MetaBCI

Release 0.2

TBC-TJU

Jan 24, 2024

CONTENTS:

CHAPTER

ONE

METABCI

1.1 Welcome!

MetaBCI is an open-source platform for non-invasive brain computer interface. The project of MetaBCI is led by Prof. Minpeng Xu from Tianjin University, China. MetaBCI has 3 main parts:

- brainda: for importing dataset, pre-processing EEG data and implementing EEG decoding algorithms.
- brainflow: a high speed EEG online data processing framework.
- brainstim: a simple and efficient BCI experiment paradigms design module.

This is the first release of MetaBCI, our team will continue to maintain the repository. If you need the handbook of this repository, please contact us by sending email to TBC_TJU_2022@163.com with the following information:

- Name of your teamleader
- Name of your university(or organization)

We will send you a copy of the handbook as soon as we receive your information.

1.2 What are we doing?

1.2.1 The problem

- BCI datasets come in different formats and standards
- It's tedious to figure out the details of the data
- Lack of python implementations of modern decoding algorithms
- It's not an easy thing to perform BCI experiments especially for the online ones.

If someone new to the BCI wants to do some interesting research, most of their time would be spent on preprocessing the data, reproducing the algorithm in the paper, and also find it difficult to bring the algorithms into BCI experiments.

1.2.2 The solution

The Meta-BCI will:

- Allow users to load the data easily without knowing the details
- Provide flexible hook functions to control the preprocessing flow
- Provide the latest decoding algorithms
- Provide the experiment UI for different paradigms (e.g. MI, P300 and SSVEP)
- Provide the online data acquiring pipeline.
- Allow users to bring their pre-trained models to the online decoding pipeline.

The goal of the Meta-BCI is to make researchers focus on improving their own BCI algorithms and performing their experiments without wasting too much time on preliminary preparations.

1.3 Features

- Improvements to MOABB APIs
  - add hook functions to control the preprocessing flow more easily
  - use joblib to accelerate the data loading
  - add proxy options for network connection issues
  - add more information in the meta of data
  - other small changes
- Supported Datasets
  - MI Datasets
    - AlexMI
    - BNCI2014001, BNCI2014004
    - PhysionetMI, PhysionetME
    - Cho2017
    - MunichMI

        * Schirrmeister2017
        * Weibo2014
        * Zhou2016
    - SSVEP Datasets
        * Nakanishi2015
        * Wang2016
        * BETA
- Implemented BCI algorithms
    - Decomposition Methods
        * SPoC, CSP, MultiCSP and FBCSP
        * CCA, itCCA, MsCCA, ExtendCCA, ttCCA, MsetCCA, MsetCCA-R, TRCA, TRCA-R, SSCOR and TDCA
        * DSP
    - Manifold Learning
        * Basic Riemannian Geometry operations
        * Alignment methods
        * Riemann Procustes Analysis
    - Deep Learning
        * ShallowConvNet
        * EEGNet
        * ConvCA
        * GuneyNet
    - Transfer Learning
        * MEKT
        * LST

1.4 Installation

1. Clone the repo .. code-block:: sh

       git clone https://github.com/TBC-TJU/MetaBCI.git

2. Change to the project directory .. code-block:: sh

       cd MetaBCI

3. Install all requirements .. code-block:: sh

       pip install -r requirements.txt

4. Install brainda package with the editable mode .. code-block:: sh

       pip install -e .

   ## Who are we?

The MetaBCI project is carried out by researchers from

- Academy of Medical Engineering and Translational Medicine, Tianjin University, China
- Tianjin Brain Center, China

1.5 What do we need?

You! In whatever way you can help.

We need expertise in programming, user experience, software sustainability, documentation and technical writing and project management.

We'd love your feedback along the way.

1.6 Contributing

Contributions are what make the open source community such an amazing place to be learn, inspire, and create. **Any contributions you make are greatly appreciated**. Especially welcome to submit BCI algorithms.

1. Fork the Project
2. Create your Feature Branch (`git checkout -b feature/AmazingFeature`)
3. Commit your Changes (`git commit -m 'Add some AmazingFeature'`)
4. Push to the Branch (`git push origin feature/AmazingFeature`)
5. Open a Pull Request

1.7 License

Distributed under the GNU General Public License v2.0 License. See `LICENSE` for more information.

1.8 Contact

Email: TBC_TJU_2022@163.com

1.9 Acknowledgements

- MNE
- MOABB
- pyRiemann
- TRCA/eTRCA
- EEGNet
- RPA
- MEKT

CHAPTER

TWO

METABCI

2.1 metabci package

2.1.1 Subpackages

metabci.brainda package

Subpackages

metabci.brainda.algorithms package

Subpackages

metabci.brainda.algorithms.decomposition package

Submodules

metabci.brainda.algorithms.decomposition.SKLDA module

Shrinkage Linear Discriminant Analysis (SKLDA) algorithm, through the optimization of local features to achieve the purpose ofreducing the dimensionality of the data, can improve the small sample problem of the LDA algorithm to some extent.

author: OrionHan

email: jinhan9165@gmail.com

Created on: date (e.g.2022-02-15)

update log:
2023/12/08 by Yin ZiFan, promise010818@gmail.com, update code annotation

Refer: [1] Blankertz, et al. "Single-trial analysis and classification of ERP components—a tutorial."
NeuroImage 56.2 (2011): 814-825.

Application:

class metabci.brainda.algorithms.decomposition.SKLDA.**SKLDA**

Bases: BaseEstimator, TransformerMixin, ClassifierMixin

Shrinkage Linear discriminant analysis (SKLDA) for BCI.

`avg_feats1`

mean feature vector of class 1.

Type
ndarray of shape (n_features,)

`avg_feats2`

mean feature vector of class 2.

Type
ndarray of shape (n_features,)

`sigma_c1`

empirical covariance matrix of class 1.

Type
ndarray of shape (n_features, n_features)

`sigma_c2`

empirical covariance matrix of class 2.

Type
ndarray of shape (n_features, n_features)

`D`

the dimensionality of the feature space.

Type
int, (=n_features)

`nu_c1`

for sigma penalty calculation in class 1.

Type
float

`nu_c2`

for sigma penalty calculation in class 2.

Type
float

`classes_`

Class labels.

Type
ndarray

`n_features`

Number of features of the training data.

Type
int

`n_samples_c2`

Number of samples in class 2.

Type
int

n_samples_c1

Number of samples in class 1.

Type
int

Tip:

Listing 1: A example using SKLDA

```
import numpy as np
from metabci.brainda.algorithms.decomposition import SKLDA
Xtrain = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
y = np.array([1, 1, 1, 2, 2, 2])
Xtest = np.array([[-0.8, -1], [-1.2, -1], [1.2, 1], [0.5, 2]])
clf2 = SKLDA()
clf2.fit(Xtrain, y)
print(clf2.transform(Xtest))
```

---

fit(*X: ndarray*, *y: ndarray*)

Train the model, Fit SKLDA.

Parameters

- **X1** (*ndarray of shape (n_samples, n_features)*) – samples for class 1 (i.e. positive samples)
- **X2** (*ndarray of shape (n_samples, n_features)*) – samples for class 2 (i.e. negative samples)
- **X** (*array-like of shape (n_samples, n_features)*) – Training data.
- **y** (*array-like of shape (n_samples,)*) – Target values, {-1, 1} or {0, 1}.

Returns
self – Some parameters (sigma_c1, sigma_c2, D) of SKLDA.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *SKLDA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

set_transform_request(*, *Xtest: bool | None | str = '$UNCHANGED$'*) → *SKLDA*

Request metadata passed to the `transform` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `transform` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `transform`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Xtest (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Xtest` parameter in `transform`.

Returns
self – The updated object.

Return type
object

transform(*Xtest: ndarray*)

Project data and Get the decision values.

Parameters
Xtest (*ndarray of shape (n_samples, n_features).*) – Input test data.

Returns
proba – decision values of all test samples.

Return type
ndarray of shape (n_samples,)

metabci.brainda.algorithms.decomposition.STDA module

The Spatial-Temporal Discriminant Analysis (STDA) algorithm maximizes the discriminability of the projected features between target and non-target classes by alternately and synergistically optimizing the spatial and temporal dimensions of the EEG in order to learn two projection matrices. Using the learned two projection matrices to transform each of the constructed spatial-temporal two-dimensional samples into new one-dimensional samples with significantly lower dimensions effectively improves the covariance matrix parameter estimation and enhances the generalization ability of the learned classifiers under small training sample sets.

author: Jin Han

email: jinhan9165@gmail.com

Created on: 2022-05

update log:
2023/12/08 by Yin ZiFan, promise010818@gmail.com, update code annotation

Refer: [1] Zhang, Yu, et al. "Spatial-temporal discriminant analysis for ERP-based brain-computer interface."
IEEE Transactions on Neural Systems and Rehabilitation Engineering 21.2 (2013): 233-243.

Application: Spatial-Temporal Discriminant Analysis (STDA)

class metabci.brainda.algorithms.decomposition.STDA.**STDA**(*L: int = 6*, *max_iter: int = 400*, *eps: float = 1e-05*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

Spatial-Temporal Discriminant Analysis (STDA). Note that the parameters naming are exactly the same as in the paper for convenient application.

Parameters

- **L** (*int*) – the number of eigenvectors retained for projection matrices.
- **max_iter** (*int, default=400*) – Max iteration times.
- **eps** (*float, default=1e-5, also can be 1e-10.*) – Error to guarantee convergence. Error = norm2(W(n) - W(n-1)), see more details in paper[1].

W1

Weight vector. Actually, D1=n_chs.

Type
ndarray of shape (D1, self.L)

W2

Weight vector. Actually, D2=n_features.

Type
ndarray of shape (D2, self.L)

iter_times

Iteration times of STDA.

Type
int

wf

Weight vector of LDA after the raw features are projected by STDA.

Type
ndarray of shape (1, L*L)

References

[1] Zhang, Yu, et al. "Spatial-temporal discriminant analysis for ERP-based brain-computer interface." IEEE Transactions on Neural Systems and Rehabilitation Engineering 21.2 (2013): 233-243.

Tip:

Listing 2: A example using STDA

```
import numpy as np
from metabci.brainda.algorithms.decomposition import STDA
Xtrain2 = np.random.randint(-10, 10, (100*2, 16, 19))
y2 = np.hstack((np.ones(100, dtype=int), np.ones(100, dtype=int) * -1))
Xtest2 = np.random.randint(-10, 10, (4, 16, 19))
clf3 = STDA()
clf3.fit(Xtrain2, y2)
z=clf3.transform(Xtest2)
print(clf3.transform(Xtest2))
```

---

fit(*X*, *y*)

Fit Spatial-Temporal Discriminant Analysis (STDA) model.

Parameters

- **X** (*array-like of shape (n_samples, n_chs, n_features)*) – Training data.
- **y** (*array-like of shape (n_samples,)*) – Target values. {-1, 1} or {0, 1}

Returns
self – Fitted estimator (i.e. self.W1, self.W2).

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *STDA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in score.

Returns

self – The updated object.

Return type

object

set_transform_request(*, *Xtest: bool | None | str = '$UNCHANGED$'*) → *STDA*

Request metadata passed to the transform method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to transform if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to transform.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters

Xtest (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for Xtest parameter in transform.

Returns

self – The updated object.

Return type

object

transform(*Xtest*)

Project data and Get the decision values.

Parameters
Xtest (*ndarray of shape (n_samples, n_features).*) – Input test data.

Returns
H_dv – decision values.

Return type
ndarray of shape (n_samples,)

metabci.brainda.algorithms.decomposition.STDA.**lda_kernel**(*X1: ndarray*, *X2: ndarray*)

Linear Discriminant analysis kernel that is appliable to binary problems.

Parameters

- **X1** (*ndarray of shape (n_samples, n_features)*) – samples for class 1 (i.e. positive samples)
- **X2** (*ndarray of shape (n_samples, n_features)*) – samples for class 2 (i.e. negative samples)

Returns

- **weight_vec** (*ndarray of shape (1, n_features)*) – weight vector.
- **lda_threshold** (*float*)

Note:

The test samples should be formatted as (n_samples, n_features).
test sample is positive, if W @ test_sample.T > lda_thold. test sample is negative, if W @ test_sample.T <= lda_thold.

---

metabci.brainda.algorithms.decomposition.STDA.**lda_proba**(*test_samples: ndarray*, *weight_vec: ndarray*, *lda_threshold: float*)

Calculate decision value.

Parameters

- **test_samples** (*2-D, (n_samples, n_features)*) –
- **weight_vec** (*from LDA_kernel.*) –
- **lda_threshold** (*from LDA_kernel.*) –

Returns
proba

Return type
ndarray of shape (n_samples,)

metabci.brainda.algorithms.decomposition.base module

class metabci.brainda.algorithms.decomposition.base.**FilterBank**(*base_estimator: BaseEstimator*, *filterbank: List[ndarray]*, *n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin

Filter bank decomposition is a bandpass filter array that divides the input signal into multiple subband components and obtains the eigenvalues of each subband component.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **base_estimator** (*class*) – Estimator for model training and feature extraction.
- **filterbank** (*list[ndarray]*) – A bandpass filter bank used to divide the input signal into multiple subband components.
- **n_jobs** (*int*) – Sets the number of CPU working cores. The default is None.

References

Proceedings of the national academy of sciences, 2015, 112(44): E6058-E6067.

fit(*X: ndarray*, *y: ndarray | None = None*, ***kwargs*)

Training model

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **X** (*None*) – Training signal (parameters can be ignored, only used to maintain code structure).
- **y** (*None*) – Label data (ibid., ignorable).
- **Yf** (*None*) – Reference signal (ibid., ignorable).

transform(*X: ndarray*, ***kwargs*)

The parameters stored in self are used to convert X into features, and X is filtered through the filter bank to obtain the eigenvalues of each subband component.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – Test the signal.

Returns
feat – Feature array.

Return type
ndarray, shape(n_trials, n_fre)

transform_filterbank(*X: ndarray*)

The input signal is filtered through a filter bank.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – Input signal.

Returns
Xs – Individual subband components of the input signal.

Return type
ndarray, shape(Nfb, n_trials, n_channels, n_samples)

class metabci.brainda.algorithms.decomposition.base.**FilterBankSSVEP**(*filterbank: List[ndarray]*, *base_estimator: BaseEstimator*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBank*

Filter bank analysis for SSVEP. The SSVEP is analyzed using filter banks, that is, multiple filters are combined to decompose the SSVEP signal into specific segments (subbands containing the original data) and obtain its characteristic data.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **filterbank** (*list[ndarray]*) – The filter bank.
- **base_estimator** (*class*) – Estimator for model training and feature extraction.
- **filterweights** (*ndarray*) – Filter weight, default is None.
- **n_jobs** (*int*) – Sets the number of CPU working cores. The default is None.

transform(*X: ndarray*)

X is converted into features by using the parameters stored in self, and the eigenvalues of each subband component are obtained after the input signal is filtered by the filter bank.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – Test the signal.

Returns
features – Feature array.

Return type
ndarray, shape(n_trials, n_fre)

class metabci.brainda.algorithms.decomposition.base.**TimeDecodeTool**(*dataset: BaseTimeEncodingDataset*, *feature_operation: str = 'sum'*)

Bases: object

Decoding tool set for TDMA coding paradigm. Applicable data sets include P300 speller data set and aVEP speller data.The main functions include: dividing the trial according to the minor event, downsampling the data, and determining the target character (or instruction) according to the judgment result of the trial.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **dataset** (BaseTimeEncodingDataset) – The data set to be decoded.
- **feature_operation** (*str*) – An operation performed after feature extraction for each attempt of the same class.

decode(*key: str*, *feature: ndarray*, *fold_num=6*, *paradigm='avep'*)

The data is decoded according to character large label (used to determine the encoding sequence length, which can be any large label) characteristics, stimulus repetition cycles (fold_num), and normal form types.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **key** (*str*) – Character large label.
- **feature** (*ndarray, shape(n_trials, n_class)*) – A multidimensional array of the features of multiple attempts. The size of the array is the number of attempts x the number of template categories. Where the number of attempts is equal to the number of stimulus repeats * the length of the encoding sequence (key_encode_len).
- **fold_num** (*int*) – The stimulation was repeated.
- **paradigm** (*str*) – Type of paradigm.

Returns
command – The character to be tested is predicted according to the class sequence of the test.

Return type
str

epoch_sort(*X*, *y*)

A trial-ordering method designed specifically for the classic column P300 speller. The trials are sorted in ascending order according to the trial label of a single round of characters.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **X** (*list*) – Pre-sort data for multiple characters, where each element represents the data for all attempts of a character.
- **y** (*list*) – A multi-character trial tag, where each element represents the label value of all the tries of a character, and the label value represents the currently blinking row or column.

Returns

- **X_sort** (*list*) – The sorted data of multiple characters is arranged in ascending order of the label value, where each element represents the data of all attempts of a character.
- **Y_sort** (*list*) – After the sorting of multiple characters, each element in the ascending order of the label value represents the label value of all the tries of a character. The label value represents the current blinking row or column.

`resample`(*x*, *fs_old*, *fs_new*, *axis=None*)

Each element is all the small labels that correspond to a character (labeled "target" and "non-target").

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **x** (*ndarray*) – Each element is a character corresponding to all the try labels.
- **fs_old** (*float*) – The original sampling rate of x.
- **fs_new** (*float*) – Sampling rate of resampling.
- **axis** – Dimensions of resampling.

Returns
x_1 – Data after resampling.

Return type
ndarray

`target_calibrate`(*y*, *key*)

A trial identification method specifically designed for the classic column P300 speller. According to the trial label (y) and character label (key) of the labeled column in the P300 data set, the trial label is converted into a small label that can label "target" and "non-target".

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **y** (*list*) – Each element is a character corresponding to all the try labels.
- **key** – A large label, which contains the label value (key.index) and the character corresponding to the label (key.value).

Returns
y_tar – Each element is all the small labels corresponding to a character (labeled "target" and "non-target").

Return type
list

`metabci.brainda.algorithms.decomposition.base.generate_cca_references`(*freqs: ndarray | int | float*, *srate*, *T*, *phases: ndarray | int | float | None = None*, *n_harmonics: int = 1*)

Construct a sine-cosine reference signal for canonical correlation analysis (CCA).

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **freqs** (*int or float*) – Frequency.
- **srate** (*int*) – Sampling rate.
- **T** (*int*) – Sampling time.
- **phases** (*int or float*) – Phase, default is None.
- **n_harmonics** (*int*) – The number of harmonics. The default value is 1.

Returns
Sine and cosine reference signal.

Return type
Yfndarray, shape(srate*T, n_harmonics*2)

metabci.brainda.algorithms.decomposition.base.**generate_filterbank**(*passbands: List[Tuple[float, float]]*, *stopbands: List[Tuple[float, float]]*, *srate: int*, *order: int | None = None*, *rp: float = 0.5*)

Create a filter bank, that is, obtain a bandpass filter coefficient that can divide the input signal into multiple subband components.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **passbands** (*list or tuple(float, float)*) – Passband parameters.
- **stopbands** (*list or tuple(float, float)*) – Stopband parameters.
- **srate** (*float*) – Sampling rate.
- **order** (*int*) – Filter order.
- **rp** (*float*) – The maximum ripple allowed in the passband below the unit gain is 0.5 by default.

Returns
Filter bank coefficient.

Return type
Filterbankndarray, shape(len(passbands), N, 6)

metabci.brainda.algorithms.decomposition.base.**robust_pattern**(*W: ndarray*, *Cx: ndarray*, *Cs: ndarray*) → ndarray

Transform spatial filters to spatial patterns based on paper [1]_.
Referring to the method mentioned in article [1],the constructed spatial filter only shows how to combine information from different channels to extract signals of interest from EEG signals, but if our goal is neurophysiological interpretation or visualization of weights, activation patterns need to be constructed from the obtained spatial filters.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **W** (*ndarray*) – Spatial filters, shape (n_channels, n_filters).
- **Cx** (*ndarray*) – Covariance matrix of eeg data, shape (n_channels, n_channels).
- **Cs** (*ndarray*) – Covariance matrix of source data, shape (n_channels, n_channels).

Returns
A – Spatial patterns, shape (n_channels, n_patterns), each column is a spatial pattern.

Return type
ndarray

References

metabci.brainda.algorithms.decomposition.base.**sign_flip**(*u*, *s*, *vh=None*)

Flip signs of SVD or EIG using the method in paper [1]_.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **u** (*ndarray*) – left singular vectors, shape (M, K).
- **s** (*ndarray*) – singular values, shape (K,).
- **vh** (*ndarray or None*) – transpose of right singular vectors, shape (K, N).

Returns

- **u** (*ndarray*) – corrected left singular vectors.
- **s** (*ndarray*) – singular values.
- **vh** (*ndarray*) – transpose of corrected right singular vectors.

References

metabci.brainda.algorithms.decomposition.cca module

class metabci.brainda.algorithms.decomposition.cca.**ECCA**(*n_components: int = 1*, *n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin, ClassifierMixin

The Extended Canonical Correlation Analysis (eCCA) method combines the advantages of sCCA and itCCA while applying the individual averaging templates and the positive cosine reference signal correlation information, thus obtaining better recognition performance[1]_.

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

Yf_

Reference signals.

Type
ndarray

Us_

Spatial filter.

Type
ndarray

Vs_

Spatial filter.

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,).
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples).

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *ECCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for Yf parameter in fit.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *ECCA*

Request metadata passed to the score method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to score if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to score.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in score.

Returns
self – The updated object.

Return type
object

transform(*X: ndarray*)

Transform X into features and calculate the correlation coefficients of the signals from different trials

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
rhos – The correlation coefficients, shape(n_trials, n_fre)

Return type
ndarray

class metabci.brainda.algorithms.decomposition.cca.**FBECCA**(*filterbank: List[ndarray]*, *n_components: int = 1*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, ClassifierMixin

Filter bank eCCA method, i.e., an eCCA method that combines the application of multiple filters in order to decompose the SSVEP signal into specific subbands [1]_.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list
- **filterweights** (*ndarray*) – Weights of filter bank
- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

References

Tip:

Listing 3: A example using FBECCA

```
import sys
import numpy as np
from brainda.algorithms.decomposition import FBECCA
from brainda.algorithms.decomposition.base import generate_filterbank, generate_cca_
↪references
wp=[(5,90),(14,90),(22,90),(30,90),(38,90)]
ws=[(3,92),(12,92),(20,92),(28,92),(36,92)]
filterbank = generate_filterbank(wp,ws,srate=250,order=15,rp=0.5)
filterweights = [(idx_filter+1) ** (-1.25) + 0.25 for idx_filter in range(5)]
estimator = FBECCA(filterbank=filterbank,n_components=1,filterweights=np.
↪array(filterweights),n_jobs=-1)
accs = []
for k in range(kfold):
```

(continues on next page)

(continued from previous page)

```
    train_ind, validate_ind, test_ind = match_kfold_indices(k, meta, indices)
    train_ind = np.concatenate((train_ind, validate_ind))
    p_labels = estimator.fit(X=X[train_ind],y=y[train_ind], Yf=Yf).predict(X[test_
↪ind])
    accs.append(np.mean(p_labels==y[test_ind]))
print(np.mean(accs))
```

---

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *FBECCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBECCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

class metabci.brainda.algorithms.decomposition.cca.**FBItCCA**(*filterbank: List[ndarray]*, *n_components: int = 1*, *method: str = 'itcca2'*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, `ClassifierMixin`

The filter bank ItCCA method, i.e., the ItCCA method that combines the application of multiple filters in order to decompose the SSVEP signal into specific subbands[1]_.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **filterweights** (*ndarray*) – Filter weights, defaults to None.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.
- **method** (*str*) – Two pattern feature extraction and fitting classifier model methods judgment, defaulting to 'itcca2'.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *FBItCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBItCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

class metabci.brainda.algorithms.decomposition.cca.**FBMsCCA**(*filterbank: List[ndarray]*, *n_components: int = 1*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, `ClassifierMixin`

The filter bank MsetCCA method, i.e., the MsetCCA method that combines the application of multiple filters in order to decompose the SSVEP signal into specific subbands[1]_.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list.
- **filterweights** (*ndarray*) – Weights of filter banks
- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **method** (*str*) – Two pattern feature extraction and fitting classifier model methods judgment, defaulting to 'itcca2'.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *FBMsCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBMsCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

class metabci.brainda.algorithms.decomposition.cca.**FBMsetCCA**(*filterbank: List[ndarray]*, *n_components: int = 1*, *method: str = 'msetcca2'*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, ClassifierMixin

The filter bank MsetCCA method, i.e., the MsetCCA method that combines the application of multiple filters in order to decompose the SSVEP signal into specific subbands[1]_.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list.
- **filterweights** (*ndarray*) – Weights of filter banks.
- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.
- **methods** (*str*) – Two Pattern Feature Extraction and Fitting Classifier Model Methods Judgment, defaulting to 'msetcca2'.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,).
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples).

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(**, Yf: bool | None | str = '$UNCHANGED$'*) → *FBMsetCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBMsetCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

class metabci.brainda.algorithms.decomposition.cca.**FBMsetCCAR**(*filterbank: List[ndarray]*, *n_components: int = 1*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, ClassifierMixin

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(**, Yf: bool | None | str = '$UNCHANGED$'*) → *FBMsetCCAR*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBMsetCCAR*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

class metabci.brainda.algorithms.decomposition.cca.**FBSCCA**(*filterbank: List[ndarray]*, *n_components: int = 1*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, `ClassifierMixin`

Filter bank SCCA methods, i.e., SCCA methods that combine the application of multiple filters in order to decompose the SSVEP signal into specific subbands[1]_ .This class is a FBSCCA classifier.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **filterweights** (*ndarray*) – Filter weights, defaults to None.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

References

Tip:

Listing 4: A example using FBSCCA

```
import sys
import numpy as np
from brainda.algorithms.decomposition import FBSCCA
from brainda.algorithms.decomposition.base import generate_filterbank, generate_cca_
↪references
wp=[(5,90),(14,90),(22,90),(30,90),(38,90)]
ws=[(3,92),(12,92),(20,92),(28,92),(36,92)]
filterbank = generate_filterbank(wp,ws,srate=250,order=15,rp=0.5)
filterweights = [(idx_filter+1) ** (-1.25) + 0.25 for idx_filter in range(5)]
estimator = FBSCCA(filterbank=filterbank,n_components=1,filterweights=np.
↪array(filterweights),n_jobs=-1)
accs = []
for k in range(kfold):
    train_ind, validate_ind, test_ind = match_kfold_indices(k, meta, indices)
    # merge train and validate set
    train_ind = np.concatenate((train_ind, validate_ind))
    p_labels = estimator.fit(X=X[train_ind],y=y[train_ind], Yf=Yf).predict(X[test_
↪ind])
    accs.append(np.mean(p_labels==y[test_ind]))
    print(np.mean(accs))
```

---

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBSCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.

- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

class metabci.brainda.algorithms.decomposition.cca.**FBTRCA**(*filterbank: List[ndarray]*, *n_components: int = 1*, *ensemble: bool = True*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, `ClassifierMixin`

Filter bank TRCA (filter bank Task-Related Component Analysis, fbTRCA) adds the filter bank analysis method to TRCA by combining the fundamental and harmonic components of the signal. The EEG signal is first filtered using multiple subband filters with different cutoff frequencies to obtain the subband filtered signal. Subsequently, the correlation coefficients of the subband signals are summed according to a weighting function, and finally this weighted correlation coefficient sum is used as the feature discriminant [1]_.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list
- **filterweights** (*ndarray*) – Filter weights, defaults to None.
- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.
- **ensemble** (*bool*) – Whether to perform spatial filter ensemble for each category of signals, the default is True to perform ensemble.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type

ndarray

templates_

Individual average template

Type
ndarray

Us_

Spatial filters obtained for each class of training signals.

Type
ndarray

References

Tip:

Listing 5: A example using FBTRCA

```
import numpy as np
from brainda.algorithms.decomposition import FBTRCA
X = np.zeros((4,22,22))
for i in range(4):
    X[i,...] = np.identity(22)*0.5 + np.random.normal(-1,3,(22,22))*2
y = np.array([1,1,2,2])
filterbank = [np.ones((3,6))]
filterweights = np.array([[0.3, -0.1], [0.5, -0.1]])
estimator = FBTRCA(filterbank=filterbank,n_components=1, ensemble=True,
↪filterweights=np.array(filterweights),n_jobs=-1)
p_labels = estimator.fit(X, y)
print(estimator.predict(np.identity(22)))
```

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal, shape(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *FBTRCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBTRCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

class metabci.brainda.algorithms.decomposition.cca.**FBTRCAR**(*filterbank: List[ndarray]*, *n_components: int = 1*, *ensemble: bool = True*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, `ClassifierMixin`

The filter bank TRCA-R algorithm (filter bank TRCA-R, fbTRCA-R) adds a filter bank analysis method to the TRCA-R algorithm, combining the fundamental and harmonic components of the signal. Multiple subband filters with different cutoff frequencies are utilized to filter the EEG signal to obtain the subband filtered signal. Subsequently, the correlation coefficients of the subband signals are summed according to a weighting function, and finally this weighted correlation coefficient sum is used as the feature discriminant[1]_.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list
- **filterweights** (*ndarray*) – Filter weights, defaults to None.
- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

`classes_`

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type

ndarray

`ensemble`

Whether to perform spatial filter ensemble for each category of signals, the default is True to perform ensemble.

Type

bool

`templates_`

Individual average template

Type

ndarray

Us_

Spatial filters obtained for each class of training signals.

Type
ndarray

Yf

Reference signal(n_classes, 2*n_harmonics, n_samples)

Type
ndarray

References

Tip:

Listing 6: A example using FBTRCAR

```
import numpy as np
from brainda.algorithms.decomposition import FBTRCAR
X = np.zeros((4,22,22))
for i in range(4):
X[i,...] = np.identity(22)*0.3 + np.random.normal(-1,3,(22,22))*5
y = np.array([1,1,2,2])
Yf = X
filterbank = [np.ones((3,6))]
filterweights = np.array([[0.3, -0.1], [0.5, -0.1]])
estimator = FBTRCAR(filterbank=filterbank,n_components=1,ensemble=True,
↪filterweights=np.array(filterweights),n_jobs=-1)
p_labels = estimator.fit(X, y, Yf)
print(estimator.predict(np.identity(22)))
```

---

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

`set_fit_request`(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *FBTRCAR*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns

self – The updated object.

Return type

object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBTRCAR*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

class metabci.brainda.algorithms.decomposition.cca.**FBTtCCA**(*filterbank: List[ndarray]*, *n_components: int = 1*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, `ClassifierMixin`

Filter bank TtCCA method, i.e., a TtCCA method that combines the application of multiple filters in order to decompose the SSVEP signal into specific subbands[1]_.

Parameters

- **filterbank** (*list[ndarray]*) – Filter bank list
- **filterweights** (*ndarray*) – Weights of filter banks
- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type

ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*, *y_sub: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,).
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples).
- **y_sub** (*ndarray*) – Existing source subject data.

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

`set_fit_request`(*, *Yf: bool | None | str = '$UNCHANGED$'*, *y_sub: bool | None | str = '$UNCHANGED$'*) → *FBTtCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

- **Yf** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.
- **y_sub** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `y_sub` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBTtCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.

- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

class metabci.brainda.algorithms.decomposition.cca.**ItCCA**(*n_components: int = 1*, *method: str = 'itcca2'*, *n_jobs: int | None = None*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

The Individual Template-based Canonical Correlation Analysis (It-CCA) method is an extension of the CCA method in which the reference signal is a VEP template obtained by averaging multiple EEG trials from each individual's calibration data, and the individual SSVEP training data is used in the CCA method to improve the frequency detection of SSVEP [1]_.This class is a itCCA classifier

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **method** (*str*) – Two pattern feature extraction and fitting classifier model methods judgment, defaulting to 'itcca2'.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

Yf_

Reference signal.

Type

ndarray

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type

ndarray

templates_

Test data after spatial filtering

Type
ndarray

Us_

Spatial filter

Type
ndarray

Vs_

Spatial filter

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *ItCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for Yf parameter in fit.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *ItCCA*

Request metadata passed to the score method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to score if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to score.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in score.

Returns
self – The updated object.

Return type
object

transform(*X: ndarray*)

Transform the X into features and calculate the correlation coefficients of different trials

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
rhos – Correlation coefficients, shape(n_trials, n_fre).

Return type
ndarray

class metabci.brainda.algorithms.decomposition.cca.**MsCCA**(*n_components: int = 1*, *n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin, ClassifierMixin

Since the sine-cosine signal may not be the ideal reference signal, the Multiset Canonical Correlation Analysis (MsetCCA) method uses joint spatial filtering of multiple sets of data to create an optimized reference signal that extracts common SSVEP features from multiple sets of EEG data recorded at the same stimulus frequency[1]_. Note: MsCCA heavily depends on Yf, thus the phase information should be included when designs Yf.

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **method** (*str*) – Two pattern feature extraction and fitting classifier model methods judgment, defaulting to 'itcca2'.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

Yf_

Reference signals

Type
ndarray

Us_

Spatial filter

Type
ndarray

Ts_

Spatial filter

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *MsCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *MsCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

transform(*X: ndarray*)

Transform X into features and calculate the correlation coefficients of the signals from different trials.

Parameters

X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns

rhos – The correlation coefficients, shape(n_trials, n_fre).

Return type

ndarray

class metabci.brainda.algorithms.decomposition.cca.**MsetCCA**(*n_components: int = 1*, *method: str = 'msetcca2'*, *n_jobs: int | None = None*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

Since the sine-cosine signal may not be the ideal reference signal, the Multiset Canonical Correlation Analysis (MsetCCA) method uses joint spatial filtering of multiple sets of data to create an optimized reference signal that extracts common SSVEP features from multiple sets of EEG data recorded at the same stimulus frequency[1]_.

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.
- **methods** (*str*) – Two Pattern Feature Extraction and Fitting Classifier Model Methods Judgment, defaulting to 'msetcca2'.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

templates_

Template signals

Type
ndarray

Yf_

Reference signals

Type
ndarray

Us_

Spatial filter

Type
ndarray

Ts_

Spatial filter

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *MsetCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *MsetCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

`transform`(*X: ndarray*)

Transform X into features and calculate the correlation coefficients of the signals from different trials.

Parameters

X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns

rhos – The correlation coefficients, shape(n_trials, n_fre)

Return type

ndarray

`class metabci.brainda.algorithms.decomposition.cca.MsetCCAR`(*n_components: int = 1*, *n_jobs: int | None = 1*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

`fit`(*X: ndarray*, *y: ndarray*, *Yf: ndarray*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

`predict`(*X: ndarray*)

Predict the labels

Parameters

X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns

labels – Predicting labels, shape(n_trials,).

Return type

ndarray

`set_fit_request`(**, Yf: bool | None | str = '$UNCHANGED$'*) → *MsetCCAR*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *MsetCCAR*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

`transform`(*X: ndarray*)

Transform X into features and calculate the correlation coefficients of the signals from different trials

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
rhos – The correlation coefficients, shape(n_trials, n_fre)

Return type
ndarray

`class metabci.brainda.algorithms.decomposition.cca.SCCA`(*n_components: int = 1*, *n_jobs: int | None = None*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

Standard CCA (sCCA).The Canonical Correlation Analysis (CCA) method finds the coefficients of the linear combination between the test signal and the Fourier series reference signal for a given frequency-periodic signal to find the maximum correlation between the two sets of signals. To identify the frequency of the SSVEP, CCA calculates the typical correlation between the multichannel SSVEP and the reference signal corresponding to each stimulus frequency, and the frequency of the reference signal with the largest correlation is regarded as the frequency of the SSVEP[1]_[2]_.SCCA is the standard CCA method.

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

`Yf_`

The reference signal provided, defaults to None.

Type
ndarray

Raises
ValueError – None

References

Tip:

Listing 7: A example using SCCA

```
from metabci.brainda.algorithms.decomposition.cca import SCCA
estimator = SCCA()
p_labels = estimator.fit(X=X[train_ind],y=y[train_ind], Yf=Yf).predict(X[test_ind])
```

fit(*X: ndarray | None = None*, *y: ndarray | None = None*, *Yf: ndarray | None = None*)

model training

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Label, shape(n_trials,)
- **Yf** (*ndarray*) – Sine and cosine reference signal, shape(n_classes, 2*n_harmonics, n_samples).

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(**, Yf: bool | None | str = '$UNCHANGED$'*) → *SCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *SCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

transform(*X: ndarray*)

The correlation coefficients of the signals from different trials were obtained by converting X into features.

Parameters

X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns

rhos – he correlation coefficients, shape(n_trials, n_fre)

Return type

ndarray

class metabci.brainda.algorithms.decomposition.cca.**TRCA**(*n_components: int = 1*, *ensemble: bool = True*, *n_jobs: int | None = None*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

The core idea of Task-Related Component Analysis (TRCA) algorithm is to extract task-related components by improving the repeatability between trials, specifically, the algorithm is based on inter-trial covariance matrix maximization to achieve the extraction of task-related components, which belongs to the supervised learning method[1]_.

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **ensemble** (*bool*) – Whether to perform spatial filter ensemble for each category of signals, the default is True to perform ensemble.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

templates_

Individual average template

Type
ndarray

Us_

Spatial filters obtained for each class of training signals.

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *TRCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.

- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TRCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

`transform`(*X: ndarray*)

Transform X into features and calculate the correlation coefficients of the signals from different trials

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
rhos – The correlation coefficients, shape(n_trials, n_fre)

Return type
ndarray

`class metabci.brainda.algorithms.decomposition.cca.TRCAR`(*n_components: int = 1*, *ensemble: bool = True*, *n_jobs: int | None = None*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

The task-related component analysis algorithm with sine-cosine reference signal (TRCA with sine-cosine reference signal, TRCA-R) is based on the TRCA algorithm, and the main improvement point is to add the step of orthogonal projection of the signal to the subspace of sine-cosine template during the training process, which further extracts the components of the EEG signal that are more correlated with the sine-cosine fluctuations of SSVEP[1]_.

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.
- **ensemble** (*bool*) – Whether to perform spatial filter ensemble for each category of signals, the default is True to perform ensemble.

`classes_`

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

`templates_`

Individual average template.

Type
ndarray

`Yf_`

Sine-Cosine reference signal.

Type
ndarray

`Us_`

Spatial filters obtained for each class of training signals.

Type
ndarray

References

Tip:

Listing 8: A example using TRCAR

```
import numpy as np
from brainda.algorithms.decomposition import TRCAR
X = np.array([[[0, -1],[2, -1]], [[2, -1],[0, 1]], [[1, -1],[3, 2]],[[-1, 2],[1,␣
↪0]]])
y = np.array([1, 1, 2, 2])
Yf = np.array([[[0, -0.5],[1, -1]], [[0.2, -1],[0, 1]]])
estimator = TRCAR(n_components=1, ensemble=True, n_jobs=-1)
p_labels = estimator.fit(X, y, Yf)
print(estimator.predict(np.array([[[0, -1.2],[0.5, -1]]])))
```

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *TRCAR*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TRCAR*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

`transform`(*X: ndarray*)

Transform X into features and calculate the correlation coefficients of the signals from different trials

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
rhos – The correlation coefficients, shape(n_trials, n_fre)

Return type
ndarray

class metabci.brainda.algorithms.decomposition.cca.**TtCCA**(*n_components: int = 1*, *n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin, ClassifierMixin

The Transfer Template-based Canonical Correlation Analysis (tt-CCA) method migrates SSVEP templates from existing subjects to new subjects to enhance SSVEP detection. EEG templates were generated for the new subjects using the existing source subject dataset, i.e., migrating EEG templates to capture the frequency and phase information of SSVEP[1]_.

Parameters

- **n_components** (*int*) – The number of feature dimensions after dimensionality reduction, the dimension of the spatial filter, defaults to 1.
- **n_jobs** (*int*) – The number of CPU working cores, default is None.

classes_

Predictive labels, obtained from labeled data by removing duplicate elements from it.

Type
ndarray

templates_

Individual average template signals.

Type
ndarray

Yf_

Reference signals.

Type
ndarray

Us_

Spatial filter.

Type
ndarray

Vs_

Spatial filter.

Type
ndarray

References

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray*, *y_sub: ndarray | None = None*)

model train

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Labels, shape(n_trials,)
- **Yf** (*ndarray*) – Reference signal(n_classes, 2*n_harmonics, n_samples)
- **y_sub** (*ndarray*) – Existing source subject data

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
labels – Predicting labels, shape(n_trials,).

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*, *y_sub: bool | None | str = '$UNCHANGED$'*) → *TtCCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

- **Yf** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.
- **y_sub** (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `y_sub` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TtCCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

`transform`(*X: ndarray*)

Transform X into features and calculate the correlation coefficients of the signals from different trials

Parameters
X (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).

Returns
rhos – The correlation coefficients, shape(n_trials, n_fre)

Return type
ndarray

metabci.brainda.algorithms.decomposition.csp module

Common Spatial Patterns and his happy little buddies!

class metabci.brainda.algorithms.decomposition.csp.**CSP**(*n_components: int | None = None*, *max_components: int | None = None*)

Bases: BaseEstimator, TransformerMixin

common spatial pattern (CSP)

author: Swolf <swolfforever@gmail.com>

Created on:2021-1-07

update log:
2023-11-06 by Yupeng Wang <zy_wyp@tju.edu.cn>

Parameters

- **n_component** (*int*) – Spatial filter dimension
- **max_component** (*int*) – The maximum dimension of the selected spatial filter does not exceed half of the number of leads

n_component

Spatial filter dimension

Type
int

max_component

The maximum dimension of the selected spatial filter does not exceed half of the number of leads

Type
int

classes_

number of classes.

Type
ndarray

W

Spatial filter

Type
ndarray, shape(n_channels, n_filters)

D

Eigenvector of spatial filter

Type
ndarray, shape(n_filters)

A

Spatial pattern

Type
ndarray, shape(n_channels, n_filters)

best_n_components

If the number of spatial filters is not set, the optimal number of choices is calculated automatically.

Type
int

fit(*X: ndarray*, *y: ndarray*)

model training

Parameters

- **X** (*Optional, [ndarray]*) – Test signal, default is None.
- **y** (*Optional, [ndarray]*) – Label, default is None.

transform(*X: ndarray*)

Convert X to a feature using the arguments stored in self.

Parameters
X (*ndarray*) – Test signal, shape(n_trials, n_channels, n_samples).

Returns
features – Find feature model, shape(n_trials, n_components).

Return type
ndarray

class metabci.brainda.algorithms.decomposition.csp.**FBCSP**(*n_components: int | None = None*, *max_components: int | None = None*, *n_mutualinfo_components: int | None = None*, *filterbank: List[ndarray] = []*)

Bases: *FilterBank*

FBCSP.

FilterBank CSP based on paper [1]_.

author: Swolf <swolfforever@gmail.com>

Created on:2021-1-07

update log:
2023-11-06 by Yupeng Wang <zy_wyp@tju.edu.cn>

Parameters

- **n_component** (*int*) – Spatial filter dimension
- **max_component** (*int*) – The maximum dimension of the selected spatial filter does not exceed half of the number of leads
- **n_mutualinfo_components** (*int*) – Multiple classification strategy, one to many.
- **filterbank** (*Optional, [List,[ndarray]]*) – Spatial filter band division range.

n_component

Spatial filter dimension

Type
int

`max_component`

The maximum dimension of the selected spatial filter does not exceed half of the number of leads

Type
int

`n_mutualinfo_components`

Multiple classification strategy, one to many.

Type
int

`filterbank`

Spatial filter band division range.

Type
Optional, [List,[ndarray]]

`ajd_method`

Covariance matrix joint diagonalization method

Type
str,'uwedge'

`classes_`

number of classes.

Type
ndarray

`W`

Spatial filter

Type
ndarray, shape(n_channels, n_filters)

`mutualinfo_values_`

The selected mutual information feature vector has dimension k

Type
ndarray, shape(k,)

`A`

Spatial pattern

Type
ndarray, shape(n_channels, n_filters)

`best_n_components`

If the number of spatial filters is not set, the optimal number of choices is calculated automatically.

Type
int

Raises
`ValueError` – None

References

interface[C]//2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence). IEEE, 2008: 2390-2397.

fit(*X: ndarray*, *y: ndarray*)

model training

Parameters

- **X** (*Optional, [ndarray]*) – Test signal, default is None.
- **y** (*Optional, [ndarray]*) – Label, default is None.

transform(*X: ndarray*)

Convert X to a feature using the arguments stored in self.

Parameters
X (*ndarray*) – Test signal, shape(n_trials, n_channels, n_samples).

Returns
features – Find feature model, shape(n_trials, n_components).

Return type
ndarray

class metabci.brainda.algorithms.decomposition.csp.**FBMultiCSP**(*n_components: int | None = None*, *max_components: int | None = None*, *multiclass: str = 'ovr'*, *ajd_method: str = 'uwedge'*, *n_mutualinfo_components: int | None = None*, *filterbank: List[ndarray] = []*)

Bases: *FilterBank*

FBMultiCSP.

The MultiCSP method based on filter banks is a decoding algorithm formed after adding filter banks and feature selection strategies to the MultiCSP algorithm.

author: Swolf <swolfforever@gmail.com>

Created on:2021-1-07

update log:
2023-11-06 by Yupeng Wang <zy_wyp@tju.edu.cn>

Parameters

- **n_component** (*int*) – Spatial filter dimension
- **max_component** (*int*) – The maximum dimension of the selected spatial filter does not exceed half of the number of leads
- **n_mutualinfo_components** (*int*) – Multiple classification strategy, one to many.
- **filterbank** (*Optional, [List,[ndarray]]*) – Spatial filter band division range.

n_component

Spatial filter dimension

Type
int

`max_component`

The maximum dimension of the selected spatial filter does not exceed half of the number of leads

Type
int

`n_mutualinfo_components`

Multiple classification strategy, one to many.

Type
int

`filterbank`

Spatial filter band division range.

Type
Optional, [List,[ndarray]]

`ajd_method`

Covariance matrix joint diagonalization method

Type
str,'uwedge'

`classes_`

number of classes.

Type
ndarray

`W`

Spatial filter

Type
ndarray, shape(n_channels, n_filters)

`mutualinfo_values_`

The selected mutual information feature vector has dimension k

Type
ndarray, shape(k,)

`A`

Spatial pattern

Type
ndarray, shape(n_channels, n_filters)

`best_n_components`

If the number of spatial filters is not set, the optimal number of choices is calculated automatically.

Type
int

Raises
`ValueError` – None

References

interface[C]//2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence). IEEE, 2008: 2390-2397.

fit(*X: ndarray*, *y: ndarray*)

model training

Parameters

- **X** (*Optional, [ndarray]*) – Test signal, default is None.
- **y** (*Optional, [ndarray]*) – Label, default is None.

transform(*X: ndarray*)

Convert X to a feature using the arguments stored in self.

Parameters
X (*ndarray*) – Test signal, shape(n_trials, n_channels, n_samples).

Returns
features – Find feature model, shape(n_trials, n_components).

Return type
ndarray

class metabci.brainda.algorithms.decomposition.csp.**MultiCSP**(*n_components: int | None = None*, *max_components: int | None = None*, *multiclass: str = 'ovr'*, *ajd_method: str = 'uwedge'*)

Bases: BaseEstimator, TransformerMixin

Multi common spatial pattern (MultiCSP) [1]_.

author: Swolf <swolfforever@gmail.com>

Created on:2021-1-07

update log:
2023-11-06 by Yupeng Wang <zy_wyp@tju.edu.cn>

Parameters

- **n_component** (*int*) – Spatial filter dimension
- **max_component** (*int*) – The maximum dimension of the selected spatial filter does not exceed half of the number of leads
- **multiclass** (*int*) – Multiple classification strategy, one to many.

n_component

Spatial filter dimension

Type
int

max_component

The maximum dimension of the selected spatial filter does not exceed half of the number of leads

Type
int

`multiclass`

Multiple classification strategy, one to many.

Type
int

`ajd_method`

Covariance matrix joint diagonalization method

Type
str,'uwedge'

`classes_`

number of classes.

Type
ndarray

`W`

Spatial filter

Type
ndarray, shape(n_channels, n_filters)

`mutualinfo_values_`

The selected mutual information feature vector has dimension k

Type
ndarray, shape(k,)

`A`

Spatial pattern

Type
ndarray, shape(n_channels, n_filters)

`best_n_components`

If the number of spatial filters is not set, the optimal number of choices is calculated automatically.

Type
int

Raises
`ValueError` – None

References

feature extraction. " Biomedical Engineering, IEEE Transactions on 55, no. 8 (2008): 1991-2000.

`fit`(*X: ndarray*, *y: ndarray*)

model training

Parameters

- `X` (*Optional, [ndarray]*) – Test signal, default is None.
- `y` (*Optional, [ndarray]*) – Label, default is None.

transform(*X: ndarray*)

Convert X to a feature using the arguments stored in self.

Parameters
X (*ndarray*) – Test signal, shape(n_trials, n_channels, n_samples).

Returns
features – Find feature model, shape(n_trials, n_components).

Return type
ndarray

class metabci.brainda.algorithms.decomposition.csp.**SPoC**(*n_components: int | None = None*, *max_components: int | None = None*)

Bases: BaseEstimator, TransformerMixin

Source Power Comodulation (SPoC).

For continuous data, not verified.

fit(*X: ndarray*, *y: ndarray*)

transform(*X: ndarray*)

metabci.brainda.algorithms.decomposition.csp.**ajd**(*X: ndarray*, *method: str = 'uwedge'*) → Tuple[ndarray, ndarray]

Wrapper of AJD methods.

Parameters

- **X** (*ndarray*) – Input covariance matrices, shape (n_trials, n_channels, n_channels)
- **method** (*str, optional*) – AJD method (default uwedge).

Returns

- **V** (*ndarray*) – The diagonalizer, shape (n_channels, n_filters), usually n_filters == n_channels.
- **D** (*ndarray*) – The mean of quasi diagonal matrices, shape (n_channels,).

metabci.brainda.algorithms.decomposition.csp.**csp_feature**(*W: ndarray*, *X: ndarray*, *n_components: int = 2*) → ndarray

Return CSP features in paper [1]_.

Parameters

- **W** (*ndarray*) – spatial filters from csp_kernel, shape (n_channels, n_filters)
- **X** (*ndarray*) – eeg data, shape (n_trials, n_channels, n_samples)
- **n_components** (*int, optional*) – the first k components to use, usually even number, by default 2

Returns
features of shape (n_trials, n_features)

Return type
ndarray

Raises
ValueError – n_components should less than the number of channels

References

metabci.brainda.algorithms.decomposition.csp.**csp_kernel**(*X: ndarray*, *y: ndarray*) → Tuple[ndarray, ndarray, ndarray]

The kernel in CSP algorithm based on paper [1]_.

Parameters

- **X** (*ndarray*) – eeg data, shape (n_trials, n_channels, n_samples).
- **y** (*ndarray*) – labels of X, shape (n_trials,).

Returns

- **W** (*ndarray*) – Spatial filters, shape (n_channels, n_filters).
- **D** (*ndarray*) – Eigenvalues of spatial filters, shape (n_filters,).
- **A** (*ndarray*) – Spatial patterns, shape (n_channels, n_patterns).

References

metabci.brainda.algorithms.decomposition.csp.**gw_csp_kernel**(*X: ndarray*, *y: ndarray*, *ajd_method: str = 'uwedge'*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Grosse-Wentrup AJD method based on paper [1]_.

Parameters

- **X** (*ndarray*) – eeg data, shape (n_trials, n_channels, n_samples).
- **y** (*ndarray*) – labels, shape (n_trials).
- **ajd_method** (*str, optional*) – ajd methods, 'uwedge' 'rjd' and 'ajd_pham', by default 'uwedge'.

Returns

- **W** (*ndarray*) – Spatial filters, shape (n_channels, n_filters).
- **D** (*ndarray*) – Eigenvalues of spatial filters, shape (n_filters,).
- **A** (*ndarray*) – Spatial patterns, shape (n_channels, n_patterns).
- **mutual_info** (*ndarray*) – Mutual informaiton values, shape (n_filters).

References

metabci.brainda.algorithms.decomposition.csp.**spoc_kernel**(*X: ndarray*, *y: ndarray*) → Tuple[ndarray, ndarray, ndarray]

Source Power Comodulation (SPoC) based on paper [1]_.

It is a continous CSP-like method.

Parameters

- **X** (*ndarray*) – eeg data, shape (n_trials, n_channels, n_samples)
- **y** (*ndarray*) – labels, shape (n_trials)

Returns

- **W** (*ndarray*) – Spatial filters, shape (n_channels, n_filters).

- **D** (*ndarray*) – Eigenvalues of spatial filters, shape (n_filters,).
- **A** (*ndarray*) – Spatial patterns, shape (n_channels, n_patterns).

References

metabci.brainda.algorithms.decomposition.dsp module

class metabci.brainda.algorithms.decomposition.dsp.**DCPM**(*n_components: int = 1*, *transform_method: str = 'corr'*)

Bases: *DSP*, ClassifierMixin

DCPM: discriminative canonical pattern matching [1]_.

Author: Junyang Wang <2144755928@qq.com>

Create on: 2022-6-26

Update log:

Parameters

- **n_components** (*int*) – length of the spatial filters, first k components to use, by default 1
- **transform_method** (*str*) – method of template matching, by default 'corr' (pearson correlation coefficient)
- **n_rpts** (*int*) – repetition times in a block

n_components

length of the spatial filters, first k components to use, by default 1

Type
int

transform_method

method of template matching, by default 'corr' (pearson correlation coefficient)

Type
str

n_rpts

repetition times in a block

Type
int

classes_

number of classes

Type
int

combinations_

combinations of two classes in all classes

Type
list, ([int, int], ...)

n_combinations

numbers of combinations

Type
int

Ws

spatial filter, shpe(n_channels, n_components * n_combinations)

Type
ndarray, shape(n_channels, n_components *n_combinations)

templates

templates of train data, shape(n_classes, n_components * n_combinations, n_samples)

Type
ndarray, shape(n_classes, n_components*n_combinations, n_samples)

M

mean value of all classes and trials, i.e. common mode signals, shape(n_channels, n_samples)

Type
ndarray, shape(n_channels, n_samples)

References

Tip:

Listing 9: An example using DCPM

```
from brainda.algorithms.decomposition.dsp import DCPM
X = np.array(data.get('X'))     #data(n_trials, n_channels, n_times)
y = data.get('Y')               #labels(n_trials)
estimator = DCPM(n_components=2,transform_method='corr', n_rpts=1)
accs = []
# use 'fit' to get the model of train data;
# use 'predict' to get the prediction labels of test data;
p_labels=estimator.fit(X[train_ind], y[train_ind]).predict(X[test_ind])
accs.append(np.mean(p_labels==y[test_ind]))
print(np.mean(accs))
```

See also:

pearson_features
calculate pearson correlation coefficients

fit(*X: ndarray*, *y: ndarray*)

Import the train data to get a model: Ws, templates, M.

Parameters

- **X** (*ndarray, shape(n_trials, n_channels, n_samples)*) – train data, shape(n_trials, n_channels, n_samples)
- **y** (*ndarray, shape(n_trials,)*) – labels of train data, shape(n_trials,)

Returns

- **Ws** (*ndarray*) – spatial filters of train data, shape(n_channels, n_components * n_combinations)
- **templates** (*ndarray*) – templates of train data, shape(n_classes, n_components*n_combinations, n_samples)
- **M** (*ndarray*) – mean of train data (common-mode signals), shape(n_channels, n_samples)

`predict`(*X: ndarray*)

Import the templates and the test data to get prediction labels.

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – test data, shape(n_trials, n_channels, n_samples)

Returns
labels – prediction labels of test data, shape(n_trials,)

Return type
ndarray, shape(n_trials,)

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *DCPM*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

`transform`(*X: ndarray*)

Import the test data to get features.

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – test data, shape(n_trials, n_channels, n_samples)

Returns
feature – features of test data, shape(n_trials, n_classes)

Return type
ndarray, shape(n_trials,n_classes)

class metabci.brainda.algorithms.decomposition.dsp.**DSP**(*n_components: int = 1*, *transform_method: str = 'corr'*)

Bases: `BaseEstimator`, `TransformerMixin`, `ClassifierMixin`

DSP: Discriminal Spatial Patterns

Author: Swolf <swolfforever@gmail.com>

Created on: 2021-1-07

Update log:

Parameters

- **n_components** (*int*) – length of the spatial filter, first k components to use, by default 1
- **transform_method** (*str*) – method of template matching, by default 'corr' (pearson correlation coefficient)
- **classes** (*int*) – number of the EEG classes

`n_components`

length of the spatial filter, first k components to use, by default 1

Type
int

`transform_method`

method of template matching, by default 'corr' (pearson correlation coefficient)

Type
str

`classes_`

number of the EEG classes

Type
int

`W_`

Spatial filters, shape(n_channels, n_filters), in which n_channels = n_filters

Type
ndarray, shape(n_channels, n_filters)

`D_`

eigenvalues in descending order, shape(n_filters,)

Type
ndarray, shape(n_filters)

M_

mean value of all classes and trials, i.e. common mode signals, shape(n_channels, n_samples)

Type
ndarray, shape(n_channels, n_samples)

A_

spatial patterns, shape(n_channels, n_filters)

Type
ndarray, shape(n_channels, n_filters)

templates_

templates of train data, shape(n_classes, n_filters, n_samples)

Type
ndarray, shape(n_classes, n_filters, n_samples)

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

Import the train data to get a model.

Parameters

- **X** (*ndarray*) – train data, shape(n_trials, n_channels, n_samples)
- **y** (*ndarray*) – labels of train data, shape (n_trials,)
- **Yf** (*ndarray*) – optional parameter

Returns

- **W_** (*ndarray*) – spatial filters, shape (n_channels, n_filters), in which n_channels = n_filters
- **D_** (*ndarray*) – eigenvalues in descending order, shape (n_filters,)
- **M_** (*ndarray*) – template for all classes, shape (n_channel, n_samples)
- **A_** (*ndarray*) – spatial patterns, shape (n_channels, n_filters)
- **templates_** (*ndarray*) – templates of train data, shape (n_channels, n_filters, n_samples)

predict(*X: ndarray*)

Import the templates and the test data to get prediction labels.

Parameters
X (*ndarray*) – test data, shape(n_trials, n_channels, n_samples)

Returns
labels – prediction labels of test data, shape(n_trials,)

Return type
ndarray

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *DSP*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.

- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for Yf parameter in fit.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *DSP*

Request metadata passed to the score method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to score if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to score.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in score.

Returns
self – The updated object.

Return type
object

`transform`(*X: ndarray*)

Import the test data to get features.

Parameters
X (*ndarray*) – test data, shape(n_trials, n_channels, n_samples)

Returns
feature – correlation coefficients of templates of train data and features of test data, shape(n_trials, n_classes)

Return type
ndarray, shape(n_trials,n_classes)

`class metabci.brainda.algorithms.decomposition.dsp.FBDSP`(*filterbank: List[ndarray]*, *n_components: int = 1*, *transform_method: str = 'corr'*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, `ClassifierMixin`

FBDSP: FilterBank DSP

Author: Swolf <swolfforever@gmail.com>

Created on: 2021-1-07

Update log:

Parameters

- **filterbank** (*list*) – bandpass filterbank, ([float, float],...)
- **n_components** (*int*) – length of the spatial filters, first k components to use, by default 1
- **transform_method** (*str*) – method of template matching, by default 'corr' (pearson correlation coefficient)
- **filterweights** (*ndarray*) – filter weights, optional parameter, by default None
- **n_jobs** (*int*) – optional parameter, by default None

`filterbank`

bandpass filterbank, ([float, float],...)

Type
list[[float, float], ...]

`n_components`

length of the spatial filters, first k components to use, by default 1

Type
int

`transform_method`

method of template matching, by default 'corr' (pearson correlation coefficient)

Type
str

`filterweights`
filter weights, optional parameter, by default None

Type
ndarray

`n_jobs`
optional parameter, by default None

Type
int

`classes_`
number of classes

Type
int

`W_`
spatial filter, shpe(n_channels, n_filters), in which n_channels = n_filters

Type
ndarray, shape(n_channels, n_filters)

`D_`
eigenvalues in descending order, shape(n_filters,)

Type
ndarray, shape(n_filters,)

`M_`
mean value of all classes and trials, i.e. common mode signals, shape(n_channels, n_samples)

Type
ndarray, shape(n_channels, n_samples)

`A_`
spatial patterns, shape(n_channels, n_filters)

Type
ndarray, shape(n_channels, n_filters)

`templates_`
templates of train data, shape(n_classes, n_filters, n_samples)

Type
ndarray, shape(n_classes, n_filters, n_samples)

`fit`(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)
Import the test data to get features.

Parameters

- **X** (*ndarray, shape(n_trials, n_channels, n_samples)*) – train data, shape (n_trials, n_channels, n_samples)
- **y** (*ndarray, shape(n_trials,)*) – labels of train data, shape (n_trials,)
- **Yf** (*ndarray*) – optional parameter,

Returns

- **W_** (*ndarray*) – spatial filters, shape (n_channels, n_filters)

- **D_** (*ndarray*) – eigenvalues in descending order
- **M_** (*ndarray*) – template for all classes, shape (n_channel, n_samples)
- **A_** (*ndarray*) – spatial patterns, shape (n_channels, n_filters)
- **templates_** (*ndarray*) – templates of train data, shape (n_channels, n_filters, n_samples)

`predict`(*X: ndarray*)

Import the templates and the test data to get prediction labels.

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – test data, shape(n_trials, n_channels, n_samples)

Returns
labels – prediction labels of test data, shape(n_trials,)

Return type
ndarray, shape(n_trials,)

See also:

`FilterBankSSVEP()`
filterbank analysis (base)

`set_fit_request`(**, Yf: bool | None | str = '$UNCHANGED$'*) → *FBDSP*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBDSP*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

metabci.brainda.algorithms.decomposition.dsp.**pearson_features**(*X*, *templates*)

Calculate pearson correlation coefficient.

Parameters

- **X** (*ndarray*) – features of test data after spatial filters, shape(n_trials, n_components, n_samples)
- **templates** (*ndarray*) – templates of train data, shape(n_classes, n_components, n_samples)

Returns

corr – pearson correlation coefficient, shape(n_trials, n_classes)

Return type

ndarray

metabci.brainda.algorithms.decomposition.dsp.**xiang_dsp_feature**(*W: ndarray*, *M: ndarray*, *X: ndarray*, *n_components: int = 1*) → ndarray

Return DSP features in paper [1]_.

Author: Swolf <swolfforever@gmail.com>

Created on: 2021-1-07

Update log:

Parameters

- **W** (*ndarray*) – spatial filters from csp_kernel, shape (n_channels, n_filters)
- **M** (*ndarray*) – common template for all classes, shape (n_channel, n_samples)
- **X** (*ndarray*) – eeg test data, shape (n_trials, n_channels, n_samples)
- **n_components** (*int, optional*) – length of the spatial filters, first k components to use, by default 1

Returns

features – features, shape (n_trials, n_components, n_samples)

Return type

ndarray

Raises

ValueError – n_components should less than half of the number of channels

Notes

1. instead of meaning of filtered signals in paper [1]_., we directly return filtered signals.

References

metabci.brainda.algorithms.decomposition.dsp.**xiang_dsp_kernel**(*X: ndarray*, *y: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

DSP: Discriminal Spatial Patterns, only for two classes[1]_. Import train data to solve spatial filters with DSP, finds a projection matrix that maximize the between-class scatter matrix and minimize the within-class scatter matrix. Currently only support for two types of data.

Author: Swolf <swolfforever@gmail.com>

Created on: 2021-1-07

Update log:

Parameters

- **X** (*ndarray*) – EEG train data assuming removing mean, shape (n_trials, n_channels, n_samples)
- **y** (*ndarray*) – labels of EEG data, shape (n_trials,)

Returns

- **W** (*ndarray*) – spatial filters, shape (n_channels, n_filters)
- **D** (*ndarray*) – eigenvalues in descending order
- **M** (*ndarray*) – mean value of all classes and trials, i.e. common mode signals, shape (n_channel, n_samples)
- **A** (*ndarray*) – spatial patterns, shape (n_channels, n_filters)

Notes

the implementation removes regularization on within-class scatter matrix Sw.

References

metabci.brainda.algorithms.decomposition.sceTRCA module

class metabci.brainda.algorithms.decomposition.sceTRCA.**BasicFBTRCA**(*standard: bool | None = True*, *ensemble: bool | None = True*, *n_components: int | None = 1*, *n_bands: int = 1*, *ratio: float = 0.5*)

Bases: object

abstract fit(*X_train: ndarray*, *y_train: ndarray*, *sine_template: ndarray*)

Load in training dataset and train model.

Parameters

- **X_train** (*ndarray*) – (Nb,Ne*Nt,...,Np). Training dataset.
- **y_train** (*ndarray*) – (Ne*Nt,). Labels for X_train.

predict(*X_test: ndarray*)

Calculating the prediction labels based on the decision coefficients.

Parameters
X_test (*ndarray*) – (Nt*Nte,Nc,Np). Test dataset.

Returns
(Nt*Nte,). Predict labels of sc-TRCA. y_ensemble (ndarray): (Nt*Nte,). Predict labels of sc-eTRCA.

Return type
y_standard (ndarray)

transform(*X_test: ndarray*)

Using filter-bank algorithms to calculate decision coefficients.

Parameters
X_test (*ndarray*) – (Nb,Ne*Nte,Nc,Np). Test dataset.

Returns

(Ne*Nte,Ne). Decision coefficients.
Not empty when self.standard is True.

erou (ndarray): (Ne*Nte,Ne). Decision coefficients (ensemble).
Not empty when self.ensemble is True.

Return type
rou (ndarray)

class metabci.brainda.algorithms.decomposition.sceTRCA.**BasicTRCA**(*standard: bool | None = True*, *ensemble: bool | None = True*, *n_components: int | None = 1*, *ratio: float = 0.5*)

Bases: object

abstract fit(*X_train: ndarray*, *y_train: ndarray*, *sine_template: ndarray*)

Load in training dataset and train model.

Parameters

- **X_train** (*ndarray*) – (Ne*Nt,...,Np). Training dataset.
- **y_train** (*ndarray*) – (Ne*Nt,). Labels for X_train.

abstract predict(*X_test: ndarray*)

Predict test data.

Parameters
X_test (*ndarray*) – (Ne*Nte,...,Np). Test dataset.

Returns
(Ne*Nte,). Predict labels. y_ensemble (ndarray): (Ne*Nte,). Predict labels (ensemble).

Return type
y_standard (ndarray)

abstract transform(*X_test: ndarray*)

Calculating decision coefficients.

Parameters
X_test (*ndarray*) – (Ne*Nte,...,Np). Test dataset.

Returns

(Ne*Nte,Ne). Decision coefficients.
Not empty when self.standard is True.

erou (ndarray): (Ne*Nte,Ne). Decision coefficients (ensemble).
Not empty when self.ensemble is True.

Return type
rou (ndarray)

class metabci.brainda.algorithms.decomposition.sceTRCA.**FB_SC_TRCA**(*standard: bool | None = True*, *ensemble: bool | None = True*, *n_components: int | None = 1*, *n_bands: int = 1*, *ratio: float = 0.5*)

Bases: *BasicFBTRCA*

fit(*X_train: ndarray*, *y_train: ndarray*, *sine_template: ndarray*)

Train filter-bank sc-(e)TRCA model.

Parameters

- **X_train** (*ndarray*) – (Nb,Ne*Nt,Nc,Np). Training dataset. Nt>=2.
- **y_train** (*ndarray*) – (Ne*Nt,). Labels for X_train.
- **sine_template** (*ndarray*) – (Ne,2*Nh,Np). Sinusoidal template.

class metabci.brainda.algorithms.decomposition.sceTRCA.**SC_TRCA**(*standard: bool | None = True*, *ensemble: bool | None = True*, *n_components: int | None = 1*, *ratio: float = 0.5*)

Bases: *BasicTRCA*

fit(*X_train: ndarray*, *y_train: ndarray*, *sine_template: ndarray*)

Train sc-(e)TRCA model.

Parameters

- **X_train** (*ndarray*) – (Ne*Nt,Nc,Np). Training dataset. Nt>=2.
- **y_train** (*ndarray*) – (Ne*Nt,). Labels for X_train.
- **sine_template** (*ndarray*) – (Ne,2*Nh,Np). Sinusoidal template.

predict(*X_test: ndarray*)

Calculating the prediction labels based on the decision coefficients.

Parameters
X_test (*ndarray*) – (Nt*Nte,Nc,Np). Test dataset.

Returns
(Nt*Nte,). Predict labels of sc-TRCA. y_ensemble (ndarray): (Nt*Nte,). Predict labels of sc-eTRCA.

Return type
y_standard (ndarray)

transform(*X_test: ndarray*)

Using sc-(e)TRCA algorithm to compute decision coefficients.

Parameters
X_test (*ndarray*) – (Nt*Nte,Nc,Np). Test dataset.

Returns

(Nt*Nte,Ne). Decision coefficients of sc-TRCA.
Not empty when self.standard is True.

erou (ndarray): (Nt*Nte,Ne). Decision coefficients of sc-eTRCA.
Not empty when self.ensemble is True.

Return type
rou (ndarray)

metabci.brainda.algorithms.decomposition.sceTRCA.**combine_fb_feature**(*features: List[Any]*)

Coefficient-level integration specially for filter-bank design.

Parameters
features (*List[Any]*) – Coefficient matrices of different sub-bands.

Returns
Integrated coefficients.

Return type
coef (float)

metabci.brainda.algorithms.decomposition.sceTRCA.**combine_feature**(*features: ~typing.List[~numpy.ndarray], func: ~typing.Any = <function sign_sta>*)

Coefficient-level integration.

Parameters

- **features** (*List[float or int or ndarray]*) – Different features.
- **func** (*function*) – Quantization function.

Returns
Integrated coefficients.

Return type
coef (the same type with elements of features)

metabci.brainda.algorithms.decomposition.sceTRCA.**pearson_corr**(*X: ndarray*, *Y: ndarray*)

Pearson correlation coefficient (1-D or 2-D).

Parameters

- **X** (*ndarray*) – (…, n_points)
- **Y** (*ndarray*) – (…, n_points). The dimension must be same with X.

Returns
corrcoef (float)

metabci.brainda.algorithms.decomposition.sceTRCA.**pick_subspace**(*descend_order: List[Tuple[int, float]]*, *e_val_sum: float*, *ratio: float*)

Config the number of subspaces.

Parameters

- **descend_order** (*List[Tuple[int,float]]*) – See it in solve_gep() or solve_ep().
- **e_val_sum** (*float*) – Trace of covariance matrix.
- **ratio** (*float*) – 0-1. The ratio of the sum of eigenvalues to the total.

Returns
The number of subspaces.

Return type
n_components (int)

metabci.brainda.algorithms.decomposition.sceTRCA.**sctrca_compute**(*X_train: ndarray*, *y_train: ndarray*, *sine_template: ndarray*, *train_info: dict*, *n_components: int | None = 1*, *ratio: float = 0.5*)

(Ensemble) similarity-constrained TRCA (sc-(e)TRCA).

Parameters

- **X_train** (*ndarray*) – (Ne*Nt,Nc,Np). Training dataset. Nt>=2.
- **y_train** (*ndarray*) – (Ne*Nt,). Labels for X_train.
- **sine_template** (*ndarray*) – (Ne,2*Nh,Np). Sinusoidal template.
- **train_info** (*dict*) – {'event_type':ndarray (Ne,), 'n_events':int, 'n_train':ndarray (Ne,), 'n_chans':int, 'n_points':int, 'standard':True, 'ensemble':True}
- **n_components** (*int*) – Number of eigenvectors picked as filters. Set to 'None' if ratio is not 'None'.
- **ratio** (*float*) – 0-1. The ratio of the sum of eigenvalues to the total. Defaults to be 'None'.

Return: sc-(e)TRCA model (dict).
Q (ndarray): (Ne,Nc,Nc). Covariance of original data & average template. S (ndarray): (Ne,Nc,Nc). Covariance of template. u (List[ndarray]): Ne*(Nk,Nc). Spatial filters for EEG signal. v (List[ndarray]): Ne*(Nk,2*Nh). Spatial filters for sinusoidal signal. u_concat (ndarray): (Ne*Nk,Nc). Concatenated filter for EEG signal. v_concat (ndarray): (Ne*Nk,2*Nh). Concatenated filter for sinusoidal signal. uX

(List[ndarray]): Ne*(Nk,Np). sc-TRCA templates for EEG signal. vY (List[ndarray]): Ne*(Nk,Np). sc-TRCA templates for sinusoidal signal. euX (List[ndarray]): (Ne,Ne*Nk,Np). sc-eTRCA templates for EEG signal. evY (List[ndarray]): (Ne,Ne*Nk,Np). sc-eTRCA templates for sinusoidal signal.

metabci.brainda.algorithms.decomposition.sceTRCA.**sign_sta**(*x: float*)

Standardization of decision coefficient based on sign(x).

Parameters
x (*float*) –

Returns
y=sign(x)*x^2

Return type
y (float)

metabci.brainda.algorithms.decomposition.sceTRCA.**solve_gep**(*A: ndarray*, *B: ndarray*, *n_components: int | None = None*, *ratio: float = 0.5*, *mode: str | None = 'Max'*)

Solve generalized problems | generalized Rayleigh quotient:
f(w)=wAw^T/(wBw^T) -> Aw = lambda Bw -> B^{-1}Aw = lambda w

Parameters

- **A** (*ndarray*) – (m,m).
- **B** (*ndarray*) – (m,m).
- **n_components** (*int*) – Number of eigenvectors picked as filters. Eigenvectors are referring to eigenvalues sorted in descend order.
- **ratio** (*float*) – 0-1. The ratio of the sum of eigenvalues to the total.
- **mode** (*str*) – 'Max' or 'Min'. Depends on target function.

Returns
(Nk,m). Picked eigenvectors.

Return type
w (ndarray)

metabci.brainda.algorithms.decomposition.sscor module

SSCOR.

class metabci.brainda.algorithms.decomposition.sscor.**FBSSCOR**(*n_components: int = 1*, *ensemble: bool = False*, *n_jobs: int | None = None*, *filterbank: List[ndarray] = []*, *filterweights: ndarray | None = None*)

Bases: *FilterBank*

Filter Bank SSCOR method in paper [1]_., [2]_.

filterbank and weights suggested in the paper.

wp = [
[6, 90], [14, 90], [22, 90], [30, 90], [38, 90], [46, 90], [54, 90], [62, 90], [70, 90], [78, 90]

] ws = [

[4, 100], [10, 100], [16, 100], [24, 100], [32, 100], [40, 100], [48, 100], [56, 100], [64, 100], [72, 100]

]

filterweights:
np.arange(1, 11)**(-1.25) + 0.25

References

transform(*X: ndarray*)

The parameters stored in self are used to convert X into features, and X is filtered through the filter bank to obtain the eigenvalues of each subband component.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – Test the signal.

Returns
feat – Feature array.

Return type
ndarray, shape(n_trials, n_fre)

class metabci.brainda.algorithms.decomposition.sscor.**SSCOR**(*n_components: int = 1*, *transform_method: str | None = None*, *ensemble: bool = False*, *n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin

fit(*X: ndarray*, *y: ndarray*)

transform(*X: ndarray*)

metabci.brainda.algorithms.decomposition.sscor.**sscor_feature**(*W: ndarray*, *X: ndarray*, *n_components: int = 1*) → ndarray

Return sscor features.

Modified from https://github.com/mnakanishi/TRCA-SSVEP/blob/master/src/test_sscor.m

Parameters

- **W** (*ndarray*) – spatial filters from csp_kernel, shape (n_channels, n_filters)
- **X** (*ndarray*) – eeg data, shape (n_trials, n_channels, n_samples)
- **n_components** (*int, optional*) – the first k components to use, usually even number, by default 1

Returns
features of shape (n_trials, n_components, n_samples)

Return type
ndarray

Raises
ValueError – n_components should less than half of the number of channels

metabci.brainda.algorithms.decomposition.sscor.**sscor_kernel**(*X: ndarray*, *y: ndarray | None = None*, *n_jobs: int | None = None*) → Tuple[ndarray, ndarray, ndarray]

The kernel part in SSCOR algorithm based on paper[1]_., [2]_.

Modified from https://github.com/mnakanishi/TRCA-SSVEP/blob/master/src/train_sscor.m

Parameters

- **X** (*ndarray*) – EEG data assuming removing mean, shape (n_trials, n_channels, n_samples)
- **y** (*ndarray*) – labels, shape (n_trials,), not used here
- **n_jobs** (*int, optional*) – the number of jobs to use, default None

Returns

- **W** (*ndarray*) – filters, shape (n_channels, n_filters)
- **D** (*ndarray*) – eigenvalues in descending order
- **A** (*ndarray*) – spatial patterns, shape (n_channels, n_filters)

References

metabci.brainda.algorithms.decomposition.tdca module

Task Decomposition Component Analysis.

class metabci.brainda.algorithms.decomposition.tdca.**FBTDCA**(*filterbank: List[ndarray]*, *padding_len: int*, *n_components: int = 1*, *filterweights: ndarray | None = None*, *n_jobs: int | None = None*)

Bases: *FilterBankSSVEP*, ClassifierMixin

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray | None = None*)

Training model

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **X** (*None*) – Training signal (parameters can be ignored, only used to maintain code structure).
- **y** (*None*) – Label data (ibid., ignorable).
- **Yf** (*None*) – Reference signal (ibid., ignorable).

predict(*X: ndarray*)

set_fit_request(**, Yf: bool | None | str = '$UNCHANGED$'*) → *FBTDCA*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `Yf` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FBTDCA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

class metabci.brainda.algorithms.decomposition.tdca.**TDCA**(*padding_len: int*, *n_components: int = 1*)

Bases: BaseEstimator, TransformerMixin, ClassifierMixin

fit(*X: ndarray*, *y: ndarray*, *Yf: ndarray*)

predict(*X: ndarray*)

set_fit_request(*, *Yf: bool | None | str = '$UNCHANGED$'*) → *TDCA*

Request metadata passed to the fit method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to fit if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to fit.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters
Yf (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for Yf parameter in fit.

Returns
self – The updated object.

Return type
object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TDCA*

Request metadata passed to the score method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to score if provided. The request is ignored if metadata is not provided.

- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

transform(*X: ndarray*)

metabci.brainda.algorithms.decomposition.tdca.**aug_2**(*X: ndarray*, *n_samples: int*, *padding_len: int*, *P: ndarray*, *training: bool = True*)

metabci.brainda.algorithms.decomposition.tdca.**proj_ref**(*Yf: ndarray*)

metabci.brainda.algorithms.decomposition.tdca.**tdca_feature**(*X: ndarray*, *templates: ndarray*, *W: ndarray*, *M: ndarray*, *Ps: List[ndarray]*, *padding_len: int*, *n_components: int = 1*, *training=False*)

Module contents

metabci.brainda.algorithms.deep_learning package

Submodules

metabci.brainda.algorithms.deep_learning.base module

class metabci.brainda.algorithms.deep_learning.base.**AvgPool2dWithConv**(*kernel_size*, *stride*, *dilation=1*, *padding=0*)

Bases: `Module`

Compute average pooling using a convolution, to have the dilation parameter.

Parameters

- **kernel_size** (*(int,int)*) – Size of the pooling region.
- **stride** (*(int,int)*) – Stride of the pooling operation.
- **dilation** (*int or (int,int)*) – Dilation applied to the pooling filter.
- **padding** (*int or (int,int)*) – Padding applied before the pooling operation.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: **bool**

class metabci.brainda.algorithms.deep_learning.base.**Ensure4d**(**args*, ***kwargs*)

Bases: `Module`

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: **bool**

class metabci.brainda.algorithms.deep_learning.base.**Expression**(*expression_fn*)

Bases: `Module`

Compute given expression on forward pass.

Parameters

expression_fn (*callable*) – Should accept variable number of objects of type *torch.autograd.Variable* to compute its output.

forward(**x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: **bool**

class metabci.brainda.algorithms.deep_learning.base.**MaxNormConstraintConv2d**(**args*, *max_norm_value=1*, *norm_axis=2*, ***kwargs*)

Bases: Conv2d

bias: Tensor | None

dilation: Tuple[int, ...]

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

groups: int

in_channels: int

kernel_size: Tuple[int, ...]

out_channels: int

output_padding: Tuple[int, ...]

padding: str | Tuple[int, ...]

padding_mode: str

stride: Tuple[int, ...]

transposed: bool

weight: Tensor

class metabci.brainda.algorithms.deep_learning.base.**MaxNormConstraintLinear**(**args*, *max_norm_value=1*, *norm_axis=0*, ***kwargs*)

Bases: Linear

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

in_features: int

out_features: int

weight: Tensor

class metabci.brainda.algorithms.deep_learning.base.**NeuralNetClassifierNoLog**(*module*, **args*, *criterion=<class 'torch.nn.modules.loss.NLLLoss'>*, *train_split=<skorch.dataset.ValidSplit object>*, *classes=None*, ***kwargs*)

Bases: NeuralNetClassifier

fit(*X*, *y*, ***fit_params*)

See NeuralNet.fit.

In contrast to NeuralNet.fit, y is non-optional to avoid mistakenly forgetting about y. However, y can be set to None in case it is derived dynamically from X.

get_loss(*y_pred*, *y_true*, **args*, ***kwargs*)

Return the loss for this batch.

Parameters

- **y_pred** (*torch tensor*) – Predicted target values
- **y_true** (*torch tensor*) – True target values.
- **X** (*input data, compatible with skorch.dataset.Dataset*) – By default, you should be able to pass:
  - numpy arrays
  - torch tensors
  - pandas DataFrame or Series
  - scipy sparse CSR matrices
  - a dictionary of the former three
  - a list/tuple of the former three
  - a Dataset

  If this doesn't work with your data, you have to pass a Dataset that can deal with the data.
- **training** (*bool (default=False)*) – Whether train mode should be used or not.

class metabci.brainda.algorithms.deep_learning.base.**SkorchNet**(*module*)

Bases: object

metabci.brainda.algorithms.deep_learning.base.**adaptive_batch_norm**(*model*, *use_global=False*)

metabci.brainda.algorithms.deep_learning.base.**compute_out_size**(*input_size: int*, *kernel_size: int*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*)

metabci.brainda.algorithms.deep_learning.base.**compute_same_pad1d**(*input_size*, *kernel_size*, *stride=1*, *dilation=1*)

metabci.brainda.algorithms.deep_learning.base.**compute_same_pad2d**(*input_size*, *kernel_size*, *stride=(1, 1)*, *dilation=(1, 1)*)

metabci.brainda.algorithms.deep_learning.base.**identity**(*x*)

metabci.brainda.algorithms.deep_learning.base.**np_to_th**(*X*, *requires_grad=False*, *dtype=None*, *pin_memory=False*, ***tensor_kwargs*)

Convenience function to transform numpy array to *torch.Tensor*.

Converts *X* to ndarray using asarray if necessary.

Parameters

- **X** (*ndarray or list or number*) – Input arrays
- **requires_grad** (*bool*) – passed on to Variable constructor
- **dtype** (*numpy dtype, optional*) –
- **var_kwargs** – passed on to Variable constructor

Returns
var

Return type
torch.Tensor

metabci.brainda.algorithms.deep_learning.base.**squeeze_final_output**(*x*)

Removes empty dimension at end and potentially removes empty time
dimension. It does not just use squeeze as we never want to remove first dimension.

Returns
x – squeezed tensor

Return type
torch.Tensor

metabci.brainda.algorithms.deep_learning.base.**transpose_time_to_spat**(*x*)

Swap time and spatial dimensions.

Returns
x – tensor in which last and first dimensions are swapped

Return type
torch.Tensor

metabci.brainda.algorithms.deep_learning.convca module

Conv-CA Modified from https://github.com/yaoli90/Conv-CA

metabci.brainda.algorithms.deep_learning.deepnet module

Deep4Net. Modified from https://github.com/braindecode/braindecode/blob/master/braindecode/models/deep4.py

metabci.brainda.algorithms.deep_learning.eegnet module

EEGNet. Modified from https://github.com/vlawhern/arl-eegmodels/blob/master/EEGModels.py

class metabci.brainda.algorithms.deep_learning.eegnet.**SeparableConv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *bias=True*, *padding_mode='zeros'*, *D=1*)

Bases: Module

An equally SeparableConv2d in Keras. A depthwise conv followed by a pointwise conv.

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

metabci.brainda.algorithms.deep_learning.guney_net module

Guney's network proposed in A Deep Neural Network for SSVEP-based Brain-Computer Interfaces.

Modified from https://github.com/osmanberke/Deep-SSVEP-BCI.git

metabci.brainda.algorithms.deep_learning.shallownet module

ShallowFBCSP. Modified from https://github.com/braindecode/braindecode/blob/master/braindecode/models/shallow_fbcsp.py

class metabci.brainda.algorithms.deep_learning.shallownet.**SafeLog**(*eps=1e-06*)

Bases: Module

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class metabci.brainda.algorithms.deep_learning.shallownet.**Square**

Bases: `Module`

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

Module contents

metabci.brainda.algorithms.feature_analysis package

Submodules

metabci.brainda.algorithms.feature_analysis.freq_analysis module

class metabci.brainda.algorithms.feature_analysis.freq_analysis.**FrequencyAnalysis**(*data*, *meta*, *event*, *srate*, *latency=0*, *channel='all'*)

Bases: `object`

plot_topomap(*data*, *ch_names*, *srate=-1*, *ch_types='eeg'*)

-author: Zhou hongzhan & He Jiatong -Create on:2022-8-9 -update log:

2022-8-31 by Zhou hongzhan

Parameters

- **data** – np.array, 1D array eeg data. The default is [].
- **ch_names** – list interested channels
- **srate** – int sample rate. The default is -1.if set as default ,the initial sample ratio will be applied
- **ch_types** – string Type of channels,default value='eeg'

power_spectrum_periodogram(*x*)

-author: Zhou hongzhan & He Jiatong -Create on:2022-8-9 -update log:

2022-8-31 by Zhou hongzhan

Parameters
x – np.array 1D data.

Returns

np.array
An array of frequencies

Pxx_den
[np.array] The amplitude array respectively correspond to frequency array

Return type
f

signal_noise_ratio(*data=[]*, *srate=-1*, *T=[]*, *channel=[]*)

-author: Zhou hongzhan & He Jiatong -Create on:2022-8-9 -update log:

2022-8-31 by Zhou hongzhan

Parameters

- **data** – np.array, 1D array eeg data. The default is [].
- **srate** – int sample rate. The default is -1.if set as default ,the initial sample ratio will be applied
- **T** – int, ms the during time of data. The default is [].
- **channel** – string interested channels

Returns

np.array
frequency sequece.

snr
[np.array] SNR sequence

Return type
X1

stacking_average(*data=[]*, *_axis=0*)

-author: Zhou hongzhan -Create on:2022-8-9 -update log:

2022-8-11 by Zhou hongzhan

Parameters

- **data** – np.array (nTrials, nChannels, nTimes) EEG origin data. The default is [].
- **_axis** – int The dimension need to be stacked. The default is 0.

Returns

np.array
The data after stacked.

Return type
data_mean

sum_y(*x*, *y*, *x_inf*, *x_sup*)

-author: Zhou hongzhan -Create on:2022-8-9 -update log:

2022-8-11 by Zhou hongzhan

Parameters

- **x** – np.array(1D) An array of frequencies
- **y** – np.array(1D,SAME TYPE WITH X) The amplitude array respectively correspond to frequency array
- **x_inf** – int Infimum of freq.
- **x_sup** – int Supremum of freq.

Returns

int
freq parameter,topomap procedure needed

Return type
np.mean(sum_A)

metabci.brainda.algorithms.feature_analysis.time_analysis module

class metabci.brainda.algorithms.feature_analysis.time_analysis.**TimeAnalysis**(*data*, *meta*, *dataset*, *event*, *latency=0.0*, *channel=[0]*)

Bases: object

Chan_Neuroscan = ['FP1', 'FPZ', 'FP2', 'AF3', 'AF4', 'F7', 'F5', 'F3', 'F1', 'FZ', 'F2', 'F4', 'F6', 'F8', 'FT7', 'FC5', 'FC3', 'FC1', 'FCZ', 'FC2', 'FC4', 'FC6', 'FT8', 'T7', 'C5', 'C3', 'C1', 'CZ', 'C2', 'C4', 'C6', 'T8', 'M1', 'TP7', 'CP5', 'CP3', 'CP1', 'CPZ', 'CP2', 'CP4', 'CP6', 'TP8', 'M2', 'P7', 'P5', 'P3', 'P1', 'PZ', 'P2', 'P4', 'P6', 'P8', 'PO7', 'PO5', 'PO3', 'POZ', 'PO4', 'PO6', 'PO8', 'CB1', 'O1', 'OZ', 'O2', 'CB2']

Chan_Standard1020 = ['Fp1', 'Fpz', 'Fp2', 'AF3', 'AF4', 'F7', 'F5', 'F3', 'F1', 'Fz', 'F2', 'F4', 'F6', 'F8', 'FT7', 'FC5', 'FC3', 'FC1', 'FCz', 'FC2', 'FC4', 'FC6', 'FT8', 'T7', 'C5', 'C3', 'C1', 'Cz', 'C2', 'C4', 'C6', 'T8', 'M1', 'TP7', 'CP5', 'CP3', 'CP1', 'CPz', 'CP2', 'CP4', 'CP6', 'TP8', 'M2', 'P7', 'P5', 'P3', 'P1', 'Pz', 'P2', 'P4', 'P6', 'P8', 'PO7', 'PO5', 'PO3', 'POz', 'PO4', 'PO6', 'PO8', 'I1', 'O1', 'Oz', 'O2', 'I2']

average_amplitude(*data=[]*, *time_start=0*, *time_end=1*)

-author: Jiang Hanzhe -Created on: 2022-8-8 -updata log:

2022-8-15 by Jiang Hanzhe

Parameters

- **data** (*ndarray*) – the EEG data, by default []
- **time_start** (*int*) – beginning of peak seeking, by default 0

- **time_end** (*int*) – end of peak seeking, by default 1

Returns
signal average amplitude within the specified time quantum

Return type
ave_amp(float)

average_latency(*data=[]*, *time_start=0*, *time_end=1*)

-author: Wu Jieyu -Created on: 2022-8-8 -updata log:

2022-8-15 by Wu Jieyu

Parameters

- **data** (*ndarray*) – the EEG data, by default []
- **time_start** (*int*) – beginning of peak seeking, by default 0
- **time_end** (*int*) – end of peak seeking, by default 1

Returns
location of average amplitude ave_amp(float): signal average amplitude within the specified time quantum

Return type
ave_loc(int)

get_chan_id(*ch_name*, *channels*)

-author: Wu Jieyu -Created on: 2022-8-8 -updata log:

2022-8-15 by Wu Jieyu

Parameters

- **ch_name** (*list*) – selected channels
- **channels** (*list*) – standard channel list

Returns
index of ch_name in channels

Return type
Chan_ID(list)

peak_amplitude(*data=[]*, *time_start=0*, *time_end=1*)

-author: Jiang Hanzhe -Created on: 2022-8-8 -updata log:

2022-8-15 by Jiang Hanzhe

Parameters

- **data** (*ndarray*) – the EEG data, by default []
- **time_start** (*int*) – beginning of peak seeking, by default 0
- **time_end** (*int*) – end of peak seeking, by default 1

Returns
signal peak amplitude within the specified time quantum

Return type
peak_amp(float)

peak_latency(*data=[]*, *time_start=0*, *time_end=1*)

-author: Wu Jieyu -Created on: 2022-8-8 -updata log:

2022-8-15 by Wu Jieyu

Parameters

- **data** (*ndarray*) – the EEG data, by default []
- **time_start** (*int*) – beginning of peak seeking, by default 0
- **time_end** (*int*) – end of peak seeking, by default 1

Returns

location of peak amplitude peak_amp(float): signal peak amplitude within the specified time quantum

Return type

peak_loc(int)

plot_multi_trials(*data*, *sample_num*, *axes=None*)

-author: Wu Jieyu -Created on: 2022-8-8 -updata log:

2022-8-15 by Wu Jieyu

Parameters

- **data** (*ndarray*) – the EEG data
- **sample_num** (*int*) – total number of sampling points in data
- **axes** (*axessubplot*) – drawing area

Returns

drawing area

Return type

ax(axessubplot)

plot_single_trial(*data*, *sample_num*, *axes=None*, *amp_mark=False*, *time_start=0*, *time_end=1*)

-author: Wu Jieyu -Created on: 2022-8-8 -updata log:

2022-8-15 by Wu Jieyu

Parameters

- **data** (*ndarray*) – the EEG data
- **sample_num** (*int*) – total number of sampling points in data
- **axes** (*axessubplot*) – drawing area
- **amp_mark** (*string*) – 'peak' or 'average', call different peak marking methods, by default False (not marked)
- **time_start** (*int*) – beginning of peak seeking, by default 0
- **time_end** (*int*) – end of peak seeking, by default 1

Returns

location of amplitude amp(float): signal amplitude within the specified time quantum ax(axessubplot): drawing area

Return type
loc(int)

plot_topomap(*data*, *point*, *channels*, *fig*, *srate=-1*, *ch_types='eeg'*, *axes=None*)

-author: Wu Jieyu -Created on: 2022-8-8 -updata log:

2022-8-15 by Wu Jieyu

Parameters

- **data** (*ndarray*) – the EEG data
- **point** (*int*) – selected sampling point
- **channels** (*list*) – selected channels
- **fig** (*figure*) – figure
- **srate** (*int*) – sampling rate, by dafault self.fs
- **ch_types** (*list of str | str*) – channel types, by default 'eeg'
- **axes** (*axessubplot*) – drawing area, by default none

Returns
drawing area

Return type
aximage(axessubplot)

stacking_average(*data=[]*, *_axis=[0]*)

-author: Jiang Hanzhe -Created on: 2022-8-8 -updata log:

2022-8-15 by Jiang Hanzhe

Parameters

- **data** (*ndarray*) – the EEG data, by default []
- **_axis** (*list*) – selected the dimensions, by default [0]

Returns
array of averaged signals

Return type
data_mean(ndarray)

metabci.brainda.algorithms.feature_analysis.time_freq_analysis module

class metabci.brainda.algorithms.feature_analysis.time_freq_analysis.**TimeFrequencyAnalysis**(*fs*)

Bases: object

fun_hilbert(*X*, *N=None*, *axis=-1*)

-author: Li Xiaoyu -Created on: 2022-8-8 -updata log:

2022-8-11 by Li Xiaoyu

Parameters

- **X** (*ndarray*) – the input data

- **N** (*int*) – length of the hilbert used.

Returns
discrete-time analytic signal realEnv(ndarray): the real part of the discrete-time analytic signal. imagEnv(ndarray): the imaginary part of the discrete-time analytical signal. angle(ndarray): the angle of the discrete-time analytical signal. envModu(ndarray): the envelope of the discrete-time analytical signal

Return type
analytic_signal(ndarray)

fun_stft(*data*, *fs=None*, *window='hann'*, *nperseg=256*, *noverlap=None*, *nfft=None*, *detrend=False*, *return_onesided=True*, *boundary='zeros'*, *padded=True*, *axis=-1*)

-author: Xin Fengran -Created on: 2022-8-8 -updata log:

2022-8-11 by Xin Fengran

Parameters

- **data** (*ndarray*) – the EEG data
- **fs** (*float*) – the rasampling rate
- **window** (*str or tuple or ndarray*) – desired window to use.
- **nperseg** (*int*) – length of each segment.
- **noverlap** (*int*) – number of points to overlap between segments.
- **nfft** (*int*) – length of the FFT used.
- **detrend** (*str or function or False*) – specifies how to detrend each segment.
- **return_onesided** (*bool*) – If True, return a one-sided spectrum for real data. If False return a two-sided spectrum.
- **returned.** (*Defaults to True, but for complex data, a two-sided spectrum is always*) –
- **boundary** (*str*) – Specifies whether the input signal is extended at both ends, and how to generate the new values,
- **point.** (*in order to center the first windowed segment on the first input*) –
- **padded** (*bool*) – Specifies whether the input signal is zero-padded at the end.
- **axis** (*int*) – axis along which the STFT is computed

Returns
array of sample frequencies t(ndarray): array of segment times Zxx(ndarray): the STFT of the EEG data

Return type
f(ndarray)

fun_topoplot(*X*, *chan_names*, *sfreq=None*, *ch_types='eeg'*)

-author: Li Xiaoyu -Created on: 2022-8-8 -updata log:

2022-8-11 by Li Xiaoyu

Parameters

- **X** (*ndarray*) – the input data
- **chan_names** (*list*) – the name of channels
- **sfreq** (*float*) – the sampling rate
- **ch_types** (*str*) – the type of channel

func_morlet_wavelet(*data*, *xtimes*, *omega*, *sigma*, *fs=None*)

-author: Xin Fengran -Created on: 2022-8-8 -update log:

2022-8-11 by Xin Fengran

Parameters

- **data** – ndarray(Nchannel, Ntimes), the EEG data
- **xtimes** – ndarray(N,), timeline of the EEG data
- **omega** – float.
- **sigma** – float.
- **fs** – float, the resampling rate

Returns

ndarray(Nchannel, N, nTimes), square amplitude of Morlet wavelet transform S: ndarray(Nchannel, N, nTimes), complex values of Morlet wavelet transform

Return type

P

Module contents

metabci.brainda.algorithms.manifold package

Submodules

metabci.brainda.algorithms.manifold.riemann module

Riemannian Geometry for BCI.

class metabci.brainda.algorithms.manifold.riemann.**Alignment**(*align_method: str = 'euclid'*, *cov_method: str = 'lwf'*, *n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin

Characteristics and uses of classes Alignment

Authors: Swolf <swolfforever@gmail.com>

Date: 2021-1-23

update log:

2023-12-18 by Yuwei Liu<liuyuwei20010905@163.com> add code annotation

Riemannian Alignment (RA) uses the Riemannian mean of the covariance matrix of all trials as the reference matrix, so that the center point of the whitened covariance matrix is located in the identity matrix. By performing RA processing on each subject's data, the center point of the covariance matrix for all individuals can be aligned.

Euclidean Alignment (EA) replaces the Riemann mean covariance matrix with the Euclidean mean covariance matrix.

Parameters

- **n_jobs** (*int*) – the default is None
- **align_method** (*str*) – choose the alignment method:'riemann' or 'euclid'
- **cov_method** (*str*) – covariance estimators:'lwf'

n_jobs

the default is None

Type
int

align_method

choose the alignment method:'riemann' or 'euclid'

Type
str

cov_method

covariance estimators:'lwf'

Type
str

iC12_

aligned Riemann/Euclidean center

Type
ndarray,shape(int,int)

References

Tip:

Listing 10: An example using Alignment

```
from metabci.brainda.algorithms.manifold import Alignment
estimator = Alignment(align_method='riemann')
filterX = estimator.fit(X).transform(X)
```

fit(*X: ndarray*, *y: ndarray | None = None*)

Train the model,calculate the aligned center

Parameters
X (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals

Returns

Return type
self

transform(*X*)

Obtain the aligned individual data

Parameters
X (*ndarray,shape(n_trails,n_channels,n_samples)*) – EEG data

Returns
X – aligned EEG data

Return type
ndarray,shape(n_trails,n_channels,n_samples)

class metabci.brainda.algorithms.manifold.riemann.**FGDA**(*n_jobs=1*)

Bases: BaseEstimator, TransformerMixin

Characteristics and uses of classes FGDA

Authors: Swolf <swolfforever@gmail.com>

Created on: 2021-1-23

update log:
2023-12-18 by Yuwei Liu<liuyuwei20010905@163.com> add code annotation

Fisher Geodesic Discriminate Analysis(FGDA) is the application of Fisher Linear Discriminate Analysis in the Riemannian tangent space.FGDA first calculates the projection vectors of the sample covariance matrix of EEG signals in the Riemannian tangent space.Then, leveraging the properties of Riemannian tangent space as a Euclidean space, it performs discriminant feature extraction on the projected vectors in the tangent space based on the Fisher Linear Discriminant Analysis criterion.

Parameters
n_jobs (*int*) – the default of n_jobs is None,meaning it will utilize all available CPUs.

lda_

LDA

Type
discriminate_analysis.Linear Discriminate Analysis

P_

the average covariance matrix calculates from the Riemann matrix

Type
ndarray:shape(int,int)

W_

the weight of LDA

Type
ndarray,shape(int,int)

References

fit(*X*, *y*)

Train the model.

Parameters

- **X** (*ndarray:shape(n_trails,n_channels,n_samples)*) – train data: EEG signals
- **y** (*ndarray:shape(n_trails)*) – the labels of train data

transform(*X*)

Calculate the FGDA from the parameters stored in self

Parameters
X (*ndarray:shape(n_trails,n_channels,n_samples)*) – train data: EEG signals

Returns
Pi – the projection matrix

Return type
ndarray

class metabci.brainda.algorithms.manifold.riemann.**FgMDRM**(*n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin, ClassifierMixin

Characteristics and uses of classes FgMDRM

Authors: Swolf <swolfforever@gmail.com>

Date: 2021-1-23

update log:
2023-12-18 by Yuwei Liu<liuyuwei20010905@163.com> add code annotation

The Fisher Geodesic Minimum Distance to Riemannian Mean(FGMDRM) algorithm is a fusion of MDRM and FGDA.The algorithm first employs FGDA in the tangent space to filter the data,extracting key discriminative features,removing irrelevant noise components. Subsequently, the extracted discriminative features are remapped back to the manifold space. The covariance matrix of the filtered sample space is then calculated based on MDRM to determine the Riemannian centroids for each class. The classification of test data is performed based on the minimum distance principle.

Parameters
n_jobs (*int*) – the default of n_jobs is None,meaning it will utilize all available CPUs.

n_jobs

the default of n_jobs is None,meaning it will utilize all available CPUs.

Type
int

classes_

the class of samples

Type
ndarray,shape(int)

centroids_

Riemannian centroid of two classes

Type
ndarray,shape(int,float,float)

fgda_

Fisher Geodesic Discriminate Analysis(FGDA)

Type
algorithms.mainfold.riemann.FGDA

References

Tip:

Listing 11: An example using FgMDRM

```
from metabci.brainda.algorithms.mainfold import FgMDRM
estimator = FgMDRM()
p_labels = estimator.fit(X[train_ind],y[train_ind]).predict(X[test_ind])
```

---

fit(*X: ndarray*, *y: ndarray*, *sample_weight: ndarray | None = None*)

Train the model.

Parameters

- **X** (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals
- **y** (*ndarray,shape(n_trails)*) – the labels if train data
- **sample_weight** (*ndarray*) – the weight of the model

predict(*X: ndarray*)

Predict the labels

Parameters
X (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals

Returns
self.classes_[np.argmin(dist, axis=1)] – predicted labels

Return type
ndarray,shape(n_trails)

set_fit_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FgMDRM*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in fit.

Returns

self – The updated object.

Return type

object

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *FgMDRM*

Request metadata passed to the score method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to score if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to score.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a Pipeline. Otherwise it has no effect.

Parameters

sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for sample_weight parameter in score.

Returns

self – The updated object.

Return type

object

transform(*X: ndarray*)

Calculate the Riemann distance of each class from the parameters stored in self

Parameters
X (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals

Returns
the Riemann distance of each class

Return type
Self._transform_distance(X)

class metabci.brainda.algorithms.manifold.riemann.**MDRM**(*n_jobs: int = 1*)

Bases: BaseEstimator, TransformerMixin, ClassifierMixin

Characteristics and uses of classes MDRM

Authors: Swolf <swolfforever@gmail.com>

Date: 2021-1-23

update log:
2023-12-18 by Yuwei Liu<liuyuwei20010905@163.com> add code annotation

Minimum Distance to Riemannian Mean(MDRM) is a decoding algorithm based on Riemann distance metric. MDRM calculates the covariance matrix of EEG signals, estimates the Riemannian centroids for each class, then determines the class of a test sample by computing the minimum distance between the test data's covariance matrix and the mean point.

Parameters
n_jobs (*int*) – n_jobs the default is None,meaning it will utilize all available CPUs.

classes_

class labels

Type
ndarray,shape(int)

centroids_

Riemannian centroid of two classes

Type
ndarray,shape(int,float,float)

References

Tip:

Listing 12: An example using MDRM

```
from metabci.brainda.algorithms.mainfold import MDRM
estimator = MDRM()
p_labels = estimator.fit(X[train_ind],y[train_ind]).predict(X[test_ind])
```

---

fit(*X: ndarray*, *y: ndarray*, *sample_weight: ndarray | None = None*)

Train the model

Parameters

- **X** (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals
- **y** (*ndarray,shape(n_trails)*) – label of train data
- **sample_weight** (*ndarray*) – the weight of the samples which is optional,the default is None

Returns
self

Return type
the model

predict(*X: ndarray*)

Predict the label

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – train data: EEG signals

Returns
self.classes_[np.argmin(dist, axis=1)] – predicted labels

Return type
ndarray,shape(n_trails)

predict_proba(*X: ndarray*)

Predict label probabilities

Parameters
X (*ndarray, shape(n_trials, n_channels, n_samples)*) – train data: EEG signals

Returns
softmax(-1 * self._transform_distance(X)) – the probabilities of the predicted labels

Return type
ndarray,shape(n_trails)

set_fit_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *MDRM*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *MDRM*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

`transform`(*X: ndarray*)

Calculate the Riemann distance of each class using the parameters from self.

Parameters
X (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals

Returns
the Riemann distance of each class

Return type
self._transform_distance(X)

class metabci.brainda.algorithms.manifold.riemann.**RecursiveAlignment**(*align_method: str = 'euclid'*, *cov_method: str = 'lwf'*, *n_jobs: int | None = None*)

Bases: BaseEstimator, TransformerMixin

Characteristics and uses of classes RecursiveAlignment

Authors: Swolf <swolfforever@gmail.com>

Date: 2021-1-23

update log:
2023-12-18 by Yuwei Liu<liuyuwei20010905@163.com> add code annotation

In order to overcome the problem that the trial data gradually appear in chronological order under the online experiment, there is no initial sample size estimation center, and the calculation process of the Riemann center is complex, and it takes a lot of time to recalculate the Riemannian center in the feedback stage, the Recursive Riemannian Alignment (rRA) and Recursive Euclidean Alignment (rEA) suitable for the online stage were proposed.

Parameters

- **n_jobs** (*int*) – the default is None
- **align_method** (*str*) – choose the alignment method:'riemann' or 'euclid'
- **cov_method** (*str*) – covariance estimators:'lwf'

n_jobs

the default is None

Type
int

align_method

choose the alignment method:'riemann' or 'euclid'

Type
str

cov_method

covariance estimators:'lwf'

Type
str

iC12_

aligned Riemann/Euclidean center

Type
ndarray,shape(int,int)

n_tracked

the number of iterations

Type
int

C_

the Euclid or Riemann center after iteration

Type
ndarray

References

Tip:

Listing 13: An example using RecursiveAlignment

```
from metabci.brainda.algorithms.manifold import RecursiveAlignment
estimator = RecursiveAlignment(align_method='riemann')
filterX = estimator.fit(X).transform(X)
```

fit(*X*, *y=None*)

recursive alignment

Parameters
X (*ndarray,shape(n_trails,n_channels.n_samples)*) – EEG data

Returns

Return type
self

transform(*X*)

obtain the subject's data after recursive alignment

Parameters
X (*ndarray,shape(n_trails,n_channels.n_samples)*) – EEG data

Returns
X – the individual data after recursive alignment

Return type
ndarray,shape(n_trails,n_channels.n_samples)

class metabci.brainda.algorithms.manifold.riemann.**TSClassifier**(*clf=LogisticRegression()*, *n_jobs=None*)

Bases: BaseEstimator, ClassifierMixin

Characteristics and uses of classes TSClassifier

Authors: Swolf <swolfforever@gmail.com>

Date: 2021-1-23

update log:
2023-12-18 by Yuwei Liu<liuyuwei20010905@163.com> add code annotation

The Tangent Space Classifier (TSClassifier) is a general term for classifiers constructed in the Riemannian tangent space,which is treated as a Euclidean space. Methods such as LDA (Linear Discriminant Analysis), SVM (Support Vector Machine),Logistic Regression, and others are employed to build classifiers in this Riemannian tangent space.

Parameters

- **n_jobs** (*int*) – the default of n_jobs is None,meaning it will utilize all available CPUs.
- **clf** (*linear_model._logistic.LogisticRegression*) – Logistic Regression

n_jobs

the default of n_jobs is None,meaning it will utilize all available CPUs.

Type
int

clf

Logistic Regression

Type
linear_model._logistic.LogisticRegression

P_

The average covariance matrix returned according to the Riemann matrix

Type
ndarray,shape(int,int)

References

Tip:

Listing 14: An example using TSClassifier

```
from metabci.brainda.algorithms.manifold import TSClassifier
estimator = TSClassifier()
p_labels = estimator.fit(X[train_ind], y[train_ind]).predict(X[test_ind])
```

---

fit(*X: ndarray*, *y: ndarray*)

Train the model

Parameters

- **X** (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals
- **y** (*ndarray,shape(n_trails)*) – the labels if train data

predict(*X: ndarray*)

Predict labels

Parameters
X (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals

Returns
self.clf.predict(vSi)

Return type
ndarray,shape(n_trails) predicted labels

predict_proba(*X: ndarray*)

Predict label probabilities

Parameters
X (*ndarray,shape(n_trails,n_channels,n_samples)*) – train data: EEG signals

Returns
self.clf.predict_proba(vSi) – predicted label probabilities

Return type
ndarray,shape(n_trails)

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *TSClassifier*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
sample_weight (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `sample_weight` parameter in `score`.

Returns
self – The updated object.

Return type
object

metabci.brainda.algorithms.manifold.riemann.**distance_riemann**(*A: ndarray*, *B: ndarray*, *n_jobs: int | None = None*)

Riemannian distance between two covariance matrices A and B.

$$d = \left(\sum_i \log(\lambda_i)^2 \right)^{-1/2}$$

where λ_i are the joint eigenvalues of A and B.

Parameters

- **A** (*ndarray*) – First positive-definite matrix, shape (n_trials, n_channels, n_channels) or (n_channels, n_channels).
- **B** (*ndarray*) – Second positive-definite matrix.

Returns
Riemannian distance between A and B.

Return type
ndarray | float

metabci.brainda.algorithms.manifold.riemann.**expmap**(*Si: ndarray*, *P: ndarray*, *n_jobs: int | None = None*)

Exponential map from the tangent space to the positive-definite space.

Exponential map projects $\mathbf{S}_i \in \mathcal{T}_{\mathbf{P}}\mathcal{M}$ bach to the manifold $\mathcal{M}$.

Parameters

- **Si** (*ndarray*) – Tangent space point (in matrix form).
- **P** (*ndarray*) – Reference point.
- **n_jobs** (*int, optional*) – the number of jobs to use.

Returns
Pi – SPD matrix.

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**geodesic**(*P1: ndarray*, *P2: ndarray*, *t: float*, *n_jobs: int | None = None*)

Geodesic.

The geodesic curve between any two SPD matrices $\mathbf{P}_1, \mathbf{P}_2 \in \mathcal{M}$.

Parameters

- **P1** (*ndarray*) – SPD matrix.
- **P2** (*ndarray*) – SPD matrix, the same shape of P1.
- **t** (*float*) – $0 \leq t \leq 1$.
- **n_jobs** (*int, optional*) – the number of jobs to use.

Returns
phi – SPD matrix on the geodesic curve between P1 and P2.

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**logmap**(*Pi: ndarray*, *P: ndarray*, *n_jobs: int | None = None*)

Logarithm map from the positive-definite space to the tangent space.

Logarithm map projects $\mathbf{P}_i \in \mathcal{M}$ to the tangent space point $\mathbf{S}_i \in \mathcal{T}_{\mathbf{P}}\mathcal{M}$ at $\mathbf{P} \in \mathcal{M}$.

Parameters

- **Pi** (*ndarray*) – SPD matrix.
- **P** (*ndarray*) – Reference point.
- **n_jobs** (*int, optional*) – the number of jobs to use.

Returns
Si – Tangent space point (in matrix form).

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**mdrm_kernel**(*X: ndarray*, *y: ndarray*, *sample_weight: ndarray | None = None*, *n_jobs: int = 1*)

Minimum Distance to Riemannian Mean.

Parameters

- **X** (*ndarray*) – eeg data, shape (n_trials, n_channels, n_samples)
- **y** (*ndarray*) – labels, shape (n_trials)
- **sample_weight** (*Optional[ndarray], optional*) – sample weights, by default None
- **n_jobs** (*int*) – the number of jobs to use, by default 1

Returns
centroids of each class, shape (n_class, n_channels, n_channels).

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**mean_riemann**(*covmats*, *tol=1e-11*, *maxiter=300*, *init=None*, *sample_weight=None*, *n_jobs=None*)

Return the mean covariance matrix according to the Riemannian metric.

The procedure is similar to a gradient descent minimizing the sum of riemannian distance to the mean.

$$\mathbf{C} = \arg\min \left(\sum_i \delta_R(\mathbf{C}, \mathbf{C}_i)^2 \right)$$

where δ_R is riemann distance.

Parameters

- **covmats** (*ndarray*) – Covariance matrices set, shape (n_trials, n_channels, n_channels).
- **tol** (*float, optional*) – The tolerance to stop the gradient descent (default 1e-8).
- **maxiter** (*int, optional*) – The maximum number of iteration (default 50).
- **init** (*None|ndarray, optional*) – A covariance matrix used to initialize the gradient descent (default None), if None the arithmetic mean is used.
- **sample_weight** (*None|ndarray, optional*) – The weight of each sample (efault None), if None weights are 1 otherwise weights are normalized.

Returns
C – The Riemannian mean covariance matrix.

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**tangent_space**(*Pi: ndarray*, *P: ndarray*, *n_jobs: int | None = None*)

Logarithm map projects SPD matrices to the tangent vectors.

Parameters

- **Pi** (*ndarray*) – SPD matrices, shape (n_trials, n_channels, n_channels).
- **P** (*ndarray*) – Reference point.

Returns
vSi – Tangent vectors, shape (n_trials, n_channels*(n_channels+1)/2).

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**untangent_space**(*vSi: ndarray*, *P: ndarray*, *n_jobs: int | None = None*)

Logarithm map projects SPD matrices to the tangent vectors.

Parameters

- **vSi** (*ndarray*) – Tangent vectors, shape (n_trials, n_channels*(n_channels+1)/2).
- **P** (*ndarray*) – Reference point.

Returns
Pi – SPD matrices, shape (n_trials, n_channels, n_channels).

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**unvectorize**(*vSi: ndarray*)

unvectorize tangent space points.

Parameters
vSi (*ndarray*) – vectorized version of Si, shape (n_trials, n_channels*(n_channels+1)/2)

Returns
points in the tangent space, shape (n_trials, n_channels, n_channels)

Return type
ndarray

metabci.brainda.algorithms.manifold.riemann.**vectorize**(*Si: ndarray*)

vectorize tangent space points.

Parameters
Si (*ndarray*) – points in the tangent space, shape (n_trials, n_channels, n_channels)

Returns
vectorized version of Si, shape (n_trials, n_channels*(n_channels+1)/2)

Return type
ndarray

metabci.brainda.algorithms.manifold.rpa module

Riemannian Procrustes Analysis. Modified from https://github.com/plcrodrigues/RPA

metabci.brainda.algorithms.manifold.rpa.**get_recenter**(*X: ndarray*, *cov_method: str = 'cov'*, *mean_method: str = 'riemann'*, *n_jobs: int = 1*)

metabci.brainda.algorithms.manifold.rpa.**get_rescale**(*X: ndarray*, *cov_method: str = 'cov'*, *n_jobs: int = 1*)

metabci.brainda.algorithms.manifold.rpa.**get_rotate**(*Xs: ndarray*, *ys: ndarray*, *Xt: ndarray*, *yt: ndarray*, *cov_method: str = 'cov'*, *metric: str = 'euclid'*, *n_jobs: int = 1*)

metabci.brainda.algorithms.manifold.rpa.**recenter**(*X: ndarray*, *iM12: ndarray*)

metabci.brainda.algorithms.manifold.rpa.**rescale**(*X: ndarray*, *M: ndarray*, *scale: float*, *cov_method: str = 'cov'*, *n_jobs: int = 1*)

metabci.brainda.algorithms.manifold.rpa.**rotate**(*Xt: ndarray*, *Ropt: ndarray*)

Module contents

metabci.brainda.algorithms.transfer_learning package

Submodules

metabci.brainda.algorithms.transfer_learning.base module

metabci.brainda.algorithms.transfer_learning.lst module

Least-squares Transformation (LST).

See https://iopscience.iop.org/article/10.1088/1741-2552/abcb6e.

class metabci.brainda.algorithms.transfer_learning.lst.**LST**(*n_jobs=None*)

Bases: BaseEstimator, TransformerMixin

fit(*X: ndarray*, *y: ndarray*)

transform(*X: ndarray*, *y: ndarray*)

metabci.brainda.algorithms.transfer_learning.lst.**lst_kernel**(*S: ndarray*, *T: ndarray*)

metabci.brainda.algorithms.transfer_learning.mekt module

Manifold Embedded Knowledge Transfer. Modified from https://github.com/chamwen/MEKT.git

class metabci.brainda.algorithms.transfer_learning.mekt.**MEKT**(*subspace_dim: int = 10*, *max_iter: int = 5*, *alpha: float = 0.01*, *beta: float = 0.1*, *rho: float = 20*, *k: int = 10*, *t: int = 1*, *covariance_type='lwf'*)

Bases: BaseEstimator, TransformerMixin

Manifold Embedded Knowledge Transfer(MEKT)

fit_transform(*Xs*, *ys*, *Xt*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*) – Target values (None for unsupervised transformations).
- ****fit_params** (*dict*) – Additional fit parameters.

Returns
X_new – Transformed array.

Return type
ndarray array of shape (n_samples, n_features_new)

metabci.brainda.algorithms.transfer_learning.mekt.**anova_dimension_reduction**(*Xs*, *ys*, *d*)

Dimension reduction in MEKT.

MEKT use len(ys) as d

Parameters

- **Xs** (*ndarray*) – features, shape (n_trials, n_features)
- **ys** (*ndarray*) – labels, shape (n_trials,)
- **d** (*int*) – reduce to dimension d

Returns
f_ix – the index of selected features, shape (d,)

Return type
ndarray

metabci.brainda.algorithms.transfer_learning.mekt.**choose_multiple_subjects**(*Xs*, *Xt*, *ys*, *y_subjects*, *k=1*)

choose k most appropriate subjects according to dte.

Parameters

- **Xs** (*ndarray*) – source features, shape (n_trials*n_subjects, n_features)
- **Xt** (*ndarray*) – target features, shape (n_trials, n_features)
- **ys** (*ndarray*) – source labels, shape (n_trials*n_subjects, n_features)
- **y_subjects** (*ndarray*) – subject labels, shape (n_trials*n_subjects,)
- **k** (*int, optional*) – k subjects, by default 1

Returns

- **subject_ix** (*ndarray*) – selected subject boolean index, shape (n_trials*n_subjects,)
- **subjects** (*ndarray*) – selected subject ids, shape (k,)

metabci.brainda.algorithms.transfer_learning.mekt.**dte**(*Xs*, *Xt*, *ys*)

Domain Transferiability Estimation

Parameters: Xs: ndarray

source features, shape (n_source_trials, n_features)

Xt: ndarray
target features, shape (n_traget_trials, n_features)

ys: ndarray
source labels, shape (n_source_trials,)

metabci.brainda.algorithms.transfer_learning.mekt.**graph_laplacian**(*Xs*, *k=10*, *t=1*)

Graph Laplacian Matrix.

Currently with heat kernel implemented.

Parameters

- **Xs** (*ndarray*) – features, shape (n_trials, n_samples)
- **k** (*int, optional*) – k nearest neighbors, by default 10
- **t** (*int, optional*) – heat kernel parameter, by default 1

Returns

- **L** (*ndarray*) – unnormalized laplacian kernel, shape (n_trials, n_trials)
- **D** (*ndarray*) – degree matrix, L = D - W, shape (n_trials, n_trials)

metabci.brainda.algorithms.transfer_learning.mekt.**mekt_feature**(*X*, *covariance_type*)

Covariance Matrix Centroid Alignment and Tangent Space Feature Extraction.

Parameters

Returns

featureX – feature of X, shape (n_trials, n_feature)

Return type

ndarray

metabci.brainda.algorithms.transfer_learning.mekt.**mekt_kernel**(*Xs*, *Xt*, *ys*, *d=10*, *max_iter=5*, *alpha=0.01*, *beta=0.1*, *rho=20*, *k=10*, *t=1*)

Manifold Embedding Knowledge Transfer.

Parameters

- **Xs** (*ndarray*) – source features, shape (n_source_trials, n_features)
- **Xt** (*ndarray*) – target features, shape (n_target_trials, n_features)
- **ys** (*ndarray*) – source labels, shape (n_source_trials,)
- **d** (*int, optional*) – selected d projection vectors, by default 10
- **max_iter** (*int, optional*) – max iterations, by default 5
- **alpha** (*float, optional*) – regularized term for source domain discriminability, by default 0.01
- **beta** (*float, optional*) – regularized term for target domain locality, by default 0.1
- **rho** (*int, optional*) – regularized term for parameter transfer, by default 20
- **k** (*int, optional*) – number of nearest neighbors
- **t** (*int, optional*) – heat kernel parameter

Returns

- **A** (*ndarray*) – projection matrix for source, shape (n_features, d)
- **B** (*ndarray*) – projection matrix for target, shape (n_features, d)

metabci.brainda.algorithms.transfer_learning.mekt.**scatter_matrix**(*X*, *y*)

Compute between-class scatter matrix.

Parameters

- **X** (*ndarray*) – features, shape (n_trials, n_features)
- **y** (*ndarray*) – labels, shape (n_trials,)

Returns
Sb – between-class scatter matrix, shape (n_features, n_features)

Return type
ndarray

metabci.brainda.algorithms.transfer_learning.mekt.**source_discriminability**(*Xs*, *ys*)

Source features discriminability.

Parameters

- **Xs** (*ndarray*) – source features, shape (n_trials, n_features)
- **ys** (*ndarray*) – labels, shape (n_trials)

Returns

- **Sw** (*ndarray*) – within-class scatter matrix, shape (n_features, n_features)
- **Sb** (*ndarray*) – between-class scatter matrix, shape (n_features, n_features)

metabci.brainda.algorithms.transfer_learning.same module

source aliasing matrix estimation (SAME) and its multi-stimulus version (msSAME).

A data augmentation method named Source Aliasing Matrix Estimation (SAME) [1] to enhance the performance of state-of-the-art spatial filtering methods (i.e., eTRCA, TDCA) for SSVEP-BCIs. Based on the superposition model of SSVEPs, the task-related components are reconstructed by estimating the source aliasing matrixes. After adding noise, multiple artificial signals are generated and then added to calibrated data in an appropriate proportion.

In 2023, paper [2] proposes an extended version of SAME, called multi-stimulus SAME (msSAME), which exploits the similarity of the aliasing matrix across frequencies to enhance the performance of SSVEP-BCI with insufficient calibration trials.

souce code of SAME: https://github.com/RuixinLuo/Source-Aliasing-Matrix-Estimation-DataAugmentation-SAME-SSVEP

class metabci.brainda.algorithms.transfer_learning.same.**MSSAME**(*n_jobs=None*, *fs=250*, *flist=None*, *plist=None*, *Nh=5*, *n_Aug=5*, *n_Neig=12*, *alpha=0.05*)

Bases: `BaseEstimator`, `TransformerMixin`

multi-stimulus source aliasing matrix estimation (msSAME) [1]_.

author: Ruixin Luo <ruixin_luo@tju.edu.cn>

Created on: 2023-11-13

update log:

Parameters

- **fs** (*int*) – Sampling rate.
- **flist** (*list*) – Frequency of all class.
- **plist** (*list*) – Phase of all class.
- **Nh** (*int*) – The number of harmonics.
- **n_Aug** (*int*) – The number of generated signals
- **n_Neig** (*int*) – The number of neighborhood frequency

- **alpha** (*float*) – Intensity of noise, default 0.05.

T_

Average template for different classes of data.

Type
list

classes_

number of classes.

Type
ndarray

Raises
ValueError – None

References

Tip:

Listing 15: A example using MSSAME

```
from metabci.brainda.algorithms.transfer_learning import MSSAME
mssame = MSSAME(fs = 250, Nh = 5, flist = freq_list, plist=phase_list, n_Aug=4, n_Neig=14)
mssame.fit(X_train , y_train)
X_aug, y_aug = mssame.augment()
X_train_new = np.concatenate((X_train, X_aug), axis=0)
y_train_new = np.concatenate((y_train, y_aug), axis=0)
```

augment()

Calculating augmentation signals.

Returns

- **X_aug** (*ndarray*) – augmentation data, shape(n_events*n_aug, n_channels, n_samples).
- **y_aug** (*ndarray*) – Label of augmentation data, shape(n_events*n_aug,).

fit(*X: ndarray*, *y: ndarray*)

model training

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Label, shape(n_trials,)

class metabci.brainda.algorithms.transfer_learning.same.**SAME**(*n_jobs=None*, *fs=250*, *flist=None*, *Nh=5*, *n_Aug=5*, *alpha=0.05*)

Bases: BaseEstimator, TransformerMixin

source aliasing matrix estimation (SAME) [1]_.

author: Ruixin Luo <ruixin_luo@tju.edu.cn>

Created on: 2023-01-09

update log:

2023-09-06 by Ruixin Luo <ruixin_luo@tju.edu.cn>

2023-10-03 by Jie Mei <chmeijie@tju.edu.cn>

Parameters

- **fs** (*int*) – Sampling rate.
- **flist** (*list*) – Frequency of all class.
- **Nh** (*int*) – The number of harmonics.
- **n_Aug** (*int*) – The number of generated signals
- **alpha** (*float*) – Intensity of noise, default 0.05.

T_

Average template for different classes of data.

Type
list

classes_

number of classes.

Type
ndarray

Raises
ValueError – None

References

Tip:

Listing 16: A example using SAME

```
from metabci.brainda.algorithms.transfer_learning import SAME
same = SAME(fs = 250, Nh = 5, flist = freq_list, n_Aug = 4)
same.fit(X_train , y_train)
X_aug, y_aug = same.augment()
X_train_new = np.concatenate((X_train, X_aug), axis=0)
y_train_new = np.concatenate((y_train, y_aug), axis=0)
```

---

augment()

Calculating augmentation signals.

Returns

- **X_aug** (*ndarray*) – augmentation data, shape(n_events*n_aug, n_channels, n_samples).
- **y_aug** (*ndarray*) – Label of augmentation data, shape(n_events*n_aug,).

fit(*X: ndarray*, *y: ndarray*)

model training

Parameters

- **X** (*ndarray*) – EEG data, shape(n_trials, n_channels, n_samples).
- **y** (*ndarray*) – Label, shape(n_trials,)

metabci.brainda.algorithms.transfer_learning.same.**TRCs_estimation**(*data*, *mean_target*)

source signal estimation using LST [1]_

author: Ruixin Luo <ruixin_luo@tju.edu.cn>

Created on: 2023-01-09

update log:
2023-09-06 by Ruixin Luo <ruixin_luo@tju.edu.cn>

Parameters

- **data** (*ndarray*) – Reference signal, shape(n_channel_1, n_times).
- **mean_target** (*ndarray*) – Average template, shape(n_channel_2, n_times).

Returns
data_after – Source signal, shape(n_channel_2, n_times).

Return type
ndarray

References

metabci.brainda.algorithms.transfer_learning.same.**get_augment_noiseAfter**(*fs*, *f*, *Nh*, *n_Aug*, *mean_temp*, *alpha=0.05*)

Artificially generated signals by SAME

author: Ruixin Luo <ruixin_luo@tju.edu.cn>

Created on: 2023-01-09

update log:
2023-09-06 by Ruixin Luo <ruixin_luo@tju.edu.cn>

Parameters

- **fs** (*int*) – Sampling rate.
- **f** (*float*) – Frequency of signal.
- **Nh** (*int*) – The number of harmonics.
- **n_Aug** (*int*) – The number of generated signals.
- **mean_temp** (*ndarray*) – Average template, shape(n_channels, n_times).
- **alpha** (*float*) – Intensity of noise, default 0.05.

Returns
data_aug – Artificially generated signals, shape(n_channel, n_times, n_Aug).

Return type
ndarray

Note: Please note that we apply SAME before filter bank analysis in the MetaBCI version. This is convenient for compatibility with MetaBCI and saves computational effort. After testing, it still has a similar improvement effect.

metabci.brainda.algorithms.transfer_learning.same.**get_augment_noiseAfter_ms**(*fs*, *f_list*, *phi_list*, *Nh*, *n_Aug*, *mean_temp_all*, *iEvent*, *n_Templates*, *alpha=0.05*)

Artificially generated signals by msSAME

author: Ruixin Luo <ruixin_luo@tju.edu.cn>

Created on: 2023-11-09

update log:

Parameters

- **fs** (*int*) – Sampling rate.
- **f_list** (*list*) – The all frequency of reference signal.
- **phi_list** (*list*) – The all phase of reference signal
- **Nh** (*int*) – The number of harmonics.
- **n_Aug** (*int*) – The number of generated signals
- **mean_temp_all** (*ndarray-like (n_channel, n_times, n_events)*) – Average template of all events.
- **iEvent** (*int*) – the i-th event for the selection of neighboring frequencies
- **n_Templates** (*int*) – The number of neighboring frequencies
- **alpha** (*float*) – Intensity of noise, default 0.05.

Returns

data_aug – Artificially generated signals.

Return type

ndarray-like (n_channel, n_times, n_Aug)

Note: Please note that we apply msSAME before filter bank analysis in the MetaBCI version. This is convenient for compatibility with MetaBCI and saves computational effort. After testing, it still has a similar improvement effect.

Module contents

metabci.brainda.algorithms.utils package

Submodules

metabci.brainda.algorithms.utils.covariance module

class metabci.brainda.algorithms.utils.covariance.**Covariance**(*estimator='cov'*, *n_jobs=1*)

Bases: BaseEstimator, TransformerMixin

Estimation of covariance matrix.

Parameters

- **estimator** (*str or callable object, optional*) – Covariance estimator to use (the default is *cov*, which uses empirical covariance estimator). For regularization, consider *lwf* or *oas*.

  supported estimators

  cov: empirial covariance estimator

  lwf: ledoit wolf covariance estimator

  oas: oracle approximating shrinkage covariance estimator

  mcd: minimum covariance determinant covariance estimator

- **n_jobs** (*int or None, optional*) – The number of CPUs to use to do the computation (the default is 1, -1 for all processors).

See also:

ERPCovariance

fit(*X*, *y=None*)

Not used, only for compatibility with sklearn API.

Parameters

- **X** (*ndarray*) – EEG signal, shape (…, n_channels, n_samples).
- **y** (*ndarray*) – Labels.

Returns
self – The Covariance instance.

Return type
Covariance instance

transform(*X*)

Transform EEG to covariance matrix.

Parameters
X (*ndarray*) – EEG signal, shape (…, n_channels, n_samples).

Returns
covmats – Estimated covariances, shape (…, n_channels, n_channels)

Return type
ndarray

metabci.brainda.algorithms.utils.covariance.**covariances**(*X: ndarray*, *estimator: str | Callable[[ndarray], ndarray] = 'cov'*, *n_jobs: int = 1*) → ndarray

Estimation of covariance matrix.

Parameters

- **X** (*ndarray*) – EEG signal, shape (..., n_channels, n_samples).
- **estimator** (*str or callable object, optional*) – Covariance estimator to use (the default is *cov*, which uses empirical covariance estimator). For regularization, consider *lwf* or *oas*.

  supported estimators

  cov: empirial covariance estimator

  lwf: ledoit wolf covariance estimator

  oas: oracle approximating shrinkage covariance estimator

  mcd: minimum covariance determinant covariance estimator
- **n_jobs** (*int or None, optional*) – The number of CPUs to use to do the computation (the default is 1, -1 for all processors).

Returns
covmats – covariance matrices, shape (..., n_channels, n_channels)

Return type
ndarray

See also:

covariances_erp()

metabci.brainda.algorithms.utils.covariance.**expm**(*Ci: ndarray*, *n_jobs: int | None = None*)

Return the matrix exponential of a covariance matrix.

Parameters
Ci (*ndarray*) – Input positive-definite matrix.

Returns
Exponential matrix of Ci.

Return type
ndarray

Notes

$$\mathbf{C} = \mathbf{V} \exp(\mathbf{\Lambda}) \mathbf{V}^T$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{Ci}$.

metabci.brainda.algorithms.utils.covariance.**invsqrtm**(*Ci: ndarray*, *n_jobs: int | None = None*)

Return the inverse matrix square root of a covariance matrix.

Parameters
Ci (*ndarray*) – Input positive-definite matrix.

Returns
Inverse matrix square root of Ci.

Return type
ndarray

Notes

$$\mathbf{C} = \mathbf{V}\left(\mathbf{\Lambda}\right)^{-1/2}\mathbf{V}^{T}$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{Ci}$.

metabci.brainda.algorithms.utils.covariance.**isPD**(*B: ndarray*) → bool

Returns true when input matrix is positive-definite, via Cholesky decompositon method.

Parameters
B (*ndarray*) – Any matrix, shape (N, N)

Returns
True if B is positve-definite.

Return type
bool

Notes

Use numpy.linalg rather than scipy.linalg. In this case, scipy.linalg has unpredictable behaviors.

metabci.brainda.algorithms.utils.covariance.**logm**(*Ci: ndarray*, *n_jobs: int | None = None*)

Return the matrix logrithm of a covariance matrix.

Parameters
Ci (*ndarray*) – Input positive-definite matrix.

Returns
Logrithm matrix of Ci.

Return type
ndarray

Notes

$$\mathbf{C} = \mathbf{V}\log\left(\mathbf{\Lambda}\right)\mathbf{V}^{T}$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{Ci}$.

metabci.brainda.algorithms.utils.covariance.**matrix_operator**(*Ci: ndarray*, *operator: Callable[[ndarray], ndarray]*, *n_jobs: int | None = None*) → ndarray

Apply operator to any matrix.

Parameters

- **Ci** (*ndarray*) – Input positive definite matrix.
- **operator** (*callable object*) – Operator function or callable object.
- **n_jobs** (*int, optional*) – the number of jobs to use.

Returns
Co – Operated matrix.

Return type
ndarray

Raises
ValueError – If Ci is not positive definite.

Notes

$$\mathbf{Ci} = \mathbf{V}\left(\mathbf{\Lambda}\right)\mathbf{V}^T$$
$$\mathbf{Co} = \mathbf{V}operator\left(\mathbf{\Lambda}\right)\mathbf{V}^T$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{Ci}$.

metabci.brainda.algorithms.utils.covariance.**nearestPD**(*A: ndarray*) → ndarray

Find the nearest positive-definite matrix to input.

Parameters
A (*ndarray*) – Any square matrxi, shape (N, N)

Returns
A3 – positive-definite matrix to A

Return type
ndarray

Notes

A Python/Numpy port of John D'Errico's *nearestSPD* MATLAB code[1], which origins at[2].

References

metabci.brainda.algorithms.utils.covariance.**powm**(*Ci: ndarray*, *alpha: float*, *n_jobs: int | None = None*)

Return the matrix power of a covariance matrix.

Parameters

- **Ci** (*ndarray*) – Input positive-definite matrix.
- **alpha** (*float*) – Exponent.

Returns
Power matrix of Ci.

Return type
ndarray

[1] https://www.mathworks.com/matlabcentral/fileexchange/42885-nearestspd
[2] N.J. Higham, "Computing a nearest symmetric positive semidefinite matrix" (1988): https://doi.org/10.1016/0024-3795(88)90223-6

Notes

$$\mathbf{C} = \mathbf{V}\left(\mathbf{\Lambda}\right)^{\alpha}\mathbf{V}^{T}$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{Ci}$.

metabci.brainda.algorithms.utils.covariance.**sqrtm**(*Ci: ndarray*, *n_jobs: int | None = None*)

Return the matrix square root of a covariance matrix.

Parameters
Ci (*ndarray*) – Input positive-definite matrix.

Returns
Square root matrix of Ci.

Return type
ndarray

Notes

$$\mathbf{C} = \mathbf{V}\left(\mathbf{\Lambda}\right)^{1/2}\mathbf{V}^{T}$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{Ci}$.

metabci.brainda.algorithms.utils.model_selection module

class metabci.brainda.algorithms.utils.model_selection.**EnhancedLeaveOneGroupOut**(*return_validate: bool = True*)

Bases: LeaveOneGroupOut

Leave one method for cross-validation. Performs leave-one method cross validation that can contain validation sets.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

Parameters
return_validate (*bool*) – Whether a validation set is required, which defaults to True.

return_validate

Same as return_validate in Parameters.

Type
bool

validate_spliter

Validate set divider, valid only if return_validate is True. See sklearn.model_selection.StratifiedShuffleSplit() for details.

Type
sklearn.model_selection.StratifiedShuffleSplit()

set_split_request(*, *groups: bool | None | str = '$UNCHANGED$'*) → *EnhancedLeaveOneGroupOut*

Request metadata passed to the `split` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `split` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `split`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

groups (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `groups` parameter in `split`.

Returns

self – The updated object.

Return type

object

split(*X*, *y=None*, *groups=None*)

Returns the training, validation, and test set index subscript (return_validate is True) or the training, test set data (return_validate is False).

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:

2023-12-26 by sunchang<18822197631@163.com>

X: array-like, shape(n_samples, n_features)

Training data. n_samples indicates the number of samples, and n_features indicates the number of features.

y: array-like, shape(n_samples,)

Category label.Further adjustment is required by _generate_sequential_groups(y).

groups: None

The grouping label of the sample used when the data set is split into training, validation (return_validate is True), and test sets. The number of groups (the number of validation breaks) is calculated by this parameter. The number of groups here actually determines the sample size of the "one" part of the leave-one method. For example, a set composed of 6 samples with the group number [1,1,2,3,3] means that the set is divided into three parts, with the number of samples being

2, 1 and 3 respectively. In the reserve-one method, the set composed of 2 samples,1 samples and 3 samples is regarded as a test set, and the remaining part is regarded as a training set. groups can be entered externally or computed by an internal function based on the category label.

train: ndarray
Training set sample index subscript or training set data.

validate: ndarray
Validate set sample index index subscript (return_validate is True).

test: ndarray
Test set sample index subscript or test set data.

get_n_splitsReturns the number of packet iterators, that is, the number of packets. _generate_sequential_groupsThe sample group tag "groups" is generated.

class metabci.brainda.algorithms.utils.model_selection.**EnhancedStratifiedKFold**(*n_splits: int = 5*, *shuffle: bool = False*, *return_validate: bool = True*, *random_state: int | RandomState | None = None*)

Bases: StratifiedKFold

Enhanced Stratified KFold cross-validator.

if return_validate is True, split return (train, validate, test) indexs, else (train, test) as the sklearn StratifiedKFold.fit the validate size should be the same as the test size.

Hierarchical K-fold cross-validation. When the samples are unbalanced, the data set is divided according to the proportion of each type of sample to the total sample.

Performs hierarchical k-fold cross-validation that can contain validation sets. The sample size of the validation set will be the same as that of the test set.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **n_splits** (*int*) – Cross validation fold, default is 5.
- **shuffle** (*bool*) – Whether to scramble the sample order. The default is False.
- **return_validate** (*bool*) – Whether a validation set is required, which defaults to True.
- **random_state** (*int or numpy.random.RandomState()*) – Random initial state. When shuffle is True, random_state determines the initial ordering of the samples, hrough which the randomness of the selection of various data samples in each compromise can be controlled. See sklearn. Model_selection. StratifiedKFold () for details. The default is None.

return_validate

Same as return_validate in Parameters.

Type
bool

validate_spliter

Validate set divider, valid only if return_validate is True. See sklearn.model_selection.StratifiedShuffleSplit() for details.

Type
sklearn.model_selection.StratifiedShuffleSplit()

split(*X*, *y*, *groups=None*)

Returns the training, validation, and test set index subscript (return_validate is True) or the training, test set data (return_validate is False).

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

X: array-like, shape(n_samples, n_features)
Training data. n_samples indicates the number of samples, and n_features indicates the number of features.

y: array-like, shape(n_samples,)
Category label.

groups: None
Ignorable parameter, used only for version matching.

train: ndarray
Training set sample index subscript or training set data.

validate: ndarray
Validate set sample index index subscript (return_validate is True).

test: ndarray
Test set sample index subscript or test set data.

class metabci.brainda.algorithms.utils.model_selection.**EnhancedStratifiedShuffleSplit**(*test_size: float*, *train_size: float*, *n_splits: int = 5*, *validate_size: float | None = None*, *return_validate: bool = True*, *random_state: int | RandomState | None = None*)

Bases: StratifiedShuffleSplit

Hierarchical random cross validation. When the samples are unbalanced, the data set is divided according to the proportion of each type of sample to the total sample. Perform hierarchical random cross validation that can contain validation sets. The sample size of the validation set will be the same as that of the test set.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:

2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **test_size** (*float*) – Test set ratio (0-1).
- **train_size** (*float*) – Train set ratio (0-1).
- **n_splits** (*int*) – Cross validation fold, default is 5.
- **validate_size** (*float or None*) – The proportion of the validation set (when return_validate is True) (0-1), defaults to None.
- **return_validate** (*bool*) – Whether a validation set is required, which defaults to True.
- **random_state** (*int or numpy.random.RandomState()*) – Random initial state. See sklearn. Model_selection. StratifiedShuffleSplit () for details, the default value is None.

return_validate

Same as return_validate in Parameters.

Type
bool

validate_spliter

Validate set divider, valid only if return_validate is True. See sklearn.model_selection.StratifiedShuffleSplit() for details.

Type
sklearn.model_selection.StratifiedShuffleSplit()

split(*X*, *y*, *groups=None*)

Returns the training, validation, and test set index subscript (return_validate is True) or the training, test set data (return_validate is False).

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

X: array-like, shape(n_samples, n_features)
Training data. n_samples indicates the number of samples, and n_features indicates the number of features.

y: array-like, shape(n_samples,)
Category label.

groups: None
Ignorable parameter, used only for version matching.

train: ndarray
Training set sample index subscript or training set data.

validate: ndarray
Validate set sample index index subscript (return_validate is True).

test: ndarray
Test set sample index subscript or test set data.

metabci.brainda.algorithms.utils.model_selection.**generate_char_indices**(*meta: DataFrame*, *kfold: int = 6*, *random_state: int | RandomState | None = None*)

Generate the trail index of train set, validation set and test set. This method directly manipulate characters

author: WuJieYu

Created on: 2023-03-17

update log:2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **meta** (*DataFrame*) – meta of all trials.
- **kfold** (*int*) – Number of folds for cross validation.

- **random_state** (*Optional[Union[int, RandomState]]*) – State of random, default: None.

Returns
indices – Trial index for train set, validation set and test set. Ensemble in a tuple.

Return type
list

metabci.brainda.algorithms.utils.model_selection.**generate_kfold_indices**(*meta: DataFrame*, *kfold: int = 5*, *random_state: int | RandomState | None = None*)

The EnhancedStratifiedKFold class is invoked at the meta data structure level to generate cross-validation grouping subscripts. The subscript of K-fold cross-validation is generated based on meta class data structure.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **meta** (*pandas.DataFrame*) – metaBCI's custom data class.
- **kfold** (*int*) – Cross validation fold, default is 5.
- **random_state** (*int numpy.random.RandomState*) – Random initial state, defaults to None.

Returns
indices – The index subscript of the double-nested dictionary structure, the key of the outer dictionary is "subject name", the corresponding value classes_indices is dict format, and the content is {' e_name ': k_indices}. The key of the inner dictionary is the event class name and the value is the attempt index subscript k_indices for K-fold cross-validation. The variable is a list, and the internal elements are tuples (ix_train, ix_val, ix_test) composed of the indexes of the corresponding data sets.

Return type
dict, {'subject id': classes_indices}

metabci.brainda.algorithms.utils.model_selection.**generate_loo_indices**(*meta: DataFrame*)

The EnhancedLeaveOneGroupOut class is invoked at the meta data structure level to generate cross-validation grouping subscripts. The subscript of leave-one method cross-validation is generated based on meta class data structure.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

Parameters
meta (*pandas.DataFrame*) – metaBCI's custom data class.

Returns

indices – The index subscript of the double-nested dictionary structure, the key of the outer dictionary is "subject name", the corresponding value classes_indices is dict format, and the content is {' e_name ': k_indices}. The key of the inner dictionary is the event class name and the value is the attempt index subscript k_indices for K-fold cross-validation. The variable is a list, and the internal elements are tuples (ix_train, ix_val, ix_test) composed of the indexes of the corresponding data sets.

Return type

dict, {'subject id': classes_indices}

metabci.brainda.algorithms.utils.model_selection.**generate_shuffle_indices**(*meta: DataFrame*, *n_splits: int = 5*, *test_size: float = 0.1*, *validate_size: float = 0.1*, *train_size: float = 0.8*, *random_state: int | RandomState | None = None*)

Level in the meta data structure called EnhancedStratifiedShuffleSplit class, generating cross validation grouping subscript. Generate hierarchical random cross-validation subscripts based on meta-class data structures.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:

2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **meta** (*pandas.DataFrame*) – metaBCI's custom data class.
- **n_splits** (*int*) – Random verification fold, default is 5.
- **test_size** (*float*) – The default value is 0.1.
- **validate_size** (*int*) – The default value is 0.1, which is the same as that of the test set.
- **train_size** (*int*) – The proportion of the number of training sets is 0.8 by default (the sum of the proportion of test sets and verification sets is 1).
- **random_state** (*int numpy.random.RandomState*) – Random initial state, defaults to None.

Returns

indices – The index subscript of the double-nested dictionary structure, the key of the outer dictionary is "subject name", the corresponding value classes_indices is dict format, and the content is {' e_name ': k_indices}. The key of the inner dictionary is the event class name and the value is the attempt index subscript k_indices for K-fold cross-validation. The variable is a list, and the internal elements are tuples (ix_train, ix_val, ix_test) composed of the indexes of the corresponding data sets.

Return type

dict, {'subject id': classes_indices}

metabci.brainda.algorithms.utils.model_selection.**match_char_kfold_indices**(*k: int*, *meta: DataFrame*, *indices*)

Divide train set, validation set and test set. This method directly manipulate characters

author: WuJieYu

Created on: 2023-03-17

update log:2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **k** (*int*) – Number of folds for cross validation.
- **meta** (*DataFrame*) – meta of all trials.
- **indices** (*list*) – indices of trial index.

Returns
train_ix, val_ix, test_ix – trial index for train set, validation set and test set.

Return type
list

metabci.brainda.algorithms.utils.model_selection.**match_kfold_indices**(*k: int*, *meta: DataFrame*, *indices*)

At the level of meta data structure, hierarchical K-fold cross-validation packet subscripts are matched to generate specific indexes. Based on meta class data structure and combined with the output results of generate_kfold_indices(), the specific index is generated.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **k** (*int*) – Cross-verify the index of folds.
- **meta** (*pandas.DataFrame*) – metaBCI's custom data class.
- **indices** (*dict, {'subject id': classes_indices}*) – Subscript dictionary generated by generate_kfold_indices().

Returns

- **train_ix** (*ndarray, 'subject id': classes_indices*) – The index of the training set trials required for k-fold verification of the full class data of all subjects (i.e., meta-class data).
- **val_ix** (*ndarray, 'subject id': classes_indices*) – The validation set trial index required for validation of the meta-class data at k-fold validation.
- **test_ix** (*ndarray, 'subject id': classes_indices*) – The test set trial index required for validation of the meta-class data at the k-fold.

metabci.brainda.algorithms.utils.model_selection.**match_loo_indices**(*k: int*, *meta: DataFrame*, *indices*)

At the meta data structure level, a method is matched to cross-validate the grouping subscript and generate the specific index. Based on the meta class data structure and combined with the output of generate_loo_indices(), the specific index is generated.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **k** (*int*) – Cross-verify the index of folds.
- **meta** (*pandas.DataFrame*) – metaBCI's custom data class.
- **indices** (*dict, {'subject id': classes_indices}*) – Subscript dictionary generated by generate_loo_indices().

Returns

- **train_ix** (*ndarray, 'subject id': classes_indices*) – The index of the training set trial required by the k-fold verification of meta class data.
- **val_ix** (*ndarray, 'subject id': classes_indices*) – The validation set trial index required for validation of the meta-class data at k-fold validation.
- **test_ix** (*ndarray, 'subject id': classes_indices*) – The test set trial index required for validation of the meta-class data at the k-fold.

metabci.brainda.algorithms.utils.model_selection.**match_loo_indices_dict**(*X: Dict*, *y: Dict*, *meta: DataFrame*, *indices*, *k: int*)

metabci.brainda.algorithms.utils.model_selection.**match_shuffle_indices**(*k: int*, *meta: DataFrame*, *indices*)

Random cross-validation grouping subscripts are matched at the meta data structure level to generate specific indexes. Based on the meta class data structure and combined with the output of generate_shuffle_indices(), a specific index is generated.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:
2023-12-26 by sunchang<18822197631@163.com>

Parameters

- **k** (*int*) – Cross-verify the index of folds.
- **meta** (*pandas.DataFrame*) – metaBCI's custom data class.
- **indices** (*dict, {'subject id': classes_indices}*) – A subscript dictionary generated by generate_shuffle_indices().

Returns

- **train_ix** (*ndarray, 'subject id': classes_indices*) – The index of the training set trial required by the k-fold verification of meta class data.
- **val_ix** (*ndarray, 'subject id': classes_indices*) – The validation set trial index required for validation of the meta-class data at k-fold validation.
- **test_ix** (*ndarray, 'subject id': classes_indices*) – The test set trial index required for validation of the meta-class data at the k-fold.

metabci.brainda.algorithms.utils.model_selection.**set_random_seeds**(*seed: int*)

Set seeds for python random module numpy.random and torch.

author:Swolf <swolfforever@gmail.com>

Created on:2021-11-29

update log:

2023-12-26 by sunchang<18822197631@163.com>

Parameters

seed (*int*) – Random seed.

Module contents

Module contents

metabci.brainda.datasets package

Submodules

metabci.brainda.datasets.alex_mi module

Alex Motor imagery dataset.

class metabci.brainda.datasets.alex_mi.**AlexMI**

Bases: *BaseDataset*

Alex Motor Imagery dataset.

Motor imagery dataset from the PhD dissertation of A. Barachant[1].

This Dataset contains EEG recordings from 8 subjects, performing 2 task of motor imagination (right hand, feet or rest). Data have been recorded at 512Hz with 16 wet electrodes (Fpz, F7, F3, Fz, F4, F8, T7, C3, Cz, C4, T8, P7, P3, Pz, P4, P8) with a g.tec g.USBamp EEG amplifier.

File are provided in MNE raw file format. A stimulation channel encoding the timing of the motor imagination. The start of a trial is encoded as 1, then the actual start of the motor imagination is encoded with 2 for imagination of a right hand movement, 3 for imagination of both feet movement and 4 with a rest trial.

The duration of each trial is 3 second. There is 20 trial of each class.

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id

[1] Barachant, A., 2012. Commande robuste d'un effecteur par une interface cerveau machine EEG asynchrone (Doctoral dissertation, Université de Grenoble). https://tel.archives-ouvertes.fr/tel-01196752

- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.base module

Basic elements to describe a BCI dataset.

Modified from https://github.com/NeuroTechX/moabb

class metabci.brainda.datasets.base.**BaseDataset**(*dataset_code: str*, *subjects: List[str | int]*, *events: Dict[str, Tuple[int | str, Tuple[float, float]]]*, *channels: List[str]*, *srate: float | int*, *paradigm: str*)

Bases: object

BaseDataset for all datasets.

abstract data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed

- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

download_all(*path: str | Path | None = None*, *force_update: bool = False*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*)

Download all files.

Parameters

- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

get_data(*subjects: List[str | int]*, *verbose: bool | str | int | None = None*) → Dict[int | str, Dict[str, Dict[str, Raw]]]

Get raw data.

Parameters
subjects (*List[Union[int, str]]*) – subjects whose data should be returned

Returns

returned raw ata, structured as {

subject_id: {'sessio_id': {'run_id': Raw}}

}

Return type
Dict[Union[int, str], Dict[str, Dict[str, Raw]]]

Raises
ValueError – raise error if a subject is not valid

class metabci.brainda.datasets.base.**BaseTimeEncodingDataset**(*dataset_code: str*, *subjects: List[str | int]*, *events: Dict[str, Tuple[int | str, Tuple[float, float]]]*, *channels: List[str]*, *srate: float | int*, *paradigm: str*, *minor_events: Dict[str, Tuple[int | str, Tuple[float, float]]]*, *encode: Dict[str, List[str | int]]*, *encode_loop: int*)

Bases: *BaseDataset*

abstract data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.bids module

class metabci.brainda.datasets.bids.**matchingpennies**

Bases: *BaseDataset*

An example BIDS format dataset. This dataset is an standard example of a BIDS format dataset, that mentioned in [1], and now it can be downloaded from [2]. However, as the suggestion in [3], we download the dataset from BASE_URL instead. The source reference of this dataset is [4].

This is the "Matching Pennies" dataset. It was collected as part of a small scale replication project targeting the following reference [5]

In brief, it contains EEG data for 7 subjects raising either their left or right hand, thus giving rise to a lateralized readiness potential as measured with the EEG. For details, see the Details about the experiment section.

References: [1] Pernet, C.R., Appelhoff, S., Gorgolewski, K.J. et al.

EEG-BIDS, an extension to the brain imaging data structure for electroencephalography. Sci Data 6, 103 (2019). https://doi.org/10.1038/s41597-019-0104-8

[2] https://gin.g-node.org/sappelhoff/eeg_matchingpennies [3] https://github.com/mne-tools/mne-bids-pipeline/blob/main/mne_bids_pipeline/tests/datasets.py [4] Appelhoff, S., Sauer, D. & Gill, S. S. Matching Pennies:

A Brain Computer Interface Implementation Dataset. Open Science Framework, https://doi.org/10.17605/OSF.IO/CJ2DR (2018).

[5] Matthias Schultze-Kraft et al. "Predicting Motor Intentions with Closed-Loop Brain-Computer Interfaces". In: Springer Briefs in Electrical and Computer Engineering. Springer International Publishing, 2017, pp. 79~90.

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns

local path of a subject data, the first list is session and the second list is run

Return type

List[List[Union[str, Path]]]

metabci.brainda.datasets.bnci module

Brain/Neuro Computer Interface (BNCI) datasets.

class metabci.brainda.datasets.bnci.**BNCI2014001**

Bases: *BaseDataset*

BNCI 2014-001 Motor Imagery dataset.

Dataset IIa from BCI Competition 4 [1]_.

Dataset Description

This data set consists of EEG data from 9 subjects. The cue-based BCI paradigm consisted of four different motor imagery tasks, namely the imag- ination of movement of the left hand (class 1), right hand (class 2), both feet (class 3), and tongue (class 4). Two sessions on different days were recorded for each subject. Each session is comprised of 6 runs separated by short breaks. One run consists of 48 trials (12 for each of the four possible classes), yielding a total of 288 trials per session.

The subjects were sitting in a comfortable armchair in front of a computer screen. At the beginning of a trial (t = 0 s), a fixation cross appeared on the black screen. In addition, a short acoustic warning tone was presented. After two seconds (t = 2 s), a cue in the form of an arrow pointing either to the left, right, down or up (corresponding to one of the four classes left hand, right hand, foot or tongue) appeared and stayed on the screen for 1.25 s. This

prompted the subjects to perform the desired motor imagery task. No feedback was provided. The subjects were ask to carry out the motor imagery task until the fixation cross disappeared from the screen at t = 6 s.

Twenty-two Ag/AgCl electrodes (with inter-electrode distances of 3.5 cm) were used to record the EEG; the montage is shown in Figure 3 left. All signals were recorded monopolarly with the left mastoid serving as reference and the right mastoid as ground. The signals were sampled with. 250 Hz and bandpass-filtered between 0.5 Hz and 100 Hz. The sensitivity of the amplifier was set to 100 V . An additional 50 Hz notch filter was enabled to suppress line noise

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter `MNE_DATASETS_(dataset_code)_PATH` is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

class metabci.brainda.datasets.bnci.**BNCI2014004**

Bases: *BaseDataset*

BNCI 2014-004 Motor Imagery dataset.

Dataset B from BCI Competition 2008.

Dataset description

This data set consists of EEG data from 9 subjects of a study published in [1]_. The subjects were right-handed, had normal or corrected-to-normal vision and were paid for participating in the experiments. All volunteers were sitting in an armchair, watching a flat screen monitor placed approximately 1 m away at eye level. For each subject 5 sessions are provided, whereby the first two sessions contain training data without feedback (screening), and the last three sessions were recorded with feedback.

Three bipolar recordings (C3, Cz, and C4) were recorded with a sampling frequency of 250 Hz.They were bandpass- filtered between 0.5 Hz and 100 Hz, and a notch filter at 50 Hz was enabled. The placement of the

three bipolar recordings (large or small distances, more anterior or posterior) were slightly different for each subject (for more details see [1]). The electrode position Fz served as EEG ground. In addition to the EEG channels, the electrooculogram (EOG) was recorded with three monopolar electrodes.

The cue-based screening paradigm consisted of two classes, namely the motor imagery (MI) of left hand (class 1) and right hand (class 2). Each subject participated in two screening sessions without feedback recorded on two different days within two weeks. Each session consisted of six runs with ten trials each and two classes of imagery. This resulted in 20 trials per run and 120 trials per session. Data of 120 repetitions of each MI class were available for each person in total. Prior to the first motor im- agery training the subject executed and imagined different movements for each body part and selected the one which they could imagine best (e. g., squeezing a ball or pulling a brake).

Each trial started with a fixation cross and an additional short acoustic warning tone (1 kHz, 70 ms). Some seconds later a visual cue was presented for 1.25 seconds. Afterwards the subjects had to imagine the corresponding hand movement over a period of 4 seconds. Each trial was followed by a short break of at least 1.5 seconds. A randomized time of up to 1 second was added to the break to avoid adaptation

For the three online feedback sessions four runs with smiley feedback were recorded, whereby each run consisted of twenty trials for each type of motor imagery. At the beginning of each trial (second 0) the feedback (a gray smiley) was centered on the screen. At second 2, a short warning beep (1 kHz, 70 ms) was given. The cue was presented from second 3 to 7.5. At second 7.5 the screen went blank and a random interval between 1.0 and 2.0 seconds was added to the trial.

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter `MNE_DATASETS_(dataset_code)_PATH` is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns

local path of a subject data, the first list is session and the second list is run

Return type

List[List[Union[str, Path]]]

metabci.brainda.datasets.cattan_P300 module

P300 datasets

class metabci.brainda.datasets.cattan_P300.**Cattan_P300**(*paradigm='p300'*)

Bases: *BaseTimeEncodingDataset*

Dataset in: Grégoire Cattan, Anton Andreev, Pedro Luiz Coelho Rodrigues and Marco Congedo, "Dataset of an EEG-based BCI experiment in Virtual Reality and on a Personal Computer," in arXiv.1903.11297.1903.11297, 2019,

This dataset contains electroencephalographic recordings on 21 subjects doing a visual P300 experiment on PC (personal computer) and VR (virtual reality). The visual P300 is an event-related potential elicited by a visual stimulation, peaking 240-600 ms after stimulus onset. The experiment was designed in order to compare the use of a P300-based brain-computer interface on a PC and with a virtual reality headset, concerning the physiological, subjective and performance aspects. The brain-computer interface is based on electroencephalography (EEG). EEG data were recorded thanks to 16 electrodes. The virtual reality headset consisted of a passive head-mounted display, that is, a head-mounted display which does not include any electronics with the exception of a smartphone. A full description of the experiment is available at https://hal.archives-ouvertes.fr/hal-02078533. This experiment was carried out at GIPSA-lab (University of Grenoble Alpes, CNRS, Grenoble-INP) in 2018, and promoted by the IHMTEK Company (Interaction Homme-Machine Technologie). The study was approved by the Ethical Committee of the University of Grenoble Alpes (Comité d'Ethique pour la Recherche Non-Interventionnelle). The ID of this dataset is VR.EEG.2018-GIPSA.

This study implemented a P300 interface. Thirty-six characters were arranged as a 6 × 6 matrix displayed on the screen. The task of the subject was to focus on one of the characters.The experiment was composed of two sessions. One session ran under the PC condition and the other under the VR condition. The order of the session was randomized for all subjects. Each session comprised 12 blocks of five repetitions. All the repetitions within a block have the same target. A repetition consisted of 12 flashes of groups of six symbols chosen in such a way that after each repetition each symbol has flashed exactly two times. Thus, in each repetition the target symbol flashes twice, whereas the remaining ten flashes do not concern the target (non-target). The EEG signal was tagged corresponding to each flash.

The recorded signals were bandpass-filtered at 0.1–100 Hz, notch-filtered at 50 Hz, digitized at a rate of 500 Hz and then stored in a computer. In this script, data is downscaled to 100 Hz.

code_len = 12

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*)

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None

- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.cbcic module

China BCI Competition.

class metabci.brainda.datasets.cbcic.**CBCIC2019001**

Bases: *BaseDataset*

2019 China BCI competition Dataset for MI in preliminary contest A/B.

Motor imagery dataset from China BCI competition in 2019.

This dataset contains EEG recordings from 18 subjects, performing 2 or 3 tasks of motor imagery (left hand, right hand or feet). Data have been recored at 1000hz with 64 electrodes (59 in use except ECG, HEOR, HEOL, VEOU, VEOL channels) by an neuracle EEG amplifier.

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

class metabci.brainda.datasets.cbcic.**CBCIC2019004**

Bases: *BaseDataset*

2019 China BCI competition Dataset for MI in final competition.

Motor imagery dataset from China BCI competition in 2019.

This dataset contains EEG recordings from 18 subjects, performing 2 or 3 tasks of motor imagery (left hand, right hand or feet). Data have been recored at 1000hz with 64 electrodes (59 in use except ECG, HEOR, HEOL, VEOU, VEOL channels) by an neuracle EEG amplifier.

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns

local path of a subject data, the first list is session and the second list is run

Return type

List[List[Union[str, Path]]]

class metabci.brainda.datasets.cbcic.**XuaVEPDataset**(*paradigm='aVEP'*)

Bases: *BaseTimeEncodingDataset*

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*)

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None

- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.cho2017 module

GigaDb Motor imagery dataset.

class metabci.brainda.datasets.cho2017.**Cho2017**

Bases: *BaseDataset*

Motor Imagery dataset from Cho et al 2017.

Dataset from the paper[1].

Dataset Description

We conducted a BCI experiment for motor imagery movement (MI movement) of the left and right hands with 52 subjects (19 females, mean age ± SD age = 24.8 ± 3.86 years); Each subject took part in the same experiment, and subject ID was denoted and indexed as s1, s2, . . . , s52. Subjects s20 and s33 were both-handed, and the other 50 subjects were right-handed.

EEG data were collected using 64 Ag/AgCl active electrodes. A 64-channel montage based on the international 10-10 system was used to record the EEG signals with 512 Hz sampling rates. The EEG device used in this experiment was the Biosemi ActiveTwo system. The BCI2000 system 3.0.2 was used to collect EEG data and present instructions (left hand or right hand MI). Furthermore, we recorded EMG as well as EEG simultaneously with the same system and sampling rate to check actual hand movements. Two EMG electrodes were attached to the flexor digitorum profundus and extensor digitorum on each arm.

Subjects were asked to imagine the hand movement depending on the instruction given. Five or six runs were performed during the MI experiment. After each run, we calculated the classification accuracy over one run and gave the subject feedback to increase motivation. Between each run, a maximum 4-minute break was given depending on the subject's demands.

[1] Cho, H., Ahn, M., Ahn, S., Kwon, M. and Jun, S.C., 2017. EEG datasets for motor imagery brain computer interface. GigaScience. https://doi.org/10.1093/gigascience/gix034

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.munich2009 module

Munich MI dataset. Unkown channel names.

class metabci.brainda.datasets.munich2009.**MunichMI**

Bases: *BaseDataset*

Munich Motor Imagery dataset.

Motor imagery dataset from Grosse-Wentrup et al. 2009[1].

A trial started with the central display of a white fixation cross. After 3 s, a white arrow was superimposed on the fixation cross, either pointing to the left or the right. Subjects were instructed to perform haptic motor imagery of the left or the right hand during display of the arrow, as indicated by the direction of the arrow. After another 7 s, the arrow was removed, indicating the end of the trial and start of the next trial. While subjects were explicitly instructed to perform haptic motor imagery with the specified hand, i.e., to imagine feeling instead of visualizing how their hands moved, the exact choice of which type of imaginary movement, i.e., moving the fingers up and down, gripping an object, etc., was left unspecified. A total of 150 trials per condition were carried out by each subject, with trials presented in pseudorandomized order.

Ten healthy subjects (S1–S10) participated in the experimental evaluation. Of these, two were females, eight were right handed, and their average age was 25.6 years with a standard deviation of 2.5 years. Subject S3 had already

[1] Grosse-Wentrup, Moritz, et al. "Beamforming in noninvasive brain–computer interfaces." IEEE Transactions on Biomedical Engineering 56.4 (2009): 1209-1219.

participated twice in a BCI experiment, while all other subjects were naive to BCIs. EEG was recorded at M=128 electrodes placed according to the extended 10–20 system. Data were recorded at 500 Hz with electrode Cz as reference. Four BrainAmp amplifiers were used for this purpose, using a temporal analog high-pass filter with a time constant of 10 s. The data were re-referenced to common average reference offline. Electrode impedances were below 10 k for all electrodes and subjects at the beginning of each recording session. No trials were rejected and no artifact correction was performed. For each subject, the locations of the 128 electrodes were measured in three dimensions using a Zebris ultrasound tracking system and stored for further offline analysis.

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter `MNE_DATASETS_(dataset_code)_PATH` is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.nakanishi2015 module

Nakanishi SSVEP dataset.

class metabci.brainda.datasets.nakanishi2015.**Nakanishi2015**

Bases: *BaseDataset*

SSVEP Nakanishi 2015 dataset

This dataset contains 12-class joint frequency-phase modulated steady-state visual evoked potentials (SSVEPs) acquired from 10 subjects used to estimate an online performance of brain-computer interface (BCI) in the reference study[1].

[1] Masaki Nakanishi, Yijun Wang, Yu-Te Wang and Tzyy-Ping Jung,

References

"A Comparison Study of Canonical Correlation Analysis Based Methods for Detecting Steady-State Visual Evoked Potentials," PLoS One, vol.10, no.10, e140703, 2015. http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0140703

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

get_freq(*event: str*)

get_phase(*event: str*)

metabci.brainda.datasets.physionet module

Physionet MI.

class metabci.brainda.datasets.physionet.**BasePhysionet**(*paradigm: str*, *is_imagined: bool = True*)

Bases: *BaseDataset*

Physionet Motor Imagery dataset.

Physionet MI dataset: https://physionet.org/pn4/eegmmidb/

This data set consists of over 1500 one- and two-minute EEG recordings, obtained from 109 volunteers.

Subjects performed different motor/imagery tasks while 64-channel EEG were recorded using the BCI2000 system (http://www.bci2000.org). Each subject performed 14 experimental runs: two one-minute baseline runs (one with eyes open, one with eyes closed), and three two-minute runs of each of the four following tasks:

1. A target appears on either the left or the right side of the screen. The subject opens and closes the corresponding fist until the target disappears. Then the subject relaxes.
2. A target appears on either the left or the right side of the screen. The subject imagines opening and closing the corresponding fist until the target disappears. Then the subject relaxes.
3. A target appears on either the top or the bottom of the screen. The subject opens and closes either both fists (if the target is on top) or both feet (if the target is on the bottom) until the target disappears. Then the subject relaxes.
4. A target appears on either the top or the bottom of the screen. The subject imagines opening and closing either both fists (if the target is on top) or both feet (if the target is on the bottom) until the target disappears. Then the subject relaxes.

Parameters

- **imagined** (*bool (default True)*) – if True, return runs corresponding to motor imagination.
- **executed** (*bool (default False)*) – if True, return runs corresponding to motor execution.

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns

local path of a subject data, the first list is session and the second list is run

Return type

List[List[Union[str, Path]]]

raw_hook(*raw: Raw*, *caches: dict*, *verbose=None*)

class metabci.brainda.datasets.physionet.**PhysionetME**

Bases: *BasePhysionet*

class metabci.brainda.datasets.physionet.**PhysionetMI**

Bases: *BasePhysionet*

metabci.brainda.datasets.schirrmeister2017 module

High-gamma dataset.

class metabci.brainda.datasets.schirrmeister2017.**BBCIDataset**(*filename*, *load_sensor_names=None*)

Bases: object

Loader class for files created by saving BBCI files in matlab (make sure to save with '-v7.3' in matlab, see https://de.mathworks.com/help/matlab/import_export/mat-file-versions.html#buk6i87) :param filename: :type filename: str :param load_sensor_names: Also speeds up loading if you only load some sensors.

None means load all sensors.

Parameters

- **2017** (*Copyright Robin Schirrmeister,*) –
- **2018** (*Altered by Vinay Jayaram,*) –

static **get_all_sensors**(*filename*, *pattern=None*)

Get all sensors that exist in the given file.

Parameters

- **filename** (*str*) –
- **pattern** (*str, optional*) – Only return those sensor names that match the given pattern.

Returns

sensor_names – Sensor names that match the pattern or all sensor names in the file.

Return type

list of str

load()

class metabci.brainda.datasets.schirrmeister2017.**Schirrmeister2017**

Bases: *BaseDataset*

High-gamma dataset discribed in Schirrmeister et al. 2017

Our "High-Gamma Dataset" is a 128-electrode dataset (of which we later only use 44 sensors covering the motor cortex, (see Section 2.7.1), obtained from 14 healthy subjects (6 female, 2 left-handed, age 27.2 ± 3.6 (mean ± std)) with roughly 1000 (963.1 ± 150.9, mean ± std) four-second trials of executed movements divided into 13 runs per subject. The four classes of movements were movements of either the left hand, the right hand, both feet, and rest (no movement, but same type of visual cue as for the other classes). The training set consists of the approx. 880 trials of all runs except the last two runs, the test set of the approx. 160 trials of the last 2 runs. This dataset was acquired in an EEG lab optimized for non-invasive detection of high- frequency movement-related EEG components (Ball et al., 2008; Darvas et al., 2010).

Depending on the direction of a gray arrow that was shown on black back- ground, the subjects had to repetitively clench their toes (downward arrow), perform sequential finger-tapping of their left (leftward arrow) or

right (rightward arrow) hand, or relax (upward arrow). The movements were selected to require little proximal muscular activity while still being complex enough to keep subjects in- volved. Within the 4-s trials, the subjects performed the repetitive movements at their own pace, which had to be maintained as long as the arrow was showing. Per run, 80 arrows were displayed for 4 s each, with 3 to 4 s of continuous random inter-trial interval. The order of presentation was pseudo-randomized, with all four arrows being shown every four trials. Ideally 13 runs were performed to collect 260 trials of each movement and rest. The stimuli were presented and the data recorded with BCI2000 (Schalk et al., 2004). The experiment was approved by the ethical committee of the University of Freiburg.

References

neural networks for EEG decoding and visualization." Human brain mapping 38.11 (2017): 5391-5420.

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter `MNE_DATASETS_(dataset_code)_PATH` is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.tsinghua module

Tsinghua BCI Lab.

class metabci.brainda.datasets.tsinghua.**BETA**

Bases: *BaseDataset*

BETA SSVEP dataset[1].

EEG data after preprocessing are store as a 4-way tensor, with a dimension of channel x time point x block x condition. Each trial comprises 0.5-s data before the event onset and 0.5-s data after the time window of 2 s or

[1] Liu B, Huang X, Wang Y, et al. BETA: A Large Benchmark Database

3 s. For S1-S15, the time window is 2 s and the trial length is 3 s, whereas for S16-S70 the time window is 3 s and the trial length is 4 s. Additional details about the channel and condition information can be found in the following supplementary information.

Eight supplementary information is comprised of personal information, channel information, frequency and initial phase associated to each condition, SNR and sampling rate. The personal information contains age and gender of the subject. For the channel information, a location matrix (64 x 4) is provided, with the first column indicating channel index, the second column and third column indicating the degree and radius in polar coordinates, and the last column indicating channel name. The SNR information contains the mean narrow-band SNR and wide-band SNR matrix for each subject, calculated in (3) and (4), respectively. The initial phase is in radius.

3-100Hz bandpass filtering (eegfilt), downsampled to 250 Hz

References

Toward SSVEP-BCI Application[J]. Frontiers in neuroscience, 2020, 14: 627.

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

get_freq(*event: str*)

get_phase(*event: str*)

class metabci.brainda.datasets.tsinghua.**Wang2016**

Bases: *BaseDataset*

SSVEP dataset from Yijun Wang.

This dataset gathered SSVEP-BCI recordings of 35 healthy subjects (17 females, aged 17-34 years, mean age: 22 years) focusing on 40 characters flickering at different frequencies (8-15.8 Hz with an interval of 0.2 Hz).

For each subject, the experiment consisted of 6 blocks. Each block contained 40 trials corresponding to all 40 characters indicated in a random order. Each trial started with a visual cue (a red square) indicating a target stimulus. The cue appeared for 0.5 s on the screen. Subjects were asked to shift their gaze to the target as soon as possible within the cue duration. Following the cue offset, all stimuli started to flicker on the screen concurrently and lasted 5 s. After stimulus offset, the screen was blank for 0.5 s before the next trial began, which allowed the subjects to have short breaks between consecutive trials. Each trial lasted a total of 6 s. To facilitate visual fixation, a red triangle appeared below the flickering target during the stimulation period. In each block, subjects were asked to avoid eye blinks during the stimulation period. To avoid visual fatigue, there was a rest for several minutes between two consecutive blocks. EEG data were acquired using a Synamps2 system (Neuroscan, Inc.) with a sampling rate of 1000 Hz. The amplifier frequency passband ranged from 0.15 Hz to 200 Hz. Sixty-four channels covered the whole scalp of the subject and were aligned according to the international 10-20 system. The ground was placed on midway between Fz and FPz. The reference was located on the vertex. Electrode impedances were kept below 10 K. To remove the common power-line noise, a notch filter at 50 Hz was applied in data recording. Event triggers generated by the computer to the amplifier and recorded on an event channel synchronized to the EEG data.

The continuous EEG data was segmented into 6 s epochs (500 ms pre-stimulus, 5.5 s post-stimulus onset). The epochs were subsequently downsampled to 250 Hz. Thus each trial consisted of 1500 time points. Finally, these data were stored as double-precision floating-point values in MATLAB and were named as subject indices (i.e., S01.mat, …, S35.mat). For each file, the data loaded in MATLAB generate a 4-D matrix named 'data' with dimensions of [64, 1500, 40, 6]. The four dimensions indicate 'Electrode index', 'Time points', 'Target index', and 'Block index'. The electrode positions were saved in a '64-channels.loc' file. Six trials were available for each SSVEP frequency. Frequency and phase values for the 40 target indices were saved in a 'Freq_Phase.mat' file.

Information for all subjects was listed in a 'Sub_info.txt' file. For each subject, there are five factors including 'Subject Index', 'Gender', 'Age' 'Handedness', and 'Group'. Subjects were divided into an 'experienced' group (eight subjects, S01-S08) and a 'naive' group (27 subjects, S09-S35) according to their experience in SSVEP-based BCIs.

Frequency Table 8 9 10 11 12 13 14 15 8.2 9.2 10.2 11.2 12.2 13.2 14.2 15.2 8.4 9.4 10.4 11.4 12.4 13.4 14.4 15.4 8.6 9.6 10.6 11.6 12.6 13.6 14.6 15.6 8.8 9.8 10.8 11.8 12.8 13.8 14.8 15.8

Notes

1. sub5 is not available from the download url.

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter `MNE_DATASETS_(dataset_code)_PATH` is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False

- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

get_freq(*event: str*)

get_phase(*event: str*)

metabci.brainda.datasets.tunerl module

TUNERL Datasets

Weibo2014

class metabci.brainda.datasets.tunerl.**Weibo2014**

Bases: *BaseDataset*

Motor Imagery dataset from Weibo et al 2014.

Dataset from the article *Evaluation of EEG oscillatory patterns and cognitive process during simple and compound limb motor imagery*[1].

It contains data recorded on 10 subjects, with 60 electrodes.

This dataset was used to investigate the differences of the EEG patterns between simple limb motor imagery and compound limb motor imagery. Seven kinds of mental tasks have been designed, involving three tasks of simple limb motor imagery (left hand, right hand, feet), three tasks of compound limb motor imagery combining hand with hand/foot (both hands, left hand combined with right foot, right hand combined with left foot) and rest state.

At the beginning of each trial (8 seconds), a white circle appeared at the center of the monitor. After 2 seconds, a red circle (preparation cue) appeared for 1 second to remind the subjects of paying attention to the character indication next. Then red circle disappeared and character indication ('Left Hand', 'Left Hand & Right Foot', et al) was presented on the screen for 4 seconds, during which the participants were asked to perform kinesthetic motor imagery rather than a visual type of imagery while avoiding any muscle movement. After 7 seconds, 'Rest' was presented for 1 second before next trial (Fig. 1(a)). The experiments were divided into 9 sections, involving 8 sections consisting of 60 trials each for six kinds of MI tasks (10 trials for each MI task in one section) and one section consisting of 80 trials for rest state. The sequence of six MI tasks was randomized. Intersection break was about 5 to 10 minutes.

[1] Yi, Weibo, et al. "Evaluation of EEG oscillatory patterns and cognitive process during simple and compound limb motor imagery." PloS one 9.12 (2014). https://doi.org/10.1371/journal.pone.0114853

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns

local path of a subject data, the first list is session and the second list is run

Return type

List[List[Union[str, Path]]]

metabci.brainda.datasets.xu2018_minavep module

aVEP datasets

class metabci.brainda.datasets.xu2018_minavep.**Xu2018MinaVep**(*paradigm='aVEP'*)

Bases: *BaseTimeEncodingDataset*

Dataset in: M. Xu, X. Xiao, Y. Wang, H. Qi, T. -P. Jung and D. Ming, "A Brain–Computer Interface Based on Miniature-Event-Related Potentials Induced by Very Small Lateral Visual Stimuli," in IEEE Transactions on Biomedical Engineering, vol. 65, no. 5, pp. 1166-1175, May 2018, doi: 10.1109/TBME.2018.2799661.

This study implemented a miniature aVEP-based BCI speller, and proposed a new scheme for BCI encoding. Thirty-two alphanumeric characters were arranged as a 4 × 8 matrix displayed on a computer screen and encoded by a new SCDMA scheme, in which the left and right lateral visual stimuli constituted two parallel spatial channels while two different lateral visual stimuli sequences made up the basic communication codes '0' and '1'. Specifically, the 'left-right' stimulus sequence, which lasted 200 ms, was regarded as code '0', while 'right-left' stimulus was coded '1'. Thirty-two different code sequences were created using 5 bits in this study, which were arbitrarily allocated to different characters. Specifically, character A' was encoded by '01100'. In spelling, the lateral visual stimuli would be presented simultaneously for all characters with different sequences. To obtain a reliable output, the same code sequence was repeated 6 times for the offline spelling and individually optimized times for the online spelling. The character specified to output in the offline spelling would be indicated by a star-shaped cue underneath for 0.8 seconds, which would be offset for another 0.2 seconds to wipe out the cue effect. There was a time interval of 0.2 seconds with no stimulation between two successive sequences.

EEG was recorded using a Neuroscan Synamps2 system with 64 electrodes located in the positions following the 10/20 system. The reference electrode was put in the central area near Cz and the ground electrode was put on the frontal lobe. The recorded signals were bandpass-filtered at 0.1–100 Hz, notch-filtered at 50 Hz, digitized at a rate of 1000 Hz and then stored in a computer.

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*)

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

metabci.brainda.datasets.zhou2016 module

Zhou2016.

class metabci.brainda.datasets.zhou2016.**Zhou2016**

Bases: *BaseDataset*

Motor Imagery dataset from Zhou et al 2016.

Dataset from the article *A Fully Automated Trial Selection Method for Optimization of Motor Imagery Based Brain-Computer Interface*[1]. This dataset contains data recorded on 4 subjects performing 3 type of motor imagery: left hand, right hand and feet.

Every subject went through three sessions, each of which contained two consecutive runs with several minutes inter-run breaks, and each run comprised 75 trials (25 trials per class). The intervals between two sessions varied from several days to several months.

A trial started by a short beep indicating 1 s preparation time, and followed by a red arrow pointing randomly to three directions (left, right, or bottom) lasting for 5 s and then presented a black screen for 4 s. The subject was instructed to immediately perform the imagination tasks of the left hand, right hand or foot movement respectively according to the cue direction, and try to relax during the black screen.

[1] Zhou B, Wu X, Lv Z, Zhang L, Guo X (2016) A Fully Automated Trial Selection Method for Optimization of Motor Imagery Based Brain-Computer Interface. PLoS ONE 11(9). https://doi.org/10.1371/journal.pone.0162657

References

data_path(*subject: str | int*, *path: str | Path | None = None*, *force_update: bool = False*, *update_path: bool | None = None*, *proxies: Dict[str, str] | None = None*, *verbose: bool | str | int | None = None*) → List[List[str | Path]]

Get path to local copy of a subject data.

Parameters

- **subject** (*Union[str, int]*) – subject id
- **path** (*Optional[Union[str, Path]], optional*) – Location of where to look for the data storing location. If None, the environment variable or config parameter MNE_DATASETS_(dataset_code)_PATH is used. If it doesn't exist, the "~/mne_data" directory is used. If the dataset is not found under the given path, the data will be automatically downloaded to the specified folder, by default None
- **force_update** (*bool, optional*) – force update of the dataset even if a local copy exists, by default False
- **update_path** (*Optional[bool], optional*) – If True, set the MNE_DATASETS_(dataset)_PATH in mne-python config to the given path. If None, the user is prompted, by default None
- **proxies** (*Optional[Union[bool, str, int]], optional*) – proxies if needed
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of a subject data, the first list is session and the second list is run

Return type
List[List[Union[str, Path]]]

Module contents

metabci.brainda.paradigms package

Submodules

metabci.brainda.paradigms.avep module

aVEP Paradigm.

class metabci.brainda.paradigms.avep.**aVEP**(*channels: List[str] | None = None*, *events: List[str] | None = None*, *intervals: List[Tuple[float, float]] | None = None*, *minor_event_intervals: List[Tuple[float, float]] | None = None*, *srate: float | None = None*)

Bases: *BaseTimeEncodingParadigm*

is_valid(*dataset*)

Verify the dataset is compatible with the paradigm.

This method is called to verify dataset is compatible with the paradigm.

This method should raise an error if the dataset is not compatible with the paradigm. This is for example the case if the dataset is an ERP dataset for motor imagery paradigm, or if the dataset does not contain any of the required events.

Parameters
dataset (BaseDataset) – dataset

metabci.brainda.paradigms.base module

Base Paradigm Design.

class metabci.brainda.paradigms.base.**BaseParadigm**(*channels: List[str] | None = None*, *events: List[str] | None = None*, *intervals: List[Tuple[float, float]] | None = None*, *srate: float | None = None*)

Bases: object

Abstract Base Paradigm.

get_data(*dataset:* BaseDataset, *subjects: List[int | str] = []*, *label_encode: bool = True*, *return_concat: bool = False*, *n_jobs: int = -1*, *verbose: bool | None = None*) → Tuple[Dict[str, ndarray | DataFrame] | ndarray | DataFrame, ...]

Get data from dataset with selected subjects.

Parameters

- **dataset** (BaseDataset) – dataset
- **subjects** (*List[Union[int, str]],*) – selected subjects, by default empty
- **label_encode** (*bool, optional,*) – if True, return y in label encode way
- **return_concat** (*bool, optional*) – if True, return concated ndarray object, otherwise return dict of events, by default False
- **n_jobs** (*int, optional*) – Parallel jobs, by default -1
- **verbose** (*Optional[bool], optional*) – verbose, by default None

Returns
Xs, ys, metas, corresponding to data, label and meta data

Return type
Tuple[Union[Dict[str, Union[np.ndarray, pd.DataFrame]], Union[np.ndarray, pd.DataFrame]], ...]

Raises
TypeError – raise error if dataset is not avaliable for the paradigm

abstract is_valid(*dataset:* BaseDataset) → bool

Verify the dataset is compatible with the paradigm.

This method is called to verify dataset is compatible with the paradigm.

This method should raise an error if the dataset is not compatible with the paradigm. This is for example the case if the dataset is an ERP dataset for motor imagery paradigm, or if the dataset does not contain any of the required events.

Parameters
dataset (BaseDataset) – dataset

register_data_hook(*hook*)

Register data hook before return data.

Parameters

hook (*callable object*) – Callable object to process ndarray data before return it. Its' signature should look like:

hook(X, y, meta, caches) -> X, y, meta, caches

where caches is an dict storing information, X, y are ndarray object, meta is a pandas DataFrame instance.

register_epochs_hook(*hook*)

Register epochs hook after epoch operation.

Parameters

hook (*callable object*) – Callable object to process Epochs object after epoch operation. Its' signature should look like:

hook(epochs, caches) -> epochs, caches

where caches is an dict storing information, epochs is MNE Epochs instance.

register_raw_hook(*hook*)

Register raw hook before epoch operation.

Parameters

hook (*callable object*) – Callable object to process Raw object before epoch operation. Its signature should look like:

hook(raw, caches) -> raw, caches

where caches is an dict stroing information, raw is MNE Raw instance.

unregister_data_hook()

Register data hook before return data.

unregister_epochs_hook()

Register epochs hook after epoch operation.

unregister_raw_hook()

Unregister raw hook before epoch operation.

class metabci.brainda.paradigms.base.**BaseTimeEncodingParadigm**(*channels: List[str] | None = None*, *events: List[str] | None = None*, *intervals: List[Tuple[float, float]] | None = None*, *minor_event_intervals: List[Tuple[float, float]] | None = None*, *srate: float | None = None*)

Bases: *BaseParadigm*

get_data(*dataset:* BaseTimeEncodingDataset, *subjects: List[int | str] = []*, *return_concat: bool = False*, *n_jobs: int = -1*, *verbose: bool | None = None*)

Get data from dataset with selected subjects.

Parameters

- **dataset** (BaseDataset) – dataset
- **subjects** (*List[Union[int, str]],*) – selected subjects, by default empty

- **label_encode** (*bool, optional,*) – if True, return y in label encode way
- **return_concat** (*bool, optional*) – if True, return concated ndarray object, otherwise return dict of events, by default False
- **n_jobs** (*int, optional*) – Parallel jobs, by default -1
- **verbose** (*Optional[bool], optional*) – verbose, by default None

Returns
Xs, ys, metas, corresponding to data, label and meta data

Return type
Tuple[Union[Dict[str, Union[np.ndarray, pd.DataFrame]], Union[np.ndarray, pd.DataFrame]], ...]

Raises
TypeError – raise error if dataset is not avaliable for the paradigm

is_valid(*dataset*)

Verify the dataset is compatible with the paradigm.

This method is called to verify dataset is compatible with the paradigm.

This method should raise an error if the dataset is not compatible with the paradigm. This is for example the case if the dataset is an ERP dataset for motor imagery paradigm, or if the dataset does not contain any of the required events.

Parameters
dataset (BaseDataset) – dataset

register_trial_hook(*hook*)

Register trial hook before trial operation.

Parameters
hook (*callable object to process Raw object before epoch operation.*) – Different from the raw_hook, the trial hook allows you to do some specific operation BEFORE epoch operation (i.e. smallest encode unit) and AFTER raw continuous data operation

Its signature should look like:

hook(raw, caches) -> raw, caches

where caches is a dict storing information, raw is MNE Raw instance

unregister_trial_hook()

metabci.brainda.paradigms.base.**label_encoder**(*y*, *labels*)

metabci.brainda.paradigms.imagery module

Motor Imagery Paradigm.

class metabci.brainda.paradigms.imagery.**MotorImagery**(*channels: List[str] | None = None*, *events: List[str] | None = None*, *intervals: List[Tuple[float, float]] | None = None*, *srate: float | None = None*)

Bases: BaseParadigm

Basic motor imagery paradigm.

is_valid(*dataset*)

Verify the dataset is compatible with the paradigm.

This method is called to verify dataset is compatible with the paradigm.

This method should raise an error if the dataset is not compatible with the paradigm. This is for example the case if the dataset is an ERP dataset for motor imagery paradigm, or if the dataset does not contain any of the required events.

Parameters
dataset (BaseDataset) – dataset

metabci.brainda.paradigms.movement_intention module

Movement intention paradigms.

class metabci.brainda.paradigms.movement_intention.**MovementIntention**(*channels: List[str] | None = None*, *events: List[str] | None = None*, *intervals: List[Tuple[float, float]] | None = None*, *srate: float | None = None*)

Bases: *BaseParadigm*

Basic movement intention paradigm.

is_valid(*dataset*)

Verify the dataset is compatible with the paradigm.

This method is called to verify dataset is compatible with the paradigm.

This method should raise an error if the dataset is not compatible with the paradigm. This is for example the case if the dataset is an ERP dataset for motor imagery paradigm, or if the dataset does not contain any of the required events.

Parameters
dataset (BaseDataset) – dataset

metabci.brainda.paradigms.p300 module

P300 Paradigm.

class metabci.brainda.paradigms.p300.**P300**(*channels: List[str] | None = None*, *events: List[str] | None = None*, *intervals: List[Tuple[float, float]] | None = None*, *minor_event_intervals: List[Tuple[float, float]] | None = None*, *srate: float | None = None*)

Bases: *BaseTimeEncodingParadigm*

is_valid(*dataset*)

Verify the dataset is compatible with the paradigm.

This method is called to verify dataset is compatible with the paradigm.

This method should raise an error if the dataset is not compatible with the paradigm. This is for example the case if the dataset is an ERP dataset for motor imagery paradigm, or if the dataset does not contain any of the required events.

Parameters
dataset (BaseDataset) – dataset

metabci.brainda.paradigms.ssvep module

SSVEP Paradigm.

class metabci.brainda.paradigms.ssvep.**SSVEP**(*channels: List[str] | None = None*, *events: List[str] | None = None*, *intervals: List[Tuple[float, float]] | None = None*, *srate: float | None = None*)

Bases: *BaseParadigm*

is_valid(*dataset*)

Verify the dataset is compatible with the paradigm.

This method is called to verify dataset is compatible with the paradigm.

This method should raise an error if the dataset is not compatible with the paradigm. This is for example the case if the dataset is an ERP dataset for motor imagery paradigm, or if the dataset does not contain any of the required events.

Parameters
dataset (BaseDataset) – dataset

Module contents

metabci.brainda.utils package

Submodules

metabci.brainda.utils.channels module

metabci.brainda.utils.channels.**pick_channels**(*ch_names: List[str]*, *pick_chs: List[str]*, *ordered: bool = True*, *match_case: str | bool = 'auto'*) → List[int]

Wrapper of mne.pick_channels with match_case option.

Parameters

- **ch_names** (*List[str]*) – all channel names
- **pick_chs** (*List[str]*) – channel names to pick
- **ordered** (*bool, optional*) – if Ture, return picked channels in pick_chs order, by default True
- **match_case** (*str, optional*) – if True, pick channels in strict mode, by default 'auto'

Returns
indices of picked channels

Return type
List[int]

metabci.brainda.utils.channels.**upper_ch_names**(*raw: Raw*) → Raw

Uppercase all channel names in MNE Raw object.

Parameters
raw (*Raw*) – MNE Raw object.

Returns
MNE Raw object.

Return type
Raw

metabci.brainda.utils.download module

metabci.brainda.utils.download.**mne_data_path**(*url: str*, *sign: str*, *path: str | Path | None = None*, *proxies: Dict[str, str] | None = None*, *force_update: bool = False*, *update_path: bool = True*, *verbose: bool | str | int | None = None*, ***kwargs*) → str

Get the local path of the target file.

This function returns the local path of the target file, downloading it if needed or requested. The local path keeps the same structure as the url.

Parameters

- **url** (*str*) – url of the target file.
- **sign** (*str*) – the unique identifier to which the file belongs
- **path** (*Optional[Union[str, Path]], optional*) – local folder to save the file, by default None
- **proxies** (*Optional[Dict[str, str]], optional*) – use proxies to download files, e.g. {'https': 'socks5://127.0.0.1:1080'}, by default None
- **force_update** (*bool, optional*) – whether to re-download the file, by default False
- **update_path** (*bool, optional*) – whether to update mne config, by default True
- **verbose** (*Optional[Union[bool, str, int]], optional*) – [description], by default None

Returns
local path of the target file

Return type
str

metabci.brainda.utils.io module

metabci.brainda.utils.io.**loadmat**(*mat_file: str | Path*) → dict

Wrapper of scipy.io loadmat function, works for matv7.3.

Parameters
mat_file (*Union[str, Path]*) – file path

Returns
data

Return type
dict

metabci.brainda.utils.performance module

class metabci.brainda.utils.performance.**Performance**(*estimators_list=['Acc', 'pITR']*, *Tw=None*, *Ts=None*, *isdraw=False*)

Bases: BaseEstimator, TransformerMixin

Evaluation of BCI performance.

update log:

2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **Tw** (*float*) – Signal duration (in second).
- **Ts** (*float*) – Eye shift time (in second).
- **estimators_list** (*list*) –

  supported estimators

  Acc: Accuracy classification score.

  bAcc: balanced accuracy to deal with imbalanced datasets.

  tITR: theoretical ITR.

  pITR: practical ITR.

  TPR: true positive rate(TPR).

  FNR: false negative rate(FNR).

  FPR: false positive rate (FPR).

  TNR: true negative rate (TNR).

  AUC: Area under the curve.
- **isdraw** (*bool*) – Whether to draw the ROC curve.

estimators_list

supported estimators

Acc: Accuracy classification score.

bAcc: balanced accuracy to deal with imbalanced datasets.

tITR: theoretical ITR.

pITR: practical ITR.

TPR: true positive rate(TPR).

FNR: false negative rate(FNR).

FPR: false positive rate (FPR).

TNR: true negative rate (TNR).

AUC: Area under the curve.

Type

list

Tw

Signal duration (in second).

Type
float

Ts

Eye shift time (in second).

Type
float

isdraw

Whether to draw the ROC curve.

Type
bool

Tip:

Listing 17: Example

```
1.from metabci.brainda.utils.performance import Performance.

2.performance = Performance(estimators_list=["Acc","pITR","TPR","AUC"], Tw=0.5,␣
↪Ts=0.5).

3.results = performance.evaluate(y_true=y[test_ind], y_pred=p_labels, y_score=p_
↪corr).
```

---

evaluate(*y_true*, *y_pred*, *y_score=None*)

Transform EEG to covariance matrix.

update log:
2023-12-10 by Leyi Jia <18020095036@163.com>, Add code annotation

Parameters

- **y_true** (*1d array-like*) – Ground truth (correct) labels.
- **y_pred** (*1d array-like*) – Predicted labels.
- **y_score** (*array-like of (n_samples, n_classes)*) – Target scores.

Returns
results – Evaluate the results and form a dictionary.

Return type
list

metabci.brainda.utils.performance.**profile**(*func*)

Module contents

Module contents

metabci.brainflow package

Submodules

metabci.brainflow.amplifiers module

metabci.brainflow.logger module

Logging system.

metabci.brainflow.logger.**disable_log**()

disable system logger. -author: Lichao Xu -Created on: 2021-04-01 -update log:

Nonw

metabci.brainflow.logger.**get_logger**(*log_name*)

get system logger. -author: Lichao Xu -Created on: 2021-04-01 -update log:

Nonw

Parameters

log_name (*str,*) – Name of logger.

metabci.brainflow.workers module

Start another process, define a framework for offline modeling and online processing with three functions:

pre(): for offline modeling;

consume(): for online prediction;

post(): for subsequent custom operations.

In the actual usage process, you only need to customize the operations of the above functions.

class metabci.brainflow.workers.**ProcessWorker**(*timeout: float = 0.001*, *name: str | None = None*)

Bases: Process

Online processing.

author: Lichao Xu

Created on: 2021-04-01

update log:

2022-08-10 by Wei Zhao

Parameters

- **timeout** (*float*) – Timer setting.
- **name** (*str*) – Custom name for the online processing process.

daemon

Type
bool

_exit

Multiprocess event handling.

_in_queue

Data sharing between the online processing process and the main process.

Type
queue

Tip:

Listing 18: A example using brainflow. worker

```
from brainflow. worker import ProcessWorker
class FeedbackWorker(ProcessWorker):
    def __init__():
        #Initialization

    def pre(self):
        #Off-line modeling

        #Online processing of data flow between stimulus interfaces
        info = StreamInfo(
            name='meta_feedback',
            type='Markers',
            channel_count=1,
            nominal_srate=0,
            channel_format='int32',
            source_id=self.lsl_source_id)
        self.outlet = StreamOutlet(info)
        print('waiting connection...')
        while not self._exit:
            if self.outlet.wait_for_consumers(1e-3):
                break
        print('Connected')

    def consume(self, data) :
        #Online processing
        if self.outlet.have_consumers ():
            self.outlet.push_sample("online resultslist")

    def post(self):
        pass
```

---

clear_queue()

Clear the queue.

author: Lichao Xu

Created on: 2021-04-01

update log:
2022-08-10 by Wei Zhao

`abstract consume`(*data*)

Custom function to process online data.

author: Lichao Xu

Created on: 2021-04-01

update log:
2022-08-10 by Wei Zhao

Parameters
data (*ndarray, shape(n_samples, n_channels+1)*) – Single trial of online data.

`abstract post`()

`abstract pre`()

Custom function to build a model using offline data.

author: Lichao Xu

Created on: 2021-04-01

update log:
2022-08-10 by Wei Zhao

`put`(*data*)

Put the data in the queue

author: Lichao Xu

Created on: 2021-04-01

update log:
2022-08-10 by Wei Zhao

Parameters
data (*ndarray, shape(n_samples, n_channels+1)*) – Single trial of online data.

`run`()

author: Lichao Xu

Created on: 2021-04-01

update log:
2022-08-10 by Wei Zhao

Online processing process:
Customize the *pre()* function to build a model using offline data.

Clear the queue and wait for data retrieval thread in the main process to get data within a fixed time.

Customize the *consume()* function to process online data and provide feedback.

Customize the *post()* function to perform subsequent operations.

Wait for the next online label to start the next online processing.

Close the online processing process, clear the queue, and stop online experiments.

settimeout(*timeout=0.01*)

Set the timer.

author: Lichao Xu

Created on: 2021-04-01

update log:
2022-08-10 by Wei Zhao

stop()

Stop the online processing process.

author: Lichao Xu

Created on: 2021-04-01

update log:
2022-08-10 by Wei Zhao

Module contents

metabci.brainstim package

Submodules

metabci.brainstim.framework module

metabci.brainstim.paradigm module

metabci.brainstim.utils module

Module contents

2.1.2 Module contents

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

S

T