatial_median(data)
```

Species Diversity

EcoPy contains several methods for estimating species diversity:

- *diversity()*
- *rarefy()*
- :py:func‘div_partition‘
- :py:func‘beta_dispersion‘

diversity (*x*, *method='shannon'*, *breakNA=True*, *num_equiv=True*)
Calculate species diversity for every site in a site x species matrix

Parameters

x: numpy.ndarray or pandas.DataFrame (*required*) A site *x* species matrix, where sites are rows and columns are species.

method: ['shannon' | 'gini-simpson' | 'simpson' | 'dominance' | 'spRich' | 'even'] *shannon*: Calculates Shannon's H

$$H = -\sum_1^k p_k \log p_k$$

where p_k is the relative abundance of species *k*

gini-simpson: Calculates the Gini-Simpson coefficient

$$D = 1 - \sum_1^k p_k^2$$

simpon: Calculates Simpson's D

$$D = \sum_1^k p_k^2$$

dominance: Dominance index. $\max p_k$

spRich: Species richness (# of non-zero columns)

even: Evenness of a site. Shannon's H divided by log of species richness.

breakNA: [True | False] Whether null values should halt the process. If False, then null values are removed from all calculations.

num_equiv: [True | False] Whether or not species diversity is returned in number-equivalents, which has better properties than raw diversity. Number equivalents are calculated as follows:

shannon: $exp(H)$

gini-simpson: $\frac{1}{1-D}$

simpson: $\frac{1}{D}$

'spRich': No conversion needed.

Example

Calculate Shannon diversity of the 'varespec' dataset from R:

```
import ecopy as ep
varespec = ep.load_data('varespec')
shannonH = ep.diversity(varespec, 'shannon')
```

rarefy (*x*, *method='rarefy'*, *size=None*, *breakNA=True*)

Returns either rarefied species richness or draws a rarefaction curve for each row. Rarefied species richness is calculated based on the smallest sample (default) or allows user-specified sample sizes.

Parameters

x: numpy.ndarray or pandas.DataFrame (*required*) A site x species matrix, where sites are rows and columns are species.

method: ['rarefy' | 'rarecurve'] *rarefy*: Returns rarefied species richness.

$$S = \sum_1^i 1 - \frac{\binom{N-N_i}{size}}{\binom{N}{size}}$$

where *N* is the total number of individuals in the site, N_i is the number of individuals of species *i*, and *size* is the sample size for rarefaction

rarecurve: Plots a rarefaction curve for each site (row). The curve is calculated as

$$S_n - \frac{\sum_1^i \binom{N-N_i}{size}}{\binom{N}{size}}$$

where S_n is the total number of species in the matrix and *size* ranges from 0 to the total number of individuals in each site.

Example

Calculate rarefied species richness for the BCI dataset:

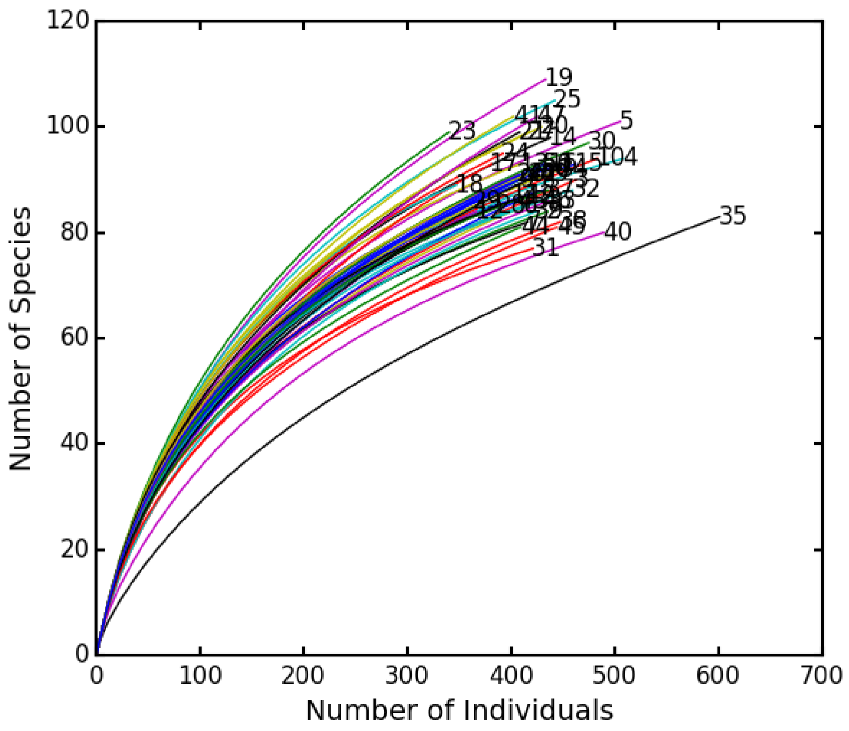
```
import ecopy as ep
BCI = ep.load_data('BCI')
rareRich = ep.rarefy(BCI, 'rarefy')
```

Show rarefaction curves for each site:

```
ep.rarefy(BCI, 'rarecurve')
```

div_partition (*x*, *method='shannon'*, *breakNA=True*, *weights=None*)

Partitions diversity into alpha, beta, and gamma components. First, diversity is calculated for each site (see `diversity()`). Then, a weighted average of each diversity metric is calculated to yield an average alpha diversity. This average alpha diversity is converted to number equivalents D_α (see `diversity()`). Next,



gamma diversity is calculated using the species totals from the entire matrix (i.e. summing down columns) and converted to a number equivalent D_γ. Beta diversity is then:

$$D_\beta = \frac{D_\gamma}{D_\alpha}$$

Parameters

x: numpy.ndarray or pandas.DataFrame (*required*) A site x species matrix, where sites are rows and columns are species.

method: ['shannon' | 'gini-simpson' | 'simpson' | 'spRich'] See `diversity()`

breakNA: [True | False] Whether null values should halt the process. If False, then null values are removed from all calculations.

weights: list or np.ndarray Weights given for each row (site). Defaults to the sum of each row divided by the sum of the matrix. This yields weights based on the number of individuals in a site for raw abundance data or equal weights for relative abundance data.

Example

Partition diversity into alpha, beta, and gamma components for the 'varespec' data:

```
import ecopy as ep
varespec = ep.load_data('varespec')
D_alpha, D_beta, D_gamma = ep.div_partition(varespec, 'shannon')
```

beta_dispersion (*X*, *groups*, *test='anova'*, *scores=False*, *center='median'*, *n_iter=99*)
Calculate beta dispersion among groups for a given distance matrix. First, the data is subject to transformation as described in PCoA. Next, eigenvalues and eigenvectors are calculated for the transformed matrix. Eigenvectors

are split into two matrices, those pertaining to non-negative eigenvalues and those pertaining to negative eigenvalues. Next, centroids for the positive and negative eigenvector matrices are determined (using spatial_median if center='median'). Z-distances are calculated as:

$$z_{ij} = np.sqrt(\delta^2(x_{ij} - c_i^+) - \delta^2(x_{ij} - c_i^-))$$

where $\delta^2(x_{ij}, c_i^+)$ is the squared euclidean distance between observation **ij** and the center of group **i**, and +/- denote the non-negative and negative eigenvector matrices.

A one-way ANOVA is conducted on the z-distances.

Parameters

X: numpy.ndarray, pandas.DataFrame A square, symmetric distance matrix

groups: list, pandas.Series, pandas.DataFrame A column or list containing the group identification for each observation

test: ['anova' | 'permute'] Whether significance is calculated using the ANOVA approximation or permutation of residuals.

scores: [True | False] Whether or not the calculated z-distances should be returned for subsequent plotting

center: ['median' | 'centroid'] Which central tendency should be used for calculating z-distances.

n_iter: integer Number of iterations for the permutation test

Example

Conduct beta dispersion test on the 'varespec' dataset from R:

```
varespec = ep.load_data('varespec')
dist = ep.distance(varespec, method='bray')

groups = ['grazed']*16 + ['ungrazed']*8
ep.beta_dispersion(dist, groups, test='permute', center='median', scores=False)
```

Matrix Transformations

EcoPy makes it easy to prep matrices for analysis. It assumes that all matrices have observations as rows (*i.e.* sites) and descriptors as columns (*i.e.* species). Although designed for site *x* species analyses, these techniques can apply to any matrix.

- `transform()`
- `distance()`

`transform`(*x*, *method='wisconsin'*, *axis=1*, *breakNA=True*)

Takes an input matrix, performs a transformation, and returns an output matrix. It will accept either a pandas.DataFrame or numpy.ndarray, and will return an object of the same class. Matrices consist of *i* rows and *k* columns.

Parameters

x: a numpy.ndarray or pandas.DataFrame (*required*) A site *x* species matrix, where sites are rows and columns are species.

method: ['total' | 'max' | 'normalize', 'range', 'standardize', 'hellinger', 'log', 'logp1', 'pa', 'wisconsin']

total: Divides each observation by row or column sum.

max: Divides each observation by row or column max.

normalize: Chord transformation, also euclidean normalization, making the length of each row or column 1.

$$y_{ik} = \frac{y_{ik}}{\sqrt{\sum_1^k y_{ik}^2}}$$

range: Converts the range of the data to 0 and 1.

standardize: Standardizes each observation (*i.e.* z-score).

hellinger: Square-root of the *total* transformation.

log: Returns ln(x+1)

logp1: Returns ln(x) + 1, if x > 0. Otherwise returns 0.

pa: Converts data to binary absence (0) presence (1) data.

wisconsin: First divides an observation by the max of the column, then the sum of the row. That is, it applies 'max' down columns then 'total' across rows.

axis: [0 | 1] Axis for the transformation.

breakNA: [True | False] Whether NA values should halt the transformation.

Example

Convert the 'varespec' data to relative abundance:

```
import ecopy as ep
varespec = ep.load_data('varespec')
relAbund = ep.transform(varespec, method='total', axis=1)
```

distance (*x*, *method='euclidean'*, *transform="1"*, *breakNA=True*)

Takes an input matrix and returns a square-symmetric array of distances among rows. **NOTE:** Be sure the appropriate transformation has already been applied. This function contains a variety of both similarity (S) and distance (D) metrics. However, for consistency all similarities are converted to distances D = 1 - S. Methods annotated with SIMILARITY follow this procedure.

In the case of binary 0/1 data, the two rows are converted to a contingency table, where A is the number of double presences, B and C are the number of single presences in x_1 and x_2, respectively, and D is the number of double absences. Matrices consist of *i* rows and *k* species. Methods that only work on binary data will result in an error if non-binary data is passed. However, binary data can be passed to all methods, and sometimes give equivalent results (i.e. passing binary data to method 'bray' is identical to using method 'sorensen').

Parameters

x: a numpy.ndarray or pandas.DataFrame (*required*) A site *x* species matrix, where sites are rows and columns are species.

method: ['euclidean' | 'gow_euclidean' | chord' | 'manhattan' | 'meanChar' | 'whittaker' | 'canberra' | 'hellinger' | 'mod_g Note, some methods do not allow negative values.

euclidean: Calculates euclidean distance between rows.

gow_euclidean: Calculates euclidean distance between rows, removing missing values.

$$D_{1,2} = \sqrt{\frac{\sum_k^p \delta_k (x_{1k} - x_{2k})^2}{\sum_k^p \delta_k}}$$

where δ_k =1 if the observation is present in both rows and 0 otherwise.

chord: Euclidean distance of normalized rows.

manhattan: 'City-block' distance

$$D_{1,2} = \sum_1^k |x_{1k} - x_{2k}|$$

meanChar: Czekanowski's mean character difference, where M is the number of columns.

$$D_{1,2} = \frac{1}{M} \sum_1^k |x_{1k} - x_{2k}|$$

whittaker: Whittaker's index of association. Rows are first standardized by row totals (if the transformation as already been applied above, this will not affect it as row totals will equal 1)

$$D_{1,2} = 0.5 \sum_1^k |x_{1k} - x_{2k}|$$

canberra: Canberra metric

$$\frac{1}{M} \sum_1^k \frac{x_{1k} - x_{2k}}{x_{1k} + x_{2k}}$$

hellinger: Hellinger distance. This is the same as 'chord', but square-root transformed first.

mod_gower: Modified Gower distance. This is the same as 'meanChar', except M is the number of columns that are not double zero. This discounts double-absences from the 'meanChar' method.

bray: Bray-Curtis percentage dissimilarity coefficient

$$D_{1,2} = 1 - \frac{2 * \sum_1^k \min(x_{1k}, x_{2k})}{\sum x_1 + \sum x_2}$$

kulcznski: Kulcznski's coefficient (SIMILARITY)

$$S_{1,2} = 0.5(\frac{\sum_1^k \min(x_{1k}, x_{2k})}{\sum x_1} + \frac{\sum_1^k \min(x_{1k}, x_{2k})}{\sum x_2})$$

gower: Gower asymmetrical coefficient (SIMILARITY)

$$S_{1,2} = \frac{1}{M}(1 - \sum_1^k \frac{|x_{1k} - x_{2k}|}{\max x_k - \min x_k})$$

The denominator is the maximum of column *k* minus the minimum of column *k* in the entire matrix. Double zeroes are excluded in this calculation.

simple: simple matching of BINARY data (SIMILARITY)

$$S_{1,2} = \frac{A + D}{A + B + C + D}$$

rogers: Rogers and Tanimoto coefficient for BINARY data (SIMILARITY)

$$S_{1,2} = \frac{A + D}{A + 2B + 2C + D}$$

sokal: Sokal and Sneath coefficient for BINARY data (SIMILARITY)

$$S_{1,2} = \frac{2A + 2D}{2A + B + C + 2D}$$

jaccard: Jaccard's coefficient for BINARY data (SIMILARITY)

$$S_{1,2} = \frac{A}{A + B + C}$$

sorensen: Sorensen's coefficient for BINARY data (SIMILARITY)

$$S_{1,2} = \frac{2A}{2A + B + C}$$

transform: [”1” | “sqrt”] Determines the final transformation of the distance metric. “1” returns the raw distance D. “sqrt” returns sqrt(D). Sometimes sqrt(D) has more desirable properties, depending on the subsequent analyses (see Legendre and Legendre - Numerical Ecology).

breakNA: [True | False] Whether null values should halt the process.

Examples

Calculate the Bray-Curtis dissimilarity among rows of the ‘varespec’ data:

```
import ecopy as ep
varespec = ep.load_data('varespec')
brayDist = ep.distance(varespec, method='bray')
```

If attempting a binary method with non-binary data, an error will be raise:

```
jacDist = ep.distance(varespec, method='jaccard')

>>ValueError: For method jaccard, data must be binary

varespec2 = ep.transform(varespec, method='pa')
jacDist = ep.distance(varespec2, method='jaccard')
```

Ordination

Ecopy contains numerous methods for ordination, that is, plotting points in reduced space. Techniques include, but are not limited to, principle components analysis (PCA), correspondence analysis (CA), principle coordinates analysis (PCoA), and multidimensional scaling (nMDS).

- `pca` (Principle Components Analysis)
- `ca` (Correspondance Analysis)
- `pcoa` (Principle Coordinates Analysis)
- `MDS` (Multidimensional Scaling)
- `hillsmith` (Hill and Smith Ordination)
- `ord_plot()` (Ordination plotting)

class pca (*x*, *scale=True*, *varNames=None*)

Takes an input matrix and performs principle components analysis. It will accept either pandas.DataFrames or numpy.ndarrays. It returns on object of class :py:class: *pca*, with several methods and attributes. This function uses SVD and can operate when rows < columns. NOTE: PCA will NOT work with missing observations, as it is up to the user to decide how best to deal with those. Returns object of class `pca`.

Parameters

x: a numpy.ndarray or pandas.DataFrame A matrix for ordination, where objects are rows and descriptors/variables as columns. Can be either a pandas.DataFrame or numpy. ndarray.

scale: [True | False] Whether or not the columns should be standardized prior to PCA. If 'True', the PCA then operates on a correlation matrix, which is appropriate if variables are on different measurement scales. If variables are on the same scale, use 'False' to have PCA operate on the covariance matrix.

varNames: list If using a numpy.ndarray, pass a list of column names for to help make PCA output easier to interpret. Column names should be in order of the columns in the matrix. Otherwise, column names are represented as integers during summary.

Attributes

`evals`
Eigenvalues in order of largest to smallest.

`evecs`
Normalized eigenvectors corresponding to each eigenvalue (i.e. the principle axes).

`scores`
Principle component scores of each object (row) on each principle axis. This returns the raw scores $\mathbf{F}$ calculated as $\mathbf{F} = \mathbf{YU}$ where $\mathbf{U}$ is the matrix of eigenvectors and $\mathbf{Y}$ are the original observations.

Methods

classmethod `summary_imp`()
Returns a data frame containing information about the principle axes.

classmethod `summary_rot`()
Returns a data frame containing information on axes rotations (i.e. the eigenvectors).

classmethod `summary_desc`()
Returns a data frame containing the cumulative variance explained for each predictor along each principle axis.

classmethod `biplot`(*xax=1*, *yax=2*, *type='distance'*, *obsNames=False*)
Create a biplot using a specified transformation.

xax: integer Specifies which PC axis to plot on the x-axis

yax: integer Specifies which PC axis to plot on the y-axis

type: ['distance' | 'correlation'] Type 'distance' plots the raw scores $\mathbf{F}$ and the raw vectors $\mathbf{U}$ of the first two principle axes.

Type 'correlation' plots scores and vectors scaled by the eigenvalues corresponding to each axis: $\mathbf{F\Lambda}^{-0.5}$ and $\mathbf{U\Lambda}^{0.5}$, where $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues.

obsNames: [True | False] Denotes whether to plot a scatterplot of points (False) or to actually show the names of the observations, as taken from the DataFrame index (True).

Examples

Principle components analysis of the USArrests data. First, load the data:

```
import ecopy as ep
USArrests = ep.load_data('USArrests')
USArrests.set_index('State', inplace=True)
```

Next, run the PCA:

```
arrests_PCA = ep.pca(USArrests, scale=True)
```

Check the importance of the different axes by examining the standard deviations, which are the square root of the eigenvalues, and the proportions of variance explained by each axis:

```
impPC = arrests_PCA.summary_imp()
print(impPC)
             PC1      PC2       PC3      PC4
Std Dev 1.574878 0.994869 0.597129 0.416449
Proportion 0.620060 0.247441 0.089141 0.043358
Cum Prop 0.620060 0.867502 0.956642 1.000000
```

Next, examine the eigenvectors and loadings to determine which variables contribute to which axes:

```
rotPC = arrests_PCA.summary_rot()
print(rotPC)
         PC1       PC2      PC3        PC4
Murder 0.535899 0.418181 -0.341233 0.649228
Assault 0.583184 0.187986 -0.268148 -0.743407
UrbanPop 0.278191 -0.872806 -0.378016 0.133878
Rape 0.543432 -0.167319 0.817778 0.089024
```

Then, look to see how much of the variance among predictors is explained by the first two axes:

```
print(arrests_PCA.summary_desc())
            PC1       PC2      PC3   PC4
Murder 0.712296 0.885382 0.926900 1
Assault 0.843538 0.878515 0.904153 1
Urban Pop 0.191946 0.945940 0.996892 1
Rape 0.732461 0.760170 0.998626 1
```

Show the biplot using the 'correlation' scaling. Instead of just a scatterplot, use obsNames=True to show the actual names of observations:

```
arrests_PCA.biplot(type='correlation', obsNames=True)
```

class ca (*x*, *siteNames=None*, *spNames=None*, *scaling=1*)

Takes an input matrix and performs principle simple correspondence analysis. It will accept either pandas.DataFrames or numpy.ndarrays. Data MUST be 0's or positive numbers. **NOTE:** Will NOT work with missing observations, as it is up to the user to decide how best to deal with those. Returns on object of class `ca`.

Parameters

x: a numpy.ndarray or pandas.DataFrame A matrix for ordination, where objects are rows and descriptors/variables as columns. Can be either a pandas.DataFrame or numpy.ndarray. **NOTE:** If the matrix has more variables (columns) than objects (rows), the matrix will be transposed prior to analysis, which reverses the meanings of the matrices as noted.

The matrix is first scaled to proportions by dividing each element by the matrix sum, $p_{ik} = y_{ik} / \sum_1^i \sum_1^k$. Row (site) weights w_i are calculated as the sums of row probabilities and column (species) weights w_k are the sum of column probabilities. NOTE: If $r < c$ in the original matrix, then row weights give species weights and column weights give site weights due to transposition.

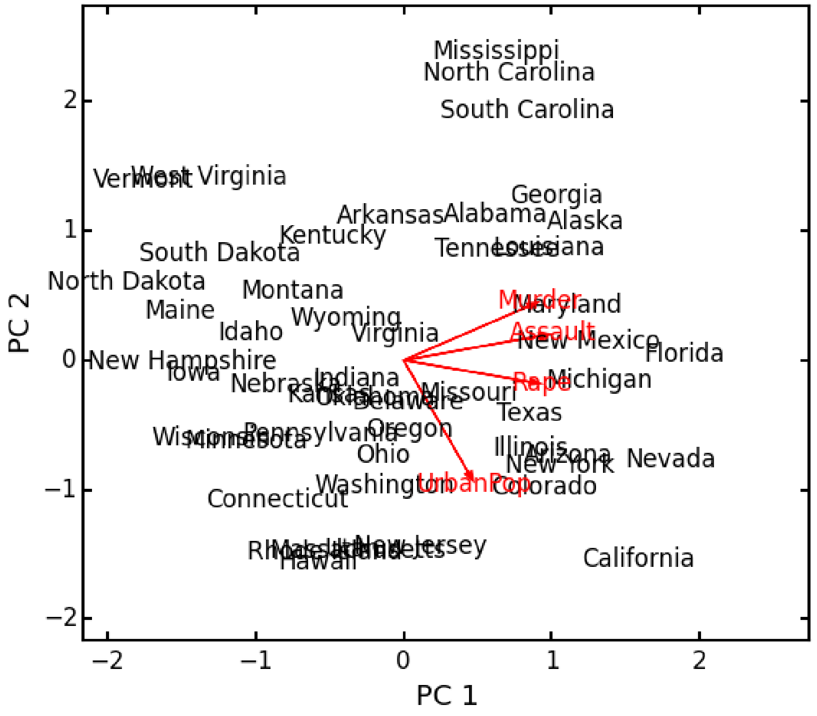
A matrix of chi-squared deviations is then calculated as:

$$\mathbf{Q} = \frac{p_{ik} - w_i w_k}{\sqrt{w_i w_k}}$$

This is then converted into a sum-of-squared deviations as

$$\mathbf{QQ} = \mathbf{Q}'\mathbf{Q}$$

Eigen-decomposition of $\mathbf{QQ}$ yields a diagonal matrix of eigenvalues $\mathbf{\Lambda}$ and a matrix of eigenvectors $\mathbf{U}$. Left-hand eigenvectors $\hat{\mathbf{U}}$ (as determined by SVD) are calculated as $\hat{\mathbf{U}} = \mathbf{QU\Lambda}^{-0.5}$. $\mathbf{U}$ gives the column



(species) loadings and $\hat{\mathbf{U}}$ gives the row (site) loadings. NOTE: If $r < c$ in the original matrix, the roles of these matrices are reversed.

siteNames: list A list of site names. If left blank, site names are taken as the index of the pandas.DataFrame or the row index from the numpy.ndarray.

spNames: list A list of species names. If left blank, species names are taken as the column names of the pandas.DataFrame or the column index from the numpy.ndarray.

scaling: [1 | 2] Which type of scaling to use when calculating site and species scores. 1 produces a site biplot, 2 produces a species biplot. In biplots, only the first two axes are shown. The plots are constructed as follows:

Four matrices are constructed. Outer species (column) locations on CA axes $\mathbf{V}$ are given by the species (column) weights multiplied by the species (column) eigenvalues:

$$\mathbf{V} = \mathbf{D_k}^{-0.5}\mathbf{U}$$

where $\mathbf{D_k}$ is a diagonal matrix of species (column) weights *w_k*. Likewise, outer site (row) locations are given by:

$$\hat{\mathbf{V}} = \mathbf{D_i}^{-0.5}\hat{\mathbf{U}}$$

Inner site locations $\mathbf{F}$ are given as:

$$\mathbf{F} = \hat{\mathbf{V}}\mathbf{\Lambda}^{0.5}$$

Inner species locations are given as:

$$\hat{\mathbf{F}} = \mathbf{V}\mathbf{\Lambda}^{0.5}$$

Scaling 1 Biplot: Scaling 1 shows the relationships among sites within the centroids of the species. This plot is useful for examining relationships among sites and how sites are composed of species. In this, the first two columns of inner site locations $\mathbf{F}$ are plotted against the first two columns of the outer species locations $\mathbf{V}$. NOTE: If $r < c$ in the original matrix, this will be $\hat{\mathbf{F}}$ and $\hat{\mathbf{V}}$.

Scaling 2 Biplot: Scaling 2 shows the relationships among species within the centroids of the sites. This plot is useful for examining relationships among species and how species are distributed among sites. In this, the first two columns of inner species locations $\hat{\mathbf{F}}$ are plotted against the first two columns of the outer site locations $\hat{\mathbf{V}}$. NOTE: If $r < c$ in the original matrix, this will be $\mathbf{F}$ and $\mathbf{V}$.

Attributes

`w_col`
Column weights in the proportion matrix. Normally species weights unless $r < c$, in which case they are site weights.

`w_row`
Row weights in the proportion matrix. Normally site weights unless $r < c$, in which case they are species weights.

`U`
Column (species) eigenvectors (see above note on transposition).

`Uhat`
Row (site) eigenvectors (see above note on transposition).

`cumDesc_Sp`
pandas.DataFrame of the cumulative contribution of each eigenvector to each species. Matrix $\mathbf{U}$ is scaled by eigenvalues $\mathbf{U_2} = \mathbf{U}\mathbf{\Lambda}^{0.5}$. Then, the cumulative sum of each column is divided by the column total for every row. If $r < c$ in the original data, then this operation is performed on $\hat{\mathbf{U}}$ automatically.

`cumDesc_Site`
The same for cumDesc_Sp, but for each site. Normally calculated for $\hat{\mathbf{U}}$ unless $r < c$, then calculated on $\mathbf{U}$.

`siteScores`
Site scores along each CA axis. All considerations for matrix transposition and scaling have been taken into account.

`spScores`
Species scores along each CA axis. All considerations for matrix transposition and scaling have been taken into account.

Methods

classmethod `summary``()`
Returns a pandas.DataFrame of summary information for each correspondence axis, including SD's (square-root of each eigenvalue), proportion of inertia explained, and cumulative inertia explained.

classmethod `biplot`(*coords=False*, *type=1*, *xax=1*, *yax=2*, *showSp=True*, *showSite=True*, *spCol='r'*, *siteCol='k'*, *spSize=12*, *siteSize=12*, *xlim=None*, *ylim=None*)
Produces a biplot of the given CA axes.

xax: integer Specifies CA axis to plot on the x-axis.

yax: integer Specifies CA axis to plot on the y-axis.

showSp: [True | False] Whether or not to show species in the plot.

showSite: [True | False] Whether or not to show sites in the plot.

spCol: string Color of species text.

siteCol: string Color of site text.

spSize: integer Size of species text.

siteSize: integer Size of site text.

xlim: list A list of x-axis limits to override default.

ylim: list A list of y-axis limits to override default.

Examples

In Legendre and Legendre (2012), there is an example of three species varying among three lakes. Write in that data:

```
import ecopy as ep
import numpy as np
import pandas as pd
Lakes = np.array([[10, 10, 20], [10, 15, 10], [15, 5, 5]])
Lakes = pd.DataFrame(Lakes, index = ['L1', 'L2', 'L3'])
Lakes.columns = ['Sp1', 'Sp2', 'Sp3']
```

Next, run the CA:

```
lakes_CA = ep.ca(Lakes)
```

Check the variance explained by each CA axis (there will only be two):

```
CA_summary = lakes_CA.summary()
print(CA_summary)
          CA Axis 1 CA Axis 2
Std. Dev 0.310053 0.202341
Prop. 0.701318 0.298682
Cum. Prop. 0.701318 1.000000
```

Next, see how well the two axes explained variance in species and sites:

```
print(lakes_CA.cumDesc_Sp)
   CA Axis 1 CA Axis 2
Sp1 0.971877 1
Sp2 0.129043 1
Sp3 0.732340 1

print(lakes_CA.cumDesc_site)
    CA Axis 1 CA Axis 2
L1 0.684705 1
L2 0.059355 1
L3 0.967209 1
```

Make a Type 1 biplot to look at the relationship among sites:

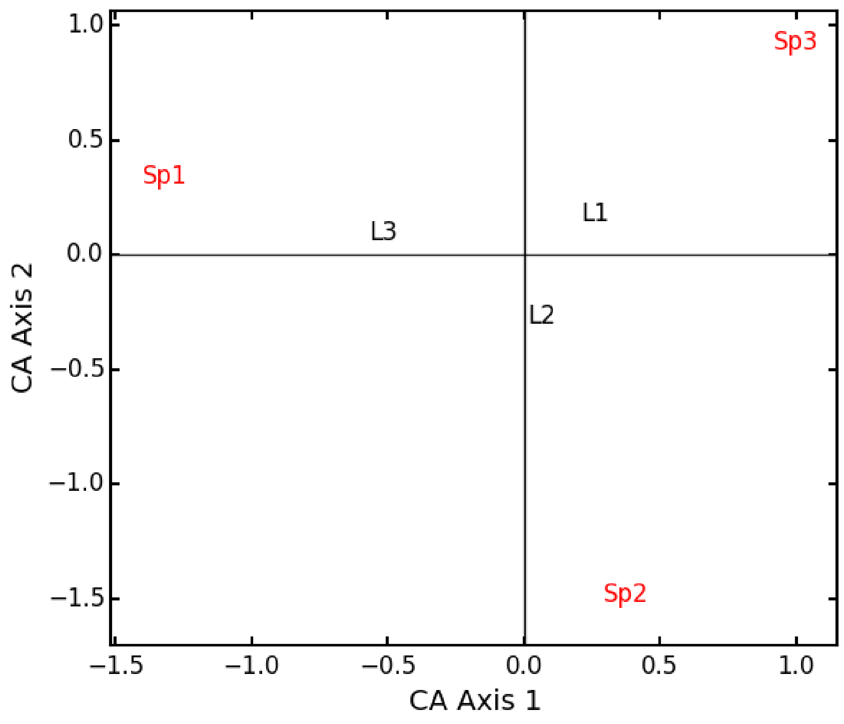
```
lakes_CA.biplot()
```

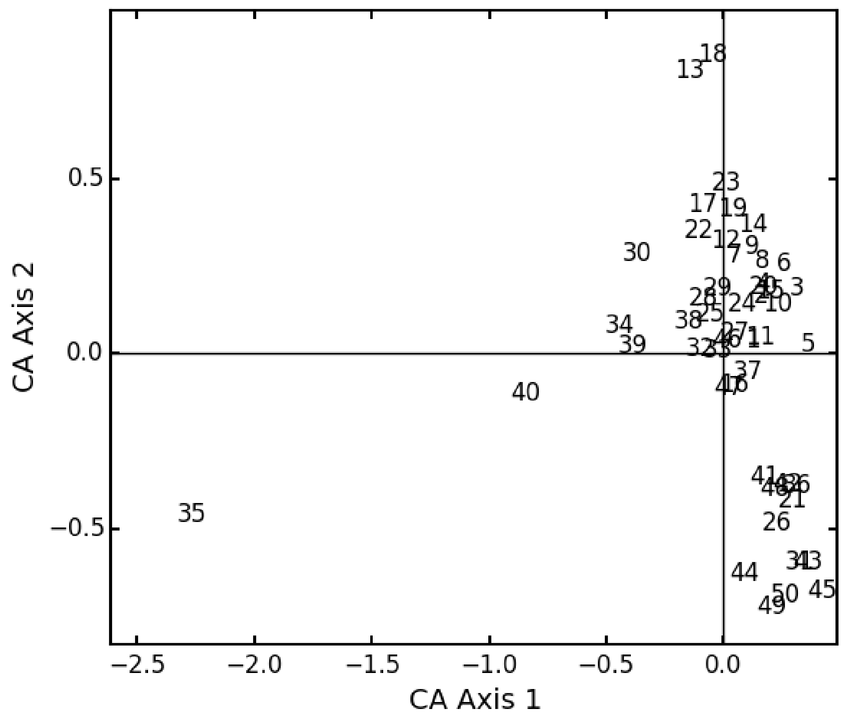
In a bigger example, run CA on the BCI dataset. **NOTE: This is an example where** $r < c$:

```
BCI = ep.load_data('BCI')
bci_ca = ep.ca(BCI)
bci_ca.biplot(showSp=False)
```

class pcoa (*x*, *correction=None*, *siteNames=None*)

Takes a square-symmetric distance matrix with no negative values as input. **NOTE:** This will not work with missing observations. Returns an object of class `pcoa`.


Sp3
Sp1
L1
L3
L2
Sp2
CA Axis 1
CA Axis 2


CA Axis 1
CA Axis 2

Parameters

x: a numpy.ndarray or pandas.DataFrame A square, symmetric distance matrix with no negative values and no missing observations. Diagonal entries should be 0.

For PCoA, distance matrix $\mathbf{x}$ is first corrected to a new matrix $\mathbf{A}$, where $a_{ij} = -0.5 * x_{ij}^2$. Elements of the new matrix $\mathbf{A}$ are centered by row and column means using the equation $\mathbf{\Delta_1} = (\mathbf{I} - \frac{\mathbf{1'1}}{\mathbf{n}})\mathbf{A}(\mathbf{I} - \frac{\mathbf{1'1}}{\mathbf{n}})$. PCoA is eigenanalysis of $\mathbf{\Delta_1}$. Eigenvectors $\mathbf{U}$ are scaled by the square root of each eigenvalue $\mathbf{U_{scl}} = \mathbf{U\Lambda^{0.5}}$ where $\mathbf{\Lambda}$ is a diagonal matrix of the eigenvalues.

correction: [None | 1 | 2] Which correction should be applied for negative eigenvalues. Accepts either '1' or '2' (must be a string). By default, no correction is applied.

Correction 1: Computes PCoA as described above. Adds the absolute value of the largest negative eigenvalue to the square original distance matrix (while keeping diagonals as 0) and then re-runs PCoA from the beginning.

Correction 2: Constructs a special matrix

$$\begin{bmatrix} \mathbf{0} & 2\mathbf{\Delta_1} \\ -\mathbf{I} & -4\mathbf{\Delta_2} \end{bmatrix}$$

Δ_1 is the centered, corrected distance matrix as described above and Δ_2 is a centered matrix (uncorrected) of $-0.5\mathbf{x}$. The largest, positive eigenvalue of this matrix is then added the original distances and PCoA run from the beginning.

siteNames: list A list of site names. If not passed, inherits from the DataFrame index or assigns integer values.

Attributes

`evals`
Eigenvalues of each principle coordinate axis.

`U`
Eignevectors describing each axis. These have already been scaled.

`correction`
The correction factor applied to correct for negative eignvalues.

Methods

classmethod `summary()`
Returns a pandas.DataFrame summarizing the variance explained by each principle coordinate axis.

classmethod `biplot`(*coords=False*, *xax=1*, *yax=2*, *descriptors=None*, *descripNames=None*, *spCol='r'*, *siteCol='k'*, *spSize=12*, *siteSize=12*)
Produces a biplot of the given PCoA axes.

coords: [True | False] If True, returns a dictionary of the plotted axes, where 'Objects' gives the coordinates of objects and 'Descriptors' gives the coordinates of the descriptors, if any.

xax: integer Specifies PCoA axis to plot on the x-axis.

yax: integer Specifies PCoA axis to plot on the y-axis.

descriptors: numpy.ndarray or pandas.DataFrame An n x m matrix of descriptors to plot on the biplot. These can be the original descriptors used to calculate distances among objects or an entirely new set. Descriptors must be quantitative. It will work for binary descriptors, but may be meaningless.

Given a new matrix $\mathbf{Y}$ of descriptors, the matrix is standardized by columns to produce a new matrix $\mathbf{Y_{scl}}$. The given principle coordinate axes denoted by xax and yax are placed into an n x 2 matrix

$\mathbf{V}$, which is also standardized by column. The covariance between the new descriptors and principle coordinates is given by

$$\mathbf{S} = \frac{1}{n-1}\mathbf{Y}'_{\mathbf{scl}}\mathbf{V}$$

The covariance $\mathbf{S}$ is then scaled by the eigenvalues corresponding to the given eigenvectors:

$$\mathbf{Y_{proj}} = \sqrt{n-1}\mathbf{S}\mathbf{\Lambda}^{-0.5}$$

Matrix $\mathbf{Y_{proj}}$ contains the coordinates of each descriptor and is what is returned as 'Descriptors' if coords=True.

descripNames: list A list containing the names of each descriptor. If None, inherits from the column names of the pandas.DataFrame or assigned integer values.

spCol: string Color of species text.

siteCol: string Color of site text.

spSize: integer Size of species text.

siteSize: integer Size of site text.

classmethod `shepard`(*xax=1*, *yax=2*)
Plots a Shepard diagram of Euclidean distances among objects in reduced space vs. original distance calculations. xax and yax as above.

Examples

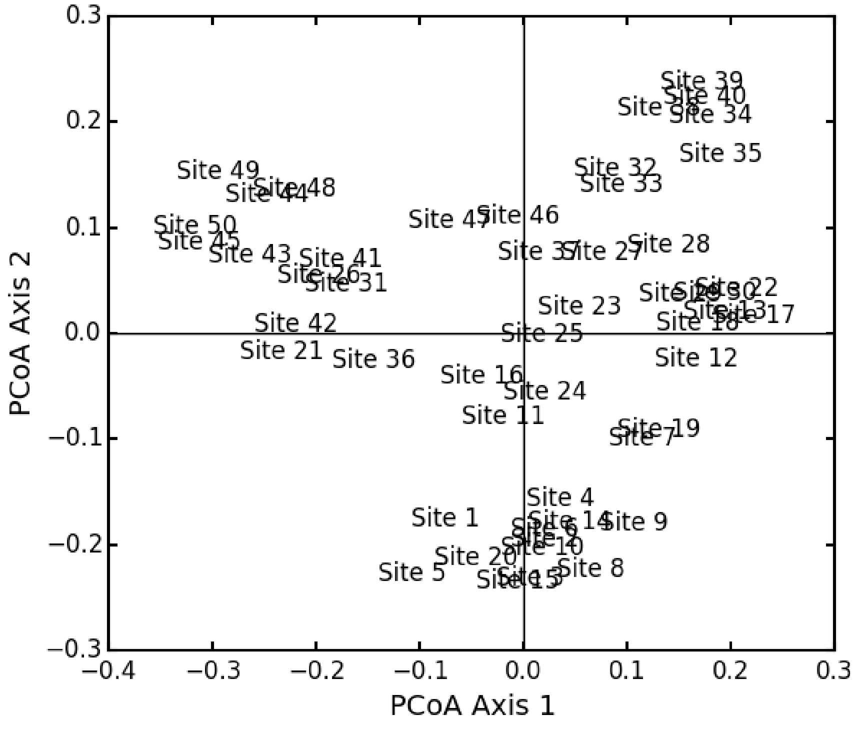
Run PCoA on the 'BCI' data:

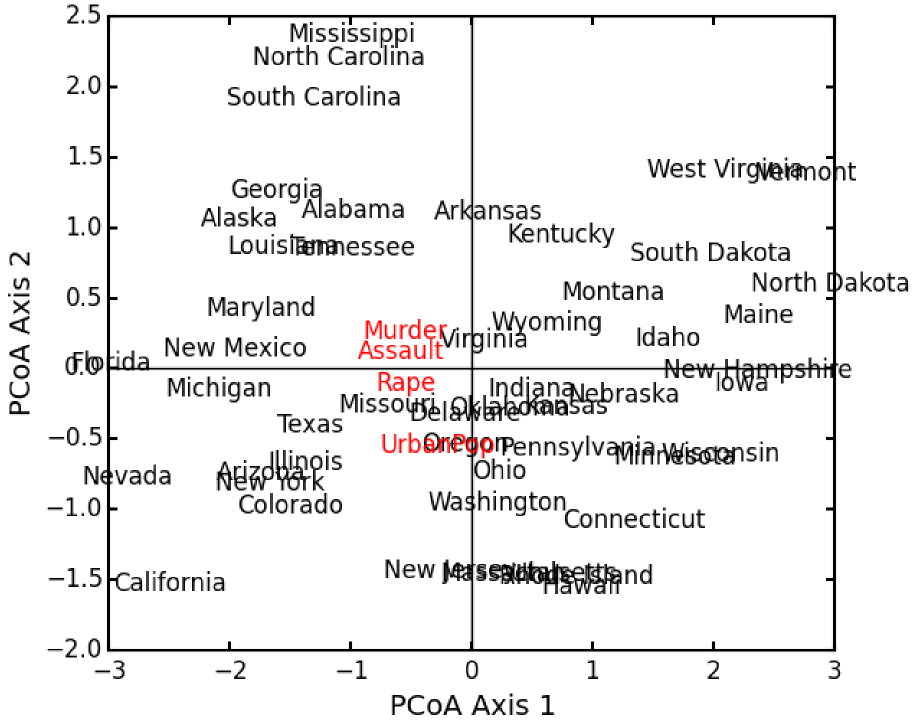
```
import ecopy as ep

BCI = ep.load_data('BCI')
brayD = ep.distance(BCI, method='bray', transform='sqrt')
pc1 = ep.pcoa(brayD)
print(pc1.summary()[['PCoA Axis 1', 'PCoA Axis 2']])

        PCoA Axis 1 PCoA Axis 2
Std. Dev 1.094943 0.962549
Prop. 0.107487 0.083065
Cum. Prop. 0.107487 0.190552

pc1.biplot()
```

Attempting to show species on the above biplot results in a messy graph. To better illustrate its use, run PCoA on the USArrests data:

```
USA = ep.load_data('USArrests')
USA.set_index('State', inplace=True)
# standardize columns first
USA = USA.apply(lambda x: (x - x.mean())/x.std(), axis=0)
eucD = ep.distance(USA, 'euclidean')

pc2 = ep.pcoa(eucD, siteNames=USA.index.values)
pc2.biplot(descriptors=USA)
```

class `MDS`(*distmat*, *siteNames=None*, *naxes=2*, *transform='monotone'*, *ntry=20*, *tolerance=1E-4*, *maxiter=3000*, *init=None*)
Takes a square-symmetric distance matrix with no negative values as input. After finding the solution that


PCoA Axis 1
PCoA Axis 2
Site 39
Site 40
Site 38
Site 34
Site 35
Site 32
Site 33
Site 49
Site 44
Site 48
Site 50
Site 45
Site 43
Site 41
Site 26
Site 31
Site 47
Site 46
Site 37
Site 27
Site 28
Site 23
Site 29
Site 30
Site 22
Site 13
Site 17
Site 18
Site 42
Site 25
Site 21
Site 36
Site 12
Site 16
Site 24
Site 11
Site 19
Site 7
Site 4
Site 1
Site 14
Site 9
Site 6
Site 10
Site 20
Site 5
Site 15
Site 8


PCoA Axis 1
PCoA Axis 2
Mississippi
North Carolina
South Carolina
West Virginia
Vermont
Georgia
Alaska
Alabama
Arkansas
Kentucky
Louisiana
Tennessee
South Dakota
Montana
North Dakota
Maryland
Maine
Murder
Assault
Wyoming
Virginia
Idaho
Florida
New Mexico
New Hampshire
Iowa
Rape
Michigan
Indiana
Nebraska
Missouri
Oklahoma
Kansas
Delaware
Texas
UrbanPop
Oregon
Pennsylvania
Minnesota
Wisconsin
Nevada
Arizona
Illinois
New York
Ohio
Colorado
Washington
Connecticut
California
New Jersey
Massachusetts
Rhode Island
Utah
Hawaii

provide the lowest stress, ecopy.MDS scales the fitted distances to have a maximum equal to the maximum observed distance. Afterwards, it uses PCA to rotate the object (site) scores so that variance is maximized along the x-axis. Returns an object of class `MDS`.

Parameters

distmat: np.ndarray or pandas.DataFrame A square-symmetric distance matrix.

siteNames: list A list of names for each object. If none, takes on integer values or the index of the pandas.DataFrame.

naxes: integer Number of ordination axes.

transform: ['absolute' | 'ratio' | 'linear' | 'monotone'] Which transformation should be used during scaling.

absolute: Conducts absolute MDS. Distances between points in ordination space should be as close as possible to observed distances.

ratio: Ordination distances are proportional to observed distances.

linear: Ordination distances are a linear function of observed distances. Uses the technique of Heiser (1991) to avoid negative ordination distances.

monotone: Constrains ordination distances simply to be ranked the same as observed distance. Typically referred to as non-metric multidimensional scaling. **Uses isotonic regression developed by Nelle Varoquaux and Andrew Tulloch from scikit-learn.**

ntry: integer Number of random starts used to avoid local minima. The returned solution is the one with the lowest final stress.

tolerance: float Minimum step size causing a break in the minimization of stress. Default = 1E-4.

maxiter: integer Maximum number of iterations to attempt before breaking if no solution is found.

init: numpy.ndarray Initial positions for the first random start. If none, the initial position of the first try is taken as the site locations from classical scaling, Principle Coordinates Analysis.

Attributes

`scores`
Final scores for each object along the ordination axes.

`stress`
Final stress.

`obs`
The observed distance matrix.

`transform`
Which transformation was used.

Methods

classmethod `biplot`(*coords=False*, *xax=1*, *yax=2*, *siteNames=True*, *descriptors=None*, *descripNames=None*, *spCol='r'*, *siteCol='k'*, *spSize=12*, *siteSize=12*)
Produces a biplot of the given MDS axes.

coords: [True | False] If True, returns a dictionary of the plotted axes, where 'Objects' gives the coordinates of objects and 'Descriptors' gives the coordinates of the descriptors, if any.

xax: integer Specifies MDS axis to plot on the x-axis.

yax: integer Specifies MDS axis to plot on the y-axis.

descriptors: numpy.ndarray or pandas.DataFrame A matrix of the original descriptors used to create the distance matrix. Descriptors (*i.e.* species) scores are calculated as the weighted average of site scores.

descripNames: list A list containing the names of each descriptor. If None, inherits from the column names of the pandas.DataFrame or assigned integer values.

spCol: string Color of species text.

siteCol: string Color of site text.

spSize: integer Size of species text.

siteSize: integer Size of site text.

classmethod `shepard`(*xax=1*, *yax=2*)

Plots a Shepard diagram of Euclidean distances among objects in reduced space vs. original distance calculations. xax and yax as above.

classmethod `correlations`()

Returns a pandas.Series of correlations between observed and fitted distances for each site.

classmethod `correlationPlots`(*site=None*)

Produces a plot of observed vs. fitted distances for a given site. If site=None, then all sites are plotted on a single graph.

Examples

Conduct nMDS on the 'dune' data:

```
import ecopy as ep
dunes = ep.load_data('dune')
dunes_T = ep.transform(dunes, 'wisconsin')
dunes_D = ep.distance(dunes_T, 'bray')
dunesMDS = ep.MDS(dunes_D, transform='monotone')
```

Plot the Shepard diagram:

```
dunesMDS.shepard()
```

Check the correlations for observed vs. fitted distances:

```
dunesMDS.correlationPlots()
```

Make a biplot, showing species locations:

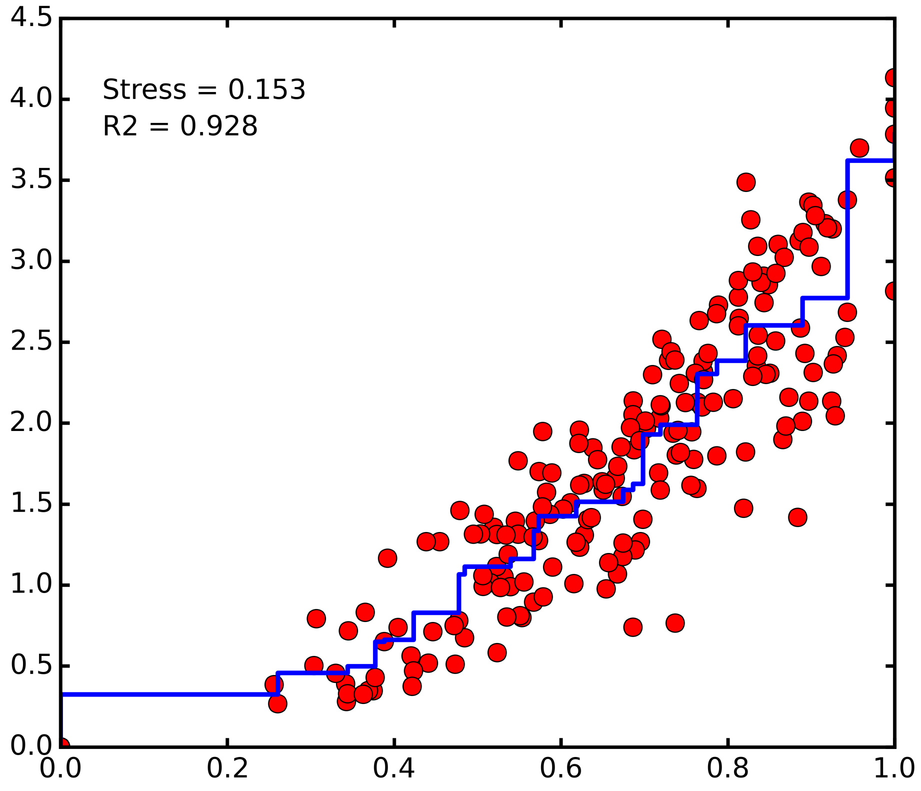
```
dunesMDS.biplot(descriptors=dunes_T)
```

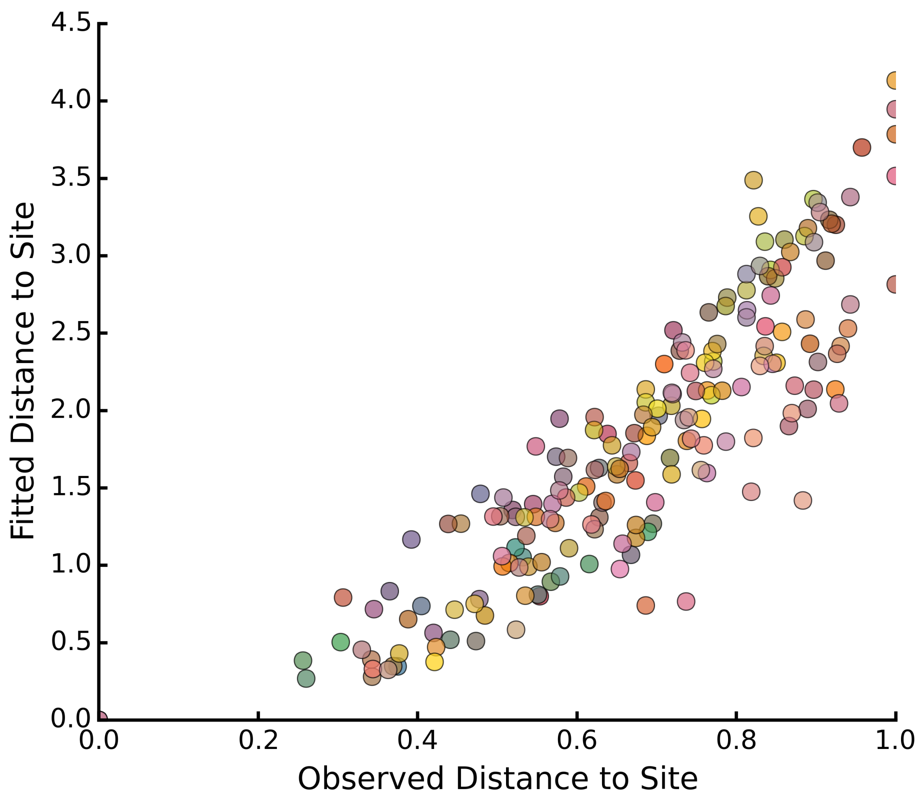
class `hillsmith`(*mat*, *wt_r=None*, *ndim=2*)

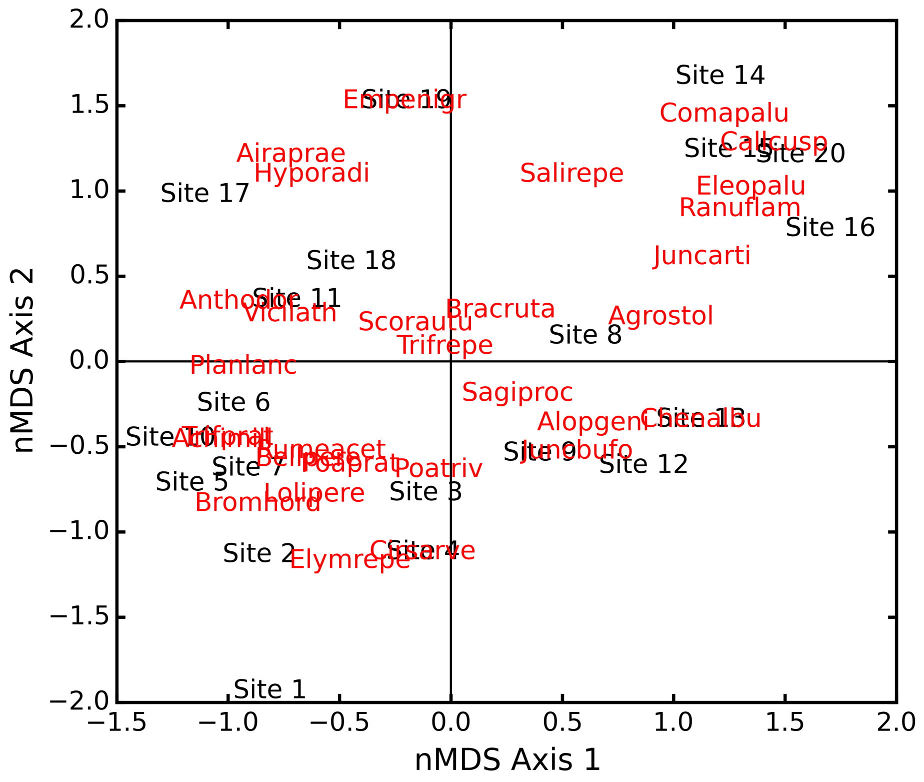
Takes an input matrix and performs ordination described by Hill and Smith (1976). Returns an object of class *`hillsmith`*, with several methods and attributes. NOTE: This will NOT work when rows < columns or with missing values.

Parameters

mat: pandas.DataFrame A matrix for ordination, where objects are rows and descriptors/variables as columns. Can have mixed data types (both quantitative and qualitative). If all columns are quantitative, this method is equivalent to PCA. If all columns are qualitative, this method is equivalent to MCA. Should not be used with ordered factors. In order to account for factors, this method creates dummy variables for each factor and then assigns weights to each dummy column based on the number of observations in each column.


Stress = 0.153
R2 = 0.928
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Fitted Distance to Site
Observed Distance to Site
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0



wt_r: list or numpy.ndarray Optional vector of row weights.

ndim: int Number of axes and components to save.

Attributes

`evals`
Eigenvalues in order of largest to smallest.

`pr_axes`
The principle axes of each column.

`row_coords`
Row coordinates along each principle axis.

`pr_components`
The principle components of each row.

`column_coords`
Column coordinates along each principle component.

Methods

classmethod `summary`()
Returns a data frame containing information about the principle axes.

classmethod `biplot`(*invert=False*, *xax=1*, *yax=2*, *obsNames=True*)
Create a biplot using a specified transformation.

invert: [True|Fasle] If False (default), plots the row coordinates as points and the principle axes of each column as arrows. If True, plots the column coordinates as points and the principle components of each row as arrows.

xax: integer Specifies which PC axis to plot on the x-axis.

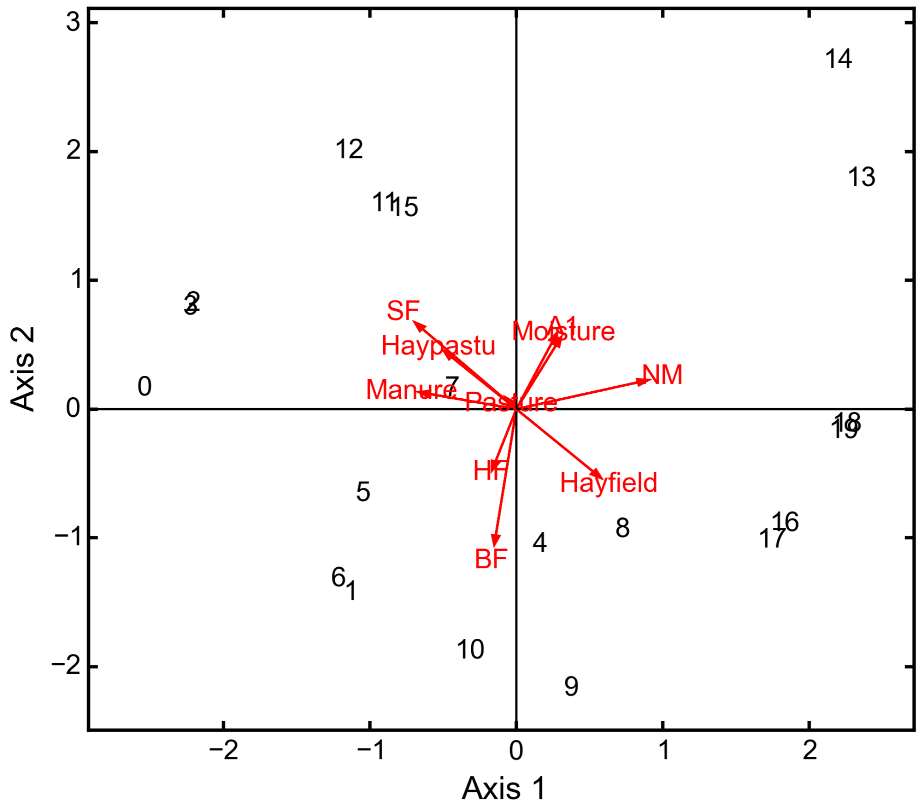
yax: integer Specifies which PC axis to plot on the y-axis.

obsNames: [True | False] Denotes whether to plot a scatterplot of points (False) or to actually show the names of the observations, as taken from the DataFrame index (True).

Examples

Hill and Smith analysis of the dune_env data:

```
import ecopy as ep
dune_env = ep.load_data('dune_env')
dune_env = dune_env[['A1', 'Moisture', 'Manure', 'Use', 'Management']]
print(ep.hillsmith(dune_env).summary().iloc[:,:2])

                        Axis 1    Axis 2
        Std. Dev   1.594392  1.363009
        Prop Var   0.317761  0.232224
        Cum Var    0.317761  0.549985

ep.hillsmith(dune_env).biplot(obsNames=False, invert=False)
```



ord_plot (*x*, *groups*, *y=None*, *colors=None*, *type='Hull'*, *label=True*, *showPoints=True*, *xlab='Axis 1'*, *ylab='Axis 2'*)
Delineates different groups in ordination (or regular) space.

Parameters

x: numpy.ndarray, pandas.DataFrame, pandas.Series (*required*) Coordinates to be plotted. Can be either a one or two column matrix. If only one column, then y must be specified.

groups: list, pandas.DataFrame, pandas.Series (*required*) Factor denoting group identification

y: numpy.ndarray, pandas.DataFrame, pandas.Series y coordinates to be plotted. Can only have one column, and must be specified is x is only one column.

colors: string, list, pandas.Series, pandas.DataFrame Gives custom colors for each group. Otherwise default colors are used.

type: ['Hull' | 'Line'] 'Hull' produces a convex hull, whereas 'Line' produces lines connected to the centroid for each point.

label: [True | False] Whether or not a label should be shown at the center of each group.

showPoints: [True | False] Whether or not the points should be shown.

xlab: string Label for the x-axis.

ylab: string Label for the y-axis.

Example

Generate fake data simulating ordination results:

```
import numpy as np
import ecopy as ep
import matplotlib.pyplot as plt

nObs = 10
X = np.random.normal(0, 1, 10*2)
Y = np.random.normal(0, 1, 10*2)
GroupID = ['A']*nObs + ['B']*nObs

Z = np.vstack((X, Y)).T
```

Make a convex hull plot where groups are red and blue:

```
ep.ord_plot(x=Z, groups=GroupID, colors=['r', 'b'])
```

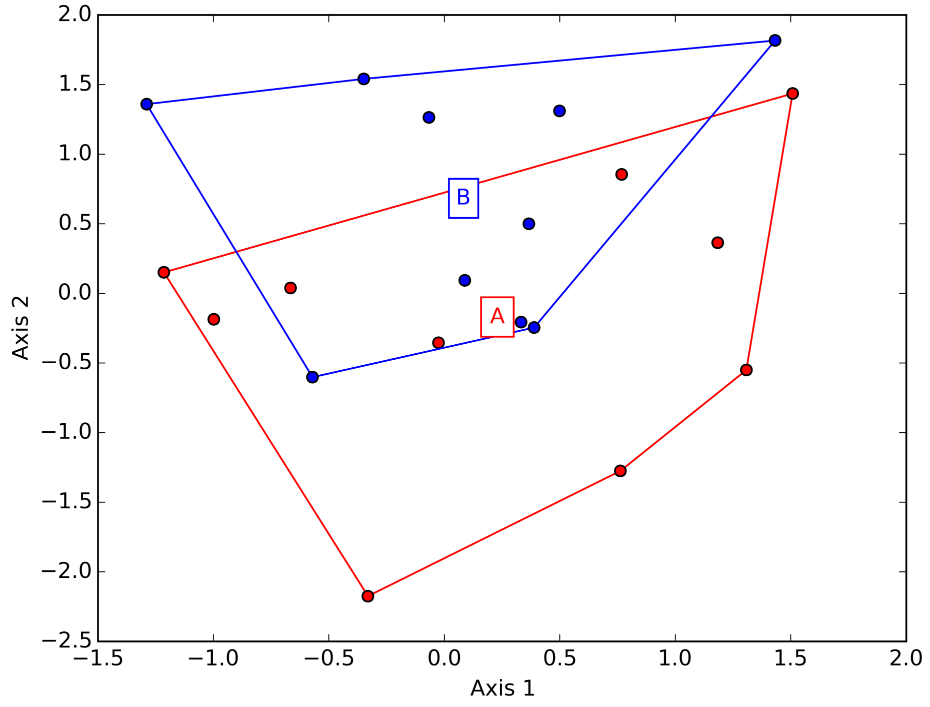
Make a line plot with coordinates in different matrices. Remove the points and the labels:

```
ep.ord_plot(x=X, y=Y, groups=GroupID, type='Line', xlab='PC1', ylab='PC2',␣
↪showPoints=False, label=False)
```

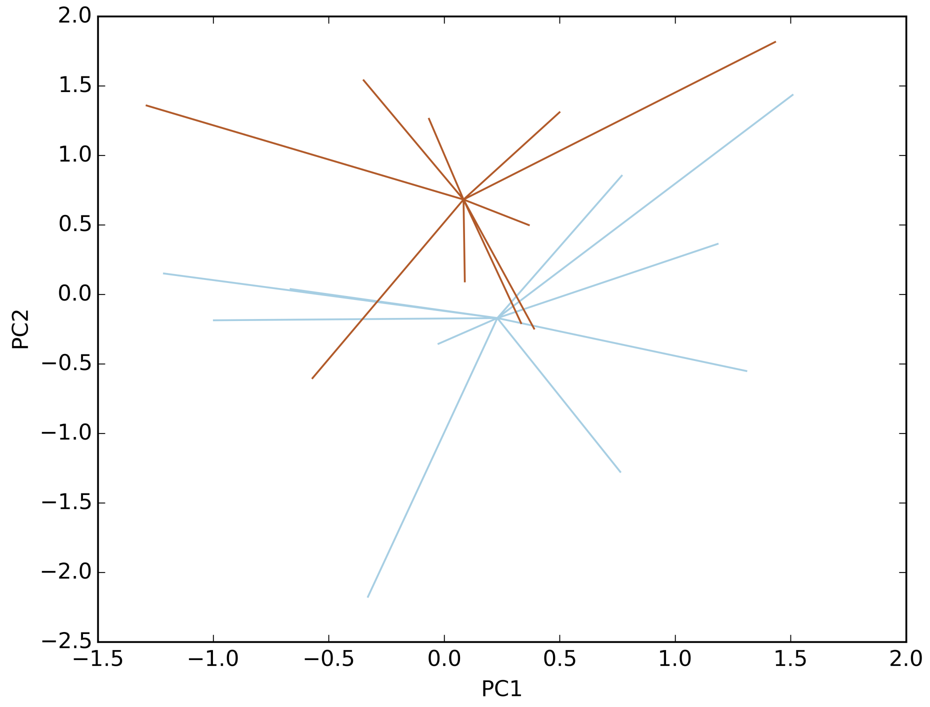
Matrix Comparisons

Ecopy contains several methods for comparing matrices. Some of these are similar to ordination, while others are more traditional null hypothesis testing.

- `Mantel` (Mantel test)
- `anosim` (Analysis of similarity)
- `simper()` (Percentage similarity calculations)
- `procrustes_test` (Procrustes test of matrix correlations)
- `corner4` (Fourth corner analysis)
- `rlq` (RLQ analysis)
- `rda` (RDA analysis)
- `cca` (CCA analysis)



B
A
Axis 1
Axis 2



PC1
PC2

- `ccor` (CCor analysis)

class Mantel (*d1*, *d2*, *d_condition=None*, *test='pearson'*, *tail='both'*, *nperm=999*)

Takes two distance matrices for a Mantel test. Returns object of class `Mantel`. Calculates the cross-product between lower triangle matrices, using either standardized variables or standardized ranks. The test statistics is the cross-product is divided by

$$\frac{n(n-1)}{2} - 1$$

where *n* is the number of objects.

If d_condition is provided, then `Mantel` conducts a partial Mantel test holding d_condition constant (Legendre and Legendre 2012). Permutations are conducted using the residual matrix as described by Legendre (2000).

Parameters

d1: numpy.ndarray or pandas.DataFrame First distance matrix.

d2: numpy.nadarray or pandas.DataFrame Second distance matrix.

test: ['pearson' | 'spearman'] 'pearson' performs Mantel test on standardized variables.

'spearman' performs Mantel test on standardized ranks.

tail: ['both' | 'greater' | 'lower'] 'greater' tests the one-tailed hypothesis that correlation is greater than predicted.

'lower' tests hypothsis that correlation is lower than predicted.

'both' is a two-tailed test.

nperm: int Number of permutations for the test.

Attributes

r_obs
Observed correlation statistic.

pval
p-value for the given hypothesis.

tail
The tested hypothesis.

test
Which of the statistics used, 'pearson' or 'spearman'.

perm
Number of permutations.

Methods

classmethod summary ()
Prints a summary output table.

Examples

Load the data:

```
import ecopy as ep
v1 = ep.load_data('varespec')
v2 = ep.load_data('varechem')
```

Standardize the chemistry variables and calculate distance matrices:

```
v2 = v2.apply(lambda x: (x - x.mean())/x.std(), axis=0)
dist1 = ep.distance(v1, 'bray')
dist2 = ep.distance(v2, 'euclidean')
```

Conduct the Mantel test:

```
mant = ep.Mantel(dist1, dist2)
print(mant.summary())

Pearson Mantel Test
Hypothesis = both

Observed r = 0.305      p = 0.004
999 permutations
```

class `anosim`(*dist*, *factor1*, *factor2=None*, *nested=False*, *nperm=999*)

Conducts analysis of similarity (ANOSIM) on a distance matrix given one or two factors (groups). Returns object of *`anosim`*. Calculates the observed R-statistic as

$$R = \frac{r_b - r_w}{\frac{n(n-1)}{4}}$$

where r_w is the average within-group ranked distances, r_b is the average between-group ranked distances, and *n* is the number of objects (rows) in the distance matrix. The factor is then randomly permuted and R recalculated to generate a null distribution.

Parameters

dist: numpy.ndarray or pandas.DataFrame Square-symmetric distance matrix.

factor1: numpy.nadarray or pandas.Series or pandas.DataFrame First factor.

factor2: numpy.nadarray or pandas.Series or pandas.DataFrame Second factor.

nested: [True | False] Whether factor1 is nested within factor2. If False, then factor1 and factor2 are permuted independently. If Tue, then factor1 is permuted only within groupings of factor2.

nperm: int Number of permutations for the test.

Attributes

`r_perm1`
Permuted R-statistics for factor1.

`r_perm2`
Permuted R-statistics for factor1.

`R_obs1`
Observed R-statistic for factor1.

`R_obs2`
Observed R-statistic for factor2.

`pval`
List of p-values for factor1 and factor2.

`perm`
Number of permutations.

Methods

classmethod `summary`()
Prints a summary output table.

classmethod `plot` ()

Plots a histogram of R values.

Examples

Load the data:

```
import ecopy as ep
data1 = ep.load_data('dune')
data2 = com.load_data('dune_env')
```

Calculate Bray-Curtis dissimilarity on the 'dune' data, save the 'Management' factor as factor1 and generate factor2:

```
duneDist = ep.distance(data1, 'bray')
group1 = data2['Management']
group2map = {'SF': 'A', 'BF': 'A', 'HF': 'B', 'NM': 'B'}
group2 = group1.map(group2map)
```

Conduct the ANOSIM:

```
t1 = ep.anosim(duneDist, group1, group2, nested=True, nperm=9999)
print(t1.summary())

ANOSIM: Factor 1
Observed R = 0.299
p-value = 0.0217
9999 permutations

ANOSIM: Factor 2
Observed R = 0.25
p-value = 0.497
9999 permutations

t1.plot()
```

`simper` (*data*, *factor*, *spNames=None*)

Conducts a SIMPER (percentage similarity) analysis for a site x species matrix given a grouping factor. Returns a pandas.DataFrame containing all output for each group comparison. Percent similarity for each species is calculated as the mean Bray-Curtis dissimilarity of each species, given by:

$$\Delta_i = \frac{|y_{ik} - y_{ij}|}{\sum_i^n (y_{ik} + y_{ij})}$$

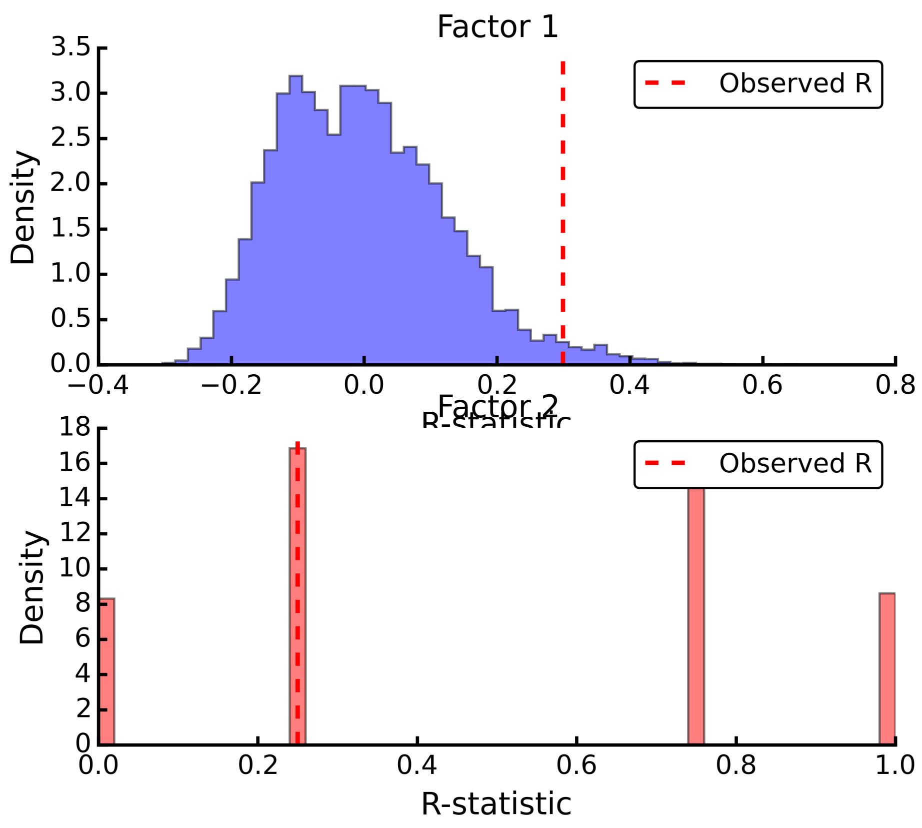
The denominator is the total number of individuals in both sites, y_{ik} is the number of individuals of species *i* in site *k*, and y_{ij} is the number of individuals in site *j*. This is performed for every pairwise combination of sites across two groups and then averaged to yield the mean percentage similarity of the species. This function also calculates the standard deviation of the percentage similarity, the signal to noise ratio (mean / sd) such that a higher ratio indicates more consistent difference, the percentage contribution of each species to the overall difference, and the cumulative percentage difference.

The output is a multi-indexed DataFrame, with the first index providing the comparison and the second index providing the species. The function lists the index comparison names as it progresses for reference.

Parameters

data: numpy.ndarray or pandas.DataFrame A site *x* species matrix.

factor: numpy.nadarray or pandas.Series or pandas.DataFrame or list Grouping factor.



spNames: list List of species names. If data is a pandas.DataFrame, then spNames is inferred as the column names. If data is a np.ndarray, then spNames is given integer values unless this argument is provided.

Examples

Conduct SIMPER on the ANOSIM data from above:

```
import ecopy as ep

data1 = ep.load_data('dune')
data2 = com.load_data('dune_env')
group1 = data2['Management']
fd = ep.simper(np.array(data1), group1, spNames=data1.columns)

Comparison indices:
BF-HF
BF-NM
BF-SF
HF-NM
HF-SF
NM-SF

print(fd.ix['BF-NM'])

 sp_mean  sp_sd  ratio  sp_pct  cumulative
Lolipere     9.07   2.64   3.44   12.43       12.43
Poatriv      5.47   4.46   1.23    7.50       19.93
Poaprat      5.25   1.81   2.90    7.19       27.12
Trifrepe     5.13   2.76   1.86    7.03       34.15
```

```
Bromhord     3.97   2.92   1.36    5.44        39.59
Bracruta     3.57   2.87   1.24    4.89        44.48
Eleopalu     3.38   3.57   0.95    4.63        49.11
Agrostol     3.34   3.47   0.96    4.58        53.69
Achimill     3.32   2.34   1.42    4.55        58.24
Scorautu     3.14   2.03   1.55    4.30        62.54
Anthodor     2.81   3.29   0.85    3.85        66.39
Planlanc     2.73   2.19   1.25    3.74        70.13
Salirepe     2.68   2.93   0.91    3.67        73.80
Bellpere     2.35   1.91   1.23    3.22        77.02
Hyporadi     2.17   2.45   0.89    2.97        79.99
Ranuflam     2.03   2.28   0.89    2.78        82.77
Elymrepe     2.00   2.93   0.68    2.74        85.51
Callcusp     1.78   2.68   0.66    2.44        87.95
Juncarti     1.77   2.60   0.68    2.43        90.38
Vicilath     1.58   1.45   1.09    2.17        92.55
Sagiproc     1.54   1.86   0.83    2.11        94.66
Airaprae     1.34   1.97   0.68    1.84        96.50
Comapalu     1.07   1.57   0.68    1.47        97.97
Alopgeni     1.00   1.46   0.68    1.37        99.34
Empenigr     0.48   1.11   0.43    0.66       100.00
Rumeacet     0.00   0.00    NaN    0.00       100.00
Cirsarve     0.00   0.00    NaN    0.00       100.00
Chenalbu     0.00   0.00    NaN    0.00       100.00
Trifprat     0.00   0.00    NaN    0.00       100.00
Juncbufo     0.00   0.00    NaN    0.00       100.00
```

class **`procrustes_test`**(*mat1*, *mat2*, *nperm=999*)

Conducts a procrustes test of matrix associations on two raw object x descriptor matrices. Returns an object of class `procrustes_test`. First, both matrices are column-centered. Then, each matrix is divided by the square root of its sum-of-squares. The test statistic m_{12}^2 is calculated as:

$$m_{12}^2 = 1 - (Trace\mathbf{W})^2$$

$\mathbf{W}$ is the diagonal matrix of eigenvalues for $\mathbf{X}'\mathbf{Y}$, which are the two transformed matrices. Then, rows of $\mathbf{X}$ are randomly permuted and the test statistic recalculated. The *p*-value is the the proportion of random test statistics less than the observed statistic.

Parameters

mat1: numpy.ndarray or pandas.DataFrame A raw object x descriptor (site x species) matrix.

factor1: numpy.nadarray or pandas.DataFrame A raw object x descriptor (site x descriptor) matrix.

nperm: int Number of permutations in the test.

Attributes

`m12_obs`
Observed m_{12}^2 statistic.

`pval`
p-value.

`perm`
Number of permutations.

Methods

classmethod **`summary`**()
Prints a summary output table.

Examples

Load the data and run the Mantel test:

```
import ecopy as ep

d1 = ep.load_data('varespec')
d2 = ep.load_data('varechem')
d = ep.procrustes_test(d1, d2)
print(d.summary())

m12 squared = 0.744
p = 0.00701
```

class corner4 (*mat1*, *mat2*, *nperm=999*, *model=1*, *test='both'*, *p_adjustment=None*)

Conducts fourth corner analysis examining associations between species traits and environmental variables. Species traits are given in a species x trait matrix **Q**, species abundances given in a site x species matrix **L**, and environmental traits given in a site x environment matrix **R**. The general concept of fourth corner analysis is to find matrix **D**:

$$\begin{bmatrix} \mathbf{L} & \mathbf{R} \\ \mathbf{Q}' & \mathbf{D} \end{bmatrix}$$

In a simple case, **R** and **Q** contain one environmental variable and one species trait. An expanded correspondance matrix is created following Dray and Legendre (2008). The association between **R** and **Q** is the calculated as follows:

- If both variables are quantitative, then association is described by Pearson's correlation coefficient *r*
- If both variables are qualitative, then association is described by χ^2 from a contingency table (see Dray and Legendre 2008, Legendre and Legendre 2011)
- If one variable is quantitative but the other is qualitative, then association is described using the *F*-statistic.

Significance of the statistics is determined using one of four permutation models (see below).

If **R** and **Q** contain more than one variable or trait, then the test iterates through all possible environment-trait combinations. The method automatically determines the appropriates statistics, depending on the data types (float=quantitative or object=qualitative). **NOTE**: As of now, this is quite slow if the number of traits and/or environmental variables is large.

Parameters

R: pandas.DataFrame A site x variable matrix containing environmental variables for each site. pandas.Series NOT allowed.

L: numpy.nadarray or pandas.DataFrame A site x species matrix of either presence/absence or abundance. Only integer values allowed.

Q: pandas.DataFrame A species x trait matrix containing trait measurements for each species. pandas.Series NOT allowed.

nperm: int Number of permutations in the test.

model: [1 | 2 | 3 | 4] Which model should be used for permutations.

1: Permutes within columns of **L** only (that is, shuffles species among sites).

2: Permutes entire rows of **L** (that is, shuffles entire species assemblages).

3: Permutes within rows of **L** (that is, shuffles the distribution of individuals within a site).

4: Permutes entire columns of **L** (that is, shuffles a species' distribution among traits, while site distributions are kept constant).

test: ['both' | 'greater' | 'lower'] Which tail of the permutation distribution should be tested against the observed statistic.

p_adjustment: [None, 'bonferroni', 'holm', 'fdr']: Which adjustment should be used for multiple comparisons. 'bonferroni' uses Bonferronni correction, 'holm' uses the Bonferroni-Holm correction, and 'fdr' uses the False Discovery Rate correction.

Methods

classmethod `summary`()

Returns a pandas.DataFrame of output.

Examples

Run fourth corner analysis on the aviurba data from R's ade4 package:

```
import ecopy as ep

traits = ep.load_data('avi_traits')
env = ep.load_data('avi_env')
sp = ep.load_data('avi_sp')

fourcorn = ep.corner4(env, sp, traits, nperm=99, p_adjustment='fdr')
results = fourcorn.summary()

print(results[['Comparison','adjusted p-value']])
                Comparison  adjusted p-value
0          farms - feed.hab             1.000
1        farms - feed.strat             1.000
2          farms - breeding             1.000
3         farms - migratory             1.000
4      small.bui - feed.hab             0.322
5    small.bui - feed.strat             0.580
6      small.bui - breeding             1.000
7     small.bui - migratory             0.909
8       high.bui - feed.hab             0.111
...      .......                     ....
41   veg.cover - feed.strat             1.000
42     veg.cover - breeding             0.033
43    veg.cover - migratory             1.000
```

class `rlq`(*R*, *L*, *Q*, *ndim=2*)

Conducts RLQ analysis which examines associations between matrices **R** (site x environment) and **Q** (species x traits) as mediated by matrix **L** (site by species). In general, a matrix **D** is constructed by:

$$\mathbf{D} = \mathbf{R}'\mathbf{D_{row}}\mathbf{L}\mathbf{D_{col}}\mathbf{Q}$$

where $\mathbf{D_{row}}$ and $\mathbf{D_{col}}$ are diagonal matrices of row and column weights derived from matrix **L**. **L** is first transformed by dividing the matrix by the total number of individuals in the matrix. Column and row weights are given by the sum of columns and rows of the transformed matrix. Matrix **L** is then transformed by diving each column by the corresponding column weight, dividing each row by the corresponding row weight, and subtracting 1 from all elements. This transformed **L** matrix is used in the above equation to generate matrix **D**.

NOTE: Both **R** and **Q** can contain a mix of factor and quantitative variables. A dummy dataframe is constructed for both **R** and **Q** as in the Hill and Smith ordination procedure.

Matrix **D** is then subject to eigen decomposition, giving site (environment) and species (trait) scores, as well as loading vectors for both environmental and trait variables.

Parameters

R: pandas.DataFrame A site x environment matrix for ordination, where objects are rows and descriptors/variables as columns. Can have mixed data types (both quantitative and qualitative). In order to account for factors, this method creates dummy variables for each factor and then assigns weights to each dummy column based on the number of observations in each column.

L: pandas.DataFrame A site x species for ordination, where objects are rows and descriptors/variables as columns.

Q: pandas.DataFrame A species x trait matrix for ordination, where objects are rows and descriptors/variables as columns. Can have mixed data types (both quantitative and qualitative). In order to account for factors, this method creates dummy variables for each factor and then assigns weights to each dummy column based on the number of observations in each column.

ndim: int Number of axes and components to save.

Attributes

`traitVecs`
A pandas.DataFrame of trait loadings.

`envVecs`
A pandas.DataFrame of environmental loadings.

`normedTraits`
Species coordinates along each axis.

`normedEnv`
Site coordinates along each axis.

`evals`
Eigenvalues for all axes (not just saved ones).

Methods

classmethod `summary`()
Returns a data frame containing information about the principle axes.

classmethod `biplot`(*xax=1*, *yax=2*)
Create a biplot. The plot contains four subplots, one each for species scores, site scores, trait vectors, and environment vectors. Species scores are plotted from normedTraits, site scores are plotted from normedEnv, trait vectors are plotted from traitVecs, and environmental vectors are plotted from envVecs. Users can mix and match which vectors to overlay with which points manually using these four attributes.

xax: integer Specifies which PC axis to plot on the x-axis,

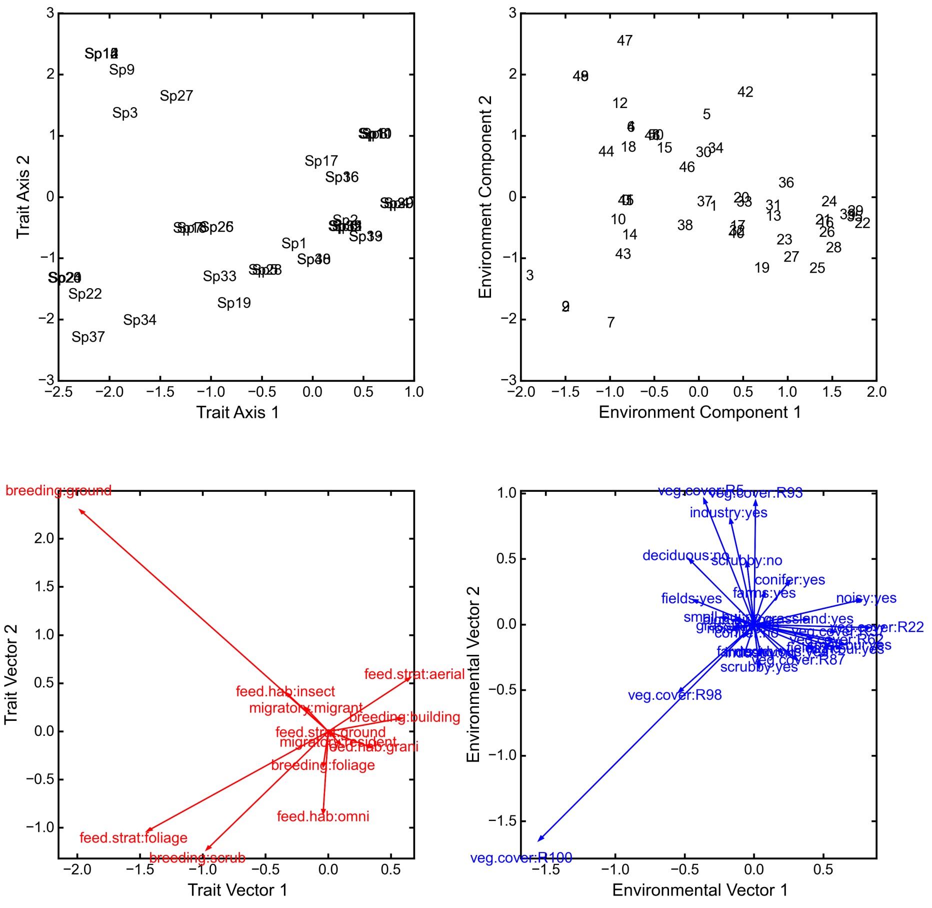
yax: integer Specifies which PC axis to plot on the y-axis.

Examples

RLQ analysis of the aviurba data:

```
vi_sp = ep.load_data('avi_sp')
avi_env = ep.load_data('avi_env')
avi_traits = ep.load_data('avi_traits')

rlq_test = ep.rlq(avi_env, avi_sp, avi_traits, ndim=2)
print(rlq_test.summary().iloc[:,:3])

                    Axis 1    Axis 2    Axis 3
Std. Dev  0.691580  0.376631  0.272509
Prop Var  0.657131  0.194894  0.102031
```

```
Cum Var    0.657131   0.852026   0.954056

rlq_test.biplot()
```



class rda (*Y*, *X*, *scale_y=True*, *scale_x=False*, *design_x=False*, *varNames_y=None*, *varNames_x=None*, *rowNames=None*, *pTypes=None*)

Conducts RDA analysis which examines the relationship between sites (rows) based on their species compositions (columns). This information is contained in matrix **Y**. However, the relationships between sites are constrained by environmental predictors contained in matrix **X**.

RDA performs a multivariate regression of **Y** against **X**, yielding linear predictors **B**:

$$\mathbf{B} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{Y}$$

These linear predictors are used to generated predicted values for each species at each site:

$$\hat{\mathbf{Y}} = \mathbf{X}\mathbf{B}$$

The variance-covariance matrix of $\hat{\mathbf{Y}}$ is then subject to eigen-analysis, yielding eigenvalues **L** and eigenvectors **U** of the predicted species values. Three new matrices are calculated:

$$\mathbf{F} = \mathbf{Y}\mathbf{U}\mathbf{Z} = \hat{\mathbf{Y}}\mathbf{U}\mathbf{C} = \mathbf{B}\mathbf{U}$$

Species scores are given by $\mathbf{UL}^{-0.5}$. Site scores are given by $\mathbf{FL}^{-0.5}$. The scores of each predictor are given in matrix **C**.

The residuals from the regression are then subject to PCA to ordinate the remaining, unconstrained variance.

Parameters

Y: pandas.DataFrame or numpy.ndarray A site *x* species for ordination, where objects are rows and descriptors/variables as columns.

X: pandas.DataFrame or numpy.ndarray A site x environment matrix for ordination, where objects are rows and descriptors/variables as columns. Only the pandas.DataFrame can have mixed data types (both quantitative and qualitative). In order to account for factors, this method creates dummy variables for each factor and then assigns weights to each dummy column based on the number of observations in each column.

scale_x: [True | False] Whether or not the matrix Y should be standardized by columns.

scale_y: [True | False] Whether or not the matrix X should be standardized by columns.

design_x: [True | False] Whether or not X has already been transformed to a design matrix. This enables the user to formulate more complicated regressions that include interactions or higher order variables.

varNames_y: list A list of variables names for each column of Y. If None, then the column names of Y are used.

varNames_x: list A list of variables names for each column of X. If None, then the column names of X are used.

rowNames: list A list of site names for each row. If none, then the index values of Y are used.

pTypes: list A list denoting whether variables in X are quantitative ('q') or factors ('f'). Can usually be ignored.

Attributes

`spScores`
A pandas.DataFrame of species scores on each RDA axis.

`linSites`
A pandas.DataFrame of linearly constrained site scores.

`siteScores`
A pandas.DataFrame of site scores on each RDA axis.

`predScores`
A pandas.DataFrame of predictor scores on each RDA axis.

`RDA_evals`
Eigenvalues for each RDA axis.

`corr`
Correlation of each predictor with each RDA axis.

`resid_evals`
Eigenvalues for residual variance.

`resid_spScores`
A pandas.DataFrame of species scores on PCA of residual variance.

`resid_siteScores`
A pandas.DataFrame of site scores on PCA of residual variance.

`imp`
Summary of importance of each RDA and PCA axis.

Methods

classmethod summary ()

Returns a data frame containing summary information.

classmethod anova (*nperm=999*)

Conducts a permutational test of global significance. The **F**-statistic is the ratio of contrained variance to unconstrained variance, where each is divided by their respective degrees of freedom. The original *Y* matrix is permuted by row and a distribution of **F**-statistics is built. The *p*-value is the proportion of permuted **F**-statistics that is greater than the observed.

classmethod triplot (*xax=1*, *yax=2*)

Creates a triplot of species scores, site scores, and predictor variable loadings. If predictors are factors, they are represented by points. Quantitative predictors are represented by arrows.

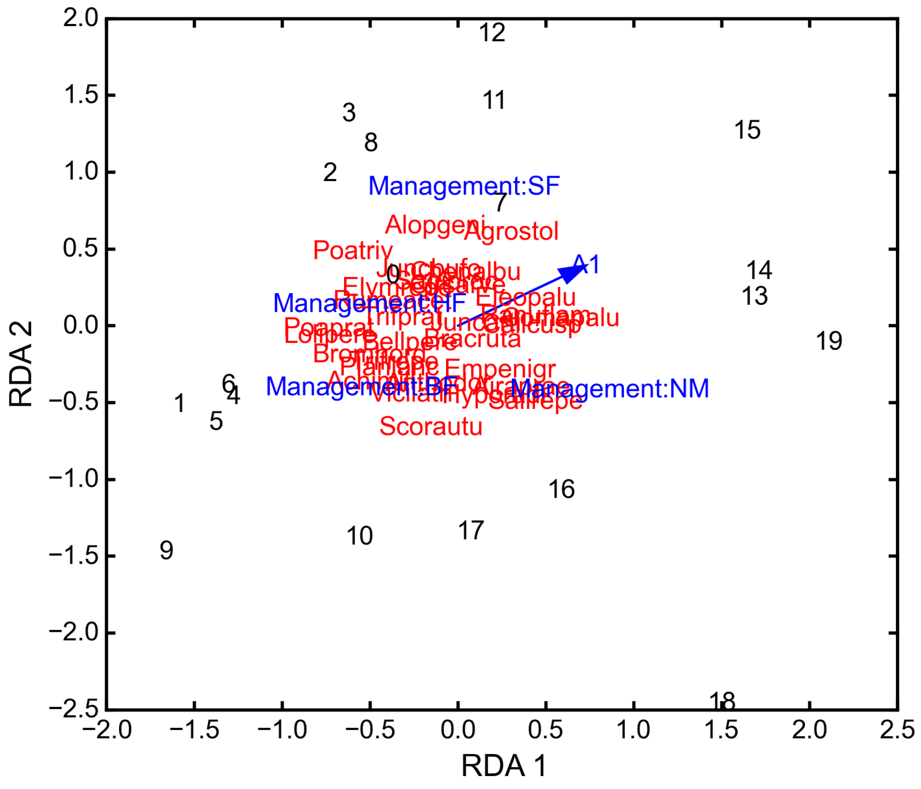
xax: integer Specifies which RDA axis to plot on the x-axis.

yax: integer Specifies which RDA axis to plot on the y-axis.

Examples

RDA on dune data:

```
import ecopy as ep

dune = ep.load_data('dune')
dune_env = ep.load_data('dune_env')

RDA = ep.rda(dune, dune_env[['A1', 'Management']])
RDA.triplot()
```



class cca (*Y*, *X*, *varNames_y=None*, *varNames_x=None*, *rowNames=None*, *scaling=1*)

Conducts CCA analysis which examines the relationship between sites (rows) based on their species compositions (columns). This information is contained in matrix $\mathbf{Y}$. However, the relationships between sites are constrained by environmental predictors contained in matrix $\mathbf{X}$.

CCA first transforms the species matrix $\mathbf{Y}$ into matrix $\bar{\mathbf{Q}}$ as in correspondance analysis. The predictor matrix $\mathbf{X}$ is then standardized using the row weights from matrix $\mathbf{Y}$ to calculate the mean and standard deviation of each column, resulting in a new matrix $\mathbf{X}_{scale}$. This matrix, along with a diagonal matrix of row weghts $\mathbf{D}$ is used in a multivariate regression of $\bar{\mathbf{Q}}$ against $\mathbf{X}_{scale}$, yielding linear predictors $\mathbf{B}$:

$$\mathbf{B} = (\mathbf{X}'_{scale}\mathbf{D}\mathbf{X}_{scale})^{-1}\mathbf{X}'_{scale}\mathbf{D}^{\mathbf{0.5}}\mathbf{Y}$$

These linear predictors are used to generated predicted values for each species at each site:

$$\hat{\mathbf{Y}} = \mathbf{D}^{\mathbf{0.5}}\mathbf{X}_{\mathbf{scale}}\mathbf{B}$$

The cross-product matrix of $\hat{\mathbf{Y}}$ is then subject to eigen-analysis, yielding eigenvalues $\mathbf{L}$ and eigenvectors $\mathbf{U}$ of the predicted species values. Five new matrices are calculated using diagonal matrices of row $\mathbf{D}_r$ and column $\mathbf{D}_c$ weights:

$$\hat{\mathbf{U}} = \bar{\mathbf{Q}}\mathbf{U}\mathbf{L}^{-\mathbf{0.5}}\mathbf{V} = \mathbf{D}_{\mathbf{c}}^{-\mathbf{0.5}}\mathbf{U}\hat{\mathbf{V}} = \mathbf{D}_{\mathbf{r}}^{-\mathbf{0.5}}\hat{\mathbf{U}}\mathbf{F} = \hat{\mathbf{V}}\mathbf{L}^{\mathbf{0.5}}\hat{\mathbf{F}} = \mathbf{V}\mathbf{L}^{\mathbf{0.5}}$$

In scaling type 1, species scores are given by $\mathbf{V}$ and site scores are given by $\mathbf{F}$. Fitted site scores are given by $\mathbf{D_r}\hat{\mathbf{Y}}\mathbf{U}$. To calculate the predictor scores, the fitted site scores are standardized using row weights as was done for $\mathbf{X}_{scale}$, yielding $\mathbf{Z}_{scale}$. Predictor variable scores are then calculated as $\mathbf{X}'_{scale}\mathbf{D_r}\mathbf{Z}_{scale}\mathbf{L}^{\mathbf{0.5}}$.

In scaling type 2, species scores are given by $\hat{\mathbf{F}}$ and site scores are given by $\hat{\mathbf{V}}$. Fitted site scores are given by $\mathbf{D_r}\hat{\mathbf{Y}}\mathbf{U}\mathbf{L}^{-\mathbf{0.5}}$. To calculate the predictor scores, the fitted site scores are standardized using row weights as was done for $\mathbf{X}_{scale}$, yielding $\mathbf{Z}_{scale}$. Predictor variable scores are then calculated as $\mathbf{X}'_{scale}\mathbf{D_r}\mathbf{Z}_{scale}$.

Residuals from the constrained ordination are available in order to subject them to CA.

Parameters

Y: pandas.DataFrame or numpy.ndarray A pandas.DataFrame or numpy.ndarray containing species abundance data (site *x* species).

X: pandas.DataFrame or numpy.ndarray A pandas.DataFrame or numpy.ndarray containing predictor variables for constrained ordination (site *x* variable).

varNames_y: list A list of variables names for each column of Y. If None, then the column names of Y are used.

varNames_x: list A list of variables names for each column of X. If None, then the column names of X are used.

rowNames: list A list of site names for each row. If none, then the index values of Y are used.

scaling: [1 | 2] Which scaling should be used. See above.

Attributes

`r_w`
Row weights.

`c_w`
Column weights.

`evals`
Constrained eigenvalues.

U
Constrained eigenvectors.

resid
A pandas.DataFrame of residuals from the constrained ordination.

spScores
A pandas.DataFrame of species scores.

siteScores
A pandas.DataFrame of site scores.

siteFitted
A pandas.DataFrame of constrained site scores.

varScores
A pandas.DataFrame variable scores.

res_evals
Residual eigenvalues

res_evecs
Residual eigenvectors

Methods

classmethod summary()
Returns summary information of each CA axis.

classmethod anova(*nperm=999*)
Conducts a permutational test of global CCA significance. The observed **F**-statistic is the ratio of constrained to unconstrained variance, each divided by their respective degrees of freedom. The original **Y** matrix is permuted by rows, CCA recomputed and a new **F**-statistic calculated for each permutation. The *p*-value is the proportion of permuted **F** values that are greater than the observed value.

classmethod triplot(*xax=1*, *yax=2*)
Creates a triplot of species scores, site scores, and predictor variable loadings.

xax: integer Specifies which CA axis to plot on the x-axis.

yax: integer Specifies which Ca axis to plot on the y-axis.

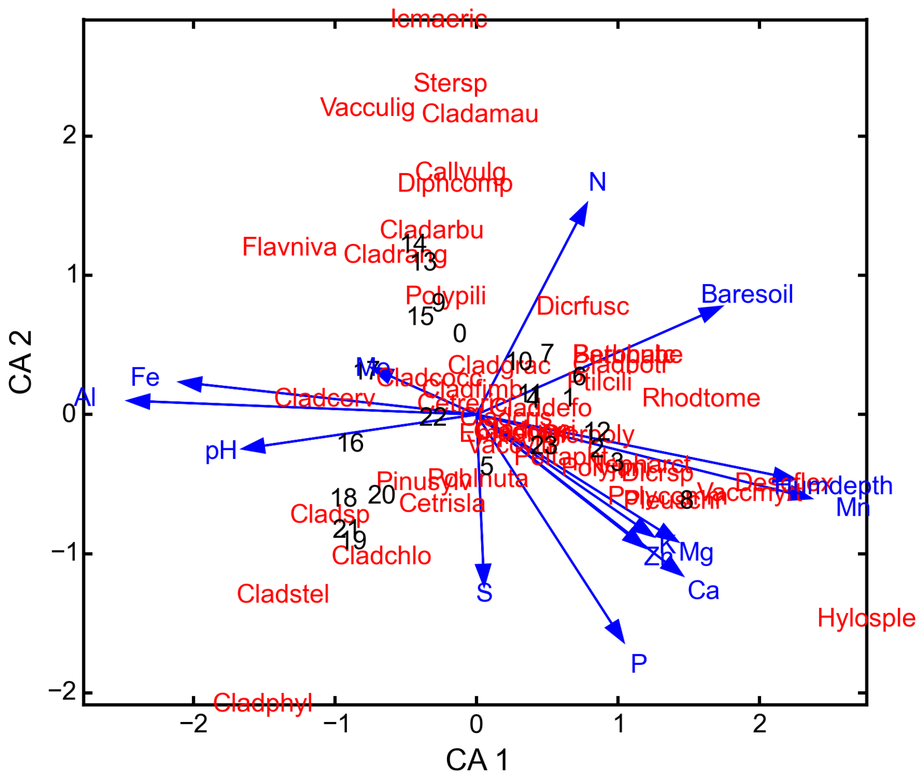
Examples

CCA on varespec data:

```
import ecopy as ep

varespec = ep.load_data('varespec')
varechem = ep.load_data('varechem')

cca_fit = ep.cca(varespec, varechem)
CCA.triplot()
```

class ccor(*self*, *Y1*, *Y2*, *varNames_1=None*, *varNames_2=None*, *stand_1=False*, *stand_2=False*, *siteNames=None*)
Conducts canonical correlation analysis (CCor) which examines the relationship between matrices **Y1** and **Y2**. CCor first calculates the variance and covariance matrices for both **Y1** and **Y2**, where $\mathbf{S}_{11}$ is the variance-covariance matrix of **Y1**, $\mathbf{S}_{22}$ is the variance-covariance matrix of **Y2**, and $\mathbf{S}_{12}$ is the covariance matrix of **Y1** and **Y2**.

A new matrix **K** is calculated as



$$\mathbf{K} = \mathbf{S}_{11}^c \mathbf{S}_{12} \mathbf{S}_{22}^c$$

where $\mathbf{S}_{11}^c$ is the Cholesky decomposition of $\mathbf{S}_{11}$ and same for $\mathbf{S}_{22}^c$.

CCor then uses SVD to calculate matrices **V**, **W**, and **U**, where **V** contains the left-hand eigenvectors, **W** contains the singular values, and **U** contains the right-hand eigenvectors. New matrices **C1** and **C2** are derived by **Y1V** and **Y2U**, respectively. Scores for matrices **Y1** are then

$$\text{Scores}_\mathbf{1} = \mathbf{Y1C1}$$

and the same for **Y2**. Variable loadings are the correlation between the original matrix and the scores.

Parameters

Y1: pandas.DataFrame or numpy.ndarray

A pandas.DataFrame or numpy.ndarray containing one set of variables.

Y2: pandas.DataFrame or numpy.ndarray

A pandas.DataFrame or numpy.ndarray containing a second set of variables.

varNames_1: list

A list of variables names for each column of **Y1**. If None, then the column names of **Y1** are used.

varNames_2: list

A list of variables names for each column of **Y2**. If None, then the column names of **Y2** are used.

siteNames: list

A list of site names for each row. If none, then the index values of **Y1** are used.

stand_1: [True | False]

Whether to standardize **Y1**.

stand_2: [True | False]

Whether to standardize **Y2**.

Attributes

`Scores1`
Site scores from matrix 1.

`Scores2`
Site scores from matrix 2.

`loadings1`
Variable loadings from matrix 1.

`loadings2`
Variable loadings from matrix 2.

`evals`
Eigenvalues.

Methods

classmethod `summary` ()
Returns summary information of each CA axis.

classmethod `biplot` (*matrix=1*, *xax=1*, *yax=2*)
Creates a biplot of site scores and predictor variable loadings.

matrix: [1 | 2] Which matrix, **Y1** or **Y2** to plot.

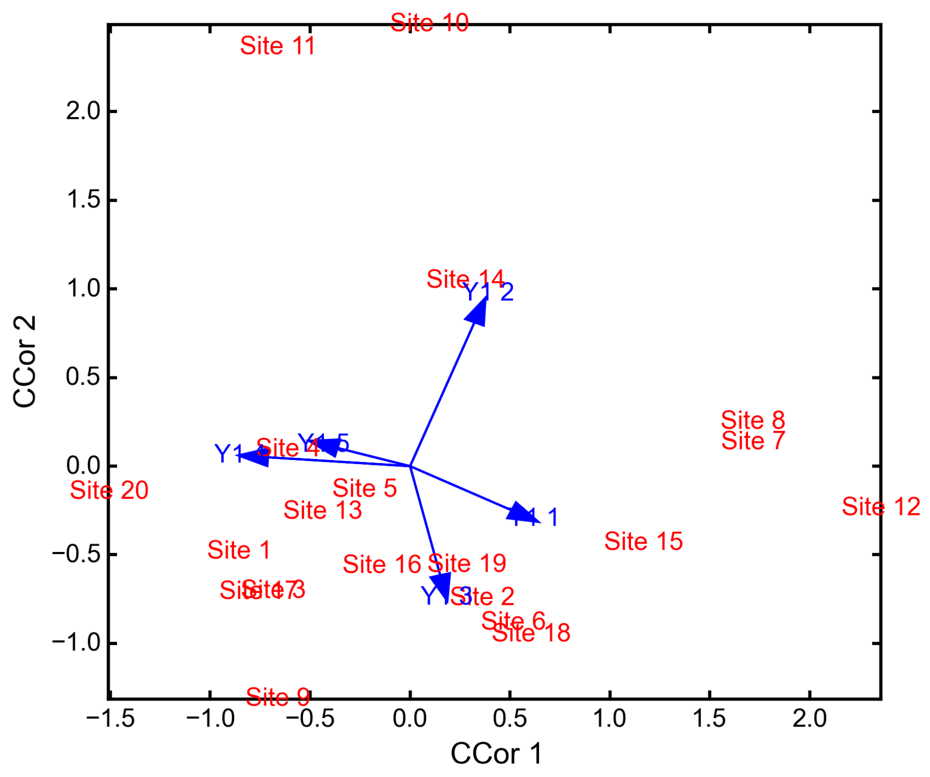
xax: integer Specifies which CCor axis to plot on the x-axis.

yax: integer Specifies which CCor axis to plot on the y-axis.

Examples

CCor analysis of random data:

```
import ecopy as ep
import numpy as np

Y1 = np.random.normal(size=20*5).reshape(20, 5)
Y2 = np.random.normal(size=20*3).reshape(20, 3)

cc = ep.ccor(Y1, Y2)
cc.summary()

Constrained variance = 1.37
Constrained variance explained be each axis
['0.722', '0.464', '0.184']
Proportion constrained variance
['0.527', '0.338', '0.135']

cc.biplot()
```



Regression

EcoPy provides a wrapper for scipy.optimize.leastsq. This wrapper allows users to specify a non-linear function and receive parameter estimates, statistics, log-likelihoods, and AIC.

- *nls*

class nls (*func*, *p0*, *xdata*, *ydata*)

nls takes a function (func), initial parameter estimates (p0), predictor variables (xdata), and a response (ydata) and passes these to scipy.optimize.leastsq. It returns an object of class *nls*.

Parameters

func: function A function that returns the quantity to be minimized. See documentation for scipy.optimize.leastsq.

p0: dictionary A dictionary of initial parameter estimates for every parameter to be estimated in the function.

xdata: numpy.ndarray A numpy.ndarray of predictor variables. See example below for how to include multiple predictors.

ydata: numpy.ndarray A numpy.ndarray of the response variable.

Attributes

cov

Variance-covariance matrix of parameters

inits

Initial parameter estimates

logLik

Log-likelihood of the function

nparm
Number of parameters estimated

parmEsts
Parameter estimates

parmSE
Standard error of each parameter

RMSE
Root mean square error

pvals
p-values of each parameter

tvals
t-values of each parameter

Methods

classmethod AIC (*k=2*)
Returns AIC for the given model. Argument k determines the correction applied to the number of parameters

classmethod summary ()
Returns a regression summary table

Examples

First, load the urchin data:

```
import ecopy as ep
import numpy as np

urchins = ep.load_data('urchins')
```

Next, make the X and Y matrices:

```
Y = np.array(urchins['Respiration])*24
X = np.array(urchins[['UrchinMass', 'Temp']])
```

Define the least-squares function to be optimized:

```
def tempMod(params, X, Y):
        a = params[0]
        b = params[1]
        c = params[2]
        mass = X[:,0]
        temp = X[:,1]
        yHat = a*mass**b*temp**c
        err = Y - yHat
        return(err)
```

Create a dictionary of initial estimates for each parameter:

```
p0 = {'a':1, 'b':1, 'c': 1}
```

Run the model and check the summary tables:

```
tMod = ep.nls(tempMod, p0, X, Y)
tMod.summary()
```

```
Non-linear least squares
Model: tempMod
Parameters:
        Estimate        Std. Error      t-value                 P(>|t|)
a        0.0002  0.0002  0.8037  0.4302
c        0.3346  0.1485  2.2533  0.0345
b        1.5209  0.3448  4.4112  0.0002

Residual Standard Error:  0.0371
Df: 22

tMod.AIC()

AIC:  -88.9664797962
```

References

Borg and Groenen (2005) Modern multidimensional scaling: theory and applications.

Clarke and Warwick (2001) Change in marine communities: an approach to statistical analysis and interpretation. PRIMER-E.

Doledec et al. (1996) Matching species traits to environmental variables: a new three-table ordination method. *Environmental and Ecological Statistics* 3:143-166.

Dray and Dufour (2007) The ade4 package: implementing the duality diagram for ecologists. *Journal of Statistical Software* 22.

Dray and Legendre (2008) Testing the species-traits-environment relationships: the fourth-corner problem revisited. *Ecology* 89:3400-3412.

Heiser (1991) A generalized majoration method for least squares multidimensional scaling of pseudodistances that may be negative. *Psychometrika* 56:7-27.

Hill (1974) Correspondance analysis: a neglected multivariate method. *Journal of the Royal Statistical Society - C* 23:340-354.

Hill and Smith (1976) Principle component analysis of taxonomic data with multi-state discrete characters. *Taxon* 25:349-255.

Jost (2007) Partitioning diversity into alpha and beta components. *Ecology* 88:2427-2439.

Legendre (2000) Comparison of permutation methods for the partial correlation and partial mantel tests. *Journal of Statistical Computation and Simulation* 67:37-73

Legendre and Legendre (2012) Numerical Ecology. Third Edition.

Little and Rubin (2002) Statistical analysis with missing data. Second Edition.

Pedregosa et al. (2001) Scikit-learn: marching learning in Python. *Journal of Machine Learning Research* 12:2825-2830.

ter Braak and Verdonschot (1995) Canonical correspondence analysis and related multivariate methods in ecology. *Aquatic Sciences* 57/3:255-289.

van Buuren et al. (2006) Fully conditional specification in multivariate imputation. *Journal of Statistical Computation and Simulation* 76:1049-1064.

van Vuuren (2012) Flexible imputation of missing data.

License

EcoPy is distributed under the MIT License

Contact

If you need help or want to contribute, you can contact me at lemoine.nathan@gmail.com.

Version History

0.1.3

- Fixed an bug causing a float error
- Fixed an bug caused by numpy not rounding Euclidean Distances of 0 to 0 (resulting in negative Euclidean distances that cannot be square-rooted)

0.1.2

- More Python 3.x compatibility
- Typos in examples

0.1.1

- More Python 3.x compatibility
- Fixed the transform function to not alter the original data matrix

0.1.0

- Updated the diversity function
- div_partition function for calculating alpha, beta, and gamma diversity
- spatial_median function for calculation multivariate medians
- fixed a bug in MDS function that provided incorrect results when using monotone transformation
- beta_dispersion function for assessing homogeneity of variances of distance matrices

0.0.9

- Missing data imputation
- nls Python 3 compatibility
- Gower's Euclidean distance for missing data
- ord_plot function for convex hull and line plots of ordination results
- Fully incorporated non-linear regression, including documentation
- Incorporated partial Mantel test in Mantel class
- Global tests of RDA significance

- Updated CCA to include correspondence analysis of residual (unconstrained) variance
- Global tests of CCA significance

0.0.8

- Updated PCA to use SVD instead of eigen decomposition

0.0.7

- CCor
- CCA
- RDA
- RLQ analysis
- Hill and Smith ordination
- weighted mean, variance, scaling

0.0.6

- procrustes test of matrix associations
- anosim class for analysis of similarity
- mantel class for Mantel tests
- corner4 class for fourth corner analysis
- load_data function for importing datasets

0.0.5

- poca class for princple coordinate analysis
- MDS class for multidimensional scaling (uses isotonic regression from scikit-learn)
- small changes and fixes to previous functions

0.0.4

- ca class for simple correspondance analysis

0.0.3

- diversity function for calculation species diversity
- rarefy function for rarefaction

0.0.2

- distance function for calculating distance matrices using a wide variety of coefficients and metrics
- transform function for transforming matrices

0.0.1

- nls class for non-linear regression
- pca class for principle components analysis

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index