
pyls

Release 0+untagged.269.gd8a19d5.dirty

pyls developers

Nov 04, 2019

CONTENTS

1	Installation requirements	3
2	Quickstart	5
3	Development and getting involved	7
4	License Information	9
4.1	User guide	9
4.2	Reference API	14
	Python Module Index	25
	Index	27

This package provides a Python interface for performing partial least squares (PLS) analyses.

INSTALLATION REQUIREMENTS

Currently, `pyls` works with Python 3.5+ and requires a few dependencies:

- `h5py`
- `numpy`
- `scikit-learn`
- `scipy`, and
- `tqdm`

Assuming you have the correct version of Python installed, you can install `pyls` by opening a terminal and running the following:

```
git clone https://github.com/rmarkello/pyls.git
cd pyls
python setup.py install
```

All relevant dependencies will be installed alongside the `pyls` module.

QUICKSTART

There are a number of ways to use `pyls`, depending on the type of analysis you would like to perform. Assuming you have two matrices `X` and `Y` representing different observations from a set of samples (i.e., subjects, neurons, brain regions), you can run a simple analysis with:

```
>>> import pyls
>>> results = pyls.behavioral_pls(X, Y)
```

For detailed information on the different methods available and how to interpret the results object, please refer to our *[user guide](#)*.

DEVELOPMENT AND GETTING INVOLVED

If you've found a bug, are experiencing a problem, or have a question about using the package, please head on over to our [GitHub issues](#) and make a new issue with some information about it! Someone will try and get back to you as quickly as possible, though please note that the primary developer for `pyls` (@rmarkello) is a graduate student so responses may take some time!

If you're interested in getting involved in the project: welcome ! We're thrilled to welcome new contributors. You should start by reading our [code of conduct](#); all activity on `pyls` should adhere to the CoC. After that, take a look at our [contributing guidelines](#) so you're familiar with the processes we (generally) try to follow when making changes to the repository! Once you're ready to jump in head on over to our issues to see if there's anything you might like to work on.

LICENSE INFORMATION

This codebase is licensed under the GNU General Public License, version 2. The full license can be found in the [LICENSE](#) file in the `ppls` distribution.

All trademarks referenced herein are property of their respective holders.

4.1 User guide

Partial least squares (PLS) is a multivariate statistical technique that aims to find shared information between two sets of variables. If you're unfamiliar with PLS and are interested in a thorough (albeit quite technical) treatment, [Abdi et al., 2013](#) is a good resource.

This user guide will go through the basic statistical concepts of the two types of PLS implemented in the current package (*Behavioral PLS* and *Mean-centered PLS*) and demonstrate how to interpret and use the results of a PLS analysis (*PLS Results*). If you still have questions after going through this guide then you can refer to the [Reference API](#)!

4.1.1 Behavioral PLS

Running a behavioral PLS using `ppls` is as simple as:

```
>>> import ppls
>>> out = ppls.behavioral_pls(X, Y)
```

What we call behavioral PLS in the `ppls` package is actually the more traditional form of PLS (and is generally not prefixed with “behavioral”). This form of PLS, at its core, attempts to find shared information between two sets of features derived from a common set of samples. However, as with all things, there are a number of ever-so-slightly different kinds of PLS that exist in the wild, so to be thorough we're going to briefly explain the exact flavor implemented here before diving into a more illustrative example.

What *exactly* do we mean by “behavioral PLS”?

Technical answer: `ppls.behavioral_pls()` employs a symmetrical, singular value decomposition (SVD) based form of PLS, and is sometimes referred to as PLS-correlation (PLS-C), PLS-SVD, or, infrequently, EZ-PLS. Notably, it is **not** the same as PLS regression (PLS-R).

Less technical answer: `ppls.behavioral_pls()` is like performing a principal components analysis (PCA) but when you have two related datasets, each with multiple features.

Differences from PLS regression (PLS-R)

You can think of the differences between PLS-C and PLS-R similar to how you might consider the differences between a Pearson correlation and a simple linear regression. Though this analogy is an over-simplification, the primary difference to take away is that behavioral PLS (PLS-C) does *not assess directional relationships between sets of data* (e.g., $X \rightarrow Y$), but rather looks at how the two sets generally covary (e.g., $X \sim Y$).

To understand this a bit more we can walk through a detailed example.

An exercise in calisthenics

Note: Descriptions of PLS are almost always accompanied by a litany of equations, and for good reason: understanding how to interpret the results of a PLS requires at least a cursory understanding of the math behind it. As such, this example is going to rely on these equations, but will always do so in the context of real data. The hope is that this approach will help make the more abstract mathematical concepts a bit more concrete (and easier to apply to new data sets!).

We'll start by loading the example dataset¹:

```
>>> from pyls.examples import load_dataset
>>> data = load_dataset('linnerud')
```

This is the same dataset as in `sklearn.datasets.load_linnerud()`; the formatting has just been lightly modified to better suit our purposes.

Our data object can be treated as a dictionary, containing all the information necessary to run a PLS analysis. The keys can be accessed as attributes, so we can take a quick look at our input matrices **X** and **Y**:

```
>>> sorted(data.keys())
['X', 'Y', 'n_boot', 'n_perm']
>>> data.X.shape
(20, 3)
>>> data.X.head()
   Chins  Situps  Jumps
0     5.0   162.0   60.0
1     2.0   110.0   60.0
2    12.0   101.0  101.0
3    12.0   105.0   37.0
4    13.0   155.0   58.0
```

The rows of our $\mathbf{X}_{n \times p}$ matrix here represent n subjects, and the columns indicate p different types of exercises these subjects were able to perform. So the first subject was able to do 5 chin-ups, 162 situps, and 60 jumping jacks.

```
>>> data.Y.shape
(20, 3)
>>> data.Y.head()
   Weight  Waist  Pulse
0   191.0   36.0   50.0
1   189.0   37.0   52.0
2   193.0   38.0   58.0
3   162.0   35.0   62.0
4   189.0   35.0   46.0
```

¹ Tenenhaus, M. (1998). La régression PLS: théorie et pratique. Editions technip.

The rows of our $\mathbf{Y}_{n \times q}$ matrix *also* represent n subjects (critically, the same subjects as in \mathbf{X}), and the columns indicate q physiological measurements taken for each subject. That same subject referenced above thus has a weight of 191 pounds, a 36 inch waist, and a pulse of 50 beats per minute.

Behavioral PLS will attempt to establish whether a relationship exists between the exercises performed and these physiological variables. If we wanted to run the full analysis right away, we could do so with:

```
>>> from pyls import behavioral_pls
>>> results = behavioral_pls(**data)
```

If you're comfortable with the down-and-dirty of PLS and want to go ahead and start understanding the `results` object, feel free to jump ahead to [PLS Results](#). Otherwise, read on for more about what's happening behind the scenes of `behavioral_pls()`

The cross-covariance matrix

Behavioral PLS works by decomposing the cross-covariance matrix $\mathbf{R}_{q \times p}$ generated from the input matrices, where $\mathbf{R} = \mathbf{Y}^T \mathbf{X}$. The results of PLS are a bit easier to interpret when \mathbf{R} is the cross-correlation matrix instead of the cross-covariance matrix, which means that we should z-score each feature in \mathbf{X} and \mathbf{Y} before multiplying them; this is done automatically by the `behavioral_pls()` function.

In our example, \mathbf{R} ends up being a 3 x 3 matrix:

```
>>> from pyls.compute import xcorr
>>> R = xcorr(data.X, data.Y)
>>> R
```

	Chins	Situps	Jumps
Weight	-0.389694	-0.493084	-0.226296
Waist	-0.552232	-0.645598	-0.191499
Pulse	0.150648	0.225038	0.034933

The q rows of this matrix correspond to the physiological measurements and the p columns to the exercises. Examining the first row, we can see that -0.389694 is the correlation between `Weight` and `Chins` across all the subjects, -0.493084 the correlation between `Weight` and `Situps`, and so on.

Singular value decomposition

Once we have generated our correlation matrix \mathbf{R} we subject it to a singular value decomposition, where $\mathbf{R} = \mathbf{U}\mathbf{S}\mathbf{V}^T$:

```
>>> from pyls.compute import svd
>>> U, S, V = svd(R)
>>> U.shape, S.shape, V.shape
((3, 3), (3, 3), (3, 3))
```

The outputs of this decomposition are two arrays of left and right singular vectors ($\mathbf{U}_{p \times l}$ and $\mathbf{V}_{q \times l}$) and a diagonal matrix of singular values ($\mathbf{S}_{l \times l}$). The rows of \mathbf{U} correspond to the exercises from our input matrix \mathbf{X} , and the rows of \mathbf{V} correspond to the physiological measurements from our input matrix \mathbf{Y} . The columns of \mathbf{U} and \mathbf{V} , on the other hand, represent new dimensions or components that have been “discovered” in the data.

The i^{th} columns of \mathbf{U} and \mathbf{V} weigh the contributions of these exercises and physiological measurements, respectively. Taken together, the i^{th} left and right singular vectors and singular value represent a *latent variable*, a multivariate pattern that weighs the original exercise and physiological measurements such that they maximally covary with each other.

The i^{th} singular value is proportional to the total exercise-physiology covariance accounted for by the latent variable. The effect size (η) associated with a particular latent variable can be estimated as the ratio of the squared singular

value (σ) to the sum of all the squared singular values:

$$\eta_i = \sigma_i^2 / \sum_{j=1}^l \sigma_j^2$$

We can use the helper function `pyls.compute.varexp()` to calculate this for us:

```
>>> from pyls.compute import varexp
>>> pctvar = varexp(S)[0, 0]
>>> print('{:.4f}'.format(pctvar))
0.9947
```

Taking a look at the variance explained, we see that a whopping ~99.5% of the covariance between the exercises and physiological measurements in **X** and **Y** are explained by this latent variable, suggesting that the relationship between these variable can be effectively explained by a single dimension.

Examining the weights from the singular vectors:

```
>>> U[:, 0]
array([0.61330742, 0.7469717 , 0.25668519])
>>> V[:, 0]
array([-0.58989118, -0.77134059, 0.23887675])
```

we see that all the exercises (`U[:, 0]`) are positively weighted, but that the physiological measurements (`V[:, 0]`) are split, with `Weight` and `Waist` measurements negatively weighted and `Pulse` positively weighted. (Note that the order of the weights is the same as the order of the original columns in our **X** and **Y** matrices.) Taken together this suggests that, for the subjects in this dataset, individuals who completed more of a given exercise tended to:

1. Complete more of the other exercises, and
2. Have a lower weight, smaller waist, and higher heart rate.

It is also worth examining how correlated the projections of the original variables on this latent variable are. To do that, we can multiply the original data matrices by the relevant singular vectors and then correlate the results:

```
>>> from scipy.stats import pearsonr
>>> XU = np.dot(data.X, U)
>>> YV = np.dot(data.Y, V)
>>> r, p = pearsonr(XU[:, 0], YV[:, 0])
>>> print('r = {:.4f}, p = {:.4f}'.format(r, p))
r = 0.4900, p = 0.0283
```

The correlation value of this latent variable (~0.49) suggests that our interpretation of the singular vectors weights, above, is only *somewhat* accurate. We can think of this correlation (ranging from -1 to 1) as a proxy for the question: “how often is this interpretation of the singular vectors true?” Correlations closer to -1 indicate that the interpretation is largely inaccurate across subjects, whereas correlations closer to 1 indicate the interpretation is largely accurate across subjects.

Latent variable significance testing

Scientists love null-hypothesis significance testing, so there’s a strong urge for researchers doing these sorts of analyses to want to find a way to determine whether observed latent variables are significant (ly different from a specified null model). The issue comes in determining what aspect of the latent variables to test!

With behavioral PLS we assess whether the **variance explained** by a given latent variable is significantly different than would be expected by a null. Importantly, that null is generated by re-computing the latent variables from random permutations of the original data, generating a non-parametric distribution of explained variances by which to measure “significance.”

Reliability of the singular vectors

<COMING SOON>

4.1.2 Mean-centered PLS

In contrast to behavioral PLS, mean-centered PLS doesn't aim to find relationships between two sets of variables. Instead, it tries to find relationships between *groupings* in a single set of variables. Indeed, you can think of it almost like a multivariate t-test or ANOVA (depending on how many groups you have).

An oenological example

```
>>> from pyls.examples import load_dataset
>>> data = load_dataset('wine')
```

This is the same dataset as in `sklearn.datasets.load_wine()`; the formatting has just been lightly modified to better suit our purposes.

Our data object can be treated as a dictionary, containing all the information necessary to run a PLS analysis. The keys can be accessed as attributes, so we can take a quick look at our input matrix:

```
>>> sorted(data.keys())
['X', 'groups', 'n_boot', 'n_perm']
>>> data.X.shape
(178, 13)
>>> data.X.columns
Index(['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium',
      'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
      'proanthocyanins', 'color_intensity', 'hue',
      'od280/od315_of_diluted_wines', 'proline'],
      dtype='object')
>>> data.groups
[59, 71, 48]
```

4.1.3 PLS Results

So you ran a PLS analysis and got some results. Congratulations! The easy part is done. Interpreting (trying to interpret) the results of a PLS analysis—similar to interpreting the results of a PCA or factor analysis or CCA or any other complex decomposition—can be difficult. The `pyls` package contains some functions, tools, and data structures to try and help.

The `PLSResults` data structure is, at its core, a Python dictionary that is designed to contain all possible results from any of the analyses available in `pyls.types`. Let's generate a small example results object to play around with. We'll use the dataset from the *Behavioral PLS* example:

```
>>> from pyls.examples import load_dataset
>>> data = load_dataset('linnerud')
```

We can generate the results file by running the behavioral PLS analysis again. We pass the `verbose=False` flag to suppress the progress bar that would normally be displayed:

```
>>> from pyls import behavioral_pls
>>> results = behavioral_pls(**data, verbose=False)
>>> results
PLSResults(x_weights, y_weights, x_scores, y_scores, y_loadings, singvals, varexp,
↳ permres, bootres, cvres, inputs)
```

Printing the `results` object gives us a helpful view of some of the different outputs available to us. While we won't go into detail about all of these (see the [Reference API](#) for info on those), we'll touch on a few of the potentially more confusing ones.

4.2 Reference API

This is the primary reference of `pyls`. Please refer to the [user guide](#) for more information on how to best implement these functions in your own workflows.

List of modules

- `pyls` - PLS decompositions
- `pyls.structures` - PLS data structures
- `pyls.io` - Data I/O functionality
- `pyls.matlab` - Matlab compatibility

4.2.1 pyls - PLS decompositions

The primary PLS decomposition methods for use in conducting PLS analyses

<code>pyls.behavioral_pls(X, Y, *[, groups, ...])</code>	Performs behavioral PLS on <i>X</i> and <i>Y</i> .
<code>pyls.meancentered_pls(X, *[, groups, ...])</code>	Performs mean-centered PLS on <i>X</i> , sorted into <i>groups</i> and <i>conditions</i> .

pyls.behavioral_pls

```
pyls.behavioral_pls(X, Y, *, groups=None, n_cond=1, n_perm=5000, n_boot=5000, n_split=0,
                    test_size=0.25, test_split=100, covariance=False, rotate=True, ci=95, permsam-
                    ples=None, bootsamples=None, seed=None, verbose=True, n_proc=None,
                    **kwargs)
```

Performs behavioral PLS on *X* and *Y*.

Behavioral PLS is a multivariate statistical approach that relates two sets of variables together. Traditionally, one of these arrays represents a set of brain features (e.g., functional connectivity estimates) and the other represents a set of behavioral variables; however, these arrays can be any two sets of features belonging to a common group of samples.

Using a singular value decomposition, behavioral PLS attempts to find linear combinations of features from the provided arrays that maximally covary with each other. The decomposition is performed on the cross-covariance matrix R , where $R = Y^T \times X$, which represents the covariation of all the input features across samples.

Parameters

- **X**((*S*, *B*) *array_like*) – Input data matrix, where *S* is samples and *B* is features
- **Y**((*S*, *T*) *array_like*) – Input data matrix, where *S* is samples and *T* is features
- **groups**((*G*,) *list of int*) – List with the number of subjects present in each of *G* groups. Input data should be organized as subjects within groups (i.e., groups should be vertically stacked). If there is only one group this can be left blank.
- **n_cond**(*int*) – Number of conditions observed in data. Note that all subjects must have the same number of conditions. If both conditions and groups are present then the input data should be organized as subjects within conditions within groups (i.e., g1c1s[1-S], g1c2s[1-S], g2c1s[1-S], g2c2s[1-S]).
- **n_perm**(*int*, *optional*) –
Number of permutations to use for testing significance of components. Default: 5000
- **n_boot**(*int*, *optional*) – Number of bootstraps to use for testing reliability of data features. Default: 5000
- **n_split**(*int*, *optional*) – Number of split-half resamples to assess during permutation testing. Default: 0
- **test_split**(*int*, *optional*) – Number of splits for generating test sets during cross-validation. Default: 100
- **test_size**([0, 1] *float*, *optional*) – Proportion of data to partition to test set during cross-validation. Default: 0.25
- **covariance**(*bool*, *optional*) – Whether to use the cross-covariance matrix instead of the cross-correlation during the decomposition. Only set if you are sure this is what you want as many of the results may become more difficult to interpret (i.e., `behavcorr` will no longer be interpretable as Pearson correlation values). Default: False
- **rotate**(*bool*, *optional*) – Whether to perform Procrustes rotations during permutation testing. Can inflate false-positive rates; see Kovacevic et al., (2013) for more information. Default: True
- **ci**([0, 100] *float*, *optional*) – Confidence interval to use for assessing bootstrap results. This roughly corresponds to an alpha rate; e.g., the 95%ile CI is approximately equivalent to a two-tailed $p \leq 0.05$. Default: 95
- **permsamples**(*array_like*, *optional*) – Re-sampling array to be used during permutation test (if `n_perm > 0`). If not specified a set of unique permutations will be generated. Default: None
- **bootsamples**(*array_like*, *optional*) – Resampling array to be used during bootstrap resampling (if `n_boot > 0`). If not specified a set of unique bootstraps will be generated. Default: None
- **seed**({*int*, `numpy.random.RandomState`, `None`}, *optional*) – Seed to use for random number generation. Helps ensure reproducibility of results. Default: None
- **verbose**(*bool*, *optional*) – Whether to show progress bars as the analysis runs. Note that progress bars will not persist after the analysis is completed. Default: True
- **n_proc**(*int*, *optional*) – How many processes to use for parallelizing permutation testing and bootstrap resampling. If not specified will default to serialized processing (i.e., one processor). Can optionally specify 'max' to use all available processors. Default: None

Returns **results** – Dictionary-like object containing results from the PLS analysis

Return type `pyls.structures.PLSResults`

Notes

The singular value decomposition generates mutually orthogonal latent variables (LVs), comprised of left and right singular vectors and a diagonal matrix of singular values. The i -th pair of singular vectors detail the contributions of individual input features to an overall, multivariate pattern (the i -th LV), and the singular values explain the amount of variance captured by that pattern.

Statistical significance of the LVs is determined via permutation testing. Bootstrap resampling is used to examine the contribution and reliability of the input features to each LV. Split-half resampling can optionally be used to assess the reliability of the LVs. A cross-validated framework can optionally be used to examine how accurate the decomposition is when employed in a predictive framework.

References

- McIntosh, A. R., Bookstein, F. L., Haxby, J. V., & Grady, C. L. (1996). Spatial pattern analysis of functional brain images using partial least squares. *NeuroImage*, 3(3), 143-157.
- McIntosh, A. R., & Lobaugh, N. J. (2004). Partial least squares analysis of neuroimaging data: applications and advances. *NeuroImage*, 23, S250-S263.
- Krishnan, A., Williams, L. J., McIntosh, A. R., & Abdi, H. (2011). Partial Least Squares (PLS) methods for neuroimaging: a tutorial and review. *NeuroImage*, 56(2), 455-475.
- Kovacevic, N., Abdi, H., Beaton, D., & McIntosh, A. R. (2013). Revisiting PLS resampling: comparing significance versus reliability across range of simulations. In *New Perspectives in Partial Least Squares and Related Methods* (pp. 159-170). Springer, New York, NY. Chicago
- Misic, B., Betzel, R. F., de Reus, M. A., van den Heuvel, M.P., Berman, M. G., McIntosh, A. R., & Sporns, O. (2016). Network level structure-function relationships in human neocortex. *Cerebral Cortex*, 26, 3285-96.

pyls.meancentered_pls

```
pyls.meancentered_pls(X, *, groups=None, n_cond=1, mean_centering=0, n_perm=5000,
                      n_boot=5000, n_split=0, rotate=True, ci=95, permsamples=None, boot-
                      samples=None, seed=None, verbose=True, n_proc=None, **kwargs)
```

Performs mean-centered PLS on X , sorted into *groups* and *conditions*.

Mean-centered PLS is a multivariate statistical approach that attempts to find sets of variables in a matrix which maximally discriminate between subgroups within the matrix.

While it carries the name PLS, mean-centered PLS is perhaps more related to principal components analysis than it is to `pyls.behavioral_pls`. In contrast to behavioral PLS, mean-centered PLS does not construct a cross-covariance matrix. Instead, it operates by averaging the provided data (X) within groups and/or conditions. The resultant matrix M is mean-centered, generating a new matrix $R_{mean_centered}$ which is submitted to singular value decomposition.

Parameters

- **X** ((S, B) array_like) – Input data matrix, where S is samples and B is features
- **groups** ($(G,)$ list of int) – List with the number of subjects present in each of G groups. Input data should be organized as subjects within groups (i.e., groups should be vertically stacked). If there is only one group this can be left blank.
- **n_cond** (int) – Number of conditions observed in data. Note that all subjects must have the same number of conditions. If both conditions and groups are present then the input data should be organized as subjects within conditions within groups (i.e., g1c1s[1-S], g1c2s[1-S], g2c1s[1-S], g2c2s[1-S]).

- **mean_centering** (*{0, 1, 2}, optional*) – Mean-centering method to use. This will determine how the mean-centered matrix is generated and what effects are “boosted” during the SVD. Default: 0
- **n_perm** (*int, optional*) –
Number of permutations to use for testing significance of components. Default: 5000
- **n_boot** (*int, optional*) – Number of bootstraps to use for testing reliability of data features. Default: 5000
- **n_split** (*int, optional*) – Number of split-half resamples to assess during permutation testing. Default: 0
- **rotate** (*bool, optional*) – Whether to perform Procrustes rotations during permutation testing. Can inflate false-positive rates; see Kovacevic et al., (2013) for more information. Default: True
- **ci** (*[0, 100] float, optional*) – Confidence interval to use for assessing bootstrap results. This roughly corresponds to an alpha rate; e.g., the 95%ile CI is approximately equivalent to a two-tailed $p \leq 0.05$. Default: 95
- **permsamples** (*array_like, optional*) – Re-sampling array to be used during permutation test (if `n_perm > 0`). If not specified a set of unique permutations will be generated. Default: None
- **bootsamples** (*array_like, optional*) – Resampling array to be used during bootstrap resampling (if `n_boot > 0`). If not specified a set of unique bootstraps will be generated. Default: None
- **seed** (*{int, numpy.random.RandomState, None}, optional*) – Seed to use for random number generation. Helps ensure reproducibility of results. Default: None
- **verbose** (*bool, optional*) – Whether to show progress bars as the analysis runs. Note that progress bars will not persist after the analysis is completed. Default: True
- **n_proc** (*int, optional*) – How many processes to use for parallelizing permutation testing and bootstrap resampling. If not specified will default to serialized processing (i.e., one processor). Can optionally specify ‘max’ to use all available processors. Default: None

Returns **results** – Dictionary-like object containing results from the PLS analysis

Return type `pyls.structures.PLSResults`

Notes

The provided `mean_centering` argument can be changed to highlight or “boost” potential group / condition differences by modifying how $R_{mean_centered}$ is generated:

- `mean_centering=0` will remove group means collapsed across conditions, emphasizing potential differences between conditions while removing overall group differences
- `mean_centering=1` will remove condition means collapsed across groups, emphasizing potential differences between groups while removing overall condition differences
- `mean_centering=2` will remove the grand mean collapsed across both groups _and_ conditions, permitting investigation of the full spectrum of potential group and condition effects.

The singular value decomposition generates mutually orthogonal latent variables (LVs), comprised of left and right singular vectors and a diagonal matrix of singular values. The i -th pair of singular vectors detail the contributions of individual input features to an overall, multivariate pattern (the i -th LV), and the singular values explain the amount of variance captured by that pattern.

Statistical significance of the LVs is determined via permutation testing. Bootstrap resampling is used to examine the contribution and reliability of the input features to each LV. Split-half resampling can optionally be used to assess the reliability of the LVs. A cross-validated framework can optionally be used to examine how accurate the decomposition is when employed in a predictive framework.

References

- McIntosh, A. R., Bookstein, F. L., Haxby, J. V., & Grady, C. L. (1996). Spatial pattern analysis of functional brain images using partial least squares. *NeuroImage*, 3(3), 143-157.
- McIntosh, A. R., & Lobaugh, N. J. (2004). Partial least squares analysis of neuroimaging data: applications and advances. *NeuroImage*, 23, S250-S263.
- Krishnan, A., Williams, L. J., McIntosh, A. R., & Abdi, H. (2011). Partial Least Squares (PLS) methods for neuroimaging: a tutorial and review. *NeuroImage*, 56(2), 455-475.
- Kovacevic, N., Abdi, H., Beaton, D., & McIntosh, A. R. (2013). Revisiting PLS resampling: comparing significance versus reliability across range of simulations. In *New Perspectives in Partial Least Squares and Related Methods* (pp. 159-170). Springer, New York, NY. Chicago

4.2.2 pyls.structures - PLS data structures

Data structures to hold PLS inputs and results objects

<code>pyls.structures.PLSResults(**kwargs)</code>	Dictionary-like object containing results of PLS analysis
<code>pyls.structures.PLSPermResults(**kwargs)</code>	Dictionary-like object containing results of PLS permutation testing
<code>pyls.structures.PLSBootResults(**kwargs)</code>	Dictionary-like object containing results of PLS bootstrap resampling
<code>pyls.structures.PLSSplitHalfResults(**kwargs)</code>	Dictionary-like object containing results of PLS split-half resampling
<code>pyls.structures.PLSCrossValidationResults(...)</code>	Dictionary-like object containing results of PLS cross-validation testing
<code>pyls.structures.PLSInputs(*args, **kwargs)</code>	PLS input information

pyls.structures.PLSResults

class `pyls.structures.PLSResults` (***kwargs*)

Dictionary-like object containing results of PLS analysis

x_weights

Weights of B features used to project X matrix into PLS-derived component space

Type (B, L) *numpy.ndarray*

y_weights

Weights of J features used to project Y matrix into PLS-derived component space; not available with `pls_regression()`

Type (J, L) *numpy.ndarray*

x_scores

Projection of X matrix into PLS-derived component space

Type (S, L) *numpy.ndarray*

y_scores

Projection of Y matrix into PLS-derived component space

Type (S, L) *numpy.ndarray*

y_loadings

Covariance of features in Y with projected x_scores

Type (J, L) *numpy.ndarray*

singvals

Singular values for PLS-derived component space; not available with `pls_regression()`

Type (L, L) *numpy.ndarray*

varexp

Variance explained in each of the PLS-derived components

Type (L,) *numpy.ndarray*

permres

Results of permutation testing, as applicable

Type *PLSPermResults*

bootres

Results of bootstrap resampling, as applicable

Type *PLSBootResults*

splitres

Results of split-half resampling, as applicable

Type *PLSSplitHalfResults*

cvres

Results of cross-validation testing, as applicable

Type *PLSCrossValidationResults*

inputs

Inputs provided to original PLS

Type *PLSInputs*

pyls.structures.PLSPermResults

class `pyls.structures.PLSPermResults` (**kwargs)

Dictionary-like object containing results of PLS permutation testing

pvals

Non-parametric p-values used to examine whether components from original decomposition explain more variance than permuted components

Type (L,) *numpy.ndarray*

permsamples

Resampling array used to permute S samples over P permutations

Type (S, P) *numpy.ndarray*

pyls.structures.PLSBootResults

```
class pyls.structures.PLSBootResults (**kwargs)
    Dictionary-like object containing results of PLS bootstrap resampling

    x_weights_normed
        x_weights normalized by their standard error, obtained from bootstrap resampling (see x_weights_stderr)

        Type (B, L) numpy.ndarray

    x_weights_stderr
        Standard error of x_weights, used to generate x_weights_normed

        Type (B, L) numpy.ndarray

    y_loadings
        Covariance of features in Y with projected x_scores; not available with meancentered_pls()

        Type (J, L) numpy.ndarray

    y_loadings_boot
        Distribution of y_loadings across all bootstrap resamples; not available with meancentered_pls()

        Type (J, L, R) numpy.ndarray

    y_loadings_ci
        Lower (... , 0) and upper (... , 1) bounds of confidence interval for y_loadings; not available with
        meancentered_pls()

        Type (J, L, 2) numpy.ndarray

    contrast
        Group x condition averages of brainscores_demeaned. Can be treated as a contrast indicating group
        x condition differences. Only obtained from meancentered_pls.

        Type (J, L) numpy.ndarray

    contrast_boot
        Bootstrapped distribution of contrast; only available with meancentered_pls()

        Type (J, L, R) numpy.ndarray

    contrast_ci
        Lower (... , 0) and upper (... , 1) bounds of confidence interval for contrast; only available with
        meancentered_pls()

        Type (J, L, 2) numpy.ndarray

    bootsamples
        Indices of bootstrapped samples S across R resamples.

        Type (S, R) numpy.ndarray
```

pyls.structures.PLSSplitHalfResults

```
class pyls.structures.PLSSplitHalfResults (**kwargs)
    Dictionary-like object containing results of PLS split-half resampling

    ucorr, vcorr
        Average correlations between split-half resamples in original (non- permuted) data for left/right singular
        vectors. Can be interpreted as reliability of L latent variables

        Type (L,) numpy.ndarray
```


ucorr_pvals, vcorr_pvals

Number of permutations where correlation between split-half resamples exceeded original correlations, normalized by the total number of permutations. Can be interpreted as the statistical significance of the reliability of L latent variables

Type (L,) *numpy.ndarray*

ucorr_uplim, vcorr_uplim

Upper bound of confidence interval for correlations between split halves for left/right singular vectors

Type (L,) *numpy.ndarray*

ucorr_lolim, vcorr_lolim

Lower bound of confidence interval for correlations between split halves for left/right singular vectors

Type (L,) *numpy.ndarray*

pyls.structures.PLSCrossValidationResults

class `pyls.structures.PLSCrossValidationResults` (**kwargs)

Dictionary-like object containing results of PLS cross-validation testing

r_squared

R-squared (“determination coefficient”) for each of T predicted behavioral scores against true behavioral scores across I train / test split

Type (T, I) *numpy.ndarray*

pearson_r

Pearson’s correlation for each of T predicted behavioral scores against true behavioral scores across I train / test split

Type (T, I) *numpy.ndarray*

pyls.structures.PLSInputs

class `pyls.structures.PLSInputs` (*args, **kwargs)

PLS input information

X

Input data matrix, where S is observations and B is features.

Type (S, B) array_like

Y

Behavioral matrix, where S is observations and T is features. If from *behavioral_pls*, this is the provided behavior matrix; if from *meancentered_pls*, this is a dummy-coded group/condition matrix.

Type (S, T) array_like

groups

List with the number of subjects present in each of G groups. Input data should be organized as subjects within groups (i.e., groups should be vertically stacked). If there is only one group this can be left blank.

Type (G,) list of int

n_cond

Number of conditions observed in data. Note that all subjects must have the same number of conditions. If both conditions and groups are present then the input data should be organized as subjects within conditions within groups (i.e., g1c1s[1-S], g1c2s[1-S], g2c1s[1-S], g2c2s[1-S]).

Type int

mean_centering

Mean-centering method to use. This will determine how the mean-centered matrix is generated and what effects are “boosted” during the SVD. Default: 0

Type {0, 1, 2}, optional

covariance

Whether to use the cross-covariance matrix instead of the cross- correlation during the decomposition. Only set if you are sure this is what you want as many of the results may become more difficult to interpret (i.e., `behavcorr` will no longer be interpretable as Pearson correlation values). Default: False

Type bool, optional

n_perm

Number of permutations to use for testing significance of components. Default: 5000

Type int, optional

n_boot

Number of bootstraps to use for testing reliability of data features. Default: 5000

Type int, optional

rotate

Whether to perform Procrustes rotations during permutation testing. Can inflate false-positive rates; see Kovacevic et al., (2013) for more information. Default: True

Type bool, optional

ci

Confidence interval to use for assessing bootstrap results. This roughly corresponds to an alpha rate; e.g., the 95%ile CI is approximately equivalent to a two-tailed $p \leq 0.05$. Default: 95

Type [0, 100] float, optional

seed

Seed to use for random number generation. Helps ensure reproducibility of results. Default: None

Type {int, `numpy.random.RandomState`, None}, optional

verbose

Whether to show progress bars as the analysis runs. Note that progress bars will not persist after the analysis is completed. Default: True

Type bool, optional

n_proc

How many processes to use for parallelizing permutation testing and bootstrap resampling. If not specified will default to serialized processing (i.e., one processor). Can optionally specify ‘max’ to use all available processors. Default: None

Type int, optional

4.2.3 `pyls.io` - Data I/O functionality

Functions for saving and loading PLS data objects

<code>pyls.save_results(fname, results)</code>	Saves PLS <i>results</i> to hdf5 file <i>fname</i>
<code>pyls.load_results(fname)</code>	Load PLS results stored in <i>fname</i> , generated by <code>pyls.save_results()</code>

pyls.save_results

`pyls.save_results(fname, results)`

Saves PLS *results* to hdf5 file *fname*

If *fname* does not end with '.hdf5' it will be appended

Parameters

- **fname** (*str*) – Filepath to where hdf5 file should be created and *results* stored
- **results** (`pyls.structures.PLSResults`) – PLSResults object to be saved

Returns *fname* – Filepath to created file

Return type *str*

pyls.load_results

`pyls.load_results(fname)`

Load PLS results stored in *fname*, generated by `pyls.save_results()`

Parameters **fname** (*str*) – Filepath to HDF5 file containing PLS results

Returns *results* – Loaded PLS results

Return type `pyls.structures.PLSResults`

4.2.4 pyls.matlab - Matlab compatibility

Utilities for handling PLS results generated using the Matlab PLS toolbox

<code>pyls.import_matlab_result(fname[, data=mat])</code>	Imports <i>fname</i> PLS result from Matlab
---	---

pyls.import_matlab_result

`pyls.import_matlab_result(fname, datamat='datamat_lst')`

Imports *fname* PLS result from Matlab

Parameters

- **fname** (*str*) – Filepath to output mat file obtained from Matlab PLS toolbox. Should contain at least a result struct object.
- **datamat** (*str, optional*) – Variable name of datamat ('X' array) provided to original PLS if it exists *fname*. By default the datamat is not stored in the PLS results structure, but if it was saved in *fname* it can be loaded and cached in the returned results object. Default: 'datamat_lst'

Returns *results* – Matlab results in a Python-friendly format

Return type *PLSResults*

PYTHON MODULE INDEX

p

`pyls.io`, [22](#)
`pyls.matlab`, [23](#)
`pyls.structures`, [18](#)
`pyls.types`, [14](#)

B

`behavioral_pls()` (in module `pyls`), 14
`bootres` (`pyls.structures.PLSResults` attribute), 19
`bootsamples` (`pyls.structures.PLSBootResults` attribute), 20

C

`ci` (`pyls.structures.PLSInputs` attribute), 22
`contrast` (`pyls.structures.PLSBootResults` attribute), 20
`contrast_boot` (`pyls.structures.PLSBootResults` attribute), 20
`contrast_ci` (`pyls.structures.PLSBootResults` attribute), 20
`covariance` (`pyls.structures.PLSInputs` attribute), 22
`cvres` (`pyls.structures.PLSResults` attribute), 19

G

`groups` (`pyls.structures.PLSInputs` attribute), 21

I

`import_matlab_result()` (in module `pyls`), 23
`inputs` (`pyls.structures.PLSResults` attribute), 19

L

`load_results()` (in module `pyls`), 23

M

`mean_centering` (`pyls.structures.PLSInputs` attribute), 22
`meancentered_pls()` (in module `pyls`), 16

N

`n_boot` (`pyls.structures.PLSInputs` attribute), 22
`n_cond` (`pyls.structures.PLSInputs` attribute), 21
`n_perm` (`pyls.structures.PLSInputs` attribute), 22
`n_proc` (`pyls.structures.PLSInputs` attribute), 22

P

`pearson_r` (`pyls.structures.PLSCrossValidationResults` attribute), 21

`permres` (`pyls.structures.PLSResults` attribute), 19
`permsamples` (`pyls.structures.PLSPermResults` attribute), 19
`PLSBootResults` (class in `pyls.structures`), 20
`PLSCrossValidationResults` (class in `pyls.structures`), 21
`PLSInputs` (class in `pyls.structures`), 21
`PLSPermResults` (class in `pyls.structures`), 19
`PLSResults` (class in `pyls.structures`), 18
`PLSSplitHalfResults` (class in `pyls.structures`), 20
`pvals` (`pyls.structures.PLSPermResults` attribute), 19
`pyls.io` (module), 22
`pyls.matlab` (module), 23
`pyls.structures` (module), 18
`pyls.types` (module), 14

R

`r_squared` (`pyls.structures.PLSCrossValidationResults` attribute), 21
`rotate` (`pyls.structures.PLSInputs` attribute), 22

S

`save_results()` (in module `pyls`), 23
`seed` (`pyls.structures.PLSInputs` attribute), 22
`singvals` (`pyls.structures.PLSResults` attribute), 19
`splitres` (`pyls.structures.PLSResults` attribute), 19

V

`varexp` (`pyls.structures.PLSResults` attribute), 19
`verbose` (`pyls.structures.PLSInputs` attribute), 22

X

`X` (`pyls.structures.PLSInputs` attribute), 21
`x_scores` (`pyls.structures.PLSResults` attribute), 18
`x_weights` (`pyls.structures.PLSResults` attribute), 18
`x_weights_normed` (`pyls.structures.PLSBootResults` attribute), 20
`x_weights_stderr` (`pyls.structures.PLSBootResults` attribute), 20

Y

`Y` (`pyls.structures.PLSInputs` attribute), 21

`y_loadings` (*pyls.structures.PLSBootResults* attribute), 20
`y_loadings` (*pyls.structures.PLSResults* attribute), 19
`y_loadings_boot` (*pyls.structures.PLSBootResults* attribute), 20
`y_loadings_ci` (*pyls.structures.PLSBootResults* attribute), 20
`y_scores` (*pyls.structures.PLSResults* attribute), 19
`y_weights` (*pyls.structures.PLSResults* attribute), 18