GLHMM

Release 0.0.1

Sonsoles Alonso

May 09, 2024

CONTENTS:

The GLHMM toolbox provides facilities to fit a variety of Hidden Markov models (HMM) based on the Gaussian distribution, which we generalise as the Gaussian-Linear HMM. Crucially, the toolbox has a focus on finding associations at various levels between brain data (EEG, MEG, fMRI, ECoG, etc) and non-brain data, such as behavioural or physiological variables.

- Official source code repo: https://github.com/vidaurre/glhmm
- GLHMM documentation: https://glhmm.readthedocs.io/en/latest/index.html

CHAPTER

ONE

DEPENDENCIES

The required dependencies to use glhmm are:

- Python >= 3.6
- NumPy
- numba
- scikit-learn
- scipy
- matplotlib
- seaborn
- cupy (only when using GPU acceleration; requires manual install)

CHAPTER

TWO

INSTALLATION

- To install the latest development version from the repository, use the following command:

```
pip install git+https://github.com/vidaurre/glhmm
```

- Alternatively, to install the latest stable release from PyPI, use the command:

```
pip install glhmm
```

CHAPTER

THREE

DOCUMENTATION

Warning The documentation of this library is under development

3.1 Tutorial

GLHMM is a Python toolbox with a focus on neuroscience applications but broadly applicable to other domains as well. It implements a generalisation of various types of Hidden Markov Model (HMM). The toolbox can be applied on multiple data modalities, including fMRI, EEG, MEG, and ECoG, and offers a comprehensive set of HMMs tailored for different data types and analysis goals. The most important configurable aspect is the state distribution, which is parameterized using a regression model. A non-exhaustive list of possible state distributions are:

- `Gaussian`: used in fMRI and other neuroimaging modalities.
- `Wishart`: employed in fMRI to specifically focus on changes in connectivity (covariance).
- `Time-delay embedded`: applied to whole-brain electrophysiological data (MEG or EEG), to capture spectral modulations in the data.
- `Autoregressive`: provides a more detailed spectral description for electrophysiological data with a limited number of channels.
- `Regression-based decoding`: describes the dynamic relationship between brain activity and ongoing stimuli.
- `Regression-based encoding`: emphasizes the spatial interpretation of brain activity in relation to stimuli.

3.1.1 Installation

If you have not done so, install the repo using:

```
pip install glhmm
```

3.1.2 Examples

Example data is provided in the `example_data` folder.

For an example of running a standard HMM using only one set of time series, see Example standard Gaussian HMM.

For an example of running a GLHMM using two sets of time series, see Example GLHMM.

3.1.3 Relations to behaviour

After estimating an HMM, we can explore its connections with an external variable not initially considered in the model. This could involve tasks like predicting age from subject-specific HMMs based on neuroimaging data or examining correlations with physiological factors. Our toolbox supports these types of analyses in the modules Prediction and Statistics

Prediction

This module enables the utilization of individual brain activity patterns for various applications, including predictions (such as cognitive abilities) and classifications (of subjects or clinical groups, for example). For a tutorial, see here

Statistics

This module provides powerful permutation testing analysis, which allows for statistical significance assessment without data distribution assumptions. It supports various test types, such as between- and within-session/subject tests. Users can choose between permutation testing with regression or correlation for a wide range of research questions. For a tutorial demonstrating the application of testing look at the following examples: - Testing across subjects . - Testing across sessions . - Testing across trials . - Testing across visits .

3.2 Modules

3.2.1 glhmm.glhmm

Gaussian Linear Hidden Markov Model @author: Diego Vidaurre 2023

class glhmm.glhmm.**glhmm**(*K=10*, *covtype='shareddiag'*, *model_mean='state'*, *model_beta='state'*, *dirichlet_diag=10*, *connectivity=None*, *Pstructure=None*, *Pistructure=None*)

Bases: `object`

Gaussian Linear Hidden Markov Model class to decode stimulus from data.

Attributes:

K
: [int, default=10] number of states in the model.

covtype
: [str, {'shareddiag', 'diag','sharedfull','full'}, default 'shareddiag'] Type of covariance matrix. Choose 'shareddiag' to have one diagonal covariance matrix for all states, or 'diag' to have a diagonal full covariance matrix for each state, or 'sharedfull' to have a shared full covariance matrix for all states, or 'full' to have a full covariance matrix for each state.

model_mean
: [str, {'state', 'shared', 'no'}, default 'state'] Model for the mean. If 'state', the mean will be modelled state-dependent. If 'shared', the mean will be modelled globally (shared between all states). If 'no' the mean of the timeseries will not be used to drive the states.

model_beta
: [str, {'state', 'shared', 'no'}, default 'state'] Model for the beta. If 'state', the regression coefficients will be modelled state-dependent. If 'shared', the regression coefficients will be modelled globally (shared between all states). If 'no' the regression coefficients will not be used to drive the states.

dirichlet_diag
[float, default=10] The value of the diagonal of the Dirichlet distribution for the transition probabilities. The higher the value, the more persistent the states will be. Note that this value is relative; the prior competes with the data, so if the timeseries is very long, the *dirichlet_diag* may have little effect unless it is set to a very large value.

connectivity
[array_like of shape (n_states, n_states), optional] Matrix of binary values defining the connectivity of the states. This parameter can only be used with a diagonal covariance matrix (i.e., *covtype='diag'*).

Pstructure
[array_like, optional] Binary matrix defining the allowed transitions between states. The default is a (n_states, n_states) matrix of all ones, allowing all possible transitions between states.

Pistructure
[array_like, optional] Binary vector defining the allowed initial states. The default is a (n_states,) vector of all ones, allowing all states to be used as initial states.

Notes:

This class requires the following modules: numpy, math, scipy, sys, warnings, copy, and time.

decode(*X*, *Y*, *indices=None*, *files=None*, *viterbi=False*, *set=None*, *serial=False*, *gpu_acceleration=0*, *gpuChunks=1*)

Calculates state time courses for all the data using either parallel or sequential processing.

Parameters:

X
[array-like of shape (n_samples, n_parcels)] The timeseries of set of variables 1.

Y
[array-like of shape (n_samples, n_parcels)] The timeseries of set of variables 2.

indices
[array-like of shape (n_sessions, 2), optional, default=None] The start and end indices of each trial/session in the input data.

files
[list of str, optional, default=None] List of filenames corresponding to the indices.

viterbi
[bool, optional, default=False] Whether or not the Viterbi algorithm should be used.

set
[int, optional, default=None] Index of the sessions set to decode.

Returns:

If viterbi=True:

vpath
[array of shape (n_samples,)] The most likely state sequence.

If viterbi=False:

Gamma
[array of shape (n_samples, n_states)] The state probability timeseries.

Xi
[array of shape (n_samples - n_sessions, n_states, n_states)] The joint probabilities of past and future states conditioned on data.

scale
[array-like of shape (n_samples,)] The scaling factors from the inference, used to compute the free energy. In normal use, we would do

Gamma,Xi,_ = hmm.decode(X,Y,indices)

Raises:

Exception
If the model has not been trained. If both 'files' and 'Y' arguments are provided.

dual_estimate(*X*, *Y*, *indices=None*, *Gamma=None*, *Xi=None*, *for_kernel=False*)

Dual estimation of HMM parameters.

Parameters:

X
[array-like of shape (n_samples, n_variables_1)] The timeseries of set of variables 1.

Y
[array-like of shape (n_samples, n_variables_2)] The timeseries of set of variables 2.

indices
[array-like of shape (n_sessions, 2), optional] The start and end indices of each trial/session in the input data. If None, a single segment spanning the entire sequence is used.

Gamma
[array-like of shape (n_samples, n_states), optional] The state probabilities. If None, it is computed from the input observations.

Xi
[array-like of shape (n_samples - n_sessions, n_states, n_states), optional] The joint probabilities of past and future states conditioned on data. If None, it is computed from the input observations.

for_kernel
[bool, optional] Whether purpose of dual estimation is kernel (gradient) computation, or not If True, function will also return Gamma and Xi (default False)

Returns:

hmm_dual
[object] A copy of the HMM object with updated dynamics and observation distributions.

`get_P()`
Returns transition probability matrix

Returns:

P: ndarray of shape (K,K), where K is the number of states

Raises:

Exception
If the model has not yet been trained.

`get_Pi()`
Returns initial probabilities

Returns:

Pi: ndarray of shape (K,), where K is the number of states

Raises:

Exception
If the model has not yet been trained.

`get_active_K()`
Returns the number of active states

Returns:

K_active
[int] Number of active states.

`get_beta(k=0)`
Returns the regression coefficients (beta) for the specified state.

Parameters:

k
[int, optional, default=0] The index of the state for which to retrieve the beta value.

Returns:

beta: ndarray of shape (n_variables_1 x n_variables_2)
The regression coefficients of each variable in X on each variable in Y for the specified state.

Raises:

Exception
If the model has not yet been trained. If the model has no beta.

`get_betas()`

Returns the regression coefficients (beta) for all states.

Returns:

betas: ndarray of shape (n_variables_1 x n_variables_2 x n_states)
The regression coefficients of each variable in X on each variable in Y for all states.

Raises:

Exception
If the model has not yet been trained. If the model has no beta.

`get_covariance_matrix(k=0)`

Returns the covariance matrix for the specified state.

Parameters:

k
[int, optional] The index of the state. Default=0.

Returns:

array of shape (n_parcels, n_parcels)
The covariance matrix for the specified state.

Raises:

Exception
If the model has not been trained.

get_fe(*X*, *Y*, *Gamma*, *Xi*, *scale=None*, *indices=None*, *todo=None*, *non_informative_prior_P=False*)
Computes the Free Energy of an HMM depending on observation model.

Parameters:

X
[array of shape (n_samples, n_parcels)] The timeseries of set of variables 1.

Y
[array of shape (n_samples, n_parcels)] The timeseries of set of variables 2.

Gamma
[array of shape (n_samples, n_states), default=None] The state timeseries probabilities.

Xi
[array-like of shape (n_samples - n_sessions, n_states, n_states)] The joint probabilities of past and future states conditioned on data.

scale
[array-like of shape (n_samples,), default=None] The scaling factors used to compute the free energy of the dataset. If None, scaling is automatically computed.

indices
[array-like of shape (n_sessions, 2), optional, default=None] The start and end indices of each trial/session in the input data.

todo: bool of shape (n_terms,) or None, default=None
Whether or not each of the 5 elements (see *fe_terms*) should be computed. Only for internal use.

non_informative_prior_P: array-like of shape (n_states, n_states), optional, default=False
Prior of transition probability matrix Only for internal use.

Returns:

fe_terms
[array of shape (n_terms,)] The variational free energy, separated into different terms: - element 1: Gamma Entropy - element 2: Data negative log-likelihood - element 3: Gamma negative log-likelihood - element 4: KL divergence for initial and transition probabilities - element 5: KL divergence for the state parameters

Raises:

Exception
If the model has not been trained.

Notes:

This function computes the variational free energy using a specific algorithm. For more information on the algorithm, see [^1].

References:

[^1] Smith, J. et al. "A variational approach to Bayesian learning of switching dynamics in dynamical systems." Journal of Machine Learning Research, vol. 18, no. 4, 2017.

`get_inverse_covariance_matrix`(*k=0*)

Returns the inverse covariance matrix for the specified state.

Parameters:

k
[int, optional] The index of the state. Default=0.

Returns:

array of shape (n_parcels, n_parcels)
The inverse covariance matrix for the specified state.

Raises:

Exception
If the model has not been trained.

`get_mean`(*k=0*)

Returns the mean for the specified state.

Parameters:

k
[int, optional, default=0] The index of the state for which to retrieve the mean.

Returns:

mean: ndarray of shape (n_variables_2,)
The mean value of each variable in Y for the specified state.

Raises:

Exception
If the model has not yet been trained. If the model has no mean.

`get_means()`

Returns the means for all states.

Returns:

means: ndarray of shape (n_variables_2, n_states)
The mean value of each variable in Y for all states.

Raises:

Exception
If the model has not yet been trained. If the model has no mean.

`get_r2(X, Y, Gamma, indices=None)`

Computes the explained variance per session/trial and per column of Y

Parameters:

X
[array of shape (n_samples, n_variables_1)] The timeseries of set of variables 1.

Y
[array of shape (n_samples, n_variables_2)] The timeseries of set of variables 2.

Gamma
[array of shape (n_samples, n_states), default=None] The state timeseries probabilities.

indices
[array-like of shape (n_sessions, 2), optional, default=None] The start and end indices of each trial/session in the input data.

Returns:

r2
[array of shape (n_sessions, n_variables_2)] The R-squared (proportion of the variance explained) for each session and each variable in Y.

Raises:

Exception
If the model has not been trained, or if it does not have neither mean or beta

Notes:

This function does not take the covariance matrix into account

`initialize`(*p*, *q*)

Initialize the parameters of the HMM with initial random values. IMPORTANT: This should not be run before training. This is only useful for sampling data, and should be combined with subsequent calls to set_beta, set_mean, set_covariance_matrix, set_P and set_Pi.

Parameters:

p : number of channels in X (not used if only Y is modelled) q : number of channels in Y

`loglikelihood`(*X*, *Y*)

Computes the likelihood of the model per state and time point given the data X and Y.

Parameters:

X
[array-like of shape (n_samples, n_parcels)] The timeseries of set of variables 1.

Y
[array-like of shape (n_samples, n_parcels)] The timeseries of set of variables 2.

Returns:

L
[array of shape (n_samples, n_states)] The likelihood of the model per state and time point given the data X and Y.

Raises:

Exception
If the model has not been trained.

`sample`(*size*, *X=None*, *Gamma=None*)

Generates Gamma and Y for timeseries of lengths specified in variable size.

Parameters:

size
[array of shape (n_sessions,) or (n_sessions, 2)] If *size* is 1-dimensional, each element represents the length of a session. If *size* is 2-dimensional, each row of *size* represents the start and end indices of a session in a timeseries.

X
[array of shape (n_samples, n_parcels), default=None] The timeseries of set of variables 1.

Gamma
[array of shape (n_samples, n_states), default=None] The state probability timeseries.

Returns:

If X is not None:

X
[array of shape (n_samples, n_parcels)] The timeseries of set of variables 1.

Y: array of shape (n_samples,n_parcels)
The timeseries of set of variables 2.

Gamma
[array of shape (n_samples, n_states)] The state probability timeseries.

`sample_Gamma`(*size*)

Generates Gamma, for timeseries of lengths specified in variable size.

Parameters:

size
[array] Array of shape (n_sessions,) or (n_sessions, 2). If *size* is 1-dimensional, each element represents the length of a session. If *size* is 2-dimensional, each row of *size* represents the start and end indices of a session in a timeseries.

Returns:

Gamma
[array of shape (n_samples, n_states)] The state probability timeseries.

`set_P`(*P*)

Set transition probability matrix. Useful to create synthetic data for simulations.

Parameters:

P: ndarray of shape (K,K), where K is the number of states

`set_Pi`(*Pi*)

Returns initial probabilities. Useful to create synthetic data for simulations.

Parameters:

Pi: ndarray of shape (K,), where K is the number of states

`set_beta`(*beta*, *k=0*)

Sets the regression coefficients (beta) to specific values. Useful to create synthetic data for simulations.

Parameters:

beta: ndarray of shape (n_variables_1 x n_variables_2)
The regression coefficients of each variable in X on each variable in Y for the specified state k.

k
[int, optional, default=0] The index of the state for which to retrieve the beta value.

`set_covariance_matrix`(*shape*, *self*, *k=0*)

Sets the covariance matrix to specific values. Useful to create synthetic data for simulations.

Parameters:

rate
[ndarray of shape (n_variables_2 x n_variables_2),] The rate parameter of the covariance

shape : int, the shape parameter of the covariance k : int, optional

The index of the state. Default=0.

`set_mean`(*mean*, *k=0*)

Sets the mean to specific values. Useful to create synthetic data for simulations.

Parameters:

mean: ndarray of shape (n_variables_2,)
The mean value of each variable in Y for the specified state.

k
[int, optional, default=0] The index of the state for which to retrieve the beta value.

`train`(*X=None*, *Y=None*, *indices=None*, *files=None*, *Gamma=None*, *Xi=None*, *scale=None*, *options=None*)

Train the GLHMM on input data X and Y, which most general formulation is Y = mu_k + X beta_k + noise where noise is Gaussian with mean zero and standard deviation Sigma_k

It supports both standard and stochastic variational learning; for the latter, data must be supplied in files format

Parameters:

X
[array-like of shape (n_samples, n_variables_1)] The timeseries of set of variables 1.

Y
[array-like of shape (n_samples, n_variables_2)] The timeseries of set of variables 2.

indices
[array-like of shape (n_sessions, 2), optional] The start and end indices of each trial/session in the input data. If None, one big segment with no cuts is assumed.

files
[str or list of str, optional] The filename(s) containing the data to load. If not None, X, Y, and indices are ignored.

Gamma
[array-like of shape (n_samples, n_states), optional] The initial values of the state probabilities.

Xi
[array-like of shape (n_samples - n_sessions, n_states, n_states), optional] The joint probabilities of past and future states conditioned on data.

scale
[array-like of shape (n_samples,), optional] The scaling factors used to compute the free energy of the dataset. If None, scaling is automatically computed.

options
[dict, optional] A dictionary with options to control the training process.

Returns:

Gamma
[array-like of shape (n_samples, n_states)] The state probabilities. To avoid unnecessary use of memory, Gamma is only returned if learning is non-stochastic; otherwise it is returned as an empty numpy array. To get Gamma after stochastic learning, use the decode method.

Xi
[array-like of shape (n_samples - n_sessions, n_states, n_states)] The joint probabilities of past and future states conditioned on data. To avoid unnecessary use of memory, Xi is only returned if learning

is non-stochastic; otherwise it is returned as an empty numpy array. To get Xi after stochastic learning, use the decode method.

fe
[array-like] The free energy computed at each iteration of the training process.

Raises:

Exception

If *files* and *Y* are both provided or if neither are provided. If *X* is not provided and the hyperparameter 'model_beta' is True.

If 'files' is not provided and stochastic learning is called upon

3.2.2 glhmm.io

Input/output functions - Gaussian Linear Hidden Markov Model @author: Diego Vidaurre 2023

`glhmm.io.load_files(`*files*, *I=None*, *do_only_indices=False*`)`

Loads data from files and returns the loaded data, indices, and individual indices for each file.

`glhmm.io.load_hmm(`*filename*, *directory=None*`)`

Load a glhmm object from the specified file.

Parameters:

filename (str):
Name of the file containing the glhmm object.

directory (str, optional), default=None:
Directory where the file is located. If None, searches in the current working directory.

Returns:

glhmm
[object] Loaded glhmm object.

`glhmm.io.load_statistics(`*filename*, *directory=None*`)`

Load statistics data from a file.

Parameters

- **filename** (*str*) – The name of the file containing the saved statistics data, with or without extension.
- **(str** (*load_directory*) – The directory path where the file is located (default is the current working directory).
- **optional)** – The directory path where the file is located (default is the current working directory).
- **default=None** – The directory path where the file is located (default is the current working directory).

Returns
data_dict – The dictionary containing the loaded statistics data.

Return type
dict

glhmm.io.**read_flattened_hmm_mat**(*file*)

Reads a MATLAB file containing hidden Markov model (HMM) parameters, and initializes a Gaussian linear hidden Markov model (GLHMM) using those parameters.

glhmm.io.**save_hmm**(*hmm*, *filename*, *directory=None*)

Save a glhmm object in the specified directory with the given filename.

Parameters:

hmm (object)
The glhmm object to be saved.

filename (str)
The name of the file to which the object will be saved.

directory (str, optional), default=None:
The directory where the file will be saved. If None, saves in the current working directory.

glhmm.io.**save_statistics**(*data_dict*, *filename='statistics'*, *directory=None*, *format='npy'*)

Save statistics data to a file in the specified directory with optional format (npy or npz).

Parameters

- **(dict)** (*data_dict*) – Dictionary containing statistics data to be saved.
- **(str** (*format*) – Name of the file.
- **optional)** – Name of the file.
- **default='statistics'** – Name of the file.
- **(str** – Directory path where the file will be saved (default is the current working directory).
- **optional)** – Directory path where the file will be saved (default is the current working directory).
- **default=None** – Directory path where the file will be saved (default is the current working directory).
- **(str** – Serialization format (‘npy’ or ‘npz’).
- **optional)** – Serialization format (‘npy’ or ‘npz’).
- **default='npy'** – Serialization format (‘npy’ or ‘npz’).

3.2.3 glhmm.preproc

Preprocessing functions - General/Gaussian Linear Hidden Markov Model @author: Diego Vidaurre 2023

glhmm.preproc.**apply_pca**(*X*, *d*, *whitening=False*, *exact=True*)

Applies PCA to the input data X.

Parameters:

X
[array-like of shape (n_samples, n_parcels)] The input data to be transformed.

d
[int or float] If int, the number of components to keep. If float, the percentage of explained variance to keep. If array-like of shape (n_parcels, n_components), the transformation matrix.

whitening
[bool, default=False] Whether to whiten the transformed data.

exact
[bool, default=True] Whether to use full SVD solver for PCA.

Returns:

X_transformed
[array-like of shape (n_samples, n_components)] The transformed data after applying PCA.

glhmm.preproc.**build_data_autoregressive**(*data*, *indices*, *autoregressive_order=1*, *connectivity=None*, *center_data=True*)

Builds X and Y for the autoregressive model, as well as an adapted indices array and predefined connectivity matrix in the right format. X and Y are centered by default.

Parameters:

data
[array-like of shape (n_samples,n_parcels)] The data timeseries.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

autoregressive_order
[int, optional, default=1] The number of lags to include in the autoregressive model.

connectivity
[array-like of shape (n_parcels, n_parcels), optional, default=None] The matrix indicating which regressors should be used for each variable.

center_data
[bool, optional, default=True] If True, the data will be centered.

Returns:

X
: [array-like of shape (n_samples - n_sessions*autoregressive_order, n_parcels*autoregressive_order)] The timeseries of set of variables 1 (i.e., the regressors).

Y
: [array-like of shape (n_samples - n_sessions*autoregressive_order, n_parcels)] The timeseries of set of variables 2 (i.e., variables to predict, targets).

indices_new
: [array-like of shape (n_sessions, 2)] The new array of start and end indices for each trial/session.

connectivity_new
: [array-like of shape (n_parcels*autoregressive_order, n_parcels)] The new connectivity matrix indicating which regressors should be used for each variable.

glhmm.preproc.**build_data_partial_connectivity**(*X*, *Y*, *connectivity=None*, *center_data=True*)

Builds X and Y for the partial connectivity model, essentially regressing out things when indicated in connectivity, and getting rid of regressors / regressed variables that are not used; it return connectivity with the right dimensions as well.

Parameters:

X
: [np.ndarray of shape (n_samples, n_parcels)] The timeseries of set of variables 1 (i.e., the regressors).

Y
: [np.ndarray of shape (n_samples, n_parcels)] The timeseries of set of variables 2 (i.e., variables to predict, targets).

connectivity
: [np.ndarray of shape (n_parcels, n_parcels), optional, default=None] A binary matrix indicating which regressors affect which targets (i.e., variables to predict).

center_data
: [bool, default=True] Center data to zero mean.

Returns:

X_new
: [np.ndarray of shape (n_samples, n_active_parcels)] The timeseries of set of variables 1 (i.e., the regressors) after removing unused predictors and regressing out the effects indicated in connectivity.

Y_new
: [np.ndarray of shape (n_samples, n_active_parcels)] The timeseries of set of variables 2 (i.e., variables to predict, targets) after removing unused targets and regressing out the effects indicated in connectivity.

connectivity_new
: [np.ndarray of shape (n_active_parcels, n_active_parcels), optional, default=None] A binary matrix indicating which regressors affect which targets The matrix has the same structure as *connectivity* after removing unused predictors and targets.

glhmm.preproc.**build_data_tde**(*data*, *indices*, *lags*, *pca=None*, *standardise_pc=True*)

Builds X for the temporal delay embedded HMM, as well as an adapted indices array.

Parameters:

data
[numpy array of shape (n_samples, n_parcels)] The data matrix.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

lags
[list or array-like] The lags to use for the embedding.

pca
[None or int or float or numpy array, default=None] The number of components for PCA, the explained variance for PCA, the precomputed PCA projection matrix, or None to skip PCA.

standardise_pc
[bool, default=True] Whether or not to standardise the principal components before returning.

Returns:

X
[numpy array of shape (n_samples - n_sessions*rwindow, n_parcels*n_lags)] The delay-embedded time-series data.

indices_new
[numpy array of shape (n_sessions, 2)] The adapted indices for each segment of delay-embedded data.

PCA can be run optionally: if pca >=1, that is the number of components; if pca < 1, that is explained variance; if pca is a numpy array, then it is a precomputed PCA projection matrix; if pca is None, then no PCA is run.

glhmm.preproc.**load_files**(*files*, *I=None*, *do_only_indices=False*)

glhmm.preproc.**preprocess_data**(*data*, *indices*, *fs=1*, *standardise=True*, *filter=None*, *detrend=False*, *onpower=False*, *pca=None*, *whitening=False*, *exact_pca=True*, *downsample=None*)

Preprocess the input data.

Parameters:

data
[array-like of shape (n_samples, n_parcels)] The input data to be preprocessed.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

fs
[int or float, default=1] The frequency of the input data.

standardise
[bool, default=True] Whether to standardize the input data.

filter
[tuple of length 2 or None, default=None] The low-pass and high-pass thresholds to apply to the input data. If None, no filtering will be applied. If a tuple, the first element is the low-pass threshold and the second is the high-pass threshold.

detrend
[bool, default=False] Whether to detrend the input data.

onpower
[bool, default=False] Whether to calculate the power of the input data using the Hilbert transform.

pca
[int or float or None, default=None] If int, the number of components to keep after applying PCA. If float, the percentage of explained variance to keep after applying PCA. If None, no PCA will be applied.

whitening
[bool, default=False] Whether to whiten the input data after applying PCA.

exact_pca
[bool, default=True] Whether to use full SVD solver for PCA.

downsample
[int or float or None, default=None] The new frequency of the input data after downsampling. If None, no downsampling will be applied.

Returns:

data_processed
[array-like of shape (n_samples_processed, n_parcels)] The preprocessed input data.

indices_processed
[array-like of shape (n_sessions_processed, 2)] The start and end indices of each trial/session in the preprocessed data.

3.2.4 glhmm.auxiliary

Auxiliary functions - Gaussian Linear Hidden Markov Model @author: Diego Vidaurre 2022

glhmm.auxiliary.**Gamma_entropy**(*Gamma*, *Xi*, *indices*)
Computes the entropy of a Gamma distribution and a sequence of transition probabilities Xi.

Parameters:

Gamma
[Array-like of shape (n_samples, n_states)] The posterior probabilities of a hidden variable.

Xi
[Array-like of shape (n_samples - n_sessions, n_states, n_states)] The joint probability of past and future states conditioned on data.

indices
[Array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

float: The entropy of the Gamma distribution and the sequence of transition probabilities.

glhmm.auxiliary.**Gamma_indices_to_Xi_indices**(*indices*)

Converts indices from Gamma array to Xi array format.

Note Xi has 1 sample less than Gamma per trial/session (i.e., n_samples - n_sessions).

Parameters:

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

indices_Xi
[array-like of shape (n_sessions, 2)] The converted indices in Xi array format.

glhmm.auxiliary.**approximate_Xi**(*Gamma*, *indices*)

Approximates Xi array based on Gamma and indices.

Parameters:

Gamma
[array-like of shape (n_samples, n_states)] The state probability time series.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

Xi
[array-like of shape (n_samples - n_sessions, n_states, n_states)] The joint probabilities of past and future states conditioned on data.

glhmm.auxiliary.**compute_alpha_beta_parallel**(*L*, *Pi*, *P*, *indices_individual*, *gpu_acceleration*)

Computes alpha and beta values and scaling factors.

Parameters:

L
[array-like of shape (n_samples, n_timeseries, n_states)] The L matrix.

Pi
[array-like with shape (n_states,)] The initial state probabilities.

P
[array-like of shape (n_states, n_states)] The transition probabilities across states.

indices_individual
[array-like of shape (n_timeseries,2)] The first and last index of the subject-specific time series

gpu_acceleration
[int] GPU acceleration setting.

Returns:

a
[GPU array-like of shape (n_samples, n_states)] The alpha values.

b
[GPU array-like of shape (n_samples, n_states)] The beta values.

sc
[GPU array-like of shape (n_samples,)] The scaling factors.

glhmm.auxiliary.**compute_alpha_beta_serial**(*L*, *Pi*, *P*)
Computes alpha and beta values and scaling factors.

Parameters:

L
[array-like of shape (n_samples, n_states)] The L matrix.

Pi
[array-like with shape (n_states,)] The initial state probabilities.

P
[array-like of shape (n_states, n_states)] The transition probabilities across states.

Returns:

a
[array-like of shape (n_samples, n_states)] The alpha values.

b
[array-like of shape (n_samples, n_states)] The beta values.

sc
[array-like of shape (n_samples,)] The scaling factors.

glhmm.auxiliary.**compute_qstar_parallel**(*L*, *Pi*, *P*, *indices_individual*)
Compute the most probable state sequence.

Parameters:

L
[array-like of shape (n_samples, n_states)] The L matrix.

Pi
[array-like with shape (n_states,)] The initial state probabilities.

P
[array-like of shape (n_states, n_states)] The transition probabilities across states.

Returns:

qstar
[array-like of shape (n_samples, n_states)] The most probable state sequence.

glhmm.auxiliary.**compute_qstar_serial**(*L*, *Pi*, *P*)

Compute the most probable state sequence.

Parameters:

L
[array-like of shape (n_samples, n_states)] The L matrix.

Pi
[array-like with shape (n_states,)] The initial state probabilities.

P
[array-like of shape (n_states, n_states)] The transition probabilities across states.

Returns:

qstar
[array-like of shape (n_samples, n_states)] The most probable state sequence.

glhmm.auxiliary.**dirichlet_kl**(*alpha_q*, *alpha_p*)

Computes the Kullback-Leibler divergence between two Dirichlet distributions with parameters alpha_q and alpha_p.

Parameters:

alpha_q
[Array of shape (n_states,)] The concentration parameters of the first Dirichlet distribution.

alpha_p
[Array of shape (n_states,)] The concentration parameters of the second Dirichlet distribution.

Returns:

float: The Kullback-Leibler divergence between the two Dirichlet distributions.

glhmm.auxiliary.**gamma_kl**(*shape_q*, *rate_q*, *shape_p*, *rate_p*)

Computes the Kullback-Leibler divergence between two Gamma distributions with shape and rate parameters.

The Kullback-Leibler divergence is a measure of how different two probability distributions are.

This implementation follows the formula presented here (https://statproofbook.github.io/P/gam-kl) from the book "KL-Divergences of Normal, Gamma, Dirichlet and Wishart densities" by Penny, William D. in 2001.

Parameters:

shape_q
[float or numpy.ndarray] The shape parameter of the first Gamma distribution.

rate_q
[float or numpy.ndarray] The rate parameter of the first Gamma distribution.

shape_p
[float or numpy.ndarray] The shape parameter of the second Gamma distribution.

rate_p
[float or numpy.ndarray] The rate parameter of the second Gamma distribution.

Returns:

D
[float or numpy.ndarray] The Kullback-Leibler divergence between the two Gamma distributions.

glhmm.auxiliary.**gauss1d_kl**(*mu_q*, *sigma_q*, *mu_p*, *sigma_p*)

Computes the KL divergence between two univariate Gaussian distributions.

Parameters:

mu_q
[float of shape (n_parcels,)] The mean of the first Gaussian distribution.

sigma_q
[float of shape (n_parcels, n_parcels)] The variance of the first Gaussian distribution.

mu_p
[float of shape (n_parcels,)] The mean of the second Gaussian distribution.

sigma_p
[float of shape (n_parcels, n_parcels)] The variance of the second Gaussian distribution.

Returns:

D
[float] The KL divergence between the two Gaussian distributions.

glhmm.auxiliary.**gauss_kl**(*mu_q*, *sigma_q*, *mu_p*, *sigma_p*)

Computes the KL divergence between two multivariate Gaussian distributions.

Parameters:

mu_q
[float of shape (n_parcels,)] The mean of the first Gaussian distribution.

sigma_q
[float of shape (n_parcels, n_parcels)] The variance of the first Gaussian distribution.

mu_p
[float of shape (n_parcels,)] The mean of the second Gaussian distribution.

sigma_p
[float of shape (n_parcels, n_parcels)] The variance of the second Gaussian distribution.

Returns:

D
[float] The KL divergence between the two Gaussian distributions.

glhmm.auxiliary.**get_T**(*idx_data*)

Returns the timepoints spent for each trial/session based on the given indices. We want to get the variable "T" when we are using the function padGamma

Parameters:

idx_data (numpy.ndarray): The indices that mark the timepoints for when each trial/session starts and ends. It should be a 2D array where each row represents the start and end index for a trial. Example: idx_data = np.array([[0, 150], [150, 300], [300, 500]])

Returns:

T (numpy.ndarray): An array containing the timepoints spent for each trial/session. For example, given idx_data = np.array([[0, 150], [150, 300], [300, 500]]), the function would return T = np.array([150, 150, 200]).

glhmm.auxiliary.**jls_extract_def**()

glhmm.auxiliary.**make_indices_from_T**(*T*)

Creates indices array from trials/sessions lengths.

Parameters:

T
[array-like of shape (n_sessions,)] Contains the lengths of each trial/session.

Returns:

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

glhmm.auxiliary.**padGamma**(*Gamma*, *T*, *options*)

Adjusts the state time courses to have the same size as the data time series.

Parameters:

Gamma (numpy.ndarray): The state time courses. T (numpy.ndarray): Timepoints spent for each trial/session. options (dict): Dictionary containing various options. - 'embeddedlags' (list): Array of lagging times if 'embeddedlags' is specified. - 'order' (int): Integer value if 'order' is specified.

Returns:

Gamma (numpy.ndarray): Adjusted state time courses.

glhmm.auxiliary.**roll_by_vector**(*arr*, *shifts*, *axis=1*, *gpu_enabled=False*)

Apply an independent roll for each dimensions of a single axis.

Parameters

- **arr** (*np.ndarray*) – Array of any shape.
- **shifts** (*np.ndarray*) – How many shifting to use for each dimension. Shape: *(arr.shape[axis],)*.
- **axis** (*int*) – Axis along which elements are shifted.
- **gpu_enabled** (*bool*) – Whether input arrays are loaded in the GPU or not.

glhmm.auxiliary.**slice_matrix**(*M*, *indices*)

Slices rows of input matrix M based on indices array along axis 0.

Parameters:

M
[array-like of shape (n_samples, n_parcels)] The input matrix.

indices
[array-like of shape (n_sessions, 2)] The indices that define the sections (i.e., trials/sessions) of the data to be processed.

Returns:

M_sliced
[array-like of shape (n_total_samples, n_parcels)] The sliced matrix.

glhmm.auxiliary.**wishart_kl**(*shape_q*, *C_q*, *shape_p*, *C_p*)

Computes the Kullback-Leibler (KL) divergence between two Wishart distributions.

Parameters:

shape_q
[float] Shape parameter of the first Wishart distribution.

C_q
[ndarray of shape (n_parcels, n_parcels)] Scale parameter of the first Wishart distribution.

shape_p
[float] Shape parameter of the second Wishart distribution.

C_p
[ndarray of shape (n_parcels, n_parcels)] Scale parameter of the second Wishart distribution.

Returns:

D
[float] KL divergence from the first to the second Wishart distribution.

3.2.5 glhmm.utils

Some public useful functions - Gaussian Linear Hidden Markov Model @author: Diego Vidaurre 2023

glhmm.utils.**get_FO**(*Gamma*, *indices*, *summation=False*)
Calculates the fractional occupancy of each state.

Parameters:

Gamma
[array-like, shape (n_samples, n_states)] The state probability time series.

indices
[array-like, shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

summation
[bool, optional, default=False] If True, the sum of each row is not normalized, otherwise it is.

Returns:

FO
[array-like, shape (n_sessions, n_states)] The fractional occupancy of each state per session.

glhmm.utils.**get_FO_entropy**(*Gamma*, *indices*)
Calculates the entropy of each session, if we understand fractional occupancies as probabilities.

Parameters:

Gamma
[array-like of shape (n_samples, n_states)] The Gamma represents the state probability timeseries.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

entropy
[array-like of shape (n_sessions,)] The entropy of each session.

glhmm.utils.**get_gamma_similarity**(*gamma1*, *gamma2*)

Computes a measure of similarity between two sets of state time courses.

These can have different numbers of states, but they must have the same number of time points.

Parameters:

gamma1
[numpy.ndarray] First set of state time courses with shape (T, K).

gamma2
[numpy.ndarray] Second set of state time courses with shape (T, K2), where K2 may be different from K.

Returns:

S
[float] Similarity, measured as the sum of joint probabilities under the optimal state alignment.

assig
[numpy.ndarray] Optimal state alignment for gamma2 (uses Munkres' algorithm).

gamma2
[numpy.ndarray] The second set of state time courses reordered to match gamma1.

glhmm.utils.**get_life_times**(*vpath*, *indices*, *threshold=0*)

Calculates the average, median and maximum life times for each state.

Parameters:

vpath
[array-like of shape (n_samples,)] The viterbi path represents the most likely state sequence.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

threshold
[int, optional, default=0] A threshold value used to exclude visits with a duration below this value.

Returns:

meanLF
[array-like of shape (n_sessions, n_states)] The average visit duration for each state in each trial/session.

medianLF
[array-like of shape (n_sessions, n_states)] The median visit duration for each state in each trial/session.

maxLF
[array-like of shape (n_sessions, n_states)] The maximum visit duration for each state in each trial/session.

Notes:

A visit to a state is defined as a contiguous sequence of time points in which the state is active. The duration of a visit is the number of time points in the sequence. This function uses the *get_visits* function to compute the visits and exclude those below the threshold.

glhmm.utils.**get_maxFO**(*Gamma*, *indices*)

Calculates the maximum fractional occupancy per session.

The first argument can also be a viterbi path (vpath).

Parameters:

Gamma
[array-like of shape (n_samples, n_states); or a vpath, array of shape (n_samples,)] The Gamma represents the state probability timeseries and the vpath represents the most likely state sequence.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

maxFO: array-like of shape (n_sessions,)
The maximum fractional occupancy across states for each trial/session

Notes:

The maxFO is useful to assess the amount of *state mixing*. For more information, see [^1].

References:

[^1]: Ahrends, R., et al. (2022). Data and model considerations for estimating time-varying functional connectivity in fMRI. NeuroImage 252, 119026.
https://pubmed.ncbi.nlm.nih.gov/35217207/)

glhmm.utils.**get_state_evoked_response**(*Gamma*, *indices*)

Calculates the state evoked response

The first argument can also be a viterbi path (vpath).

Parameters:

Gamma
[array-like of shape (n_samples, n_states), or a vpath array of shape (n_samples,)] The Gamma represents the state probability timeseries and the vpath represents the most likely state sequence.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

ser
[array-like of shape (n_samples, n_states)] The state evoked response matrix.

Raises:

Exception
If the input data violates any of the following conditions: - There is only one trial/session - Not all trials/sessions have the same length.

glhmm.utils.**get_state_evoked_response_entropy**(*Gamma*, *indices*)

Calculates the entropy of each time point, if we understand state evoked responses as probabilities.

Parameters:

Gamma: array-like of shape (n_samples, n_states)
The Gamma represents the state probability timeseries.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

entropy: array-like of shape (n_samples,)
The entropy of each time point.

glhmm.utils.**get_state_onsets**(*vpath*, *indices*, *threshold=0*)

Calculates the state onsets, i.e., the time points when each state activates.

Parameters:

vpath
[array-like of shape (n_samples, n_states)] The viterbi path represents the most likely state sequence.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

threshold
[int, optional, default=0] A threshold value used to exclude visits with a duration below this value.

Returns:

onsets
[list of lists of ints] A list of the time points when each state activates for each trial/session.

Notes:

A visit to a state is defined as a contiguous sequence of time points in which the state is active. This function uses the *get_visits* function to compute the visits and exclude those below the threshold.

glhmm.utils.**get_switching_rate**(*Gamma*, *indices*)

Calculates the switching rate.

The first argument can also be a viterbi path (vpath).

Parameters:

Gamma
[array-like of shape (n_samples, n_states), or a vpath array of shape (n_samples,)] The Gamma represents the state probability timeseries and the vpath represents the most likely state sequence.

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data.

Returns:

SR
[array-like of shape (n_sessions, n_states)] The switching rate matrix.

glhmm.utils.**get_visits**(*vpath*, *k*, *threshold=0*)

Computes a list of visits for state k, given a viterbi path (vpath).

Parameters:

vpath
[array-like of shape (n_samples,)] The viterbi path represents the most likely state sequence.

k
[int] The state for which to compute the visits.

threshold
[int, optional, default=0] A threshold value used to exclude visits with a duration below this value.

Returns:

lengths
[list of floats] A list of visit durations for state k, where each duration is greater than the threshold.

onsets
[list of ints] A list of onset time points for each visit.

Notes:

A visit to state k is defined as a contiguous sequence of time points in which state k is active.

3.2.6 glhmm.graphics

Basic graphics - Gaussian Linear Hidden Markov Model @author: Diego Vidaurre 2023

`glhmm.graphics.`**`blue_colormap`**`()`
Generate a custom blue colormap.

Returns:

A custom colormap with shades of blue.

`glhmm.graphics.`**`create_cmap_alpha`**`(`*cmap_list*, *color_array*, *alpha*`)`
Modify the colors in a colormap based on an alpha threshold.

Parameters:

cmap_list (numpy.ndarray)
List of colors representing the original colormap.

color_array (numpy.ndarray)
Array of color values corresponding to each colormap entry.

alpha (float)
Alpha threshold for modifying colors.

Returns:

Modified list of colors representing the colormap with adjusted alpha values.

`glhmm.graphics.`**`custom_colormap`**`()`
Generate a custom colormap consisting of segments from red to blue.

Returns:

A custom colormap with defined color segments.

glhmm.graphics.**interpolate_colormap**(*cmap_list*)

Create a new colormap with the modified color_array.

Parameters:

cmap_list (numpy.ndarray):
Original color array for the colormap.

Returns:

modified_cmap (numpy.ndarray):
Modified colormap array.

glhmm.graphics.**plot_FO**(*FO*, *figsize=(8, 4)*, *fontsize_labels=13*, *fontsize_title=16*, *width=0.8*, *xlabel='Subject'*, *ylabel='Fractional occupancy'*, *title='State Fractional Occupancies'*, *show_legend=True*, *num_ticks=10*)

Plot fractional occupancies for different states.

Parameters:

FO (numpy.ndarray):
Fractional occupancy data matrix.

figsize (tuple, optional), default=(8,4):
Figure size.

fontsize_labels (int, optional), default=13:
Font size for axes labels.

fontsize_title (int, optional), default=16:
Font size for plot title.

width (float, optional), default=0.5:
Width of the bars.

xlabel (str, optional), default='Subject':
Label for the x-axesis.

ylabel (str, optional), default='Fractional occupancy':
Label for the y-axesis.

title (str, optional), default='State Fractional Occupancies':
Title for the plot.

show_legend (bool, optional), default=True:
Whether to show the legend.

glhmm.graphics.**plot_average_probability**(*Gamma_reconstruct*, *title='Average probability for each state'*, *fontsize=16*, *figsize=(7, 5)*, *vertical_lines=None*, *line_colors=None*, *highlight_boxes=False*)

Plots the average probability for each state over time.

Parameters:

Gamma_reconstruct (numpy.ndarray)
3D array representing reconstructed gamma values. Shape: (num_timepoints, num_trials, num_states)

title (str, optional), default='Average probability for each state':
Title for the plot.

fontsize (int, optional), default=16:
Font size for labels and title.

figsize (tuple, optional), default=(8,6):
Figure size (width, height) in inches).

vertical_lines (list of tuples, optional), default=None:
List of pairs specifying indices for vertical lines.

line_colors (list of str or bool, optional), default=None:
List of colors for each pair of vertical lines. If True, generates random colors (unless a list is provided).

highlight_boxes (bool, optional), default=False:
Whether to include highlighted boxes for each pair of vertical lines.

glhmm.graphics.**plot_condition_difference**(*Gamma_reconstruct*, *R_trials*, *title='Average Probability and Difference'*, *fontsize=16*, *figsize=(9, 2)*, *vertical_lines=None*, *line_colors=None*, *highlight_boxes=False*)

Plots the average probability for each state over time for two conditions and their difference.

Parameters:

Gamma_reconstruct (numpy.ndarray)
3D array representing reconstructed gamma values. Shape: (num_timepoints, num_trials, num_states)

R_trials (numpy.ndarray)
1D array representing the condition for each trial. Should have the same length as the second dimension of Gamma_reconstruct.

title (str, optional), default='Average Probability and Difference':
Title for the plot.

fontsize (int, optional), default=16:
Font size for labels and title.

figsize (tuple, optional), default=(9, 2):
Figure size (width, height).

vertical_lines (list of tuples, optional), default=None:
List of pairs specifying indices for vertical lines.

line_colors (list of str or bool, optional), default=None:
List of colors for each pair of vertical lines. If True, generates random colors (unless a list is provided).

highlight_boxes (bool, optional), default=False:
Whether to include highlighted boxes for each pair of vertical lines.

Example usage:

plot_condition_difference(Gamma_reconstruct, R_trials, vertical_lines=[(10, 100)], highlight_boxes=True)

glhmm.graphics.**plot_correlation_matrix**(*corr_vals*, *performed_tests*, *normalize_vals=False*, *figsize=(9, 5)*, *title_text='Correlation Coefficients Heatmap'*, *annot=False*, *cmap_type='default'*, *cmap_reverse=True*, *xlabel=''*, *ylabel=''*, *xticklabels=None*, *xlabel_rotation=45*, *none_diagonal=False*, *num_colors=256*)

glhmm.graphics.**plot_p_value_matrix**(*pval*, *alpha=0.05*, *normalize_vals=True*, *figsize=(9, 5)*, *title_text='Heatmap (p-values)'*, *annot=False*, *cmap_type='default'*, *cmap_reverse=True*, *xlabel=''*, *ylabel=''*, *xticklabels=None*, *none_diagonal=False*, *num_colors=259*, *xlabel_rotation=0*)

glhmm.graphics.**plot_p_values_bar**(*pval*, *xticklabels=[]*, *figsize=(9, 4)*, *num_colors=256*, *xlabel=''*, *ylabel='P-values (Log Scale)'*, *title_text='Bar Plot'*, *tick_positions=[0, 0.001, 0.01, 0.05, 0.1, 0.3, 1]*, *top_adjustment=0.9*, *alpha=0.05*, *pad_title=20*, *xlabel_rotation=45*, *pval_text_hight_same=False*)

Visualize a bar plot with LogNorm and a colorbar.

Parameters:

pval (numpy.ndarray):
Array of p-values to be plotted.

xticklabels (list, optional), default=[]:
List of categories or variables.

figsize (tuple, optional), default=(9, 4):
Figure size in inches (width, height).

num_colors (int, optional), default=256:
Number of colors in the colormap.

xlabel (str, optional), default=””:
X-axis label.

ylabel (str, optional), default=”P-values (Log Scale)”:
Y-axis label.

title_text (str, optional), default=”Bar Plot”:
Title for the plot.

tick_positions (list, optional), default=[0, 0.001, 0.01, 0.05, 0.1, 0.3, 1]
Positions of ticks on the colorbar.

top_adjustment (float, optional), default=0.9:
Adjustment for extra space between title and plot.

alpha (float, optional), default=0.05:
Alpha value is the threshold we set for the p-values when doing visualization.

pad_title (int, optional), default=20:
Padding for the plot title.

glhmm.graphics.**plot_p_values_over_time**(*pval*, *figsize=(8, 4)*, *total_time_seconds=None*, *xlabel='Timepoints'*, *ylabel='P-values (Log Scale)'*, *title_text='P-values over time'*, *xlim_start=0*, *tick_positions=[0, 0.001, 0.01, 0.05, 0.1, 0.3, 1]*, *num_colors=259*, *alpha=0.05*, *plot_style='line'*, *linewidth=2.5*)

Plot a scatter plot of p-values over time with a log-scale y-axis and a colorbar.

Parameters:

pval (numpy.ndarray):
The p-values data to be plotted.

figsize
[tuple, optional, default=(8, 4):] Figure size in inches (width, height).

total_time_seconds
[float, optional, default=None] Total time duration in seconds. If provided, time points will be scaled accordingly.

xlabel (str, optional), default="Timepoints":
Label for the x-axis.

ylabel (str, optional), default="P-values (Log Scale)":
Label for the y-axis.

title_text (str, optional), default="P-values over time":
Title for the plot.

xlim_start (int, optional), default=0:
Starting point for the x-axis limit.

tick_positions (list, optional), default=[0, 0.001, 0.01, 0.05, 0.1, 0.3, 1]:
Specific values to mark on the y-axis.

num_colors (int, optional), default=259:
Resolution for the color bar.

alpha (float, optional), default=0.05:
Alpha value is the threshold we set for the p-values when doing visualization.

plot_style (str, optional), default="line":
Style of plot.

linewidth (float, optional), default=2.5:
Width of the lines in the plot.

glhmm.graphics.**plot_permutation_distribution**(*test_statistic*, *title_text='Permutation Distribution'*, *xlabel='Test Statistic Values'*, *ylabel='Density'*)

Plot the histogram of the permutation with the observed statistic marked.

Parameters:

test_statistic (numpy.ndarray)
An array containing the permutation values.

title_text (str, optional), default="Permutation Distribution":
Title text of the plot.

xlabel (str, optional), default="Test Statistic Values"
Text of the xlabel.

ylabel (str, optional), default="Density"
Text of the ylabel.

glhmm.graphics.**plot_scatter_with_labels**(*p_values*, *alpha=0.05*, *title_text=''*, *xlabel=None*, *ylabel=None*, *xlim_start=0.9*, *ylim_start=0*)

Create a scatter plot to visualize p-values with labels indicating significant points.

Parameters:

p_values (numpy.ndarray)
An array of p-values. Can be a 1D array or a 2D array with shape (1, 5).

alpha (float, optional), default=0.05:
Threshold for significance.

title_text (str, optional), default="":
The title text for the plot.

xlabel (str, optional), default=None:
The label for the x-axis.

ylabel (str, optional), default=None:
The label for the y-axis.

xlim_start (float, optional), default=-5
Start position of x-axis limits.

ylim_start (float, optional), default=-0.1
Start position of y-axis limits.

Notes:

Points with p-values less than alpha are considered significant and marked with red text.

glhmm.graphics.**plot_state_lifetimes**(*LT*, *figsize=(8, 4)*, *fontsize_labels=13*, *fontsize_title=16*, *width=0.18*, *xlabel='Subject'*, *ylabel='Lifetime'*, *title='State Lifetimes'*, *show_legend=True*, *num_ticks=10*)

Plot state lifetimes for different states.

Parameters:

LT (numpy.ndarray):
State lifetime (dwell time) data matrix.

figsize (tuple, optional), default=(8, 4):
Figure size.

fontsize_labels (int, optional), default=13:
Font size for axeses labels.

fontsize_title (int, optional), default=16:
Font size for plot title.

width (float, optional), default=0.18:
Width of the bars.

xlabel (str, optional), default='Subject':
Label for the x-axesis.

ylabel (str, optional), default='Lifetime':
Label for the y-axesis.

title (str, optional), default='State Lifetimes':
Title for the plot.

show_legend (bool, optional), default=True:
Whether to show the legend.

glhmm.graphics.**plot_state_prob_and_covariance**(*init_stateP*, *TP*, *state_means*, *state_FC*, *cmap='viridis'*, *figsize=(9, 7)*, *num_ticks=5*)

Plot HMM parameters.

Parameters:

init_stateP
[array-like] Initial state probabilities.

TP
[array-like] Transition probabilities.

state_means
[array-like] State means.

state_FC
[array-like] State covariances.

cmap
[str or Colormap, optional] The colormap to be used for plotting. Default is 'viridis'.

figsize
[tuple, optional] Figure size. Default is (9, 7).

num_ticks
[int, optional] Number of ticks for the colorbars

glhmm.graphics.**plot_switching_rates**(*SR*, *figsize=(8, 4)*, *fontsize_labels=13*, *fontsize_title=16*, *width=0.18*, *xlabel='Subject'*, *ylabel='Switching Rate'*, *title='State Switching Rates'*, *show_legend=True*, *num_ticks=10*)

Plot switching rates for different states.

Parameters:

SR (numpy.ndarray):
Switching rate data matrix.

figsize (tuple, optional), default=(8, 4):
Figure size.

fontsize_labels (int, optional), default=13:
Font size for axes labels.

fontsize_title (int, optional), default=16:
Font size for plot title.

width (float, optional), default=0.18:
Width of the bars.

xlabel (str, optional), default='Subject':
Label for the x-axesis.

ylabel (str, optional), default='Switching Rate':
Label for the y-axesis.

title (str, optional), default='State Switching Rates':
Title for the plot.

show_legend (bool, optional), default=True:
Whether to show the legend.

glhmm.graphics.**plot_vpath**(*viterbi_path*, *signal=None*, *idx_data=None*, *figsize=(7, 4)*, *fontsize_labels=13*, *fontsize_title=16*, *yticks=None*, *time_conversion_rate=None*, *xlabel='Timepoints'*, *ylabel=''*, *title='Viterbi Path'*, *signal_label='Signal'*, *show_legend=True*, *vertical_linewidth=1.5*)

Plot Viterbi path with optional signal overlay.

Parameters:

viterbi_path
The Viterbi path data matrix.

signal
[array-like, optional] Signal data to overlay on the plot. Default is None.

idx_data
[array-like, optional] Array representing time intervals. Default is None.

figsize
[tuple, optional] Figure size. Default is (7, 4).

fontsize_labels
[int, optional] Font size for axis labels. Default is 13.

fontsize_title
[int, optional] Font size for plot title. Default is 16.

yticks
[bool, optional] Whether to show y-axis ticks. Default is None.

time_conversion_rate
[float, optional] Conversion rate from time steps to seconds. Default is None.

xlabel
[str, optional] Label for the x-axis. Default is "Timepoints".

ylabel
[str, optional] Label for the y-axis. Default is "".

title
[str, optional] Title for the plot. Default is "Viterbi Path".

signal_label
[str, optional] Label for the signal plot. Default is "Signal".

show_legend
[bool, optional] Whether to show the legend. Default is True.

vertical_linewidth
[float, optional] Line width for vertical gray lines. Default is 1.5.

glhmm.graphics.**red_colormap**()

Generate a custom colormap consisting of red and warm colors.

Returns:

A custom colormap with red and warm color segments.

glhmm.graphics.**show_Gamma**(*Gamma*, *line_overlay=None*, *tlim=None*, *Hz=1*, *palette='viridis'*)

Displays the activity of the hidden states as a function of time.

Parameters:

Gamma
[array of shape (n_samples, n_states)] The state timeseries probabilities.

line_overlay
[array of shape (n_samples, 1)] A secondary related data type to overlay as a line.

tlim
[2x1 array or None, default=None] The time interval to be displayed. If None (default), displays the entire sequence.

Hz
[int, default=1] The frequency of the signal, in Hz.

palette
[str, default = 'Oranges'] The name of the color palette to use.

glhmm.graphics.**show_beta**(*hmm*, *only_active_states=True*, *recompute_states=False*, *X=None*, *Y=None*, *Gamma=None*, *show_average=None*, *alpha=1.0*)

Displays the beta coefficients of a given HMM. The beta coefficients can be extracted directly from the HMM structure or reestimated from the data; for the latter, X, Y and Gamma need to be provided as parameters. This is useful for example if one has run the model on PCA space, but wants to show coefficients in the original space.

Parameters:

hmm: HMM object
An instance of the HMM class containing the beta coefficients to be visualized.

only_active_states: bool, optional, default=False
If True, only the beta coefficients of active states are shown.

recompute_states: bool, optional, default=False
If True, the betas will be recomputed from the data and the state time courses

X: numpy.ndarray, optional, default=None
The timeseries of set of variables 1.

Y: numpy.ndarray, optional, default=None
The timeseries of set of variables 2.

Gamma: numpy.ndarray, optional, default=None
The state time courses

show_average: bool, optional, default=None
If True, an additional row of the average beta coefficients is shown.

alpha: float, optional, default=0.1
The regularisation parameter to be applied if the betas are to be recomputed.

glhmm.graphics.**show_temporal_statistic**(*Gamma*, *indices*, *statistic='FO'*, *type_plot='barplot'*)

Plots a statistic over time for a set of sessions.

Parameters:

Gamma
[array of shape (n_samples, n_states)] The state timeseries probabilities.

indices: numpy.ndarray of shape (n_sessions,)
The session indices to plot.

statistic: str, default='FO'
The statistic to compute and plot. Can be 'FO', 'switching_rate' or 'FO_entropy'.

type_plot: str, default='barplot'
The type of plot to generate. Can be 'barplot', 'boxplot' or 'matrix'.

Raises:

Exception

- Statistic is not one of 'FO', 'switching_rate' or 'FO_entropy'.
- type_plot is 'boxplot' and there are less than 10 sessions.
- type_plot is 'matrix' and there is only one session.

glhmm.graphics.**show_trans_prob_mat**(*hmm*, *only_active_states=False*, *show_diag=True*, *show_colorbar=True*)

Displays the transition probability matrix of a given HMM.

Parameters:

hmm: HMM object
An instance of the HMM class containing the transition probability matrix to be visualized.

only_active_states
[bool, optional, default=False] Whether to display only active states or all states in the matrix.

show_diag
[bool, optional, defatult=True] Whether to display the diagonal elements of the matrix or not.

show_colorbar
[bool, optional, default=True] Whether to display the colorbar next to the matrix or not.

3.2.7 glhmm.prediction

Prediction from Gaussian Linear Hidden Markov Model @author: Christine Ahrends 2023

glhmm.prediction.**classify_phenotype**(*hmm*, *Y*, *behav*, *indices*, *predictor='Fisherkernel'*, *estimator='SVM'*, *options=None*)

Classify phenotype from HMM This uses either the Fisher kernel (default) or a set of HMM summary metrics to make a classification, in a nested cross-validated way. By default, X is standardised/centered. Estimators so far include: SVM and Logistic Regression Cross-validation strategies so far include: KFold and GroupKFold Hyperparameter optimization strategies so far include: only grid search

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data

behav
[array-like of shape (n_sessions,)] phenotype, behaviour, or other external labels to be predicted

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data. Note that this function does not work if indices=None

predictor
[char (optional, default to 'Fisherkernel')] What to predict from, either 'Fisherkernel' or 'summary_metrics' (default='Fisherkernel')

estimator
[char (optional, default to 'SVM')] Model to be used for classification (default='SVM') This should be the name of a sklearn base estimator (for now either 'SVM' or 'LogisticRegression')

options
[dict (optional, default to None)]

general relevant options are:
'CVscheme': char, which CVscheme to use (default: 'GroupKFold' if group structure is specified, otherwise: KFold) 'nfolds': int, number of folds k for (outer and inner) k-fold CV loops 'group_structure': ndarray of (n_sessions, n_sessions), matrix specifying group structure: positive values if sessions(/subjects) are related, zeros otherwise 'return_scores': bool, whether to return also the model scores of each fold 'return_models': bool, whether to return also the trained models of each

fold 'return_hyperparams': bool, whether to return also the optimised hyperparameters of each fold possible hyperparameters for model, e.g. 'alpha' for (kernel) ridge regression 'return_prob': bool, whether to return also the estimated probabilities

for Fisher kernel, relevant options are:
'shape': char, either 'linear' or 'Gaussian' (TO DO) 'incl_Pi': bool, whether to include the gradient w.r.t. the initial state probabilities when computing the Fisher kernel 'incl_P': bool, whether to include the gradient w.r.t. the transition probabilities 'incl_Mu': bool, whether to include the gradient w.r.t. the state means (note that this only works if means were not set to 0 when training HMM) 'incl_Sigma': bool, whether to include the gradient w.r.t. the state covariances

for summary metrics, relevant options are:
'metrics': list of char, containing metrics to be included as features

Returns:

results
[dict] containing 'behav_pred': predicted labels on test sets 'acc': overall accuracy (if requested): 'behav_prob': predicted probabilities of each class on test set 'scores': the model scores of each fold 'models': the trained models from each fold 'hyperparams': the optimised hyperparameters of each fold

Raises:

Exception
If the hmm has not been trained or if necessary input is missing

Notes:

If behav contains NaNs, these subjects/sessions will be removed in Y and confounds

glhmm.prediction.**compute_gradient**(*hmm*, *Y*, *incl_Pi=True*, *incl_P=True*, *incl_Mu=False*, *incl_Sigma=True*)

Computes the gradient of the log-likelihood for timeseries Y with respect to specified HMM parameters

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (subject- or session-level) timeseries data

incl_Pi
[bool, default=True] whether to compute gradient w.r.t state probabilities

incl_P
[bool, default=True] whether to compute gradient w.r.t. transition probabilities

incl_Mu
[bool, default=False] whether to compute gradient w.r.t state means (only possible if state means were estimated during training)

incl_Sigma
[bool, default=False] whether to compute gradient w.r.t. state covariances (for now only for full covariance matrix)

Returns:

hmmgrad : array of shape (sum(len(requested_parameters)))

Raises:

Exception
If the model has not been trained or if requested parameters do not exist (e.g. if Mu is requested but state means were not estimated)

Notes:

Does not include gradient computation for X and beta

glhmm.prediction.**deconfound**(*Y*, *confX*, *betaY=None*, *my=None*)

Deconfound

glhmm.prediction.**get_groups**(*group_structure*)

Util function to get groups from group structure matrix such as family structure. Output can be used to make sure groups/families are not split across folds during cross validation, e.g. using sklearn's GroupKFold. Groups are defined as components in the adjacency matrix.

Parameter:

group_structure
[array-like of shape (n_sessions, n_sessions)] a matrix specifying the structure of the dataset, with positive values indicating relations between sessions(/subjects) and zeros indicating no relations. Note: The diagonal will be set to 1

Returns:

cs
[array-like of shape (n_sessions,)] 1D array containing the group each session belongs to

glhmm.prediction.**get_summ_features**(*hmm*, *Y*, *indices*, *metrics*)

Util function to get summary features from HMM. Output can be used as input features for ML

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data. Note that kernel cannot be computed if indices=None

metrics
[list] names of metrics to be extracted. For now, this should be one or more of 'FO', 'switching_rate', 'lifetimes'

Returns:

features
[array-like of shape (n_sessions, n_features)] The HMM summary metrics collected into a feature matrix

glhmm.prediction.**hmm_kernel**(*hmm*, *Y*, *indices*, *type='Fisher'*, *shape='linear'*, *incl_Pi=True*, *incl_P=True*, *incl_Mu=False*, *incl_Sigma=True*, *tau=None*, *return_feat=False*, *return_dist=False*)

Constructs a kernel from an HMM, as well as the respective feature matrix and/or distance matrix

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data. Note that kernel cannot be computed if indices=None

type
[str, optional] The type of kernel to be constructed (default: 'Fisher')

shape
[str, optional] The shape of kernel to be constructed, either 'linear' or 'Gaussian' (default: 'linear')

incl_Pi
[bool, default=True] whether to include state probabilities in kernel construction

incl_P
[bool, default=True] whether to include transition probabilities in kernel construction

incl_Mu
[bool, default=False] whether to include state means in kernel construction (only possible if state means were estimated during training)

incl_Sigma
[bool, default=False] whether to include state covariances in kernel construction (for now only for full covariance matrix)

return_feat
[bool, default=False] whether to return also the feature matrix

return_dist
[bool, default=False] whether to return also the distance matrix

Returns:

kernel
[array of shape (n_sessions, n_sessions)] HMM Kernel for subjects/sessions contained in Y

feat
[array of shape (n_sessions, sum(len(requested_parameters)))] Feature matrix for subjects/sessions contained in Y for requested parameters

dist
[array of shape (n_sessions, n_sessions)] Distance matrix for subjects/sessions contained in Y

Raises:

Exception
If the hmm has not been trained or if requested parameters do not exist (e.g. if Mu is requested but state means were not estimated) If kernel other than Fisher kernel is requested

Notes:

Does not include X and beta in kernel construction Only Fisher kernel implemented at this point

glhmm.prediction.**predict_phenotype**(*hmm*, *Y*, *behav*, *indices*, *predictor='Fisherkernel'*, *estimator='KernelRidge'*, *options=None*)

Predict phenotype from HMM This uses either the Fisher kernel (default) or a set of HMM summary metrics to predict a phenotype, in a nested cross-validated way. By default, X and Y are standardised/centered unless deconfounding is used. Estimators so far include: Kernel Ridge Regression and Ridge Regression Cross-validation strategies so far include: KFold and GroupKFold Hyperparameter optimization strategies so far include: only grid search

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data

behav
[array-like of shape (n_sessions,)] phenotype, behaviour, or other external variable to be predicted

indices
[array-like of shape (n_sessions, 2)] The start and end indices of each trial/session in the input data. Note that this function does not work if indices=None

predictor
[char (optional, default to 'Fisherkernel')] What to predict from, either 'Fisherkernel' or 'summary_metrics' (default='Fisherkernel')

estimator
[char (optional, default to 'KernelRidge')] Model to be used for prediction (default='KernelRidge') This should be the name of a sklearn base estimator (for now either 'KernelRidge' or 'Ridge')

options
[dict (optional, default to None)]

general relevant options are:
'CVscheme': char, which CVscheme to use (default: 'GroupKFold' if group structure is specified, otherwise: KFold) 'nfolds': int, number of folds k for (outer and inner) k-fold CV loops 'group_structure': ndarray of (n_sessions, n_sessions), matrix specifying group structure: positive values if sessions(/subjects) are related, zeros otherwise 'confounds': array-like of shape (n_sessions,) or (n_sessions, n_confounds) containing confounding variables 'return_scores': bool, whether to return also the model scores of each fold 'return_models': bool, whether to return also the trained models of each fold 'return_hyperparams': bool, whether to return also the optimised hyperparameters of each fold possible hyperparameters for model, e.g. 'alpha' for (kernel) ridge regression

for Fisher kernel, relevant options are:
'shape': char, either 'linear' or 'Gaussian' (TO DO) 'incl_Pi': bool, whether to include the gradient w.r.t. the initial state probabilities when computing the Fisher kernel 'incl_P': bool, whether to include the gradient w.r.t. the transition probabilities 'incl_Mu': bool, whether to include the gradient w.r.t. the state means (note that this only works if means were not set to 0 when training HMM) 'incl_Sigma': bool, whether to include the gradient w.r.t. the state covariances

for summary metrics, relevant options are:
'metrics': list of char, containing metrics to be included as features

Returns:

results
[dict] containing 'behav_pred': predicted phenotype on test sets 'corr': correlation coefficient between predicted and actual values (if requested): 'scores': the model scores of each fold 'models': the trained models from each fold 'hyperparams': the optimised hyperparameters of each fold

Raises:

Exception
If the hmm has not been trained or if necessary input is missing

Notes:

If behav contains NaNs, these subjects/sessions will be removed in Y and confounds

glhmm.prediction.**reconfound**(*Y*, *conf*, *betaY*, *my*)

Reconfound

glhmm.prediction.**test_classif**(*hmm*, *Y*, *indices*, *model_tuned*, *scaler_x*, *behav=None*, *train_indices=None*, *predictor='Fisherkernel'*, *estimator='SVM'*, *options=None*)

Test classification model from HMM This uses either the Fisher kernel (default) or a set of HMM summary metrics to make a classification, in a nested cross-validated way. The specified predictor and estimator must be the same as the ones used to train the classifier. By default, X is standardised/centered. Note: When using a kernel method (e.g. Fisher kernel), Y must be the timeseries of both training and test set to construct the correct kernel, and indices of the training sessions (train_indices) must be provided. When using summary metrics, Y must be the timeseries of only the test set, and train_indices should be None.

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data of test set

indices
[array-like of shape (n_test_sessions, 2) or (n_sessions, 2)] The start and end indices of each trial/session in the test data (when using features) or in the train and test data (when using kernel). Note that this function does not work if indices=None

model_tuned
[estimator] the trained and (if applicable) hyperparameter-optimised scikit-learn estimator

scaler_x
[estimator] the trained standard scaler/kernel centerer of the features/kernel x

behav
[array-like of shape (n_test_sessions,) (optional)] phenotype, behaviour, or other external label of test set, to be compared with the predicted labels

train_indices
[array-like of shape (n_train_sessions,) (optional, only use when using kernel)] the indices of the sessions/subjects used for training. The function assumes that test indices are all other sessions.

predictor
[char (optional, default to 'Fisherkernel')] What to predict from, either 'Fisherkernel' or 'summary_metrics' (default='Fisherkernel')

estimator
[char (optional, default to 'SVM')] Model to be used for classification (default='SVM') This should be the name of a sklearn base estimator (for now either 'SVM' or 'LogisticRegression')

options
[dict (optional, default to None)]

general relevant options are:
'return_prob': bool, whether to return also the estimated probabilities 'return_models': whether to return also the model

for Fisher kernel, relevant options are:
'shape': char, either 'linear' or 'Gaussian' (TO DO) 'incl_Pi': bool, whether to include the gradient w.r.t. the initial state probabilities when computing the Fisher kernel 'incl_P': bool, whether to include the gradient w.r.t. the transition probabilities 'incl_Mu': bool, whether to include the gradient w.r.t. the state means (note that this only works if means were not set to 0 when training HMM) 'incl_Sigma': bool, whether to include the gradient w.r.t. the state covariances

for summary metrics, relevant options are:
'metrics': list of char, containing metrics to be included as features

Returns:

results
[dict] containing 'behav_pred': predicted labels on test sets 'acc': overall accuracy (if requested): 'behav_prob': predicted probabilities of each class on test set 'scores': the model scores of each fold 'models': the trained model

Raises:

Exception
If the hmm has not been trained or if necessary input is missing

glhmm.prediction.**test_pred**(*hmm*, *Y*, *indices*, *model_tuned*, *scaler_x*, *scaler_y=None*, *behav=None*, *train_indices=None*, *CinterceptY=None*, *CbetaY=None*, *predictor='Fisherkernel'*, *estimator='KernelRidge'*, *options=None*)

Test prediction model from HMM This uses either the Fisher kernel (default) or a set of HMM summary metrics to predict a phenotype, in a nested cross-validated way. The specified predictor and estimator must be the same as the ones used to train the model. By default, X and Y are standardised/centered unless deconfounding is used. Note: When using a kernel method (e.g. Fisher kernel), Y must be the timeseries of both training and test set to construct the correct kernel, and indices of the training sessions (train_indices) must be provided. When using summary metrics, Y must be the timeseries of only the test set, and train_indices should be None. When using deconfounding, CinterceptY and CbetaY need to be specified

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data of test set

indices
[array-like of shape (n_test_sessions, 2) or (n_sessions, 2)] The start and end indices of each trial/session in the test data (when using features) or in the train and test data (when using kernel). Note that this function does not work if indices=None

model_tuned
[estimator] the trained and (if applicable) hyperparameter-optimised scikit-learn estimator

scaler_x
[estimator] the trained standard scaler/kernel centerer of the features/kernel x

scaler_y
[estimator (optional, only specify when not using deconfounding)] the trained standard scaler of the variable to be predicted y.

behav
[array-like of shape (n_test_sessions,) (optional)] phenotype, behaviour, or other external variable of test set, to be compared with the predicted values

train_indices
[array-like of shape (n_train_sessions,) (optional, only use when using kernel)] the indices of the sessions/subjects used for training. The function assumes that test indices are all other sessions.

CinterceptY
[float (optional, only specify when using deconfounding)] the estimated intercept for deconfounding

CbetaY
[array-like of shape (n_confounds) (optional, only specify when using deconfounding)] the estimated beta weights for deconfounding of each confound

predictor
[char (optional, default to 'Fisherkernel')] What to predict from, either 'Fisherkernel' or 'summary_metrics' (default='Fisherkernel')

estimator
[char (optional, default to 'KernelRidge')] Model to be used for prediction (default='KernelRidge') This should be the name of a sklearn base estimator (for now either 'KernelRidge' or 'Ridge')

options
[dict (optional, default to None)]

general relevant options are:
'confounds': array-like of shape (n_test_sessions,) or (n_test_sessions, n_confounds) containing confounding variables 'return_models': whether to return also the model

for Fisher kernel, relevant options are:
'shape': char, either 'linear' or 'Gaussian' (TO DO) 'incl_Pi': bool, whether to include the gradient w.r.t. the initial state probabilities when computing the Fisher kernel 'incl_P': bool, whether to include the gradient w.r.t. the transition probabilities 'incl_Mu': bool, whether to include the gradient w.r.t. the state means (note that this only works if means were not set to 0 when training HMM) 'incl_Sigma': bool, whether to include the gradient w.r.t. the state covariances

for summary metrics, relevant options are:
'metrics': list of char, containing metrics to be included as features

Returns:

results
[dict] containing 'behav_pred': predicted phenotype on test sets (if behav was specified): 'corr': correlation coefficient between predicted and actual values 'scores': the model scores of each fold (if requested): 'model': the trained model

Raises:

Exception
If the hmm has not been trained or if necessary input is missing

glhmm.prediction.**train_classif**(*hmm*, *Y*, *behav*, *indices*, *predictor='Fisherkernel'*, *estimator='SVM'*, *options=None*)

Train classification model from HMM This uses either the Fisher kernel (default) or a set of HMM summary metrics to make a classification, in a nested cross-validated way. By default, X is standardised/centered. Note that all outputs need to be passed on to test_classif to ensure that training and test variables are preprocessed in the same way, while avoiding leakage between training and test set. Estimators so far include: SVM and Logistic Regression Cross-validation strategies so far include: KFold and GroupKFold Hyperparameter optimization strategies so far include: grid search, no hyperparameter optimisation

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data of training set

behav
[array-like of shape (n_train_sessions,)] phenotype, behaviour, or other external labels of training set to be predicted

indices
[array-like of shape (n_train_sessions, 2)] The start and end indices of each trial/session in the training set. Note that this function does not work if indices=None

predictor
[char (optional, default to 'Fisherkernel')] What to predict from, either 'Fisherkernel' or 'summary_metrics' (default='Fisherkernel')

estimator
[char (optional, default to 'SVM')] Model to be used for classification (default='SVM') This should be the name of a sklearn base estimator (for now either 'SVM' or 'LogisticRegression')

options
[dict (optional, default to None)]

general relevant options are:

'optim_hyperparam'
[char, which hyperparameter optimisation strategy to use (default: 'GridSearchCV').] If you don't want to use hyperparameter optimisation, set this to None and specify the hyperparameter (alpha) as an option When using hyperparameter optimisation, additional relevant options are:

'CVscheme': char, which CVscheme to use (default: 'GroupKFold' if group structure is specified, otherwise: KFold) 'nfolds': int, number of folds k for (outer and inner) k-fold CV loops 'group_structure': ndarray of (n_train_sessions, n_train_sessions), matrix specifying group structure: positive values if samples are related, zeros otherwise

possible hyperparameters for model, e.g. 'C' for SVM 'return_prob': bool, whether to also estimate the probabilities

for Fisher kernel, relevant options are:
'shape': char, either 'linear' or 'Gaussian' (TO DO) 'incl_Pi': bool, whether to include the gradient w.r.t. the initial state probabilities when computing the Fisher kernel 'incl_P': bool, whether to include the gradient w.r.t. the transition probabilities 'incl_Mu': bool, whether to include the gradient w.r.t. the state means (note that this only works if means were not set to 0 when training HMM) 'incl_Sigma': bool, whether to include the gradient w.r.t. the state covariances

for summary metrics, relevant options are:
'metrics': list of char, containing metrics to be included as features

Returns:

model_tuned
[estimator] the trained and (if applicable) hyperparameter-optimised scikit-learn estimator

scaler_x
[estimator] the trained standard scaler/kernel centerer of the features/kernel x

Raises:

Exception
If the hmm has not been trained or if necessary input is missing

Notes:

If behav contains NaNs, these subjects/sessions will be removed

glhmm.prediction.**train_pred**(*hmm*, *Y*, *behav*, *indices*, *predictor='Fisherkernel'*, *estimator='KernelRidge'*, *options=None*)

Train prediction model from HMM This uses either the Fisher kernel (default) or a set of HMM summary metrics to predict a phenotype, in a nested cross-validated way. By default, X and Y are standardised/centered unless deconfounding is used. Note that all outputs except behavD, i.e. model and scalers, need to be passed on to test_pred to ensure that training and test variables are preprocessed in the same way, while avoiding leakage between training and test set. Estimators so far include: Kernel Ridge Regression and Ridge Regression Cross-validation strategies so far include: KFold and GroupKFold Hyperparameter optimization strategies so far include: grid search, no hyperparameter optimisation

Parameters:

hmm
[HMM object] An instance of the HMM class, estimated on the group-level

Y
[array-like of shape (n_samples, n_variables_2)] (group-level) timeseries data of training set

behav
[array-like of shape (n_train_sessions,)] phenotype, behaviour, or other external variable of training set

indices
[array-like of shape (n_train_sessions, 2)] The start and end indices of each trial/session in the training data. Note that this function does not work if indices=None

predictor
[char (optional, default to 'Fisherkernel')] What to predict from, either 'Fisherkernel' or 'summary_metrics' (default='Fisherkernel')

estimator
[char (optional, default to 'KernelRidge')] Model to be used for prediction (default='KernelRidge') This should be the name of a sklearn base estimator (for now either 'KernelRidge' or 'Ridge')

options
[dict (optional, default to None)]

general relevant options are:

'optim_hyperparam'
[char, which hyperparameter optimisation strategy to use (default: 'GridSearchCV').] If you don't want to use hyperparameter optimisation, set this to None and specify the hyperparameter (alpha) as an option When using hyperparameter optimisation, additional relevant options are:

'CVscheme': char, which CVscheme to use (default: 'GroupKFold' if group structure is specified, otherwise: KFold) 'nfolds': int, number of folds k for (outer and inner) k-fold CV loops 'group_structure': ndarray of (n_train_sessions, n_train_sessions), matrix specifying group structure: positive values if samples are related, zeros otherwise

'confounds': array-like of shape (n_train_sessions,) or (n_train_sessions, n_confounds) containing confounding variables possible hyperparameters for model, e.g. 'alpha' for (kernel) ridge regression

for Fisher kernel, relevant options are:
'shape': char, either 'linear' or 'Gaussian' (TO DO) 'incl_Pi': bool, whether to include the gradient w.r.t. the initial state probabilities when computing the Fisher kernel 'incl_P': bool, whether to include the gradient w.r.t. the transition probabilities 'incl_Mu': bool, whether to include the gradient w.r.t. the state means (note that this only works if means were not set to 0 when training HMM) 'incl_Sigma': bool, whether to include the gradient w.r.t. the state covariances

for summary metrics, relevant options are:
'metrics': list of char, containing metrics to be included as features

Returns:

model_tuned
[estimator] the trained and (if applicable) hyperparameter-optimised scikit-learn estimator

scaler_x
[estimator] the trained standard scaler/kernel centerer of the features/kernel x

(if not using deconfounding): scaler_y : estimator

the trained standard scaler of the variable to be predicted y.

(if using deconfounding): CinterceptY : float

the estimated intercept for deconfounding

CbetaY
[array-like of shape (n_confounds)] the estimated beta weights for deconfounding of each confound

behavD
[array-like of shape (n_train_sessions)] the phenotype/behaviour in deconfounded space

Raises:

Exception
If the hmm has not been trained or if necessary input is missing

Notes:

If behav contains NaNs, these subjects/sessions will be removed in Y and confounds

3.2.8 glhmm.statistics

Permutation testing from Gaussian Linear Hidden Markov Model @author: Nick Y. Larsen 2023

glhmm.statistics.**calculate_baseline_difference**(*vpath_array*, *R_data*, *state*, *pairwise_statistic*)

Calculate the difference between the specified statistics of a state and all other states combined.

Parameters:

vpath_data (numpy.ndarray):
The Viterbi path as of integer values that range from 1 to n_states.

R_data (numpy.ndarray):
The dependent-variable associated with each state.

state(numpy.ndarray):
The state for which the difference is calculated from.

pairwise_statistic (str)
The chosen statistic to be calculated. Valid options are "mean" or "median".

Returns:

difference (float)
The calculated difference between the specified state and all other states combined.

glhmm.statistics.**calculate_geometric_pval**(*p_values*, *test_combination*)

Calculate test statistics of z-scores converted from p-values based on the specified combination.

Parameters:

p_values (numpy.ndarray):
Matrix of p-values.

test_combination (str):
Specifies the combination method. Valid options: "True", "across_columns", "across_rows". Default is "True".

Returns:

result (numpy.ndarray):
Test statistics of z-scores converted from p-values.

glhmm.statistics.**calculate_nan_correlation_matrix**(*D_data*, *R_data*, *test_combination=False*, *reduce_pval_dims=False*)

Calculate the correlation matrix between independent variables (D_data) and dependent variables (R_data), while handling NaN values column by column of dimension p without without removing entire rows.

Parameters:

D_data (numpy.ndarray):
Input D-matrix for the independent variables.

R_data (numpy.ndarray):
Input R-matrix for the dependent variables.

Returns:

correlation_matrix (numpy.ndarray): Correlation matrix between columns in D_data and R_data.

glhmm.statistics.**calculate_nan_f_test**(*D_data*, *R_column*, *nan_values=False*)

Calculate F-statistics for each feature of D_data against categories in R_data, while handling NaN values column by column without removing entire rows.

- The function handles NaN values for each feature in D_data without removing entire rows.
- NaN values are omitted on a feature-wise basis, and the F-statistic is calculated for each feature.
- The resulting array contains F-statistics corresponding to each feature in D_data.

Parameters:

D_data (numpy.ndarray):
The input matrix of shape (n_samples, n_features).

R_column (numpy.ndarray):
The categorical labels corresponding to each sample in D_data.

Returns:

f_test (numpy.ndarray):
F-statistics for each feature in D_data against the categories in R_data.

glhmm.statistics.**calculate_nan_regression**(*Din*, *Rin*, *proj*)

Calculate the R-squared values for the regression of each dependent variable in Rin on the independent variables in Din, while handling NaN values column-wise.

Parameters:

Din (numpy.ndarray):
Input D-matrix for the independent variables.

Rin (numpy.ndarray):
Input D-matrix for the dependent variables.

proj (numpy.ndarray):
Projection matrix.

Returns:

R2_test (numpy.ndarray):
Array of R-squared values for each regression.

glhmm.statistics.**calculate_nan_regression_f_test**(*Din*, *Rin*, *proj*, *nan_values=False*)

Calculate the f-test values for the regression of each dependent variable in Rin on the independent variables in Din, while handling NaN values column-wise.

Parameters:

Din (numpy.ndarray):
Input D-matrix for the independent variables.

Rin (numpy.ndarray):
Input D-matrix for the dependent variables.

proj (numpy.ndarray):
Projection matrix.

Returns:

R2_test (numpy.ndarray):
Array of f-test values for each regression.

glhmm.statistics.**calculate_nan_t_test**(*D_data*, *R_column*, *nan_values=False*)

Calculate the t-statistics between paired independent (D_data) and dependent (R_data) variables, while handling NaN values column by column without removing entire rows.

- The function handles NaN values for each feature in D_data without removing entire rows.
- NaN values are omitted on a feature-wise basis, and the t-statistic is calculated for each feature.
- The resulting array contains t-statistics corresponding to each feature in D_data.

Parameters:

D_data (numpy.ndarray):
The input matrix of shape (n_samples, n_features).

R_column (numpy.ndarray):
The binary labels corresponding to each sample in D_data.

Returns:

t_test (numpy.ndarray):
t-statistics for each feature in D_data against the binary categories in R_data.

glhmm.statistics.**calculate_statepair_difference**(*vpath_array*, *R_data*, *state_1*, *state_2*, *stat*)

Calculate the difference between the specified statistics of two states.

Parameters:

vpath_data (numpy.ndarray):
The Viterbi path as of integer values that range from 1 to n_states.

R_data (numpy.ndarray):
The dependent-variable associated with each state.

state_1 (int):
First state for comparison.

state_2 (int):
Second state for comparison.

statistic (str):
The chosen statistic to be calculated. Valid options are "mean" or "median".

Returns:

difference (float):
The calculated difference between the two states.

glhmm.statistics.**deconfound_values**(*D_data*, *R_data*, *confounds=None*)

Deconfound the variables R_data and D_data for permutation testing.

Parameters:

D_data (numpy.ndarray):
The input data array.

R_data (numpy.ndarray or None):
The second input data array (default: None). If None, assumes we are working across visits, and R_data represents the Viterbi path of a sequence.

confounds (numpy.ndarray or None):
The confounds array (default: None).

Returns:

D_data (numpy.ndarray):
Deconfounded D_data array.

R_data (numpy.ndarray):
Deconfounded R_data array (returns None if R_data is None). If R_data is None, assumes we are working across visits

glhmm.statistics.**detect_significant_intervals**(*pval*, *alpha*)

Detect intervals of consecutive True values in a boolean array.

Parameters

- **p_values** (*numpy.ndarray*) – An array of p-values.
- **alpha** (*float, optional*) – Threshold for significance.
- **Returns**

- **----------**
- **tuple** (*list of*) – A list of tuples representing the start and end indices (inclusive) of each interval of consecutive True values.
- **Example** – array = [False, False, False, True, True, True, False, False, True, True, False] detect_intervals(array) output: [(3, 5), (8, 9)]

glhmm.statistics.**generate_vpath_1D**(*vpath*)

Convert a 2D array representing a matrix with one non-zero element in each row into a 1D array where each element is the column index of the non-zero element.

Parameters: vpath(numpy.ndarray):

A 2D array where each row has only one non-zero element. Or a 1D array where each row represents a sate number

Returns: vpath_array(numpy.ndarray):

A 1D array containing the column indices of the non-zero elements. If the input array is already 1D, it returns a copy of the input array.

glhmm.statistics.**get_concatenate_sessions**(*D_sessions*, *R_sessions=None*, *idx_sessions=None*)

Converts a 3D matrix into a 2D matrix by concatenating timepoints of every trial session into a new D-matrix.

Parameters:

D_sessions (numpy.ndarray):
D-matrix for each session.

R_sessions (numpy.ndarray):
R-matrix time for each trial.

idx_sessions (numpy.ndarray):
Indices representing the start and end of trials for each session.

Returns:

D_con (numpy.ndarray):
Concatenated D-matrix.

R_con (numpy.ndarray):
Concatenated R-matrix.

idx_sessions_con (numpy.ndarray):
Updated indices after concatenation.

glhmm.statistics.**get_concatenate_subjects**(*D_sessions*)

Converts a 3D matrix into a 2D matrix by concatenating timepoints of every subject into a new D-matrix.

Parameters:

D_sessions (numpy.ndarray):
D-matrix for each subject.

Returns:

D_con (numpy.ndarray):
Concatenated D-matrix.

glhmm.statistics.**get_indices_array**(*idx_data*)

Generates an indices array based on given data indices.

Parameters:

idx_data (numpy.ndarray):
The data indices array.

Returns:

idx_array (numpy.ndarray):
The generated indices array.

glhmm.statistics.**get_indices_from_list**(*data_list*, *count_timestamps=True*)

Generate indices representing the start and end timestamps for each subject or session from a given data list.

Parameters:

data_list (list):
List containing data for each subject or session.

count_timestamps (bool), default=True:
If True, counts timestamps for each element in data_list, otherwise assumes each element in data_list is already a count of timestamps.

Returns:

indices (ndarray):
Array with start and end indices for each subject's timestamps.

glhmm.statistics.**get_indices_session**(*data_label*)

Generate session indices in the data based on provided labels. This is done by using 'data_label' to define sessions and generates corresponding indices. The resulting 'idx_data_sessions' array represents the intervals for each session in the data.

Parameters:

data_label (ndarray):
Array representing the labels for data to be indexed into sessions.

Returns:

idx_data_sessions (ndarray):
The indices of datapoints within each session. It should be a 2D array where each row represents the start and end index for a trial.

Example: get_indices_session(np.array([1, 1, 2, 2, 2, 3, 3, 3, 3])) array([[0, 2],

[2, 5], [5, 9]])

glhmm.statistics.**get_indices_timestamp**(*n_timestamps*, *n_subjects*)

Generate indices of the timestamps for each subject in the data.

Parameters:

n_timestamps (int):
Number of timestamps.

n_subjects (int):
Number of subjects.

Returns:

indices (ndarray):
Array representing the indices of the timestamps for each subject.

Example: get_indices_timestamp(5, 3) array([[0, 5],

[5, 10], [10, 15]])

glhmm.statistics.**get_indices_update_nan**(*idx_data*, *nan_mask*)

Update interval indices based on missing values in the data.

Parameters:

idx_data (numpy.ndarray):
Array of shape (n_intervals, 2) representing the start and end indices of each interval.

nan_mask (bool):
Boolean mask indicating the presence of missing values in the data.

Returns:

idx_data_update (numpy.ndarray):
Updated interval indices after accounting for missing values.

glhmm.statistics.**get_input_shape**(*D_data*, *R_data*, *verbose*)

Computes the input shape parameters for permutation testing.

Parameters:

D_data (numpy.ndarray):
The input data array.

R_data (numpy.ndarray):
The dependent variable.

verbose (bool):
If True, display progress messages. If False, suppress progress messages.

Returns:

n_T (int):
The number of timepoints.

n_ST (int):
The number of subjects or trials.

n_p (int):
The number of features.

D_data (numpy.ndarray):
The updated input data array.

R_data (numpy.ndarray):
The updated dependent variable.

glhmm.statistics.**get_pval**(*test_statistics*, *Nperm*, *method*, *t*, *pval*, *FWER_correction=False*, *test_combination=False*)

Computes p-values and correlation matrix for permutation testing. # Ref: https://github.com/OHBA-analysis/HMM-MAR/blob/master/utils/testing/permtest_aux.m

Parameters:

test_statistics (numpy.ndarray):
The permutation array.

pval_perms (numpy.ndarray):
The p-value permutation array.

Nperm (int):
The number of permutations.

method (str):
The method used for permutation testing.

t (int):
The timepoint index.

pval (numpy.ndarray):
The p-value array.

Returns:

pval (numpy.ndarray):
Updated updated p-value .

glhmm.statistics.**identify_coloumns_for_t_and_f_tests**(*R_data*, *method*, *identify_categories=True*, *category_lim=None*)

Detect columns in R_data that are categorical. Used to detect which columns to perm t-statistics and F-statistics for later analysis.

Parameters:

R_data
[numpy.ndarray] The 3D array containing categorical values.

identify_categories
[bool or list or numpy.ndarray, optional, default=True] If True, automatically identify categorical columns. If list or ndarray, use the provided list of column indices.

method
[str, optional, default="univariate"] The method to perform the tests. Only "univariate" is currently supported.

category_lim
[int or None, optional, default=None] Maximum allowed number of categories for F-test. Acts as a safety measure for columns with integer values, like age, which may be mistakenly identified as multiple categories.

Returns:

dict
A dictionary containing the columns for t-test ("t_test_cols") and F-test ("f_test_cols").

glhmm.statistics.**initialize_arrays**(*R_data*, *n_p*, *n_q*, *n_T*, *method*, *Nperm*, *test_statistics_option*, *test_combination=False*)

glhmm.statistics.**initialize_permutation_matrices**(*method*, *Nperm*, *n_p*, *n_q*, *D_data*, *test_combination=False*)

Initializes the permutation matrices and projection matrix for permutation testing.

Parameters:

method (str):
The method to use for permutation testing.

Nperm (int):
The number of permutations.

n_p (int):
The number of features.

n_q (int):
The number of predictions.

D_data (numpy.ndarray):
The independent variable.

Returns:

test_statistics (numpy.ndarray):
The permutation array.

pval_perms (numpy.ndarray):
The p-value permutation array.

proj (numpy.ndarray or None):
The projection matrix (None for correlation methods).

glhmm.statistics.**permutation_matrix_across_subjects**(*Nperm*, *D_t*)

Generates a normal permutation matrix with the assumption that each index is independent across subjects.

Parameters:

Nperm (int):
The number of permutations.

D_t (numpy.ndarray):
D-matrix at timepoint 't'

Returns:

permutation_matrix (numpy.ndarray):
Permutation matrix of subjects it got a shape (n_ST, Nperm)

glhmm.statistics.**permutation_matrix_across_trials_within_session**(*Nperm*, *R_t*, *idx_array*, *trial_timepoints=None*)

Generates permutation matrix of within-session across-trial data based on given indices.

Parameters:

Nperm (int):
The number of permutations.

R_t (numpy.ndarray):
The preprocessed data array.

idx_array (numpy.ndarray):
The indices array.

trial_timepoints (int):
Number of timepoints for each trial (default: None)

Returns:

permutation_matrix (numpy.ndarray):
Permutation matrix of subjects it got a shape (n_ST, Nperm)

glhmm.statistics.**permutation_matrix_within_subject_across_sessions**(*Nperm*, *D_t*, *idx_array*)

Generates permutation matrix of within-session across-session data based on given indices.

Parameters:

Nperm (int):
The number of permutations.

D_t (numpy.ndarray):
The preprocessed data array.

idx_array (numpy.ndarray):
The indices array.

Returns:

permutation_matrix (numpy.ndarray):
The within-session continuos indices array.

glhmm.statistics.**permute_subject_trial_idx**(*idx_array*)

Permutes an array based on unique values while maintaining the structure.

Parameters:

idx_array (numpy.ndarray):
Input array to be permuted.

Returns:

permuted_array (numpy.ndarray):
Permuted matrix based on unique values.

glhmm.statistics.**process_family_structure**(*dict_family*, *Nperm*)

Process a dictionary containing family structure information.

Parameters:

dict_family (dict): Dictionary containing family structure information.
file_location (str): The file location of the family structure data in CSV format. M (numpy.ndarray, optional): The matrix of attributes, which is not typically required.

Defaults to None.

nP (int): The number of permutations to generate. CMC (bool, optional): A flag indicating whether to use the Conditional Monte Carlo method (CMC).

Defaults to False.

EE (bool, optional): A flag indicating whether to assume exchangeable errors, which allows permutation.
Defaults to True.

Nperm (int): Number of permutations.

Returns:

dict_mfam (dict): Modified dictionary with processed values.

EB (numpy.ndarray):
Block structure representing relationships between subjects.

M (numpy.ndarray, optional), default=None:
The matrix of attributes, which is not typically required.

nP (int):
The number of permutations to generate.

CMC (bool, optional), default=False:
A flag indicating whether to use the Conditional Monte Carlo method (CMC).

EE (bool, optional), default=True:
A flag indicating whether to assume exchangeable errors, which allows permutation.

glhmm.statistics.**pval_cluster_based_correction**(*test_statistics*, *pval*, *alpha=0.05*)

Perform cluster-based correction on test statistics using the output from permutation testing. The function corrects p-values by using the test statistics and p-values obtained from permutation testing. It converts the test statistics into z-based statistics, allowing to threshold and identify cluster sizes. The p-value map from permutation testing results is then thresholded using the cluster size derived from z-based statistics.

Parameters

- **test_statistics** (*(numpy.ndarray)*) – 2D or 3D array of test statistics. 2D if you have applied permutation testing using "regression".

- **pval** (*(numpy.ndarray)*) – 2D or 1D array of p-values obtained from permutation testing. 1D if you have applied permutation testing using "regression".
- **alpha** (*(float, optional), default=0.05*) – Significance level for cluster-based correction.

Returns
p_values – Corrected p-values after cluster-based correction.

Return type
(numpy.ndarray)

glhmm.statistics.**pval_correction**(*pval*, *method='fdr_bh'*, *alpha=0.05*, *include_nan=True*, *nan_diagonal=False*)

Adjusts p-values for multiple testing.

Parameters:

pval (numpy.ndarray):
numpy array of p-values.

method (str, optional): method used for FDR correction, default='fdr_bh.
bonferroni : one-step correction sidak : one-step correction holm-sidak : step down method using Sidak adjustments holm : step-down method using Bonferroni adjustments simes-hochberg : step-up method (independent) hommel : closed method based on Simes tests (non-negative) fdr_bh : Benjamini/Hochberg (non-negative) fdr_by : Benjamini/Yekutieli (negative) fdr_tsbh : two stage fdr correction (non-negative) fdr_tsbky : two stage fdr correction (non-negative)

alpha (float, optional):
Significance level (default: 0.05).

include_nan, default=True:
Include NaN values during the correction of p-values if True. Exclude NaN values if False.

nan_diagonal, default=False:
Add NaN values to the diagonal if True.

Returns:

pval_corrected (numpy.ndarray):
numpy array of corrected p-values.

significant (numpy.ndarray):
numpy array of boolean values indicating significant p-values.

glhmm.statistics.**reconstruct_concatenated_design**(*D_con*, *D_sessions=None*, *n_timepoints=None*, *n_trials=None*, *n_channels=None*)

Reconstructs the concatenated D-matrix to the original session variables.

Parameters:

D_con (numpy.ndarray):
Concatenated D-matrix.

D_sessions (numpy.ndarray, optional):
Original D-matrix for each session.

n_timepoints (int, optional):
Number of timepoints per trial.

n_trials (int, optional):
Number of trials per session.

n_channels (int, optional):
Number of channels.

Returns:

D_reconstruct (numpy.ndarray):
Reconstructed D-matrix for each session.

glhmm.statistics.**remove_nan_values**(*D_data*, *R_data*, *method*)

Remove rows with NaN values from input data arrays.

Parameters

- **D_data** (*numpy.ndarray*) – Input data array containing features.
- **R_data** (*numpy.ndarray*) – Input data array containing response values.

Returns

- **D_data** (*numpy.ndarray*) – Cleaned feature data (D_data) with NaN values removed.
- **R_data** (*numpy.ndarray*) – Cleaned response data (R_data) with NaN values removed.
- *nan_mask(bool)* – Array that mask the position of the NaN values with True and False for non-nan values

glhmm.statistics.**surrogate_state_time**(*perm*, *viterbi_path*, *n_states*)

Generates surrogate state-time matrix based on a given Viterbi path.

Parameters:

perm (int):
The permutation number.

viterbi_path (numpy.ndarray):
1D array or 2D matrix containing the Viterbi path.

n_states (int):
The number of states

Returns:

viterbi_path_surrogate (numpy.ndarray):
A 1D array representing the surrogate Viterbi path

glhmm.statistics.**surrogate_viterbi_path**(*viterbi_path*, *n_states*)

Generate surrogate Viterbi path based on state-time matrix.

Parameters:

viterbi_path (numpy.ndarray):
1D array or 2D matrix containing the Viterbi path.

n_states (int):
Number of states in the hidden Markov model.

Returns:

viterbi_path_surrogate (numpy.ndarray):
Surrogate Viterbi path as a 1D array representing the state indices. The number of states in the array varies from 1 to n_states

glhmm.statistics.**test_across_sessions_within_subject**(*D_data*, *R_data*, *idx_data*, *method='regression'*, *Nperm=0*, *confounds=None*, *verbose=True*, *test_statistics_option=False*, *FWER_correction=False*, *identify_categories=False*, *category_lim=10*, *test_combination=False*)

Perform permutation testing across sessions within the same subject, while keeping the trial order the same. This procedure is particularly valuable for investigating the effects of long-term treatments or monitoring changes in brain responses across sessions over time. Three options are available to customize the statistical analysis to a particular research questions:

- 'regression': Perform permutation testing using regression analysis.
- 'correlation': Conduct permutation testing with correlation analysis.
- 'cca': Apply permutation testing using canonical correlation analysis.

Parameters:

D_data (numpy.ndarray):
Input data array of shape that can be either a 2D array or a 3D array. For 2D array, it got a shape of (n, p), where n_ST represent the number of subjects, and each column represents a feature (e.g., brain region). For a 3D array,it got a shape (T, n, p), where the first dimension represents timepoints, the second dimension represents the number of trials, and the third dimension represents features/predictors.

R_data (numpy.ndarray):
The dependent-variable can be either a 2D array or a 3D array. For 2D array, it got a shape of (n, q), where n represent the number of trials, and q represents the outcome/dependent variable For a 3D array,it got a shape (T, n, q), where the first dimension represents timepoints, the second dimension represents the number of trials, and the third dimension represents a dependent variable

idx_data (numpy.ndarray):
The indices for each trial within the session. It should be a 2D array where each row represents the start and end index for a trial.

method (str, optional), default="regression":
The statistical method to be used for the permutation test. Valid options are "regression", "univariate", or "cca". Note: "cca" stands for Canonical Correlation Analysis

Nperm (int), default=0:
Number of permutations to perform.

confounds (numpy.ndarray or None, optional):
The confounding variables to be regressed out from the input data (D_data). If provided, the regression analysis is performed to remove the confounding effects. (default: None):

verbose (bool, optional), default=False:
If True, display progress messages and prints. If False, suppress messages.

test_statistics_option (bool, optional), default=False:
If True, the function will return the test statistics for each permutation.

FWER_correction (bool, optional), default=False:
Specify whether to perform family-wise error rate (FWER) correction for multiple comparisons using the MaxT method. Note: FWER_correction is not necessary if pval_correction is applied later for multiple comparison p-value correction.

identify_categories
[bool or list or numpy.ndarray, optional, default=True.] If True, automatically identify categorical columns. If list or ndarray, use the provided list of column indices.

category_lim
[int or None, optional, default=None.] Maximum allowed number of categories for F-test. Acts as a safety measure for columns with integer values, like age, which may be mistakenly identified as multiple categories.

test_combination, default=False:
Calculates geometric means of p-values using permutation testing. Valid options are: - True (bool): Return a single geometric mean per time point. - "across_rows" (str): Calculate geometric means for each row. - "across_columns" (str): Calculate geometric means for each column.

Returns:

result (dict):
A dictionary containing the following keys. Depending on the *test_statistics_option* and *method*, it can return the p-values, correlation coefficients, test statisticss. 'pval': P-values for the test with shapes based on the method:

- method=="Regression": (T, p)
- method=="univariate": (T, p, q)
- method=="cca": (T, 1)

'test_statistics': test statistics is the permutation distribution if *test_statistics_option* is True, else None.

- method=="Regression": (T, Nperm, p)
- method=="univariate": (T, Nperm, p, q)
- method=="cca": (T, Nperm, 1)

'base_statistics': Correlation coefficients for the test with shape (T, p, q) if method==”univariate”, else None. 'test_type': the type of test, which is the name of the function 'method': the method used for analysis Valid options are

“regression”, “univariate”, or “cca”, “one_vs_rest” and “state_pairs” (default: “regression”).

'max_correction': Specifies if FWER has been applied using MaxT, can either output True or False. 'Nperm' :The number of permutations that has been performed.

glhmm.statistics.**test_across_subjects**(*D_data*, *R_data*, *method='regression'*, *Nperm=0*, *confounds=None*, *dict_family=None*, *verbose=True*, *test_statistics_option=False*, *FWER_correction=False*, *identify_categories=False*, *category_lim=10*, *test_combination=False*)

Perform permutation testing across subjects. Family structure can be taken into account by inputting “dict_family”. Three options are available to customize the statistical analysis to a particular research questions:

- “regression”: Perform permutation testing using regression analysis.
- “univariate”: Conduct permutation testing with correlation analysis.
- “cca”: Apply permutation testing using canonical correlation analysis.

Parameters:

D_data (numpy.ndarray):
Input data array of shape that can be either a 2D array or a 3D array. For 2D, the data is represented as a (n, p) matrix, where n represents the number of subjects, and p represents the number of predictors. For 3D array, it has a shape (T, n, q), where the first dimension represents timepoints, the second dimension represents the number of subjects, and the third dimension represents features. For 3D, permutation testing is performed per timepoint for each subject.

R_data (numpy.ndarray):
The dependent variable can be either a 2D array or a 3D array. For 2D array, it has a shape of (n, q), where n represents the number of subjects, and q represents the outcome of the dependent variable. For 3D array, it has a shape (T, n, q), where the first dimension represents timepoints, the second dimension represents the number of subjects, and the third dimension represents a dependent variable. For 3D, permutation testing is performed per timepoint for each subject.

method (str, optional), default=”regression”:
The statistical method to be used for the permutation test. Valid options are “regression”, “univariate”, or “cca”. Note: “cca” stands for Canonical Correlation Analysis

Nperm (int), default=0:
Number of permutations to perform.

confounds (numpy.ndarray or None, optional), default=None:
The confounding variables to be regressed out from the input data (D_data). If provided, the regression analysis is performed to remove the confounding effects.

dict_family (dict):
Dictionary containing family structure information. - file_location (str): The file location of the family structure data in CSV format. - M (numpy.ndarray, optional): The matrix of attributes, which is not typically required.

Defaults to None.

- **CMC (bool, optional), default=False:**
  A flag indicating whether to use the Conditional Monte Carlo method (CMC).
- EE (bool, optional), default=True: A flag indicating whether to assume exchangeable errors, which allows permutation.

verbose (bool, optional):
If True, display progress messages. If False, suppress progress messages.

test_statistics_option (bool, optional), default=False:
If True, the function will return the test statistics for each permutation.

FWER_correction (bool, optional), default=False:
Specify whether to perform family-wise error rate (FWER) correction using the MaxT method. Note: FWER_correction is not necessary if pval_correction is applied later for multiple comparison p-value correction.

identify_categories
[bool or list or numpy.ndarray, optional, default=True] If True, automatically identify categorical columns. If list or ndarray, use the provided list of column indices.

category_lim
[int or None, optional, default=10] Maximum allowed number of categories for F-test. Acts as a safety measure for columns with integer values, like age, which may be mistakenly identified as multiple categories.

test_combination, default=False:
Calculates geometric means of p-values using permutation testing. In the context of p-values from permutation testing, calculating geometric means can be useful for summarizing results across multiple tests to get insights into the overall statistical significance across experimental conditions. Valid options are:

- True (bool): Return a single geometric mean value.
- "across_rows" (str): Calculates geometric means aggregated across rows.
- "across_columns" (str): Calculates geometric means aggregated across columns.

Returns:

result (dict):
A dictionary containing the following keys. Depending on the *test_statistics_option* and *method*, it can return the p-values, correlation coefficients, test statisticss. 'pval': P-values for the test with shapes based on the method:

- method=="Regression": (T, p)
- method=="univariate": (T, p, q)
- method=="cca": (T, 1)

'test_statistics': test statistics is the permutation distribution if *test_statistics_option* is True, else None.

- method=="Regression": (T, Nperm, p)
- method=="univariate": (T, Nperm, p, q)
- method=="cca": (T, Nperm, 1)

'base_statistics': Correlation coefficients for the test with shape (T, p, q) if method=="univariate", else None. 'test_type': the type of test, which is the name of the function 'method': the method used for analysis

Valid options are “regression”, “univariate”, or “cca”, “one_vs_rest” and “state_pairs”. ‘max_correction’: Specifies if FWER has been applied using MaxT, can either output True or False. ‘performed_tests’: A dictionary that marks the columns in the test_statistics or p-value matrix corresponding to the (q dimension) where t-tests or F-tests have been performed. ‘Nperm’ :The number of permutations that has been performed.

glhmm.statistics.**test_across_trials_within_session**(*D_data*, *R_data*, *idx_data*, *method='regression'*, *Nperm=0*, *confounds=None*, *trial_timepoints=None*, *verbose=True*, *test_statistics_option=False*, *FWER_correction=False*, *identify_categories=False*, *category_lim=10*, *test_combination=False*)

Perform permutation testing across different trials within a session. An example could be if we want to test if any learning is happening during a session that might speed up times.

Three options are available to customize the statistical analysis to a particular research questions:

- ‘regression’: Perform permutation testing using regression analysis.
- ‘correlation’: Conduct permutation testing with correlation analysis.
- ‘cca’: Apply permutation testing using canonical correlation analysis.

Parameters:

D_data (numpy.ndarray):
Input data array of shape that can be either a 2D array or a 3D array. For 2D array, it got a shape of (n, p), where n represent the number of trials, and p represents the number of predictors (e.g., brain region) For a 3D array,it got a shape (T, n, p), where the first dimension represents timepoints, the second dimension represents the number of trials, and the third dimension represents features/predictors. In the latter case, permutation testing is performed per timepoint for each subject.

R_data (numpy.ndarray):
The dependent-variable can be either a 2D array or a 3D array. For 2D array, it got a shape of (n, q), where n represent the number of trials, and q represents the outcome/dependent variable For a 3D array,it got a shape (T, n, q), where the first dimension represents timepoints, the second dimension represents the number of trials, and the third dimension represents a dependent variable

idx_data (numpy.ndarray):
The indices for each trial within the session. It should be a 2D array where each row represents the start and end index for a trial.

method (str, optional), default=”regression”:
The statistical method to be used for the permutation test. Valid options are “regression”, “univariate”, or “cca”. Note: “cca” stands for Canonical Correlation Analysis

Nperm (int), default=0:
Number of permutations to perform.

confounds (numpy.ndarray or None, optional), default=None:
The confounding variables to be regressed out from the input data (D_data). If provided, the regression analysis is performed to remove the confounding effects.

trial_timepoints (int), default=None:
Number of timepoints for each trial.

verbose (bool, optional), default=True:
If True, display progress messages. If False, suppress progress messages.

test_statistics_option (bool, optional), default=False:
If True, the function will return the test statistics for each permutation.

FWER_correction (bool, optional), default= False:
Specify whether to perform family-wise error rate (FWER) correction for multiple comparisons using the MaxT method. Note: FWER_correction is not necessary if pval_correction is applied later for multiple comparison p-value correction.

identify_categories, default=True:
bool or list or numpy.ndarray, optional. If True, automatically identify categorical columns. If list or ndarray, use the provided list of column indices.

category_lim
[int or None, optional, default=None] Maximum allowed number of categories for F-test. Acts as a safety measure for columns with integer values, like age, which may be mistakenly identified as multiple categories.

test_combination, default=False:
Calculates geometric means of p-values using permutation testing. Valid options are: - True (bool): Return a single geometric mean per time point. - "across_rows" (str): Calculate geometric means for each row. - "across_columns" (str): Calculate geometric means for each column.

Returns:

result (dict): A dictionary containing the following keys. Depending on the *test_statistics_option* and *method*, it can return the p-values,
correlation coefficients, test statisticss. 'pval': P-values for the test with shapes based on the method:

- method=="Regression": (T, p)
- method=="univariate": (T, p, q)
- method=="cca": (T, 1)

'test_statistics': test statistics is the permutation distribution if *test_statistics_option* is True, else None.

- method=="Regression": (T, Nperm, p)
- method=="univariate": (T, Nperm, p, q)
- method=="cca": (T, Nperm, 1)

'base_statistics': Correlation coefficients for the test with shape (T, p, q) if method=="univariate", else None. 'test_type': the type of test, which is the name of the function 'method': the method used for analysis Valid options are:

"regression", "univariate", or "cca", "one_vs_rest" and "state_pairs".

'max_correction': Specifies if FWER has been applied using MaxT, can either output True or False. 'Nperm' :The number of permutations that has been performed.

glhmm.statistics.**test_across_visits**(*input_data*, *vpath_data*, *n_states*, *method='regression'*, *Nperm=0*, *verbose=True*, *confounds=None*, *test_statistics_option=False*, *pairwise_statistic='mean'*, *FWER_correction=False*, *category_lim=None*, *identify_categories=False*)

glhmm.statistics.**test_statistics_calculations**(*Din*, *Rin*, *perm*, *test_statistics*, *proj*, *method*, *category_columns=[]*, *test_combination=False*)

Calculates the test_statistics array and pval_perms array based on the given data and method.

Parameters:

Din (numpy.ndarray):
The data array.

Rin (numpy.ndarray):
The dependent variable.

perm (int):
The permutation index.

pval_perms (numpy.ndarray):
The p-value permutation array.

test_statistics (numpy.ndarray):
The permutation array.

proj (numpy.ndarray or None):
The projection matrix (None for correlation methods).

method (str):
The method used for permutation testing.

Returns:

test_statistics (numpy.ndarray):
Updated test_statistics array.

pval_perms (numpy.ndarray):
Updated pval_perms array.

glhmm.statistics.**validate_condition**(*condition*, *error_message*)

Validates a given condition and raises a ValueError with the specified error message if the condition is not met.

Parameters:

condition (bool):
The condition to check.

error_message (str):
The error message to raise if the condition is not met.

glhmm.statistics.**viterbi_path_to_stc**(*viterbi_path*, *n_states*)

Convert Viterbi path to state-time matrix.

Parameters:

viterbi_path (numpy.ndarray):
1D array or 2D matrix containing the Viterbi path.

n_states (int):
Number of states in the hidden Markov model.

Returns:

stc (numpy.ndarray):
State-time matrix where each row represents a time point and each column represents a state.

3.2.9 glhmm.palm_functions

glhmm.palm_functions.**hcp2block**(*tmp*, *blocksfile=None*, *dz2sib=False*, *ids=None*)

Convert HCP-style twin data into block structure.

Parameters:

file (str):
Path to the input CSV file containing twin data.

blocksfile (str, optional), default=None:
Path to save the resulting blocks as a CSV file.

dz2sib (bool, optional), default=False:
If True, handle non-monozygotic twins as siblings.

ids (list or array-like, optional), default=None:
List of subject IDs to include.

Returns:

tuple

A tuple containing three elements:

tab
[numpy.ndarray] A modified table of twin data.

B
[numpy.ndarray] Block structure representing relationships between subjects.

famtype
[numpy.ndarray] An array indicating the type of each family.

glhmm.palm_functions.**is_single_value**(*variable*)

Check if an array contains a singlevalue.

Parameters:

variable (numpy.ndarray or list):
The array to be checked.

Returns:

bool:
True if the array contains a single value, False otherwise.

glhmm.palm_functions.**lmaxflipnode**(*Ptree*, *ns*)

Calculate the logarithm of the maximum number of sign-flips within a palm tree node.

Parameters:

Ptree (list or numpy.ndarray):
The palm tree structure.

ns (int):
The current logarithm of sign-flips (initialized to 0).

Returns:

ns (int):
The logarithm of the maximum number of sign-flips within the node.

glhmm.palm_functions.**lmaxpermnode**(*Ptree*, *n_p*)

Calculate the logarithm of the maximum number of permutations within a palm tree node.

Parameters:

Ptree (list or numpy.ndarray):
The palm tree structure.

n_p (int):
The current logarithm of permutations (initialized to 0).

Returns:

n_p (int):
The logarithm of the maximum number of permutations within the node.

glhmm.palm_functions.**lseq2np**(*S*)

Calculate the logarithm of the number of permutations for a given sequence.

Parameters:

S (numpy.ndarray):
The input sequence.

Returns:

n_p (int):
The logarithm of the number of permutations for the sequence.

glhmm.palm_functions.**maketree**(*B*, *M*, *O*, *wholeblock*, *nosf*)

Recursively construct a palm tree structure from input matrices that is representing nodes in the palm tree.

Parameters:

B (numpy.ndarray):
The input matrix where each row represents a node in the palm tree (Block definitions).

M (numpy.ndarray):
The corresponding Design-matrix, which associates nodes in B with additional data.

O (numpy.ndarray):
Observation indices.

wholeblock (bool):
A boolean indicating if the entire block is positive based on the first element of B.

nosf (bool):
A boolean indicating if there are no signflips this level.

Returns:

S (numpy.ndarray):
The palm tree structure for this branch.

Ptree (list):
The palm tree structure.

glhmm.palm_functions.**maxflipnode**(*Ptree*, *ns*)

Calculate the maximum number of sign-flips within a palm tree node.

Parameters:

Ptree (list or numpy.ndarray):
The palm tree structure.

ns (int):
The current number of sign-flips (initialized to 1).

Returns:

ns (int):
The maximum number of sign-flips within the node.

glhmm.palm_functions.**maxpermnode**(*Ptree*, *np*)

Calculate the maximum number of permutations within a palm tree node.

Parameters:

Ptree (list or numpy.ndarray):
The palm tree structure.

np (int):
The current number of permutations.

Returns:

n_p (int):
The maximum number of permutations within the node.

glhmm.palm_functions.**palm_factorial**(*N=101*)

Calculate logarithmically scaled factorials up to a given number.

Parameters:

N (int, optional), default=101:
The maximum number for which to precompute factorials.

Returns:

lf (numpy.ndarray):
An array of precomputed logarithmically scaled factorials.

glhmm.palm_functions.**palm_maxshuf**(*Ptree*, *stype='perms'*, *uselog=False*)

Calculate the maximum number of shufflings (permutations or sign-flips) for a given palm tree structure.

Parameters:

Ptree (list or numpy.ndarray):
The palm tree structure.

stype (str, optional), default='perms:
The type of shuffling to calculate ('perms' for permutations by default).

uselog (bool, optional), default=False:
A flag indicating whether to calculate using logarithmic values.

Returns:

maxb (int):
The maximum number of shufflings (permutations or sign-flips) based on the specified criteria.

glhmm.palm_functions.**palm_permtree**(*Ptree*, *nP*, *CMC=False*, *maxP=None*)

Generate permutations of a given palm tree structure represented by Ptree. Permutations are created by shuffling the branches of the palm tree.

Parameters:

Ptree (list or numpy.ndarray):
The palm tree structure to be permuted.

nP (int):
The number of permutations to generate.

CMC (bool, optional), default=False:
Whether to use Conditional Monte Carlo (CMC) method for permutation. Defaults to False.

maxP (int, optional), default=None:
The maximum number of permutations allowed. If not provided, it is calculated automatically.

Returns:

P (numpy.ndarray)
An array representing the permutations. Each row corresponds to a permutation, with the first column always representing the identity permutation.

Notes:

- If 'CMC' is False and 'nP' is greater than 'maxP' / 2, a warning message is displayed, as it may take a

considerable amount of time to find non-repeated permutations. - The function utilizes the 'pickperm' and 'randomperm' helper functions for the permutation process.

glhmm.palm_functions.**palm_quickperms**(*EB*, *M=None*, *nP=1000*, *CMC=False*, *EE=True*)

Generate a set of permutations for a given input matrix using PALM methods.

Parameters:

EB (numpy.ndarray)
Block structure representing relationships between subjects.

M (numpy.ndarray, optional), default=None:
The matrix of attributes, which is not typically required.

nP (int), default=1000:
The number of permutations to generate.

CMC (bool, optional), default=False:
A flag indicating whether to use the Conditional Monte Carlo method (CMC).

EE (bool, optional), default=True:
A flag indicating whether to assume exchangeable errors, which allows permutation.

Returns:

Pset (list):
A list containing the generated permutations.

glhmm.palm_functions.**palm_reindex**(*B*, *meth='fixleaves'*)

Reindex a 2D numpy array preserving block structure based on different reindexing methods.

Parameters:

B (numpy.ndarray):
The 2D input array to be reindexed.

meth (str, optional), default='fixleaves':

- **'fixleaves': Reindexes the input array by preserving the order of unique values in the first column**
  and recursively reindexes the remaining columns. Suitable for hierarchical data.
- **'continuous': Reindexes the input array by assigning new values to elements in a continuous,**
  non-overlapping manner within each column. Useful for continuous data.
- **'restart': Reindexes the input array by restarting the numbering from 1 for each block of**
  unique values in the first column. Suitable for data with distinct segments or blocks.
- **'mixed': Combines both 'fixleaves' and 'continuous' reindexing methods. Reindexes the first columns**
  using 'fixleaves' and the remaining columns using 'continuous', creating a mixed scheme.

Returns:

Br (numpy.ndarray):
The reindexed array, preserving the block structure based on the chosen method.

glhmm.palm_functions.**palm_shuftree**(*Ptree*, *nP*, *CMC=False*, *EE=True*)

Generate a set of shufflings (permutations or sign-flips) for a given palm tree structure.

Parameters:

Ptree (list):
The palm tree structure.

nP (int):
The number of permutations to generate.

CMC (bool, optional), default=False:
A flag indicating whether to use the Conditional Monte Carlo method (CMC).

EE (bool, optional), default=True:
A flag indicating whether to assume exchangeable errors, which allows permutation.

Returns:

Pset (list):
A list containing the generated shufflings (permutations).

glhmm.palm_functions.**palm_tree**(*B*, *M=None*)

Construct a palm tree structure from an input matrix B and an optional design-matrix M.

The palm tree represents a hierarchical structure where each node can have three branches: - The left branch contains data elements. - The middle branch represents special features (if any). - The right branch contains nested structures.

Parameters:

B (numpy.ndarray):
The input matrix where each row represents the Multi-level block definitions of the PALM tree.

M (numpy.ndarray, optional):
An optional Design-matrix that associates each node in B with additional data. Defaults to None.

Returns:

list
A list containing three elements: - Ptree[0] : numpy.ndarray or list

The left branch of the palm tree, containing data elements.

- **Ptree[1]**
  [numpy.ndarray, list, or empty list] The middle branch of the palm tree, representing special features (if any).
- **Ptree[2]**
  [numpy.ndarray or list] The right branch of the palm tree, containing nested structures.

glhmm.palm_functions.**pickperm**(*Ptree*, *P*)

Extract a permutation from a palm tree structure. It does not perform the permutation but returns the indices representing the already permuted tree.

Parameters:

Ptree (list or numpy.ndarray):
The palm tree structure.

P (numpy.ndarray):
The current state of the permutation.

Returns:

P (numpy.ndarray):
An array of indices representing the permutation of the palm tree structure.

glhmm.palm_functions.**randomperm**(*Ptree_perm*)

Create a random permutation of a palm tree structure by shuffling its branches.

Parameters:

Ptree_perm (list or numpy.ndarray):
The palm tree structure to be permuted.

Returns:

Ptree_perm (list):
The randomly permuted palm tree structure.

glhmm.palm_functions.**renumber**(*B*)

Renumber the elements in the input array B based on distinct values in its first column. Each distinct value represents a block, and the elements within each block are renumbered sequentially, while preserving the relative order of elements within each block.

Parameters:

B (numpy.ndarray):
The 2D input array to be renumbered.

Returns:

Br (numpy.ndarray):
The renumbered array, where elements are renumbered within blocks.

addcol (bool):
A boolean indicating whether a column was added during renumbering.

glhmm.palm_functions.**seq2np**(*S*)

Calculate the number of permutations for a given sequence.

Parameters:

S (numpy.ndarray):
The input sequence.

Returns:

n_p (int):
The number of permutations for the sequence.

HMM
Hidden Markov Model.

n_parcels
The number of brain parcels/regions/seeds.

n_samples
The number of total timepoints.

n_sessions
The number of total sessions.

n_states
The number of hidden states in the *HMM*.

state mixing
Refers to whether the model is capable of capturing within-session state modulations, rather than assigning the entire sessions (or the largest part of them) to a single state. For more information, visit Ahrends et al. (2022) article here.

PYTHON MODULE INDEX

g

INDEX

H

I

J

L

M

N

P

R

S

T

V

W