pyRiemann Documentation

Release 0.4

Alexandre Barachant

Feb 15, 2023

CONTENTS

pyRiemann is a Python machine learning package based on scikit-learn API. It provides a high-level interface for processing and classification of multivariate time series through the Riemannian geometry of symmetric positive definite (SPD) matrices.

pyRiemann aims at being a generic package for multivariate time series classification but has been designed around multichannel biosignals (like EEG, MEG or EMG) manipulation applied to brain-computer interface (BCI), transforming multichannel time series into covariance matrices, and classifying them using the Riemannian geometry of SPD matrices.

For a brief introduction to the ideas behind the package, you can read the *introductory notes*. More practical information is on the *installation page*. You may also want to browse the example gallery to get a sense for what you can do with pyRiemann and *API reference* to find out how.

To see the code or report a bug, please visit the github repository.

CHAPTER

ONE

INTRODUCTION TO PYRIEMANN

CHAPTER

TWO

WHAT'S NEW IN THE PACKAGE

A catalog of new features, improvements, and bug-fixes in each release.

2.1 v0.4 (Feb 2023)

- Add exponential and logarithmic maps for three main metrics: 'euclid', 'logeuclid' and 'riemann'. *pyriemann.utils.tangentspace.tangent_space()* is splitted in two steps: (i) `log_map_*()` projecting SPD matrices into tangent space depending on the metric; and (ii) *pyriemann.utils.tangentspace.upper()* taking the upper triangular part of matrices. Similarly, *pyriemann.utils.tangentspace.untangent_space()* is splitted into (i) *pyriemann.utils.tangentspace.unupper()* and (ii) `exp_map_*()`. The different metrics for tangent space mapping can now be defined into *pyriemann.tangentspace.TangentSpace*, then used for `transform()` as well as for `inverse_transform()`. #195 by @qbarthelemy
- Enhance AJD: add `init` to *pyriemann.utils.ajd.ajd_pham()* and *pyriemann.utils.ajd.rjd()*, add `warm_restart` to *pyriemann.spatialfilters.AJDC*. #196 by @qbarthelemy
- Add parameter `sampling_method` to *pyriemann.datasets.sample_gaussian_spd()*, with `rejection` accelerating 2x2 matrices generation. #198 by @Artim436
- Add geometric medians for Euclidean and Riemannian metrics: *pyriemann.utils.median_euclid()* and *pyriemann.utils.median_riemann()*, and add an example in gallery to compare means and medians on synthetic datasets. #200 by @qbarthelemy
- Add `score()` to *pyriemann.regression.KNearestNeighborRegressor*. #205 by @qbarthelemy
- Add Transfer Learning module and examples, including RPA and MDWM. #189 by @plcrodrigues, @qbarthelemy and @sylvchev
- Add class distinctiveness function to measure the distinctiveness between classes on the manifold, *pyriemann.classification.class_distinctiveness()*, and complete an example in gallery to show how it works on synthetic datasets. #215 by @MSYamamoto
- Add example on ensemble learning applied to functional connectivity, and add *pyriemann.utils.base.nearest_sym_pos_def()*. #202 by @mccorsi and @sylvchev
- Add kernel matrices representation *pyriemann.estimation.Kernels* and complete example comparing estimators. #217 by @qbarthelemy
- Add a new covariance estimator, robust fixed point covariance, and add kwds arguments for all covariance based functions and classes. #220 by @qbarthelemy
- Add example in gallery on frequency band selection using class distinctiveness measure. #219 by @MSYamamoto

- Add *pyriemann.utils.covariance.covariance_mest()* supporting three robust M-estimators (Huber, Student-t and Tyler) and available for all covariance based functions and classes; and add an example on robust covariance estimation for corrupted data. Add also *pyriemann.utils.distance.distance_mahalanobis()* between between vectors and a Gaussian distribution. #223 by @qbarthelemy

2.2 v0.3 (July 2022)

- Correct spectral estimation in *pyriemann.utils.covariance.cross_spectrum()* to obtain equivalence with SciPy. #133 by @qbarthelemy
- Add instantaneous, lagged and imaginary coherences in *pyriemann.utils.covariance.coherence()* and *pyriemann.estimation.Coherences*. #132 by @qbarthelemy
- Add `partial_fit` in *pyriemann.clustering.Potato*, useful for an online update; and update example on artifact detection. #133 by @qbarthelemy
- Deprecate *pyriemann.utils.viz.plot_confusion_matrix()* as sklearn integrate its own version. #135 by @sylvchev
- Add Ando-Li-Mathias mean estimation in *pyriemann.utils.mean.mean_covariance()*. #56 by @sylvchev
- Add Schaefer-Strimmer covariance estimator in *pyriemann.utils.covariance.covariances()*, and an example to compare estimators #59 by @sylvchev
- Refactor tests + fix refit of *pyriemann.tangentspace.TangentSpace*. #136 by @sylvchev
- Add *pyriemann.clustering.PotatoField*, and an example on artifact detection. #142 by @qbarthelemy
- Add sampling SPD matrices from a Riemannian Gaussian distribution in *pyriemann.datasets.sample_gaussian_spd()*. #140 by @plcrodrigues
- Add new function *pyriemann.datasets.make_gaussian_blobs()* for generating random datasets with SPD matrices. #140 by @plcrodrigues
- Add module `pyriemann.utils.viz` in API, add *pyriemann.utils.viz.plot_waveforms()*, and add an example on ERP visualization. #144 by @qbarthelemy
- Add a special form covariance matrix *pyriemann.utils.covariance.covariances_X()*. #147 by @qbarthelemy
- Add masked and NaN means with Riemannian metric: *pyriemann.utils.mean.maskedmean_riemann()* and *pyriemann.utils.mean.nanmean_riemann()*. #149 by @qbarthelemy and @sylvchev
- Add `corr` option in *pyriemann.utils.covariance.normalize()*, to normalize covariance into correlation matrices. #153 by @qbarthelemy
- Add block covariance matrix: *pyriemann.estimation.BlockCovariances* and *pyriemann.utils.covariance.block_covariances()*. #154 by @gabelstein
- Add Riemannian Locally Linear Embedding: *pyriemann.embedding.LocallyLinearEmbedding* and *pyriemann.embedding.locally_linear_embedding()*. #159 by @gabelstein
- Add Riemannian Kernel Function: *pyriemann.utils.kernel.kernel_riemann()*. #159 by @gabelstein
- Fix `fit` in *pyriemann.channelselection.ElectrodeSelection*. #166 by @qbarthelemy
- Add power mean estimation in *pyriemann.utils.mean.mean_power()*. #170 by @qbarthelemy and @plcrodrigues
- Add example in gallery to compare classifiers on synthetic datasets. #175 by @qbarthelemy

- Add `predict_proba` in *pyriemann.classification.KNearestNeighbor*, and correct attribute `classes_`. #171 by @qbarthelemy
- Add Riemannian Support Vector Machine classifier: *pyriemann.classification.SVC*. #175 by @gabelstein and @qbarthelemy
- Add Riemannian Support Vector Machine regressor: *pyriemann.regression.SVR*. #175 by @gabelstein and @qbarthelemy
- Add K-Nearest-Neighbor regressor: *pyriemann.regression.KNearestNeighborRegressor*. #164 by @gabelstein, @qbarthelemy and @agramfort
- Add Minimum Distance to Mean Field classifier: *pyriemann.classification.MeanField*. #172 by @qbarthelemy and @plcrodrigues
- Add example on principal geodesic analysis (PGA) for SSVEP classification. #169 by @qbarthelemy
- Add *pyriemann.utils.distance.distance_harmonic()*, and sort functions by their names in code, doc and tests. #183 by @qbarthelemy
- Parallelize functions for dataset generation: *pyriemann.datasets.make_gaussian_blobs()*. #179 by @sylvchev
- Fix dispersion when generating datasets: *pyriemann.datasets.sample_gaussian_spd()*. #179 by @sylvchev
- Enhance base and distance functions, to process ndarrays of SPD matrices. #186 and #187 by @qbarthelemy
- Enhance utils functions, to process ndarrays of SPD matrices. #190 by @qbarthelemy
- Enhance means functions, with faster implementations and warning when convergence is not reached. #188 by @qbarthelemy

2.3 v0.2.7 (June 2021)

- Add example on SSVEP classification
- Fix compatibility with scikit-learn v0.24
- Correct probas of *pyriemann.classification.MDM*
- Add `predict_proba` for *pyriemann.clustering.Potato*, and an example on artifact detection
- Add weights to Pham's AJD algorithm *pyriemann.utils.ajd.ajd_pham()*
- Add *pyriemann.utils.covariance.cross_spectrum()*, fix *pyriemann.utils.covariance.cospectrum()*; *pyriemann.utils.covariance.coherence()* output is kept unchanged
- Add *pyriemann.spatialfilters.AJDC* for BSS and gBSS, with an example on artifact correction
- Add *pyriemann.preprocessing.Whitening*, with optional dimension reduction

2.4 v0.2.6 (March 2020)

- Updated for better Scikit-Learn v0.22 support

2.5 v0.2.5 (January 2018)

- Added BilinearFilter
- Added a permutation test for generic scikit-learn estimator
- Stats module refactoring, with distance based t-test and f-test
- Removed two way permutation test
- Added FlatChannelRemover
- Support for python 3.5 in travis
- Added Shrinkage transformer
- Added Coherences transformer
- Added Embedding class.

2.6 v0.2.4 (June 2016)

- Improved documentation
- Added TSclassifier for out-of the box tangent space classification.
- Added Wasserstein distance and mean.
- Added NearestNeighbor classifier.
- Added Softmax probabilities for MDM.
- Added CSP for covariance matrices.
- Added Approximate Joint diagonalization algorithms (JADE, PHAM, UWEDGE).
- Added ALE mean.
- Added Multiclass CSP.
- API: param name changes in *CospCovariances* to comply to Scikit-Learn.
- API: attributes name changes in most modules to comply to the Scikit-Learn naming convention.
- Added *HankelCovariances* estimation
- Added *SPoC* spatial filtering
- Added Harmonic mean
- Added Kullback leibler mean

2.7 v0.2.3 (November 2015)

- Added multiprocessing for MDM with joblib.
- Added kullback-leibler divergence.
- Added Riemannian Potato.
- Added sample_weight for mean estimation and MDM.

CHAPTER

THREE

INSTALLING PYRIEMANN

The easiest way to install a stable version of pyRiemann is through pypi, the python package manager :

```
pip install pyriemann
```

For a bleeding edge version, you can clone the source code on github and install directly the package from source.

```
pip install -e .
```

The install script will install the required dependencies. If you want also to build the documentation and to run the test locally, you could install all development dependencies with

```
pip install -e .[docs,tests]
```

If you use a zsh shell, you need to write *pip install -e .[docs,tests]*. If you do not know what zsh is, you could use the above command.

3.1 Dependencies

- Python (>= 3.7)

3.1.1 Mandatory dependencies

- numpy
- scipy
- scikit-learn >=0.17
- pandas
- joblib

3.1.2 Recommended dependencies

These dependencies are recommanded to use the plotting functions of pyriemann or to run examples and tutorials, but they are not mandatory:

- mne-python
- matplotlib
- seaborn

CHAPTER

FOUR

EXAMPLES GALLERY

Contents

4.1 Classification of ERP

Using Riemannian geometry for classifying event-related potentials (ERP).

4.2 Classification of SSVEP

Using Riemannian geometry for classifying steady-state visually evoked potentials (SSVEP).

4.3 Artifact management

Using Riemannian geometry to detect, reject or correct artifacts.

4.4 Classification of motor imagery

Using Riemannian geometry for classifying motor imagery.

4.5 Covariance estimation

Examples for covariance matrix estimation.

4.6 Simulated data

Examples using datasets sampled from known probability distributions.

4.7 Permutation test

Permutation test with pyRiemann.

4.8 Transfer learning

Using Riemannian geometry for transfer learning and domain adaptation.

4.8.1 Classification of ERP

Using Riemannian geometry for classifying event-related potentials (ERP).

Embedding ERP MEG data in 2D Euclidean space

Riemannian embeddings via Laplacian Eigenmaps (LE) and Locally Linear Embedding (LLE) of a set of ERP data. Embedding via Laplacian Eigenmaps is referred to as Spectral Embedding (SE).

Locally Linear Embedding (LLE) assumes that the local neighborhood of a point on the manifold can be well approximated by the affine subspace spanned by the k-nearest neighbors of the point and finds a low-dimensional embedding of the data based on these affine approximations.

Laplacian Eigenmaps (LE) are based on computing the low dimensional representation that best preserves locality instead of local linearity in LLE[1].

```
# Authors:  Pedro Rodrigues <pedro.rodrigues01@gmail.com>,
#           Gabriel Wagner vom Berg <gabriel@bccn-berlin.de>
# License: BSD (3-clause)

from pyriemann.estimation import XdawnCovariances
from pyriemann.utils.viz import plot_embedding
```

(continues on next page)

[1] Clustering and dimensionality reduction on Riemannian manifolds A. Goh and R Vidal, in 2008 IEEE Conference on Computer Vision and Pattern Recognition.

(continued from previous page)

```
import mne
from mne import io
from mne.datasets import sample

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt


print(__doc__)
```

Set parameters and read data

```
data_path = str(sample.data_path())
raw_fname = data_path + '/MEG/sample/sample_audvis_filt-0-40_raw.fif'
event_fname = data_path + '/MEG/sample/sample_audvis_filt-0-40_raw-eve.fif'
tmin, tmax = -0., 1
event_id = dict(aud_l=1, aud_r=2, vis_l=3, vis_r=4)

# Setup for reading the raw data
raw = io.Raw(raw_fname, preload=True, verbose=False)
raw.filter(2, None, method='iir')  # replace baselining with high-pass
events = mne.read_events(event_fname)

raw.info['bads'] = ['MEG 2443']  # set bad channels
picks = mne.pick_types(raw.info, meg=True, eeg=False, stim=False, eog=False,
                       exclude='bads')

# Read epochs
epochs = mne.Epochs(raw, events, event_id, tmin, tmax, proj=False,
                    picks=picks, baseline=None, preload=True, verbose=False)

X = epochs.get_data()
y = epochs.events[:, -1]
```

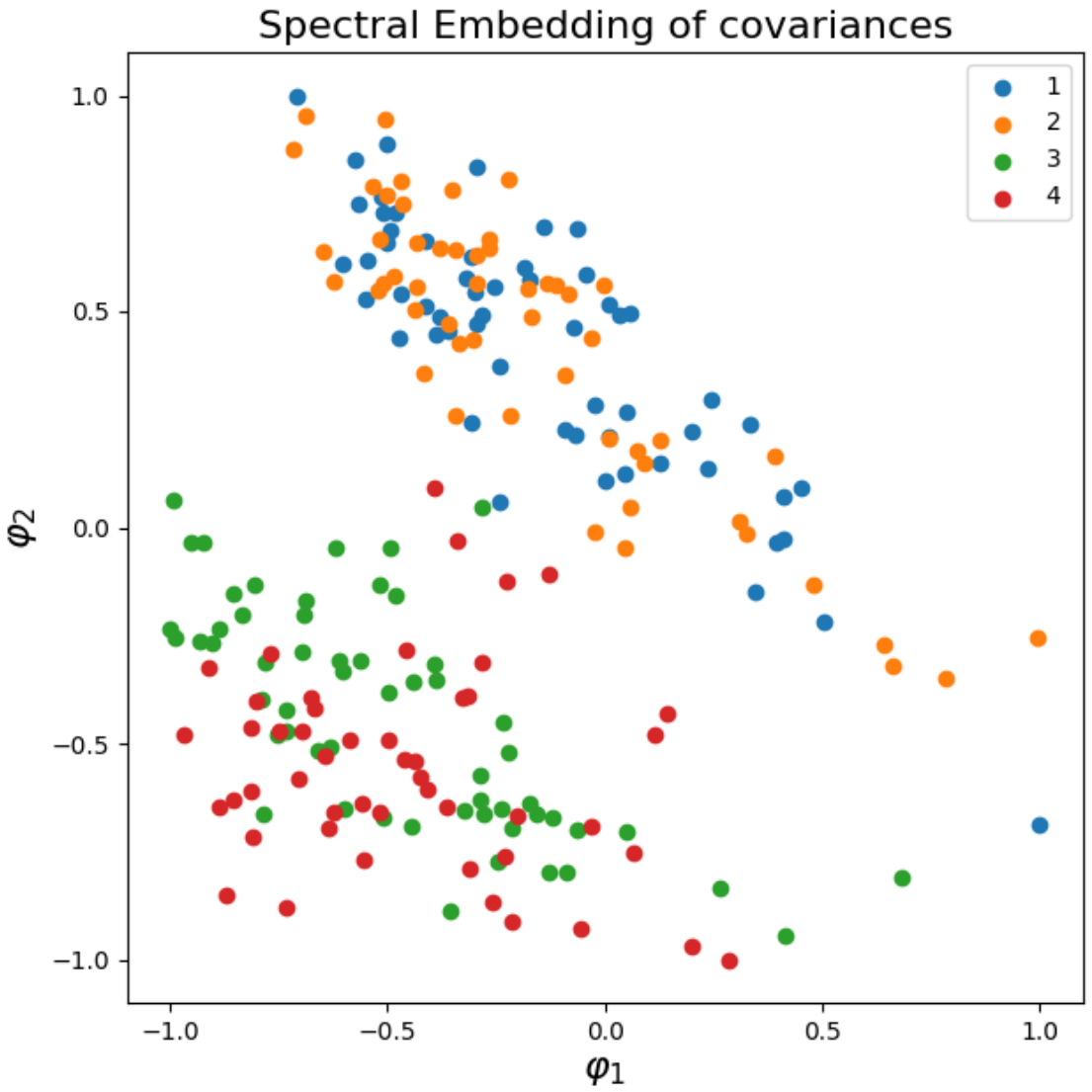
```
Filtering raw data in 1 contiguous segment
Setting up high-pass filter at 2 Hz

IIR filter parameters
---------------------
Butterworth highpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 8 (effective, after forward-backward)
- Cutoff at 2.00 Hz: -6.02 dB
```

Embedding of Xdawn covariance matrices

```
nfilter = 4
xdwn = XdawnCovariances(estimator='scm', nfilter=nfilter)
split = train_test_split(X, y, train_size=0.25, random_state=42)
Xtrain, Xtest, ytrain, ytest = split
covs = xdwn.fit(Xtrain, ytrain).transform(Xtest)
```

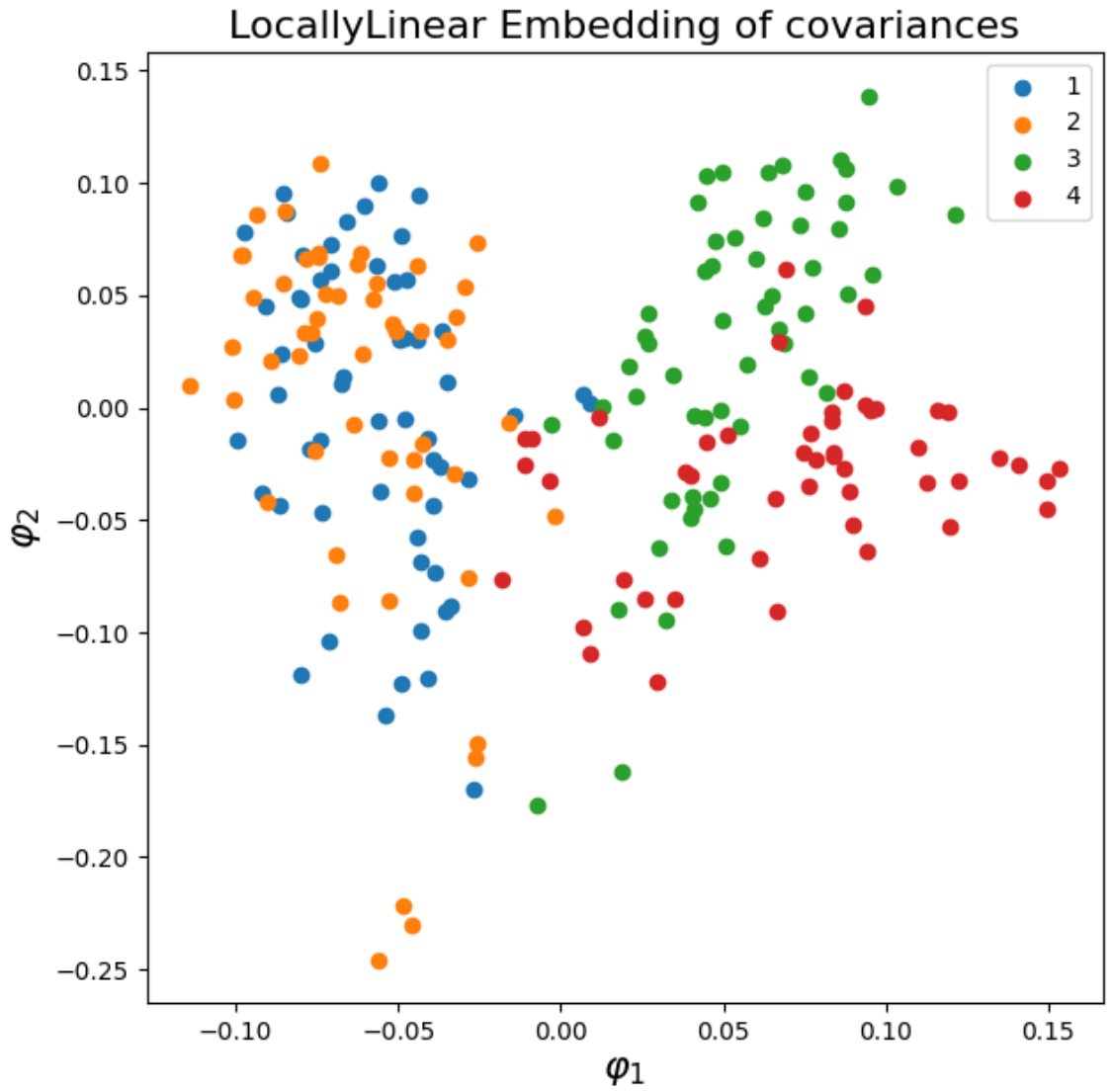
Laplacian Eigenmaps (LE), also called Spectral Embedding (SE)

```
plot_embedding(covs, ytest, metric='riemann', embd_type='Spectral',
               normalize=True)
plt.show()
```



Locally Linear Embedding (LLE)

```
plot_embedding(covs, ytest, metric='riemann', embd_type='LocallyLinear',
               normalize=False)
plt.show()
```



References

Total running time of the script: (0 minutes 20.420 seconds)

Display ERP

Different ways to display a multichannel event-related potential (ERP).

```
# Authors: Quentin Barthélemy
#
# License: BSD (3-clause)

import numpy as np
import mne
from matplotlib import pyplot as plt
from pyriemann.utils.viz import plot_waveforms
```

Load EEG data

```
# Set filenames
data_path = str(mne.datasets.sample.data_path())
raw_fname = data_path + "/MEG/sample/sample_audvis_filt-0-40_raw.fif"
event_fname = data_path + "/MEG/sample/sample_audvis_filt-0-40_raw-eve.fif"

# Read raw data, select occipital channels and high-pass filter signal
raw = mne.io.Raw(raw_fname, preload=True, verbose=False)
raw.pick_channels(['EEG 057', 'EEG 058', 'EEG 059'], ordered=True)
raw.rename_channels({'EEG 057': 'O1', 'EEG 058': 'Oz', 'EEG 059': 'O2'})
n_channels = len(raw.ch_names)
raw.filter(1.0, None, method="iir")

# Read epochs and get responses to left visual field stimulus
tmin, tmax = -0.1, 0.8
epochs = mne.Epochs(
    raw, mne.read_events(event_fname), {'vis_l': 3}, tmin, tmax, proj=False,
    baseline=None, preload=True, verbose=False)
X = 5e5 * epochs.get_data()
print('Number of trials:', X.shape[0])
times = np.linspace(tmin, tmax, num=X.shape[2])

plt.rcParams["figure.figsize"] = (7, 12)
ylims = []
```

```
Removing projector <Projection | PCA-v1, active : False, n_channels : 102>
Removing projector <Projection | PCA-v2, active : False, n_channels : 102>
Removing projector <Projection | PCA-v3, active : False, n_channels : 102>
Filtering raw data in 1 contiguous segment
Setting up high-pass filter at 1 Hz

IIR filter parameters
```

(continues on next page)

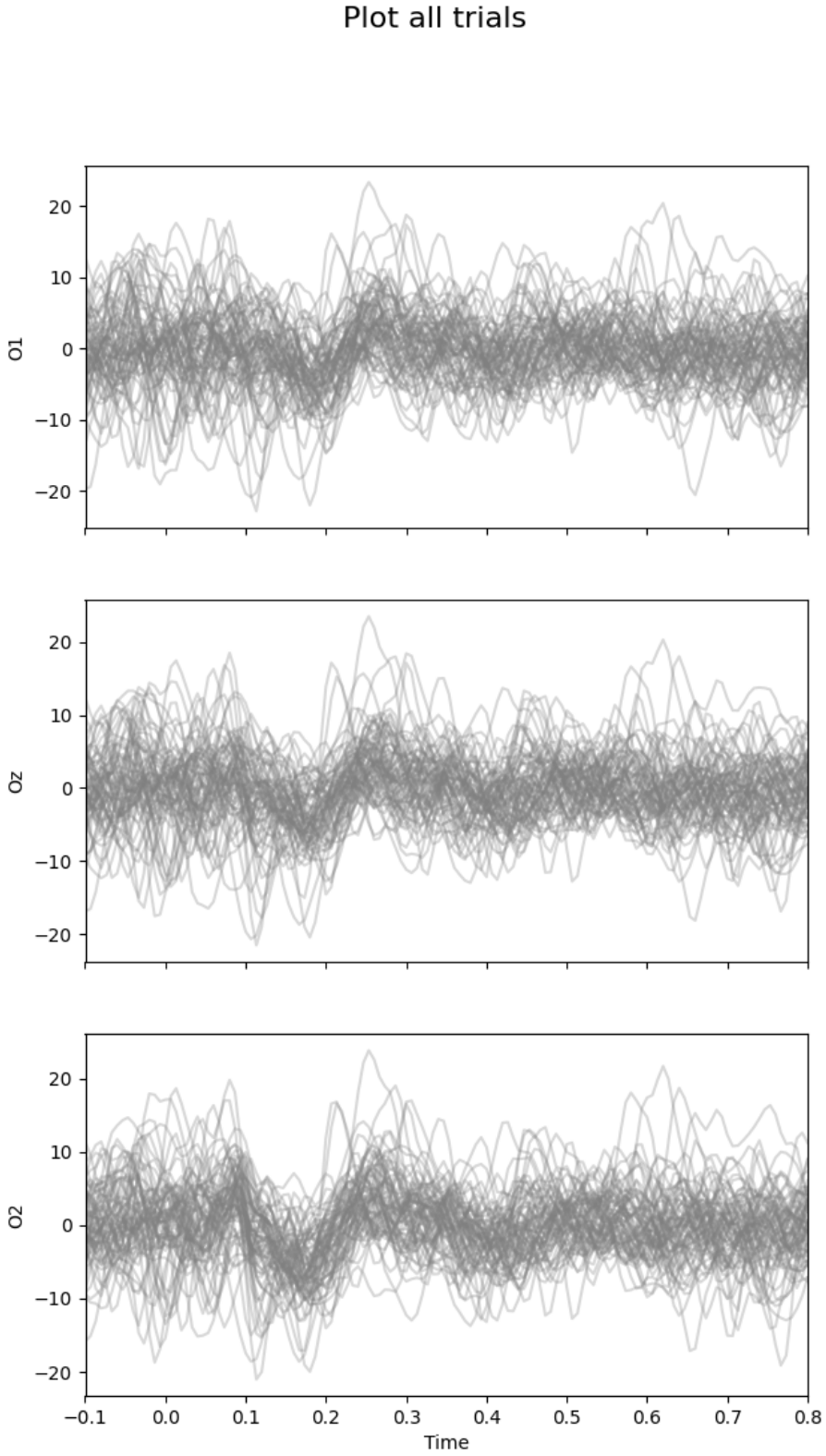
(continued from previous page)

```
---------------------
Butterworth highpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 8 (effective, after forward-backward)
- Cutoff at 1.00 Hz: -6.02 dB

Number of trials: 73
```

Plot all trials

This kind of plot is a little bit messy.

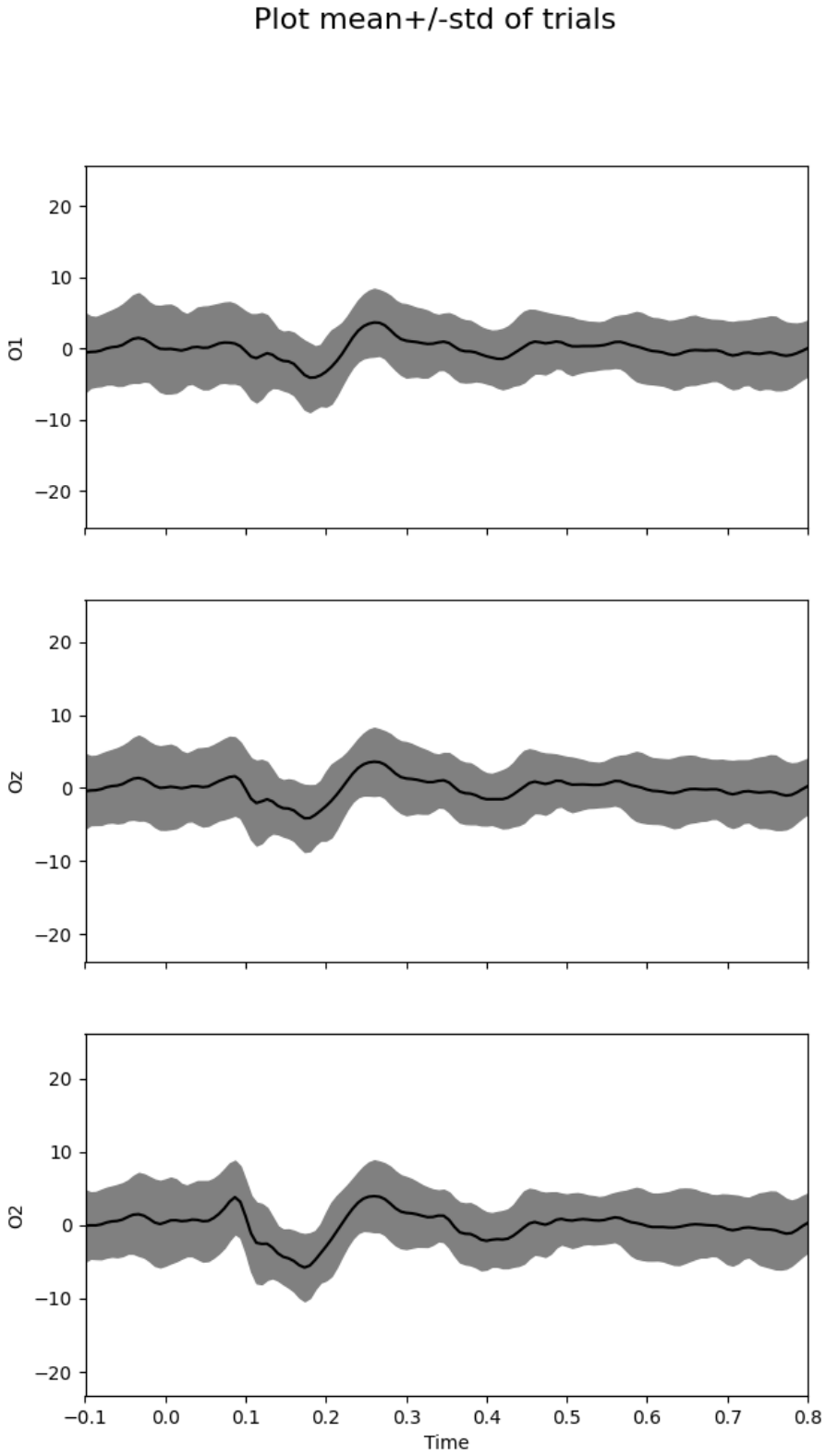
```
fig = plot_waveforms(X, 'all', times=times, alpha=0.3)
fig.suptitle('Plot all trials', fontsize=16)
for i_channel in range(n_channels):
    fig.axes[i_channel].set(ylabel=raw.ch_names[i_channel])
    fig.axes[i_channel].set_xlim(tmin, tmax)
    ylims.append(fig.axes[i_channel].get_ylim())
fig.axes[n_channels - 1].set(xlabel='Time')
plt.show()
```


Plot all trials
O1
Oz
O2
20
10
0
−10
−20
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Time

Plot central tendency and dispersion of trials

This kind of plot is well-spread, but mean and standard deviation can be contaminated by artifacts, and they make a symmetric assumption on amplitude distribution.

```
fig = plot_waveforms(X, 'mean+/-std', times=times)
fig.suptitle('Plot mean+/-std of trials', fontsize=16)
for i_channel in range(n_channels):
    fig.axes[i_channel].set(ylabel=raw.ch_names[i_channel])
    fig.axes[i_channel].set_xlim(tmin, tmax)
    fig.axes[i_channel].set_ylim(ylims[i_channel])
fig.axes[n_channels - 1].set(xlabel='Time')
plt.show()
```

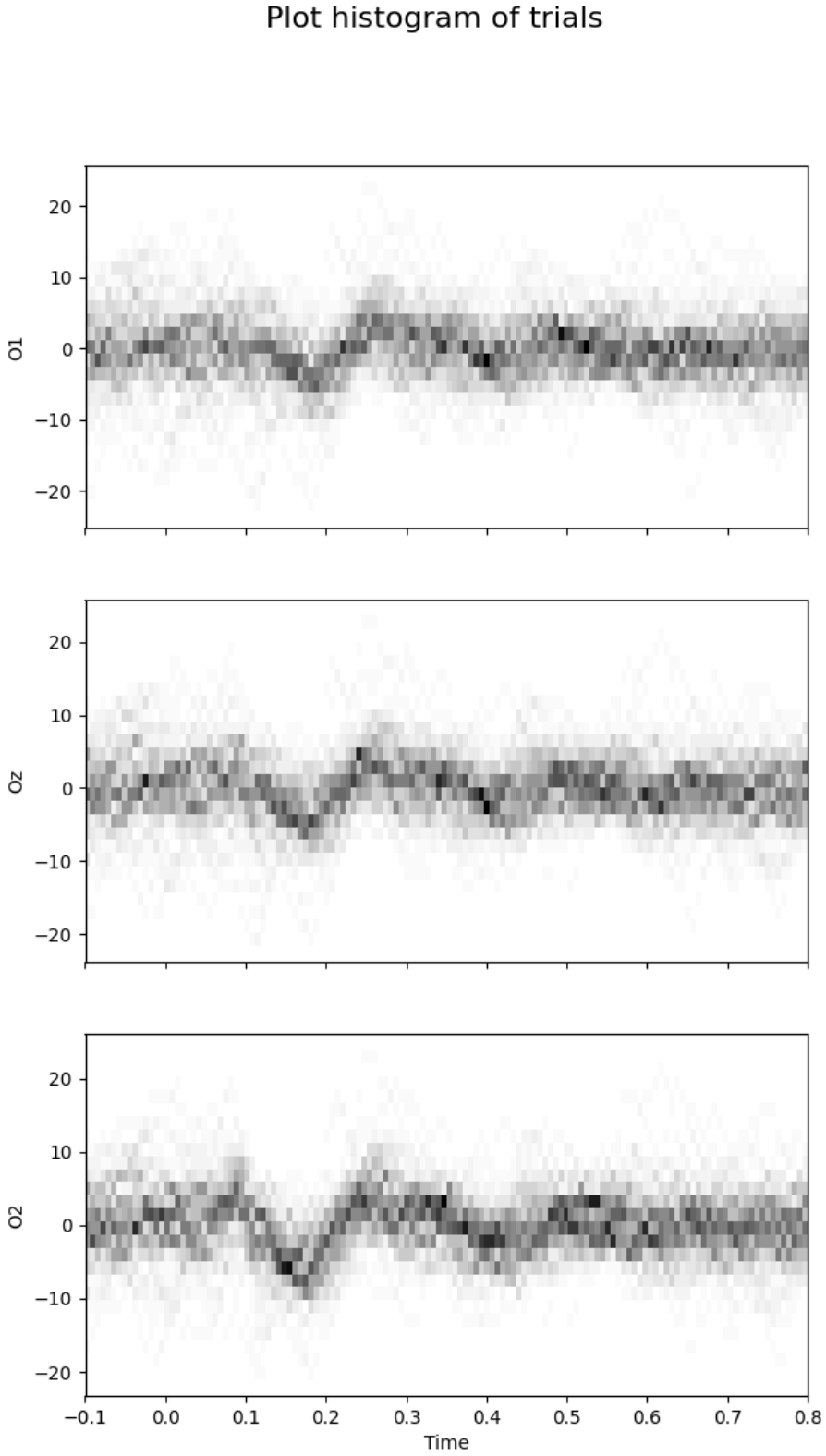

Plot mean+/-std of trials
O1
Oz
O2
20
10
0
−10
−20
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Time

Plot histogram of trials

This plot estimates a 2D histogram of trials[1].

```
fig = plot_waveforms(X, 'hist', times=times, n_bins=25, cmap=plt.cm.Greys)
fig.suptitle('Plot histogram of trials', fontsize=16)
for i_channel in range(n_channels):
    fig.axes[i_channel].set(ylabel=raw.ch_names[i_channel])
    fig.axes[i_channel].set_ylim(ylims[i_channel])
fig.axes[n_channels - 1].set(xlabel='Time')
plt.show()
```

[1] Improved estimation of EEG evoked potentials by jitter compensation and enhancing spatial filters A. Souloumiac and B. Rivet. 2013 IEEE International Conference on Acoustics, Speech and Signal Processing.


Plot histogram of trials
O1
Oz
O2
20
10
0
−10
−20
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Time

References

Total running time of the script: (0 minutes 1.148 seconds)

ERP EEG decoding in Tangent space.

Decoding applied to EEG data in sensor space decomposed using Xdawn. After spatial filtering, covariances matrices are estimated, then projected in the tangent space and classified with a logistic regression.

```
# Authors: Alexandre Barachant <alexandre.barachant@gmail.com>
#
# License: BSD (3-clause)

import numpy as np

from pyriemann.estimation import XdawnCovariances
from pyriemann.tangentspace import TangentSpace

import mne
from mne import io
from mne.datasets import sample

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.pipeline import make_pipeline

from matplotlib import pyplot as plt

print(__doc__)
```

Set parameters and read data

```
data_path = str(sample.data_path())
raw_fname = data_path + "/MEG/sample/sample_audvis_filt-0-40_raw.fif"
event_fname = data_path + "/MEG/sample/sample_audvis_filt-0-40_raw-eve.fif"
tmin, tmax = -0.0, 1
event_id = dict(aud_l=1, aud_r=2, vis_l=3, vis_r=4)

# Setup for reading the raw data
raw = io.Raw(raw_fname, preload=True, verbose=False)
raw.filter(2, None, method="iir")  # replace baselining with high-pass
events = mne.read_events(event_fname)

raw.info["bads"] = ["MEG 2443"]  # set bad channels
picks = mne.pick_types(
    raw.info, meg=False, eeg=True, stim=False, eog=False, exclude="bads"
)

# Read epochs
epochs = mne.Epochs(
    raw,
    events,
```

(continues on next page)

(continued from previous page)

```
    event_id,
    tmin,
    tmax,
    proj=False,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False,
)

labels = epochs.events[:, -1]
evoked = epochs.average()
```

```
Filtering raw data in 1 contiguous segment
Setting up high-pass filter at 2 Hz

IIR filter parameters
---------------------
Butterworth highpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 8 (effective, after forward-backward)
- Cutoff at 2.00 Hz: -6.02 dB

Removing projector <Projection | PCA-v1, active : False, n_channels : 102>
Removing projector <Projection | PCA-v2, active : False, n_channels : 102>
Removing projector <Projection | PCA-v3, active : False, n_channels : 102>
```

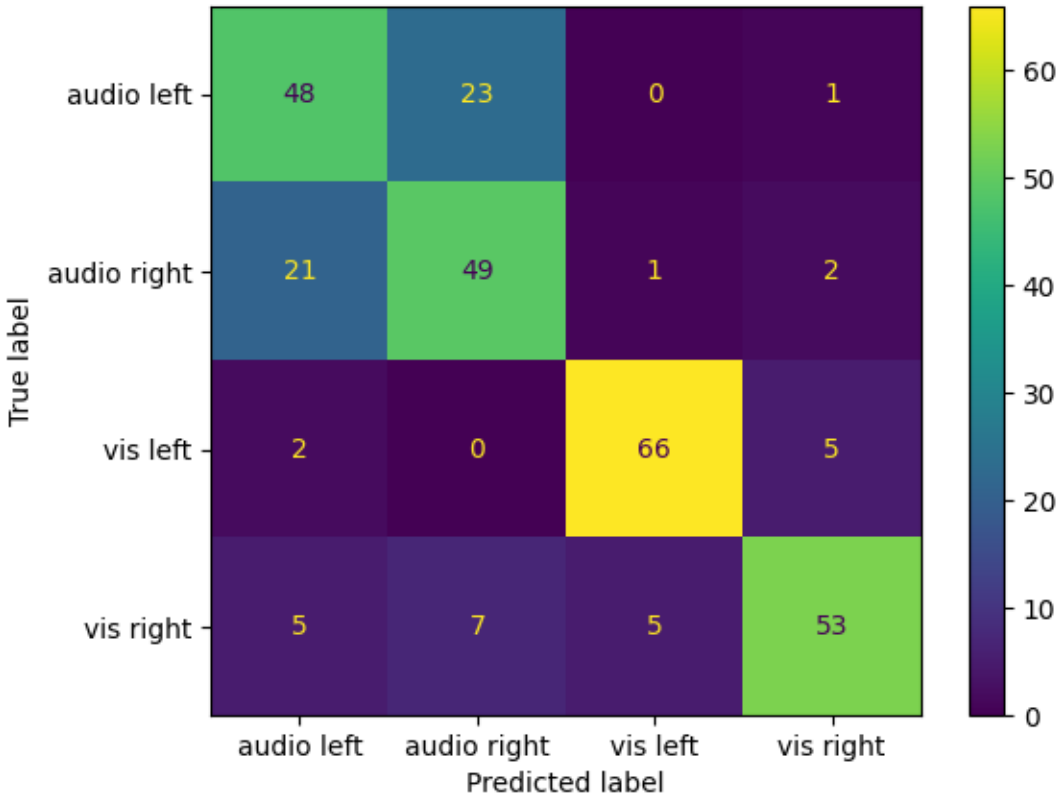
Decoding in tangent space with a logistic regression

```
n_components = 2  # pick some components

# Define a monte-carlo cross-validation generator (reduce variance):
cv = KFold(n_splits=10, shuffle=True, random_state=42)
epochs_data = epochs.get_data()


clf = make_pipeline(
    XdawnCovariances(n_components),
    TangentSpace(metric="riemann"),
    LogisticRegression(),
)

preds = np.zeros(len(labels))

for train_idx, test_idx in cv.split(epochs_data):
    y_train, y_test = labels[train_idx], labels[test_idx]

    clf.fit(epochs_data[train_idx], y_train)
    preds[test_idx] = clf.predict(epochs_data[test_idx])

# Printing the results
acc = np.mean(preds == labels)
print("Classification accuracy: %f " % (acc))
```

(continues on next page)

(continued from previous page)

```
names = ["audio left", "audio right", "vis left", "vis right"]
cm = confusion_matrix(labels, preds)
ConfusionMatrixDisplay(cm, display_labels=names).plot()
plt.show()
```




```
Classification accuracy: 0.750000
```

Total running time of the script: (0 minutes 6.012 seconds)

Multiclass MEG ERP Decoding

Decoding applied to MEG data in sensor space decomposed using Xdawn. After spatial filtering, covariances matrices are estimated and classified by the MDM algorithm (Nearest centroid).

4 Xdawn spatial patterns (1 for each class) are displayed, as per the for mean-covariance matrices used by the classification algorithm.

```
# Authors: Alexandre Barachant <alexandre.barachant@gmail.com>
#
# License: BSD (3-clause)
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from matplotlib import pyplot as plt
from pyriemann.estimation import XdawnCovariances
from pyriemann.classification import MDM

import mne
from mne import io
from mne.datasets import sample

from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    ConfusionMatrixDisplay,
)
from sklearn.model_selection import KFold
from sklearn.pipeline import make_pipeline

print(__doc__)
```

Set parameters and read data

```
data_path = str(sample.data_path())
raw_fname = data_path + "/MEG/sample/sample_audvis_filt-0-40_raw.fif"
event_fname = data_path + "/MEG/sample/sample_audvis_filt-0-40_raw-eve.fif"
tmin, tmax = -0.0, 1
event_id = dict(aud_l=1, aud_r=2, vis_l=3, vis_r=4)

# Setup for reading the raw data
raw = io.Raw(raw_fname, preload=True)
raw.filter(2, None, method="iir")  # replace baselining with high-pass
events = mne.read_events(event_fname)

raw.info["bads"] = ["MEG 2443"]  # set bad channels
picks = mne.pick_types(
    raw.info, meg="grad", eeg=False, stim=False, eog=False, exclude="bads"
)

# Read epochs
epochs = mne.Epochs(
    raw,
    events,
    event_id,
    tmin,
    tmax,
    proj=False,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False,
)

labels = epochs.events[:, -1]
```

(continues on next page)

(continued from previous page)

```
evoked = epochs.average()
```

```
Opening raw data file /home/docs/mne_data/MNE-sample-data/MEG/sample/sample_audvis_filt-
↪0-40_raw.fif...
    Read a total of 4 projection items:
        PCA-v1 (1 x 102)  idle
        PCA-v2 (1 x 102)  idle
        PCA-v3 (1 x 102)  idle
        Average EEG reference (1 x 60)  idle
    Range : 6450 ... 48149 =     42.956 ...   320.665 secs
Ready.
Reading 0 ... 41699  =      0.000 ...   277.709 secs...
Filtering raw data in 1 contiguous segment
Setting up high-pass filter at 2 Hz

IIR filter parameters
---------------------
Butterworth highpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 8 (effective, after forward-backward)
- Cutoff at 2.00 Hz: -6.02 dB

Removing projector <Projection | PCA-v1, active : False, n_channels : 102>
Removing projector <Projection | PCA-v2, active : False, n_channels : 102>
Removing projector <Projection | PCA-v3, active : False, n_channels : 102>
Removing projector <Projection | Average EEG reference, active : False, n_channels : 60>
```

Decoding with Xdawn + MDM

```
n_components = 3  # pick some components

# Define a monte-carlo cross-validation generator (reduce variance):
cv = KFold(n_splits=10, shuffle=True, random_state=42)
pr = np.zeros(len(labels))
epochs_data = epochs.get_data()

print("Multiclass classification with XDAWN + MDM")

clf = make_pipeline(XdawnCovariances(n_components), MDM())

for train_idx, test_idx in cv.split(epochs_data):
    y_train, y_test = labels[train_idx], labels[test_idx]

    clf.fit(epochs_data[train_idx], y_train)
    pr[test_idx] = clf.predict(epochs_data[test_idx])

print(classification_report(labels, pr))
```

```
Multiclass classification with XDAWN + MDM
              precision    recall  f1-score   support

           1       0.89      0.93      0.91        72
           2       0.90      0.89      0.90        73
```

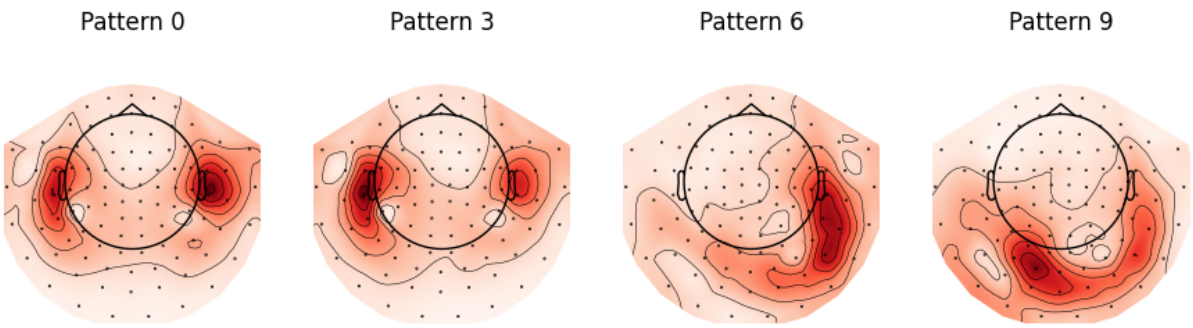
(continues on next page)

(continued from previous page)

```
              3       0.92      0.96      0.94        73
              4       0.97      0.90      0.93        70

       accuracy                           0.92       288
      macro avg       0.92      0.92      0.92       288
   weighted avg       0.92      0.92      0.92       288
```

plot the spatial patterns

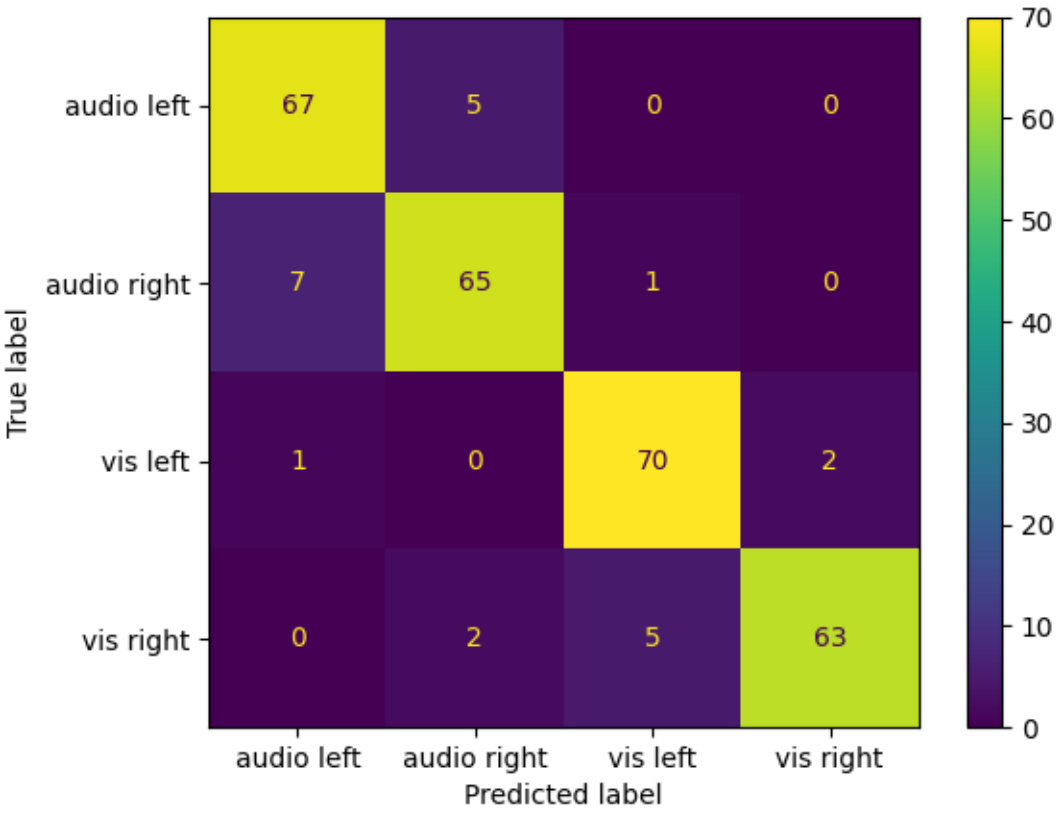
```
xd = XdawnCovariances(n_components)
xd.fit(epochs_data, labels)

info = evoked.copy().resample(1).info  # make it 1Hz for plotting
patterns = mne.EvokedArray(
    data=xd.Xd_.patterns_.T, info=info
)
patterns.plot_topomap(
    times=[0, n_components, 2 * n_components, 3 * n_components],
    ch_type="grad",
    colorbar=False,
    size=1.5,
    time_format="Pattern %d"
)
```




```
<MNEFigure size 900x287.5 with 4 Axes>
```

plot the confusion matrix

```
names = ["audio left", "audio right", "vis left", "vis right"]
cm = confusion_matrix(labels, pr)
ConfusionMatrixDisplay(cm, display_labels=names).plot()
plt.show()
```



Total running time of the script: (0 minutes 12.635 seconds)

4.8.2 Classification of SSVEP

Using Riemannian geometry for classifying steady-state visually evoked potentials (SSVEP).

Offline SSVEP-based BCI Multiclass Prediction

Building extended covariance matrices for SSVEP-based BCI. The obtained matrices are shown. A Minimum Distance to Mean classifier is trained to predict a 4-class problem for an offline setup.

```
# Authors: Sylvain Chevallier <sylvain.chevallier@uvsq.fr>,
# Emmanuel Kalunga, Quentin Barthélemy, David Ojeda
#
# License: BSD (3-clause)

import numpy as np
import matplotlib.pyplot as plt

from mne import find_events, Epochs
from mne.io import Raw
from sklearn.model_selection import cross_val_score, RepeatedKFold
```

(continues on next page)

(continued from previous page)

```
from pyriemann.estimation import BlockCovariances
from pyriemann.utils.mean import mean_riemann
from pyriemann.classification import MDM
from helpers.ssvep_helpers import download_data, extend_signal
```

Loading EEG data

The data are loaded through a MNE loader

```
# Download data
destination = download_data(subject=12, session=1)
# Read data in MNE Raw and numpy format
raw = Raw(destination, preload=True, verbose='ERROR')
events = find_events(raw, shortest_event=0, verbose=False)
raw = raw.pick_types(eeg=True)

event_id = {'13 Hz': 2, '17 Hz': 4, '21 Hz': 3, 'resting-state': 1}
sfreq = int(raw.info['sfreq'])
eeg_data = raw.get_data()
```

```
Using default location ~/mne_data for ssvep...

  0%|                                                     | 0.00/3.33M [00:00<?, ?B/s]
  1%|                                                    | 31.7k/3.33M [00:00<00:15, 215kB/s]
  2%|                                                    | 64.5k/3.33M [00:00<00:14, 219kB/s]
  4%|                                                    | 130k/3.33M [00:00<00:09, 321kB/s]
  6%|                                                   | 212k/3.33M [00:00<00:07, 413kB/s]
  8%|                                                   | 278k/3.33M [00:00<00:07, 423kB/s]
 10%|                                                   | 343k/3.33M [00:00<00:06, 429kB/s]
 12%|                                                  | 409k/3.33M [00:01<00:06, 434kB/s]
 14%|                                                  | 474k/3.33M [00:01<00:06, 436kB/s]
 16%|                                                 | 523k/3.33M [00:01<00:06, 404kB/s]
 17%|                                                 | 572k/3.33M [00:01<00:07, 382kB/s]
 19%|                                                | 638k/3.33M [00:01<00:06, 400kB/s]
 21%|                                               | 703k/3.33M [00:01<00:06, 412kB/s]
 23%|                                               | 769k/3.33M [00:01<00:06, 421kB/s]
 25%|                                              | 835k/3.33M [00:02<00:05, 427kB/s]
 27%|                                             | 900k/3.33M [00:02<00:05, 431kB/s]
 29%|                                            | 966k/3.33M [00:02<00:05, 434kB/s]
 31%|                                           | 1.03M/3.33M [00:02<00:05, 435kB/s]
 33%|                                          | 1.10M/3.33M [00:02<00:05, 437kB/s]
 35%|                                         | 1.16M/3.33M [00:02<00:04, 438kB/s]
 37%|                                         | 1.23M/3.33M [00:02<00:04, 438kB/s]
 39%|                                        | 1.29M/3.33M [00:03<00:04, 439kB/s]
 41%|                                       | 1.36M/3.33M [00:03<00:04, 440kB/s]
 43%|                                      | 1.42M/3.33M [00:03<00:04, 440kB/s]
 45%|                                      | 1.49M/3.33M [00:03<00:04, 439kB/s]
 47%|                                     | 1.56M/3.33M [00:03<00:04, 440kB/s]
 49%|                                    | 1.62M/3.33M [00:03<00:03, 441kB/s]
```

(continues on next page)

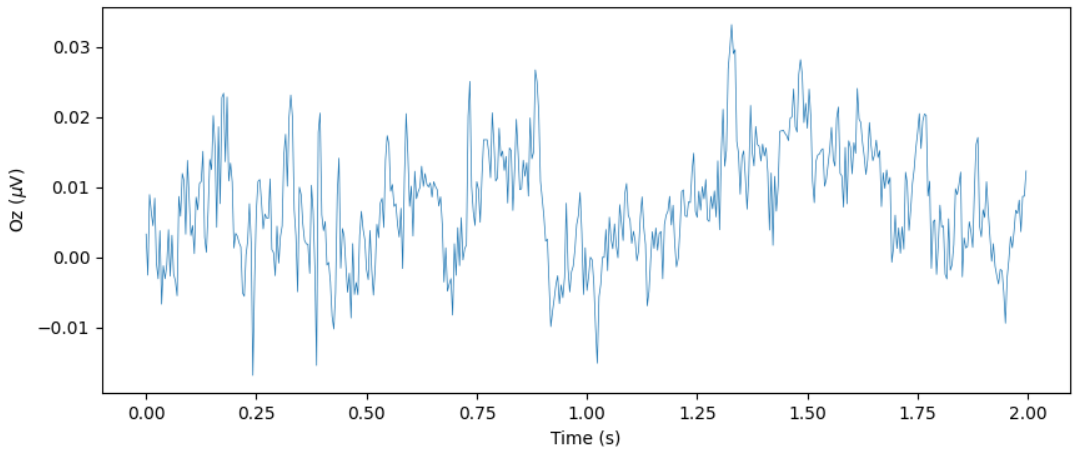
(continued from previous page)

```
 51%|                   | 1.69M/3.33M [00:04<00:03, 441kB/s]
 53%|                   | 1.75M/3.33M [00:04<00:03, 440kB/s]
 55%|                  | 1.82M/3.33M [00:04<00:03, 441kB/s]
 57%|                 | 1.88M/3.33M [00:04<00:03, 440kB/s]
 59%|                | 1.95M/3.33M [00:04<00:03, 440kB/s]
 61%|                | 2.01M/3.33M [00:04<00:02, 440kB/s]
 62%|               | 2.08M/3.33M [00:04<00:02, 440kB/s]
 64%|              | 2.15M/3.33M [00:05<00:02, 440kB/s]
 66%|             | 2.21M/3.33M [00:05<00:02, 440kB/s]
 68%|             | 2.28M/3.33M [00:05<00:02, 440kB/s]
 70%|            | 2.34M/3.33M [00:05<00:02, 441kB/s]
 72%|           | 2.41M/3.33M [00:05<00:02, 440kB/s]
 74%|           | 2.46M/3.33M [00:05<00:02, 408kB/s]
 75%|          | 2.51M/3.33M [00:05<00:02, 385kB/s]
 77%|         | 2.57M/3.33M [00:06<00:01, 401kB/s]
 79%|         | 2.64M/3.33M [00:06<00:01, 414kB/s]
 81%|        | 2.70M/3.33M [00:06<00:01, 421kB/s]
 83%|       | 2.77M/3.33M [00:06<00:01, 427kB/s]
 85%|      | 2.83M/3.33M [00:06<00:01, 431kB/s]
 87%|      | 2.90M/3.33M [00:06<00:00, 434kB/s]
 89%|     | 2.96M/3.33M [00:06<00:00, 436kB/s]
 91%|    | 3.01M/3.33M [00:07<00:00, 405kB/s]
 93%|   | 3.08M/3.33M [00:07<00:00, 415kB/s]
 94%|   | 3.13M/3.33M [00:07<00:00, 390kB/s]
 96%|  | 3.19M/3.33M [00:07<00:00, 405kB/s]
 98%|| 3.26M/3.33M [00:07<00:00, 416kB/s]
100%|| 3.32M/3.33M [00:07<00:00, 423kB/s]
  0%|                                                  | 0.00/3.33M [00:00<?, ?B/s]
100%|| 3.33M/3.33M [00:00<00:00, 10.1GB/s]
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/examples/SSVEP/
↪helpers/ssvep_helpers.py:79: RuntimeWarning: Setting non-standard config type: "MNE_
↪DATASETS_SSVEPEXO_PATH"
  data_path = fetch_dataset(dataset_params, force_update=True)
```

Visualization of raw EEG data

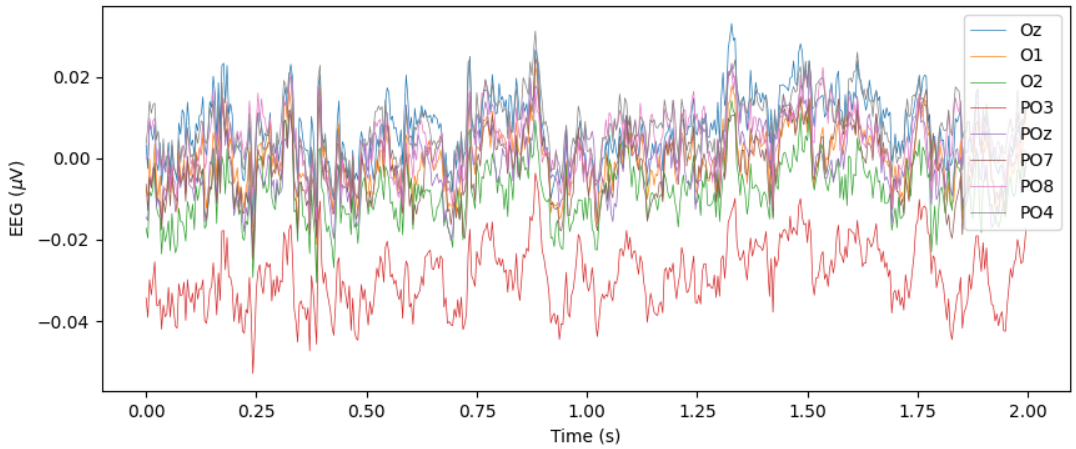
Plot few seconds of signal from the *Oz* electrode using matplotlib

```
n_seconds = 2
time = np.linspace(0, n_seconds, n_seconds * sfreq,
                   endpoint=False)[np.newaxis, :]
plt.figure(figsize=(10, 4))
plt.plot(time.T, eeg_data[np.array(raw.ch_names) == 'Oz', :n_seconds*sfreq].T,
         color='C0', lw=0.5)
plt.xlabel("Time (s)")
plt.ylabel(r"Oz ($\mu$V)")
plt.show()
```



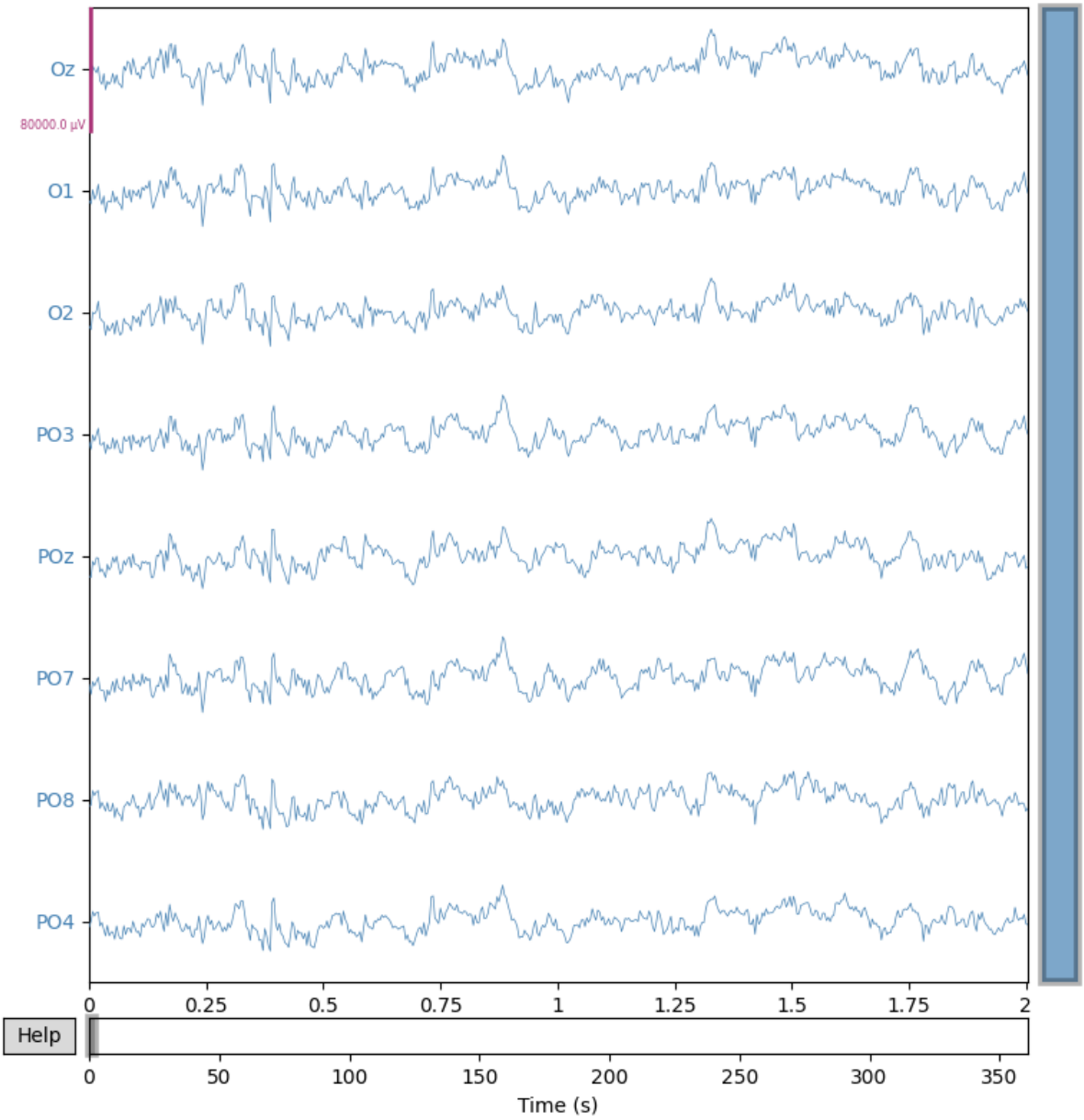
And of all electrodes:

```
plt.figure(figsize=(10, 4))
for ch_idx, ch_name in enumerate(raw.ch_names):
    plt.plot(time.T, eeg_data[ch_idx, :n_seconds*sfreq].T, lw=0.5,
             label=ch_name)
plt.xlabel("Time (s)")
plt.ylabel(r"EEG ($\mu$V)")
plt.legend(loc='upper right')
plt.show()
```



With MNE, it is much easier to visualize the data

```
raw.plot(duration=n_seconds, start=0, n_channels=8, scalings={'eeg': 4e-2},
         color={'eeg': 'steelblue'})
```




```
Using matplotlib as 2D backend.

<MNEBrowseFigure size 800x800 with 4 Axes>
```

Extended signals for spatial covariance

Using the approach proposed by[1], the SSVEP signal is extended to include the filtered signals for each stimulation frequency. We stack the filtered signals to build an extended signal.

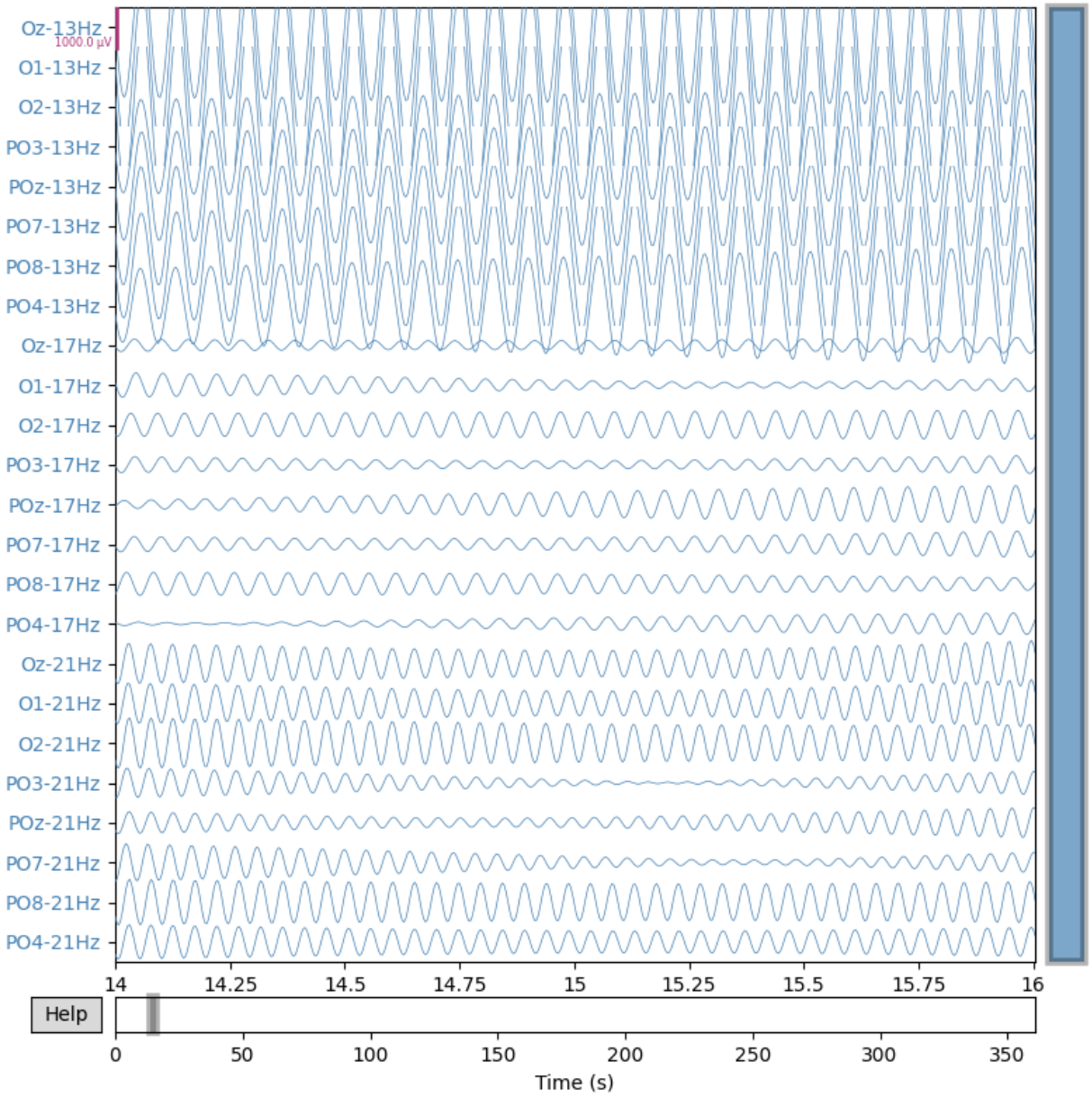
```
# We stack the filtered signals to build an extended signal
frequencies = [13, 17, 21]
freq_band = 0.1
raw_ext = extend_signal(raw, frequencies, freq_band)
```

```
Creating RawArray with float64 data, n_channels=24, n_times=92384
    Range : 0 ... 92383 =      0.000 ...   360.871 secs
Ready.
```

Plot the extended signal

```
raw_ext.plot(duration=n_seconds, start=14, n_channels=24,
             scalings={'eeg': 5e-4}, color={'eeg': 'steelblue'})
```

[1] A New generation of Brain-Computer Interface Based on Riemannian Geometry M. Congedo, A. Barachant, A. Andreev. Research report, 2013.




```
<MNEBrowseFigure size 800x800 with 4 Axes>
```

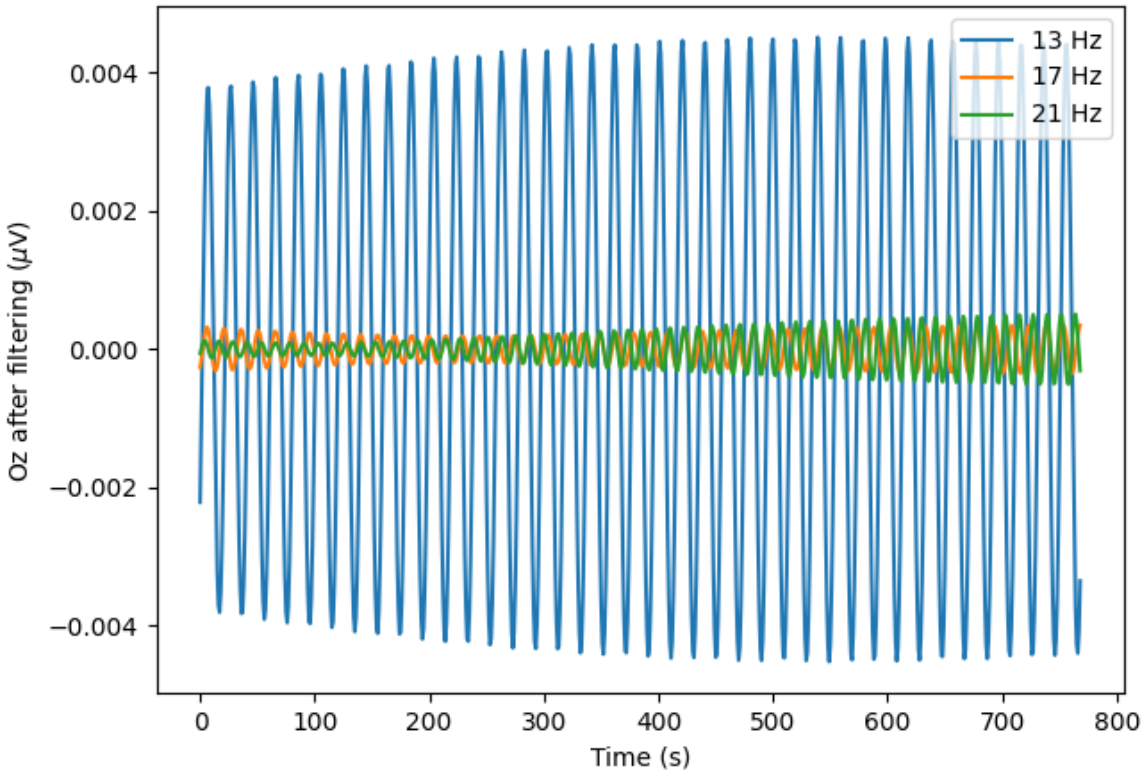
Building Epochs and plotting 3 s of the signal from electrode Oz for a trial

```
epochs = Epochs(raw_ext, events, event_id, tmin=2, tmax=5, baseline=None)

n_seconds = 3
time = np.linspace(0, n_seconds, n_seconds * sfreq,
                   endpoint=False)[np.newaxis, :]
channels = range(0, len(raw_ext.ch_names), len(raw.ch_names))
plt.figure(figsize=(7, 5))
for f, c in zip(frequencies, channels):
    plt.plot(epochs.get_data()[5, c, :].T, label=str(int(f))+' Hz')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Time (s)")
plt.ylabel(r"Oz after filtering ($\mu$V)")
plt.legend(loc='upper right')
plt.show()
```



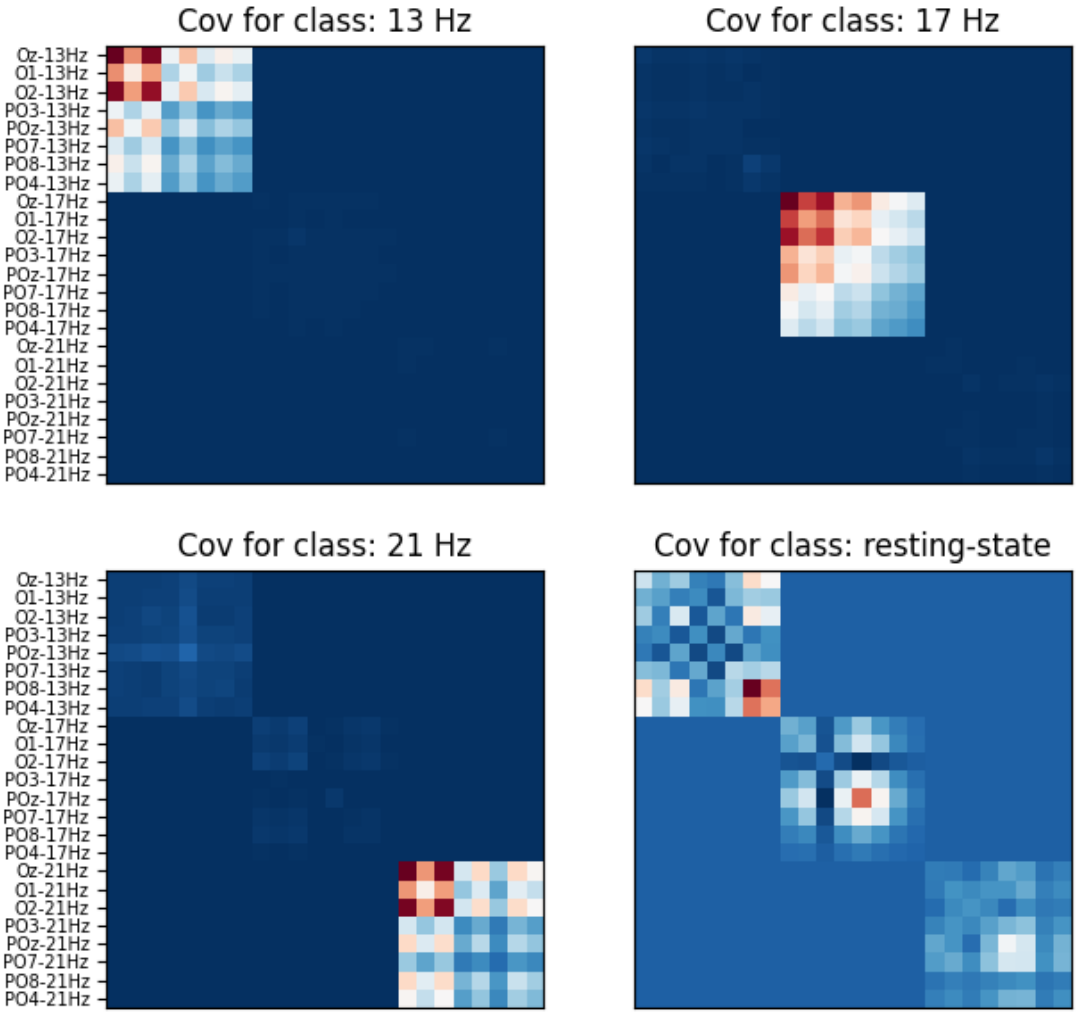

```
Not setting metadata
32 matching events found
No baseline correction applied
0 projection items activated
Using data from preloaded Raw for 32 events and 769 original time points ...
0 bad epochs dropped
Using data from preloaded Raw for 32 events and 769 original time points ...
Using data from preloaded Raw for 32 events and 769 original time points ...
```

As it can be seen on this example, the subject is watching the 13Hz stimulation and the EEG activity is showing an increase activity in this frequency band while other frequencies have lower amplitudes.

Spatial covariance for SSVEP

The covariance matrices will be estimated using the Ledoit-Wolf shrinkage estimator on the extended signal.

```
cov_ext_trials = BlockCovariances(estimator='lwf',
                                  block_size=8).transform(epochs.get_data())

# This plot shows an example of a covariance matrix observed for each class:
ch_names = raw_ext.info['ch_names']

plt.figure(figsize=(7, 7))
for i, l in enumerate(event_id):
    ax = plt.subplot(2, 2, i+1)
    plt.imshow(cov_ext_trials[events[:, 2] == event_id[l]][0],
               cmap=plt.get_cmap('RdBu_r'))
    plt.title('Cov for class: '+l)
    plt.xticks([])
    if i == 0 or i == 2:
        plt.yticks(np.arange(len(ch_names)), ch_names)
        ax.tick_params(axis='both', which='major', labelsize=7)
    else:
        plt.yticks([])
plt.show()
```



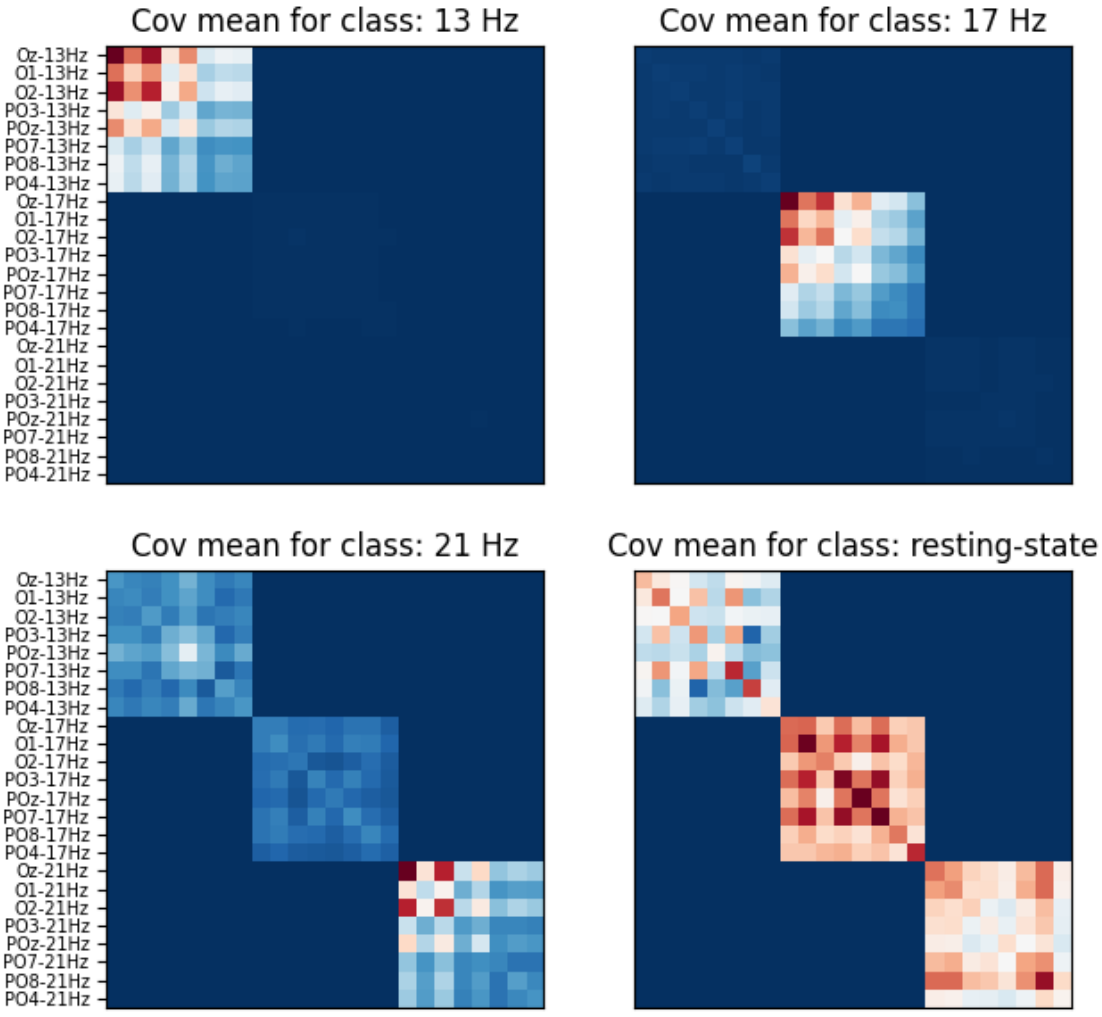

```
Using data from preloaded Raw for 32 events and 769 original time points ...
```

It appears clearly that each class yields a different structure of the covariance matrix. Each stimulation (13, 17 and 21 Hz) generating higher covariance values for EEG signal filtered at the proper bandwith and no activation at all for the other bandwiths. The resting state, where the subject focus on the center of the display and far from all blinking stimulus, shows an activity with higher correlation in the 13Hz frequency and lower but still visible activity in the other bandwiths.

Classify with MDM

Plotting mean of each class

```
cov_centers = np.empty((len(event_id), 24, 24))
for i, l in enumerate(event_id):
    cov_centers[i] = mean_riemann(cov_ext_trials[events[:, 2] == event_id[l]])

plt.figure(figsize=(7, 7))
for i, l in enumerate(event_id):
    ax = plt.subplot(2, 2, i+1)
    plt.imshow(cov_centers[i], cmap=plt.get_cmap('RdBu_r'))
    plt.title('Cov mean for class: '+l)
    plt.xticks([])
    if i == 0 or i == 2:
        plt.yticks(np.arange(len(ch_names)), ch_names)
        ax.tick_params(axis='both', which='major', labelsize=7)
    else:
        plt.yticks([])
plt.show()
```



Minimum distance to mean is a simple and robust algorithm for BCI decoding. It reproduces results of[2] for the first session of subject 12.

```
print("Number of trials: {}".format(len(cov_ext_trials)))

cv = RepeatedKFold(n_splits=2, n_repeats=10, random_state=42)
mdm = MDM(metric=dict(mean='riemann', distance='riemann'))
scores = cross_val_score(mdm, cov_ext_trials, events[:, 2], cv=cv, n_jobs=1)
print("MDM accuracy: {:.2f}% +/- {:.2f}".format(np.mean(scores)*100,
                                                np.std(scores)*100))
# The obtained results are 80.62% +/- 16.29 for this session, with a repeated
# 10-fold validation.
```

[2] Review of Riemannian distances and divergences, applied to SSVEP-based BCI S. Chevallier, E. K. Kalunga, Q. Barthélemy, E. Monacelli. Neuroinformatics, Springer, 2021, 19 (1), pp.93-106

```
Number of trials: 32
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
MDM accuracy: 80.94% +/- 16.23
```

References

Total running time of the script: (0 minutes 16.351 seconds)

Visualization of SSVEP-based BCI Classification in Tangent Space

Project extended covariance matrices of SSVEP-based BCI in the tangent space, using principal geodesic analysis (PGA).

You should have a look to "Offline SSVEP-based BCI Multiclass Prediction" before this example.

```
# Authors: Quentin Barthélemy, Emmanuel Kalunga and Sylvain Chevallier
#
# License: BSD (3-clause)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

from mne import find_events, Epochs, make_fixed_length_epochs
from mne.io import Raw
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import PCA

from pyriemann.estimation import BlockCovariances
from pyriemann.classification import MDM
from pyriemann.tangentspace import TangentSpace
from pyriemann.utils.viz import _add_alpha
from helpers.ssvep_helpers import download_data, extend_signal
```

```
clabel = ['resting-state', '13 Hz', '17 Hz', '21 Hz']
clist = plt.cm.viridis(np.array([0, 1, 2, 3])/3)
cmap = "viridis"


def plot_pga(ax, data, labels, centers):
    sc = ax.scatter(data[:, 0], data[:, 1], c=labels, marker='P', cmap=cmap)
    ax.scatter(
        centers[:, 0], centers[:, 1], c=clist, marker='o', s=100, cmap=cmap
        )
    ax.set(xlabel='PGA, 1st axis', ylabel='PGA, 2nd axis')
    for i in range(len(clabel)):
        ax.scatter([], [], color=clist[i], marker='o', s=50, label=clabel[i])
    ax.legend(loc='upper right')
    return sc
```

Load EEG and extract covariance matrices for SSVEP

```
frequencies = [13, 17, 21]
freq_band = 0.1
events_id = {'13 Hz': 2, '17 Hz': 4, '21 Hz': 3, 'resting-state': 1}

duration = 2.5    # duration of epochs
interval = 0.25   # interval between successive epochs for online processing

# Subject 12: first 4 sessions for training, last session for test

# Training set
raw = Raw(download_data(subject=12, session=1), preload=True, verbose=False)
events = find_events(raw, shortest_event=0, verbose=False)
raw = raw.pick_types(eeg=True)
ch_count = len(raw.info['ch_names'])
raw_ext = extend_signal(raw, frequencies, freq_band)
epochs = Epochs(
    raw_ext, events, events_id, tmin=2, tmax=5, baseline=None, verbose=False)
x_train = BlockCovariances(
    estimator='lwf', block_size=ch_count).transform(epochs.get_data())
y_train = events[:, 2]

# Testing set
raw = Raw(download_data(subject=12, session=4), preload=True, verbose=False)
raw = raw.pick_types(eeg=True)
raw_ext = extend_signal(raw, frequencies, freq_band)
epochs = make_fixed_length_epochs(
    raw_ext, duration=duration, overlap=duration - interval, verbose=False)
x_test = BlockCovariances(
    estimator='lwf', block_size=ch_count).transform(epochs.get_data())
```

```
Creating RawArray with float64 data, n_channels=24, n_times=92384
    Range : 0 ... 92383 =      0.000 ...   360.871 secs
Ready.
Using data from preloaded Raw for 32 events and 769 original time points ...
0 bad epochs dropped

  0%|                                                          | 0.00/5.35M [00:00<?, ?B/s]
  1%|                                       | 31.7k/5.35M [00:00<01:08, 77.3kB/s]
  1%|                                        | 64.5k/5.35M [00:00<00:41, 126kB/s]
  2%|                                         | 130k/5.35M [00:00<00:22, 230kB/s]
  3%|                                        | 179k/5.35M [00:00<00:19, 262kB/s]
  5%|                                        | 245k/5.35M [00:01<00:16, 319kB/s]
  5%|                                        | 281k/5.35M [00:01<00:17, 295kB/s]
  6%|                                       | 343k/5.35M [00:01<00:15, 334kB/s]
  8%|                                       | 409k/5.35M [00:01<00:13, 367kB/s]
  9%|                                      | 474k/5.35M [00:01<00:12, 390kB/s]
 10%|                                      | 540k/5.35M [00:01<00:11, 405kB/s]
 11%|                                     | 589k/5.35M [00:01<00:12, 383kB/s]
 12%|                                     | 654k/5.35M [00:02<00:11, 399kB/s]
 13%|                                    | 720k/5.35M [00:02<00:11, 411kB/s]
```

(continues on next page)

(continued from previous page)

```
 15%|                                  | 785k/5.35M [00:02<00:10, 419kB/s]
 16%|                                 | 851k/5.35M [00:02<00:10, 426kB/s]
 17%|                                 | 916k/5.35M [00:02<00:10, 430kB/s]
 18%|                                | 982k/5.35M [00:02<00:10, 434kB/s]
 20%|                               | 1.05M/5.35M [00:02<00:09, 435kB/s]
 21%|                               | 1.11M/5.35M [00:03<00:09, 438kB/s]
 22%|                              | 1.16M/5.35M [00:03<00:10, 405kB/s]
 23%|                              | 1.23M/5.35M [00:03<00:09, 416kB/s]
 24%|                             | 1.29M/5.35M [00:03<00:09, 423kB/s]
 26%|                             | 1.38M/5.35M [00:03<00:08, 461kB/s]
 27%|                            | 1.44M/5.35M [00:03<00:08, 455kB/s]
 28%|                            | 1.51M/5.35M [00:03<00:08, 450kB/s]
 29%|                           | 1.57M/5.35M [00:04<00:08, 448kB/s]
 30%|                           | 1.62M/5.35M [00:04<00:09, 413kB/s]
 32%|                           | 1.69M/5.35M [00:04<00:08, 422kB/s]
 32%|                          | 1.73M/5.35M [00:04<00:09, 382kB/s]
 34%|                          | 1.80M/5.35M [00:04<00:08, 412kB/s]
 35%|                         | 1.87M/5.35M [00:04<00:08, 420kB/s]
 36%|                         | 1.93M/5.35M [00:05<00:08, 427kB/s]
 37%|                        | 2.00M/5.35M [00:05<00:07, 431kB/s]
 39%|                        | 2.06M/5.35M [00:05<00:07, 435kB/s]
 40%|                        | 2.13M/5.35M [00:05<00:07, 436kB/s]
 41%|                       | 2.21M/5.35M [00:05<00:06, 470kB/s]
 43%|                      | 2.28M/5.35M [00:05<00:06, 462kB/s]
 44%|                      | 2.34M/5.35M [00:05<00:06, 456kB/s]
 45%|                      | 2.41M/5.35M [00:06<00:06, 452kB/s]
 46%|                     | 2.47M/5.35M [00:06<00:06, 449kB/s]
 47%|                     | 2.54M/5.35M [00:06<00:06, 446kB/s]
 49%|                    | 2.60M/5.35M [00:06<00:06, 444kB/s]
 50%|                    | 2.67M/5.35M [00:06<00:06, 443kB/s]
 51%|                   | 2.74M/5.35M [00:06<00:05, 443kB/s]
 52%|                   | 2.80M/5.35M [00:06<00:05, 442kB/s]
 54%|                  | 2.87M/5.35M [00:07<00:05, 442kB/s]
 55%|                  | 2.93M/5.35M [00:07<00:05, 441kB/s]
 56%|                 | 3.00M/5.35M [00:07<00:05, 440kB/s]
 57%|                 | 3.06M/5.35M [00:07<00:05, 439kB/s]
 58%|                | 3.13M/5.35M [00:07<00:05, 439kB/s]
 60%|                | 3.19M/5.35M [00:07<00:04, 439kB/s]
 61%|               | 3.26M/5.35M [00:07<00:04, 439kB/s]
 62%|               | 3.32M/5.35M [00:08<00:04, 439kB/s]
 63%|               | 3.39M/5.35M [00:08<00:04, 439kB/s]
 65%|              | 3.46M/5.35M [00:08<00:04, 439kB/s]
 66%|              | 3.52M/5.35M [00:08<00:04, 439kB/s]
 67%|             | 3.59M/5.35M [00:08<00:04, 439kB/s]
 68%|             | 3.65M/5.35M [00:08<00:03, 439kB/s]
 69%|            | 3.72M/5.35M [00:09<00:03, 439kB/s]
 71%|            | 3.78M/5.35M [00:09<00:03, 440kB/s]
 72%|           | 3.85M/5.35M [00:09<00:03, 440kB/s]
 73%|           | 3.93M/5.35M [00:09<00:03, 473kB/s]
 75%|          | 4.00M/5.35M [00:09<00:02, 462kB/s]
 76%|          | 4.06M/5.35M [00:09<00:02, 455kB/s]
 77%|         | 4.13M/5.35M [00:09<00:02, 450kB/s]
```

(continues on next page)

(continued from previous page)

```
 78%|         | 4.19M/5.35M [00:10<00:02, 445kB/s]
 79%|        | 4.24M/5.35M [00:10<00:02, 411kB/s]
 81%|        | 4.31M/5.35M [00:10<00:02, 419kB/s]
 82%|        | 4.37M/5.35M [00:10<00:02, 425kB/s]
 83%|       | 4.44M/5.35M [00:10<00:02, 429kB/s]
 84%|       | 4.50M/5.35M [00:10<00:01, 433kB/s]
 85%|      | 4.57M/5.35M [00:10<00:01, 435kB/s]
 87%|      | 4.64M/5.35M [00:11<00:01, 437kB/s]
 88%|     | 4.70M/5.35M [00:11<00:01, 438kB/s]
 89%|     | 4.77M/5.35M [00:11<00:01, 438kB/s]
 90%|    | 4.83M/5.35M [00:11<00:01, 439kB/s]
 92%|    | 4.90M/5.35M [00:11<00:01, 439kB/s]
 93%|   | 4.96M/5.35M [00:11<00:00, 439kB/s]
 94%|   | 5.03M/5.35M [00:12<00:00, 438kB/s]
 95%|  | 5.09M/5.35M [00:12<00:00, 439kB/s]
 96%|  | 5.16M/5.35M [00:12<00:00, 439kB/s]
 98%|  | 5.23M/5.35M [00:12<00:00, 439kB/s]
 99%|| 5.29M/5.35M [00:12<00:00, 439kB/s]
  0%|                                                | 0.00/5.35M [00:00<?, ?B/s]
100%|| 5.35M/5.35M [00:00<00:00, 13.7GB/s]
Creating RawArray with float64 data, n_channels=24, n_times=148544
    Range : 0 ... 148543 =      0.000 ...   580.246 secs
Ready.
Using data from preloaded Raw for 2312 events and 640 original time points ...
0 bad epochs dropped
```

Classification with minimum distance to mean (MDM)

Classification for a 4-class SSVEP BCI, including resting-state class.

```
print("Number of training trials: {}".format(len(x_train)))

mdm = MDM(metric=dict(mean='riemann', distance='riemann'))
mdm.fit(x_train, y_train)
```

```
Number of training trials: 32

MDM(metric={'distance': 'riemann', 'mean': 'riemann'})
```

Projection in tangent space with principal geodesic analysis (PGA)

Project covariance matrices from the Riemannian manifold into the Euclidean tangent space at the grand average, and apply a principal component analysis (PCA) to obtain an unsupervised dimension reduction[1].

```
pga = make_pipeline(
    TangentSpace(metric="riemann", tsupdate=False),
    PCA(n_components=2)
```

(continues on next page)

[1] Principal geodesic analysis for the study of nonlinear statistics of shape P.T. Fletcher, C. Lu, S. M. Pizer, S. Joshi. IEEE Transactions on Medical Imaging (Volume: 23, Issue: 8, August 2004).

(continued from previous page)

```
)

ts_train = pga.fit_transform(x_train)
ts_means = pga.transform(np.array(mdm.covmeans_))
```

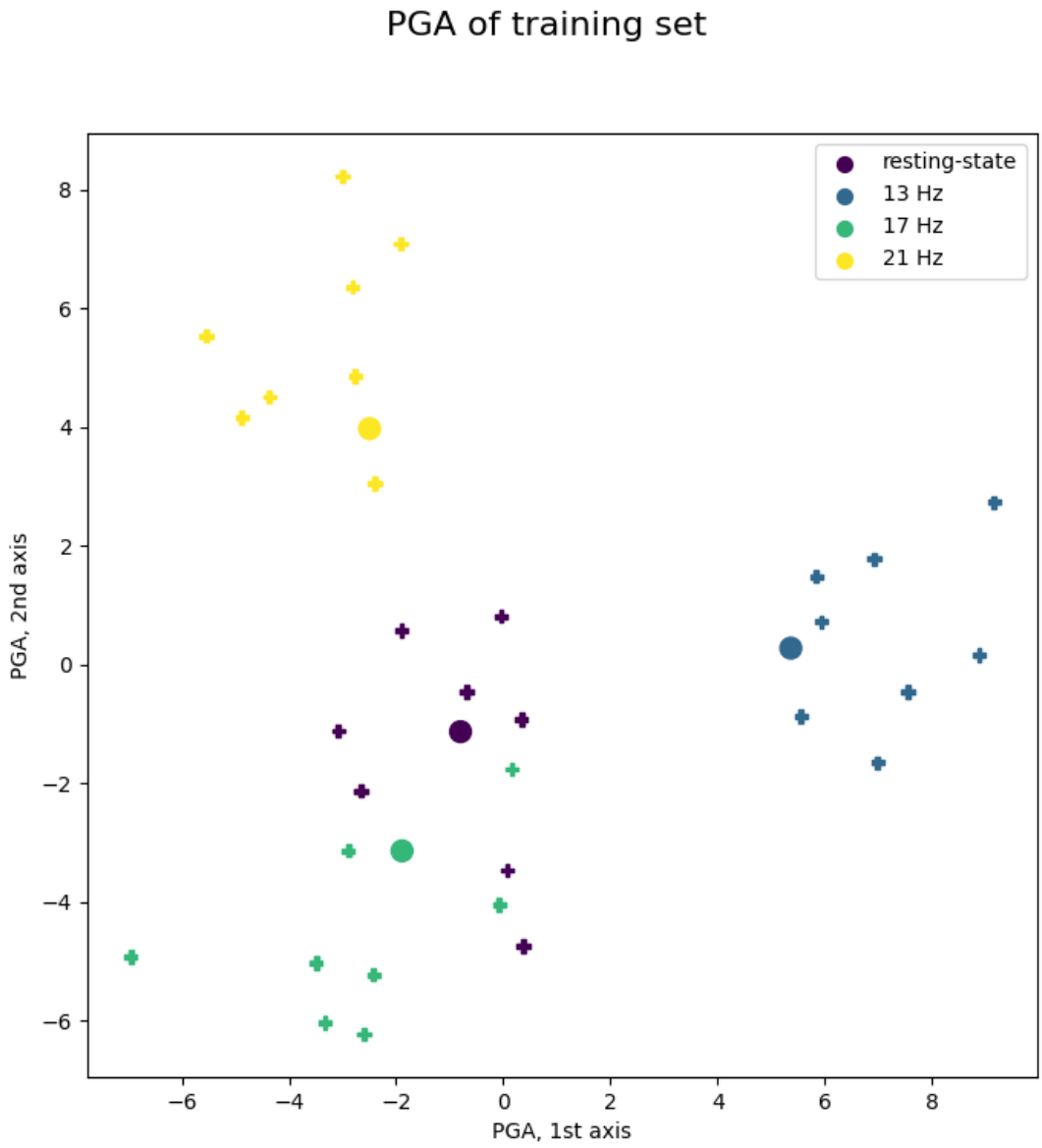
Offline training of MDM visualized by PGA

These figures show the trajectory on the tangent space taken by covariance matrices during a 4-class SSVEP experiment, and how they are classified epoch by epoch.

This figure reproduces Fig 3(c) of reference[2], showing training trials of best subject.

```
fig, ax = plt.subplots(figsize=(8, 8))
fig.suptitle('PGA of training set', fontsize=16)
plot_pga(ax, ts_train, y_train, ts_means)
plt.show()
```

[2] Online SSVEP-based BCI using Riemannian geometry E. K. Kalunga, S. Chevallier, Q. Barthélemy, K. Djouani, E. Monacelli, Y. Hamam. Neurocomputing, Elsevier, 2016, 191, pp.55-68.



Online classification by MDM visualized by PGA

This figure reproduces Fig 6 of reference[Page 48, 2], with an animation to imitate an online acquisition, processing and classification of EEG time-series.

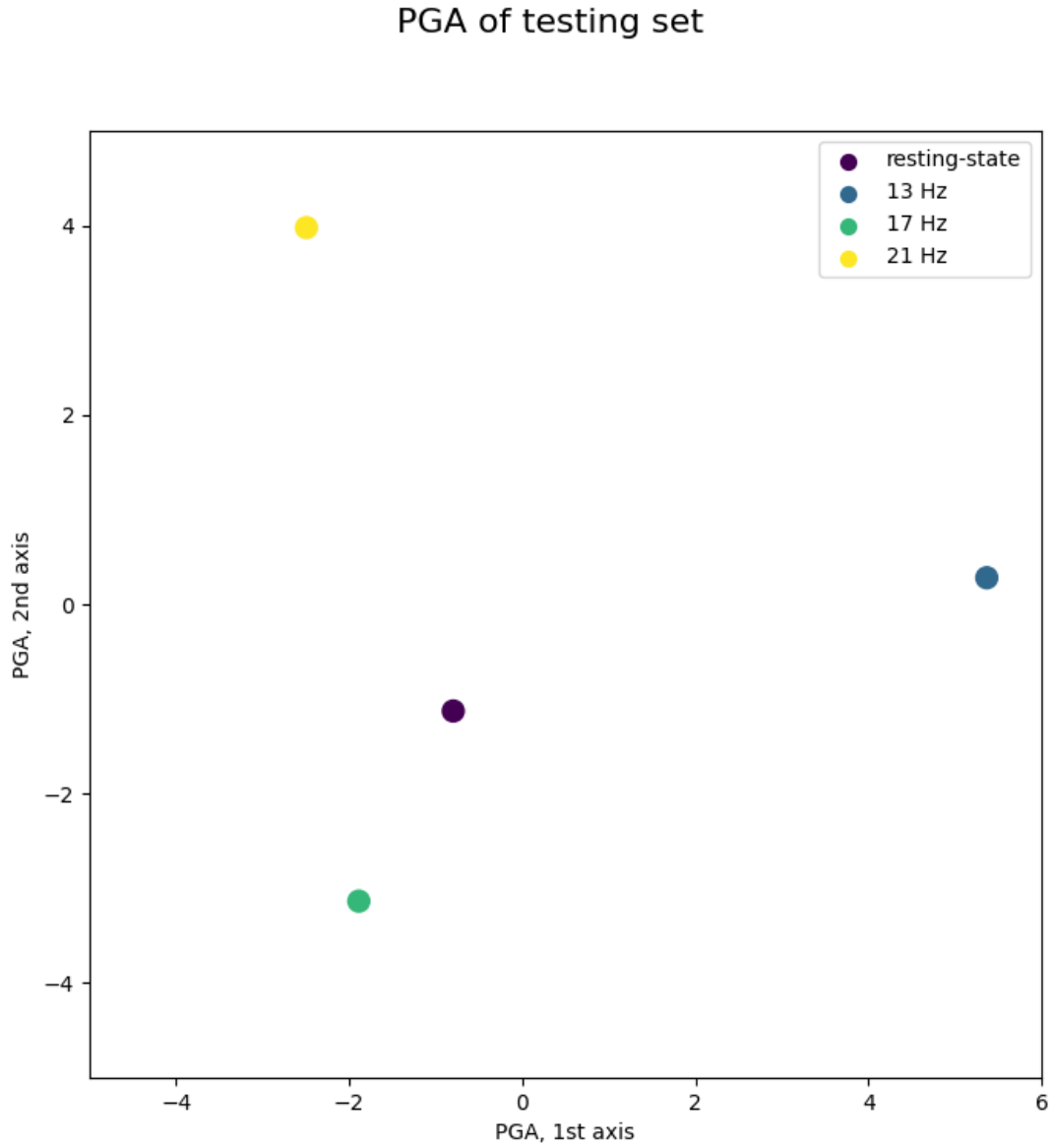
Warning:[Page 48, 2] uses a curved based online classification, while a single trial classification is used here.

```
# Prepare data for online classification
test_visu = 50      # nb of matrices to display simultaneously
colors, ts_visu = [], np.empty([0, 2])
alphas = np.linspace(0, 1, test_visu)

fig, ax = plt.subplots(figsize=(8, 8))
```

(continues on next page)

(continued from previous page)

```
fig.suptitle('PGA of testing set', fontsize=16)
pl = plot_pga(ax, ts_visu, colors, ts_means)
pl.axes.set_xlim(-5, 6)
pl.axes.set_ylim(-5, 5)
```




```
(-5.0, 5.0)
```

```
# Prepare animation for online classification
def online_classify(t):
    global colors, ts_visu

    # Online classification
```

(continues on next page)

(continued from previous page)

```
    y = mdm.predict(x_test[np.newaxis, t])
    color = clist[int(y[0] - 1)]
    ts_test = pga.transform(x_test[np.newaxis, t])

    # Update data
    colors.append(color)
    ts_visu = np.vstack((ts_visu, ts_test))
    if len(ts_visu) > test_visu:
        colors.pop(0)
        ts_visu = ts_visu[1:]
    colors = _add_alpha(colors, alphas)

    # Update plot
    pl.set_offsets(np.c_[ts_visu[:, 0], ts_visu[:, 1]])
    pl.set_color(colors)
    return pl


interval_display = 1.0  # can be changed for a slower display

visu = FuncAnimation(fig, online_classify,
                     frames=range(0, len(x_test)),
                     interval=interval_display, blit=False, repeat=False)
```

```
# Plot online classification

# Plot complete visu: a dynamic display is required
plt.show()

# Plot only 10s, for animated documentation
try:
    from IPython.display import HTML
except ImportError:
    raise ImportError("Install IPython to plot animation in documentation")

plt.rcParams["animation.embed_limit"] = 10
HTML(visu.to_jshtml(fps=5, default_mode='loop'))
```

```
Animation size has reached 10525072 bytes, exceeding the limit of 10485760.0. If you're␣
↪sure you want a larger animation embedded, set the animation.embed_limit rc parameter␣
↪to a larger value (in MB). This and further frames will be dropped.
```

References

Total running time of the script: (4 minutes 10.612 seconds)

4.8.3 Artifact management

Using Riemannian geometry to detect, reject or correct artifacts.

Artifact Correction by AJDC-based Blind Source Separation

Blind source separation (BSS) based on approximate joint diagonalization of Fourier cospectra (AJDC), applied to artifact correction of EEG[1].

```
# Authors: Quentin Barthélemy & David Ojeda.
# EEG signal kindly shared by Marco Congedo.
#
# License: BSD (3-clause)

import gzip
import numpy as np
from scipy.signal import welch
from matplotlib import pyplot as plt

from mne import create_info
from mne.io import RawArray
from mne.viz import plot_topomap
from mne.preprocessing import ICA

from pyriemann.spatialfilters import AJDC
from pyriemann.utils.viz import plot_cospectra
```

```
def read_header(fname):
    """Read the header of sample-blinks.txt"""
    with gzip.open(fname, 'rt') as f:
        content = f.readline().split()
        return content[:-1], int(content[-1])
```

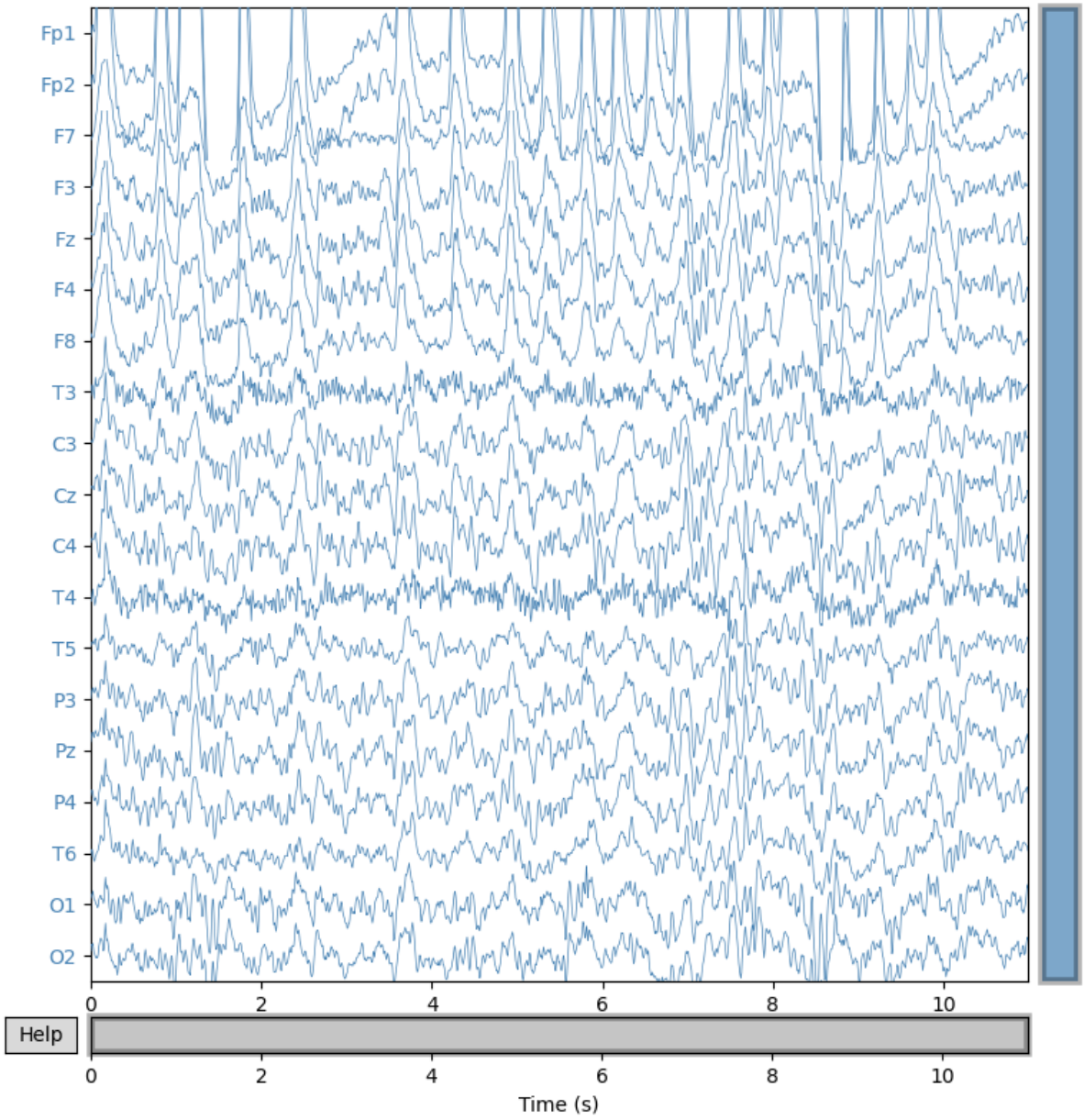
Load EEG data

```
fname = '../data/sample-blinks.txt.gz'
signal_raw = np.loadtxt(fname, skiprows=1).T
ch_names, sfreq = read_header(fname)
ch_count = len(ch_names)
duration = signal_raw.shape[1] / sfreq
```

[1] Online denoising of eye-blinks in electroencephalography Q. Barthélemy, L. Mayaud, Y. Renard, D. Kim, S.-W. Kang, J. Gunkelman and M. Congedo. Clinical Neurophysiology, Elsevier Masson, 2017, 47 (5-6), pp.371-391

Channel space

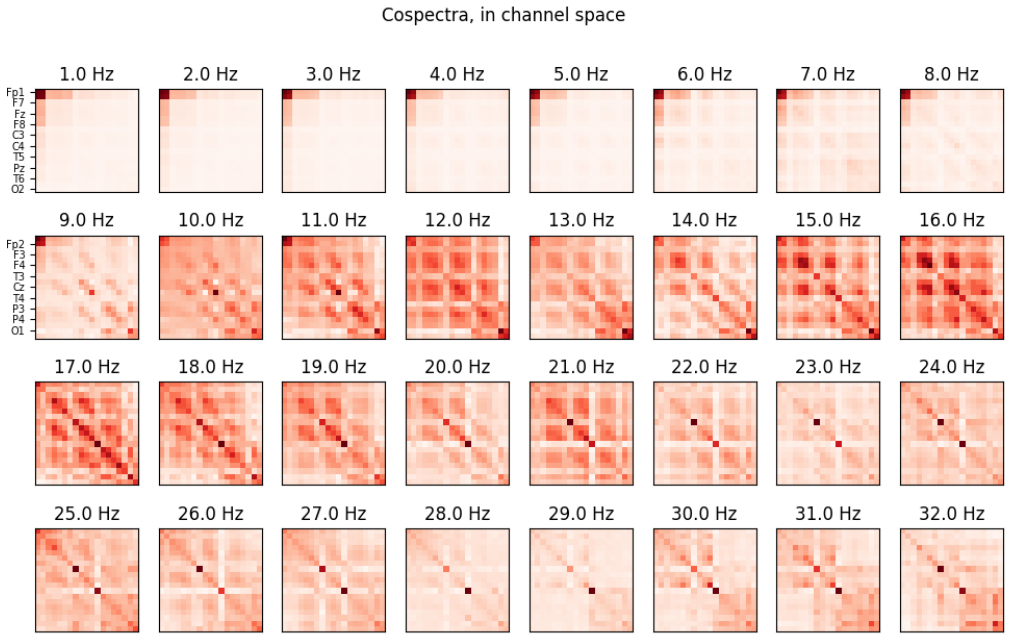
```
# Plot signal X
ch_info = create_info(ch_names=ch_names, ch_types=['eeg'] * ch_count,
                      sfreq=sfreq)
ch_info.set_montage('standard_1020')
signal = RawArray(signal_raw, ch_info, verbose=False)
signal.plot(duration=duration, start=0, n_channels=ch_count,
            scalings={'eeg': 3e1}, color={'eeg': 'steelblue'},
            title='Original EEG signal', show_scalebars=False)
```




```
<MNEBrowseFigure size 800x800 with 4 Axes>
```

AJDC: Second-Order Statistics (SOS)-based BSS, diagonalizing cospectra

```
# Compute and diagonalize Fourier cospectral matrices between 1 and 32 Hz
window, overlap = sfreq, 0.5
fmin, fmax = 1, 32
ajdc = AJDC(window=window, overlap=overlap, fmin=fmin, fmax=fmax, fs=sfreq,
            dim_red={'max_cond': 100})
ajdc.fit(signal_raw[np.newaxis, np.newaxis, ...])
freqs = ajdc.freqs_

# Plot cospectra in channel space, after trace-normalization by frequency: each
# cospectrum, associated to a frequency, is a covariance matrix
plot_cospectra(ajdc._cosp_channels, freqs, ylabels=ch_names,
               title='Cospectra, in channel space')
```



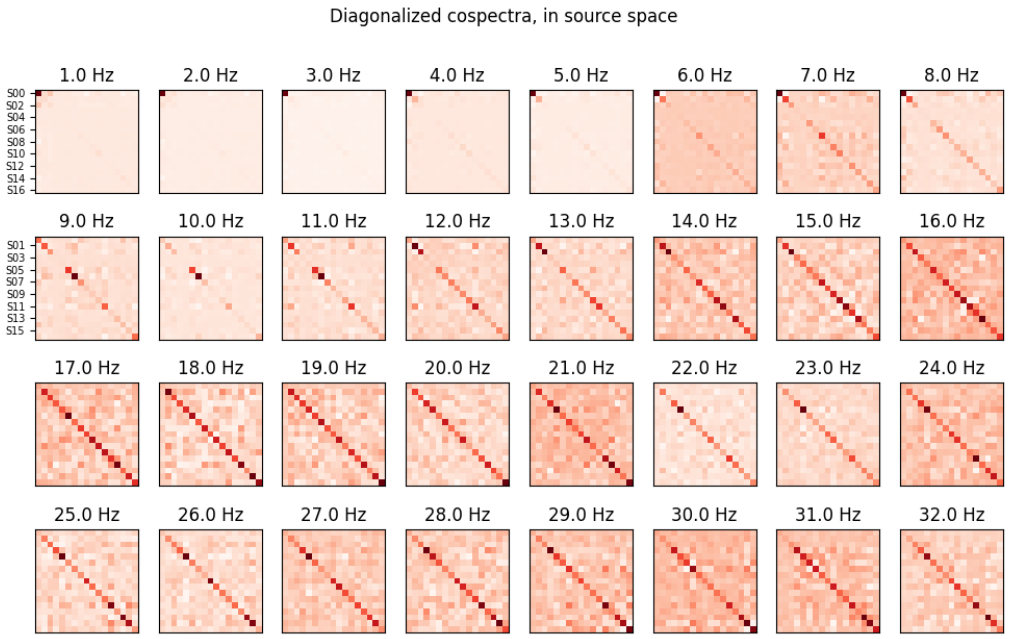

```
Condition numbers:
 array([  1.        ,   2.29766117,   4.09457756,   4.86981696,
         6.09760458,   9.15865458,  13.21748535,  17.74436118,
        26.1024296 ,  27.31744246,  33.78134725,  45.22515539,
        50.61007053,  60.36895283,  73.48533473,  74.73247287,
        92.15600097, 121.30282659, 164.52162547])
Dimension reduction of Whitening on 17 components

<Figure size 1200x700 with 32 Axes>
```

```
# Plot diagonalized cospectra in source space
sr_count = ajdc.n_sources_
sr_names = ['S' + str(s).zfill(2) for s in range(sr_count)]
plot_cospectra(ajdc._cosp_sources, freqs, ylabels=sr_names,
```

(continues on next page)

(continued from previous page)

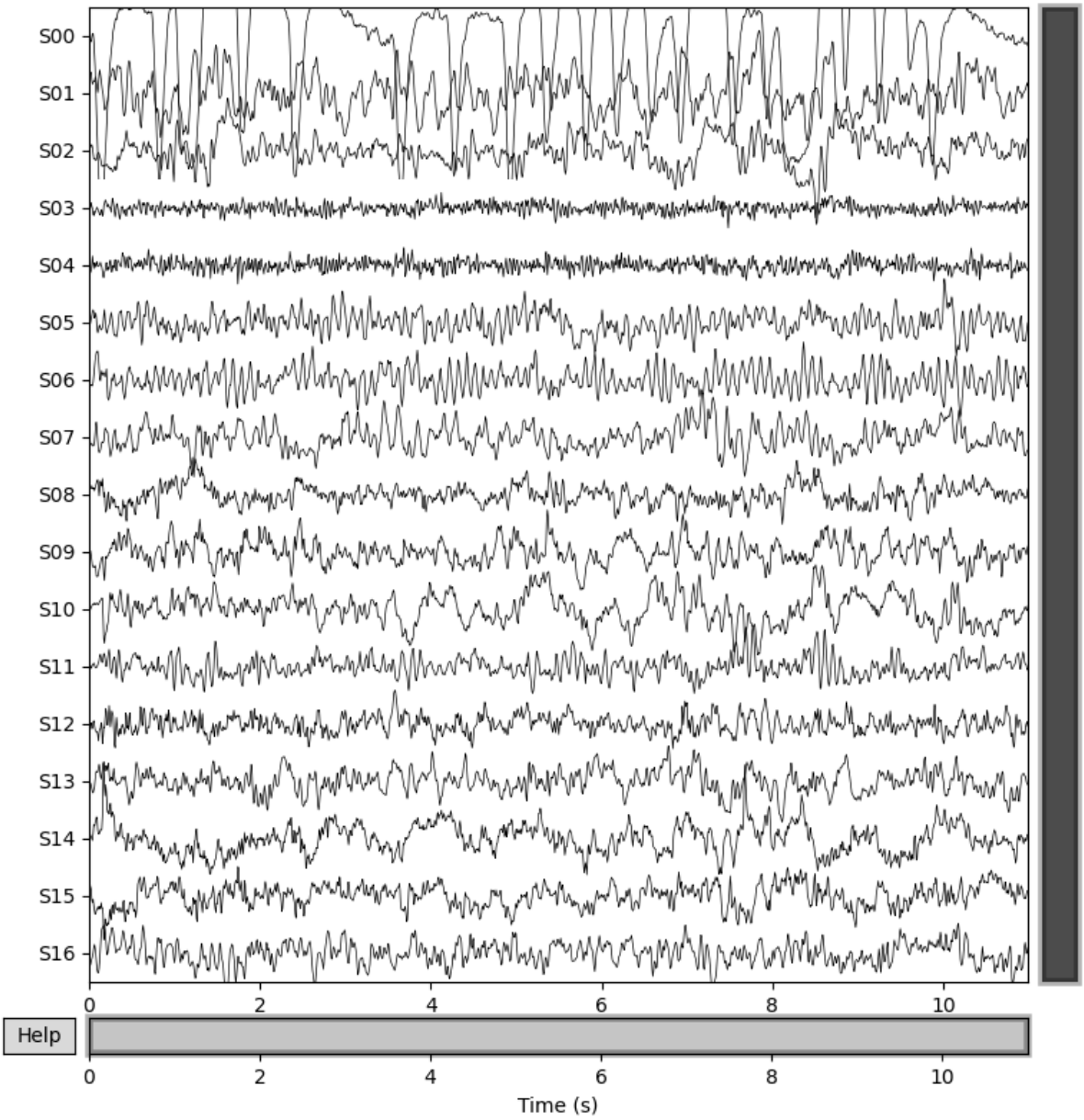
```
                title='Diagonalized cospectra, in source space')
```




```
<Figure size 1200x700 with 32 Axes>
```

Source space

```
# Estimate sources S applying forward filters B to signal X: S = B X
source_raw = ajdc.transform(signal_raw[np.newaxis, ...])[0]

# Plot sources S
sr_info = create_info(ch_names=sr_names, ch_types=['misc'] * sr_count,
                      sfreq=sfreq)
source = RawArray(source_raw, sr_info, verbose=False)
source.plot(duration=duration, start=0, n_channels=sr_count,
            scalings={'misc': 2e2}, title='EEG sources estimated by AJDC',
            show_scalebars=False)
```




```
<MNEBrowseFigure size 800x800 with 4 Axes>
```

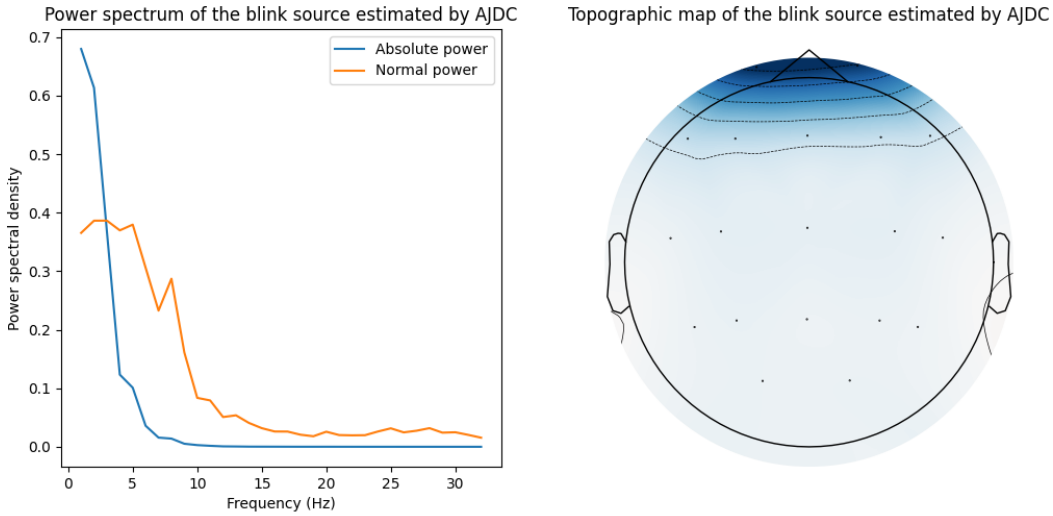
Artifact identification

```
# Identify artifact by eye: blinks are well separated in source S0
blink_idx = 0

# Get normal spectrum, ie power spectrum after trace-normalization
blink_spectrum_norm = ajdc._cosp_sources[:, blink_idx, blink_idx]
blink_spectrum_norm /= np.linalg.norm(blink_spectrum_norm)
```

(continues on next page)

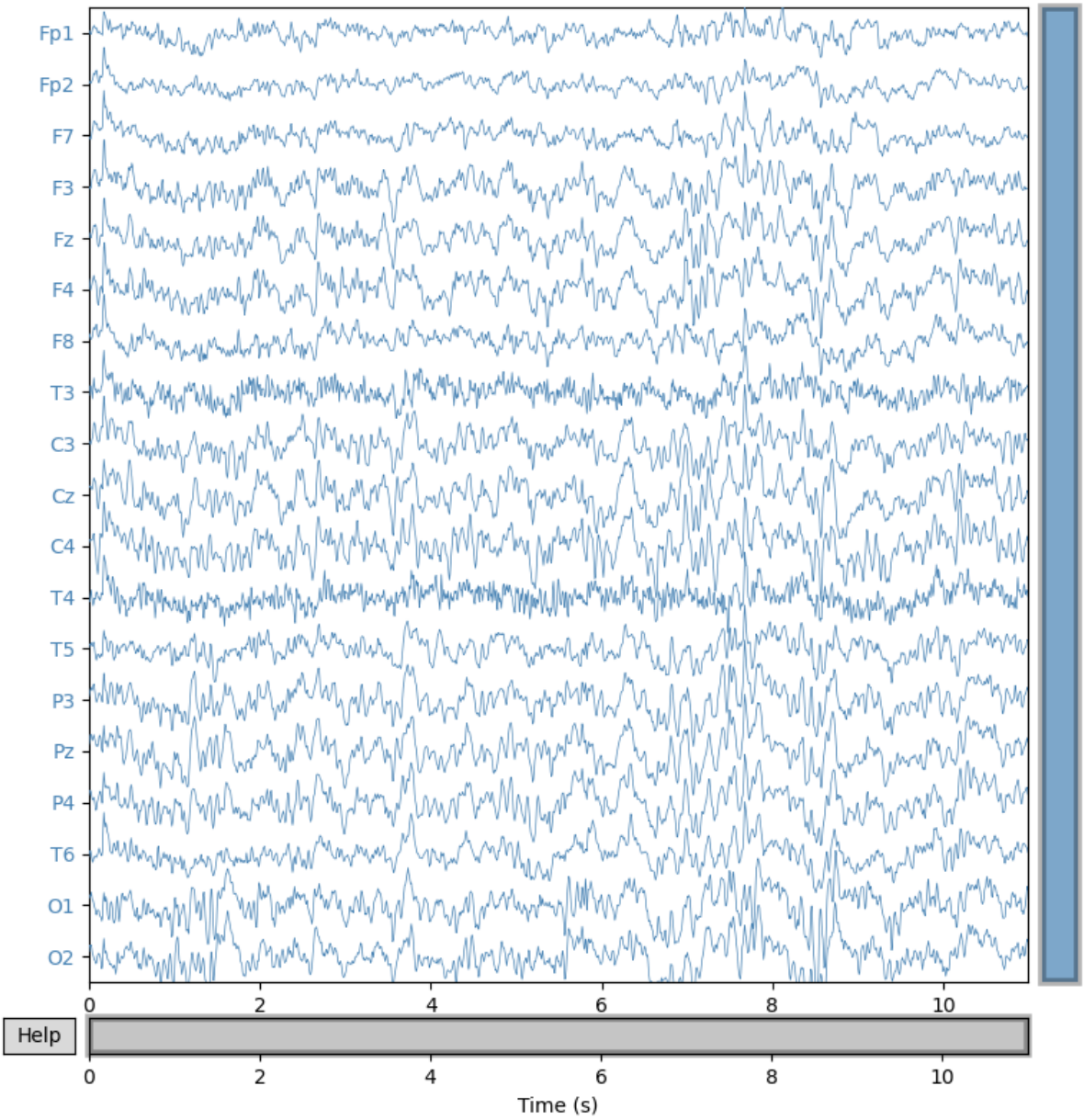
(continued from previous page)

```
# Get absolute spectrum, ie raw power spectrum of the source
f, spectrum = welch(source.get_data(picks=[blink_idx]), fs=sfreq,
                    nperseg=window, noverlap=int(window * overlap))
blink_spectrum_abs = spectrum[0, (f >= fmin) & (f <= fmax)]
blink_spectrum_abs /= np.linalg.norm(blink_spectrum_abs)

# Get topographic map
blink_filter = ajdc.backward_filters_[:, blink_idx]

# Plot spectrum and topographic map of the blink source separated by AJDC
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))
axs[0].set(title='Power spectrum of the blink source estimated by AJDC',
           xlabel='Frequency (Hz)', ylabel='Power spectral density')
axs[0].plot(freqs, blink_spectrum_abs, label='Absolute power')
axs[0].plot(freqs, blink_spectrum_norm, label='Normal power')
axs[0].legend()
axs[1].set_title('Topographic map of the blink source estimated by AJDC')
plot_topomap(blink_filter, pos=ch_info, axes=axs[1], extrapolate='box')
plt.show()
```



Artifact correction by BSS denoising

```
# BSS denoising: blink source is suppressed in source space using activation
# matrix D, and then applying backward filters A to come back to channel space
# Denoised signal: Xd = A D S
signal_denois_raw = ajdc.inverse_transform(source_raw[np.newaxis, ...],
                                           supp=[blink_idx])[0]

# Plot denoised signal Xd
signal_denois = RawArray(signal_denois_raw, ch_info, verbose=False)
signal_denois.plot(duration=duration, start=0, n_channels=ch_count,
                   scalings={'eeg': 3e1}, color={'eeg': 'steelblue'},
                   title='Denoised EEG signal by AJDC', show_scalebars=False)
```




```
<MNEBrowseFigure size 800x800 with 4 Axes>
```

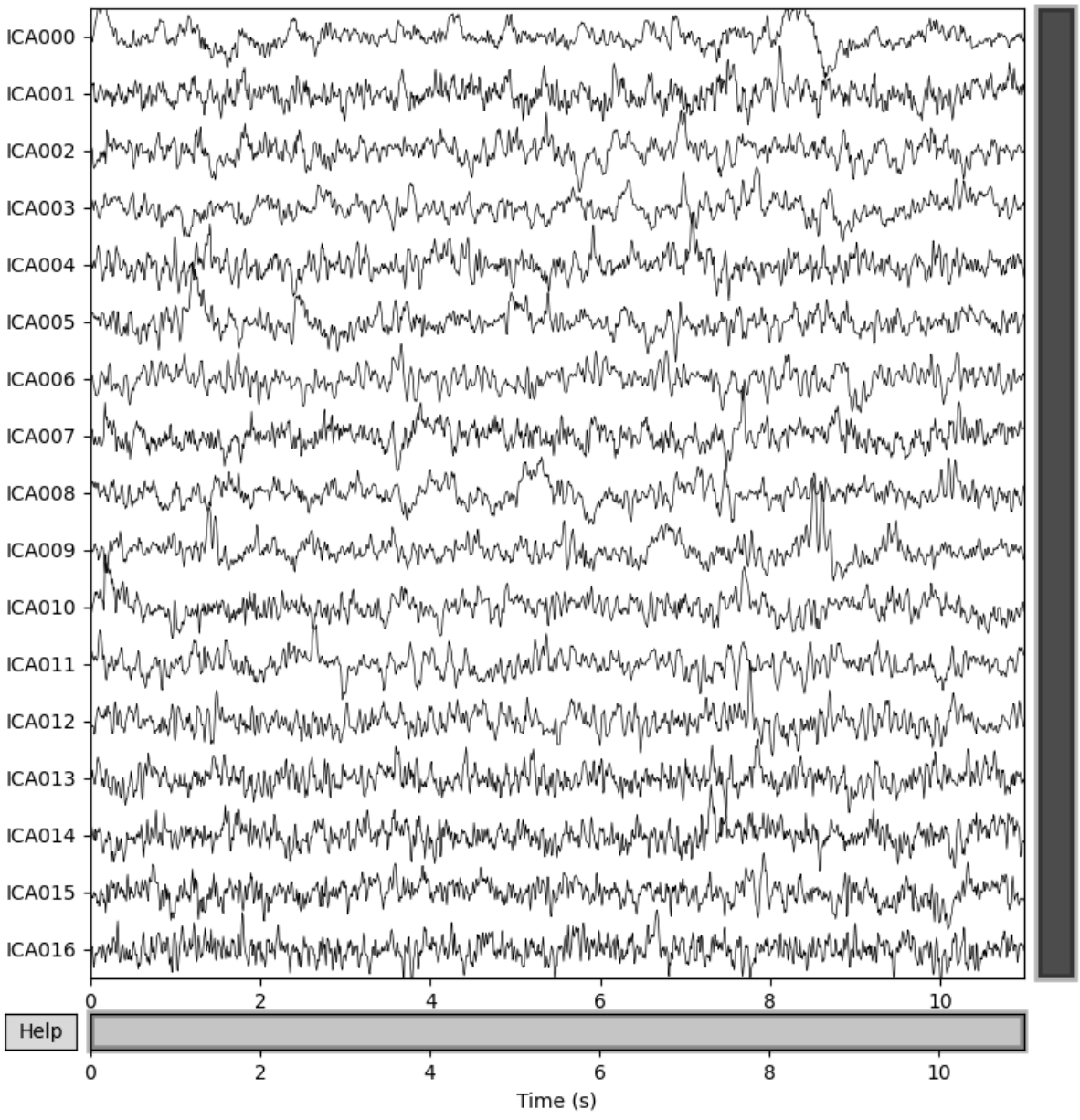
Comparison with Independent Component Analysis (ICA)

```
# Infomax-based ICA is a Higher-Order Statistics (HOS)-based BSS, minimizing
# mutual information
ica = ICA(n_components=ajdc.n_sources_, method='infomax', random_state=42)
ica.fit(signal, picks='eeg')

# Plot sources separated by ICA
ica.plot_sources(signal, title='EEG sources estimated by ICA')
```

(continues on next page)

(continued from previous page)

```
# Can you find the blink source?
```



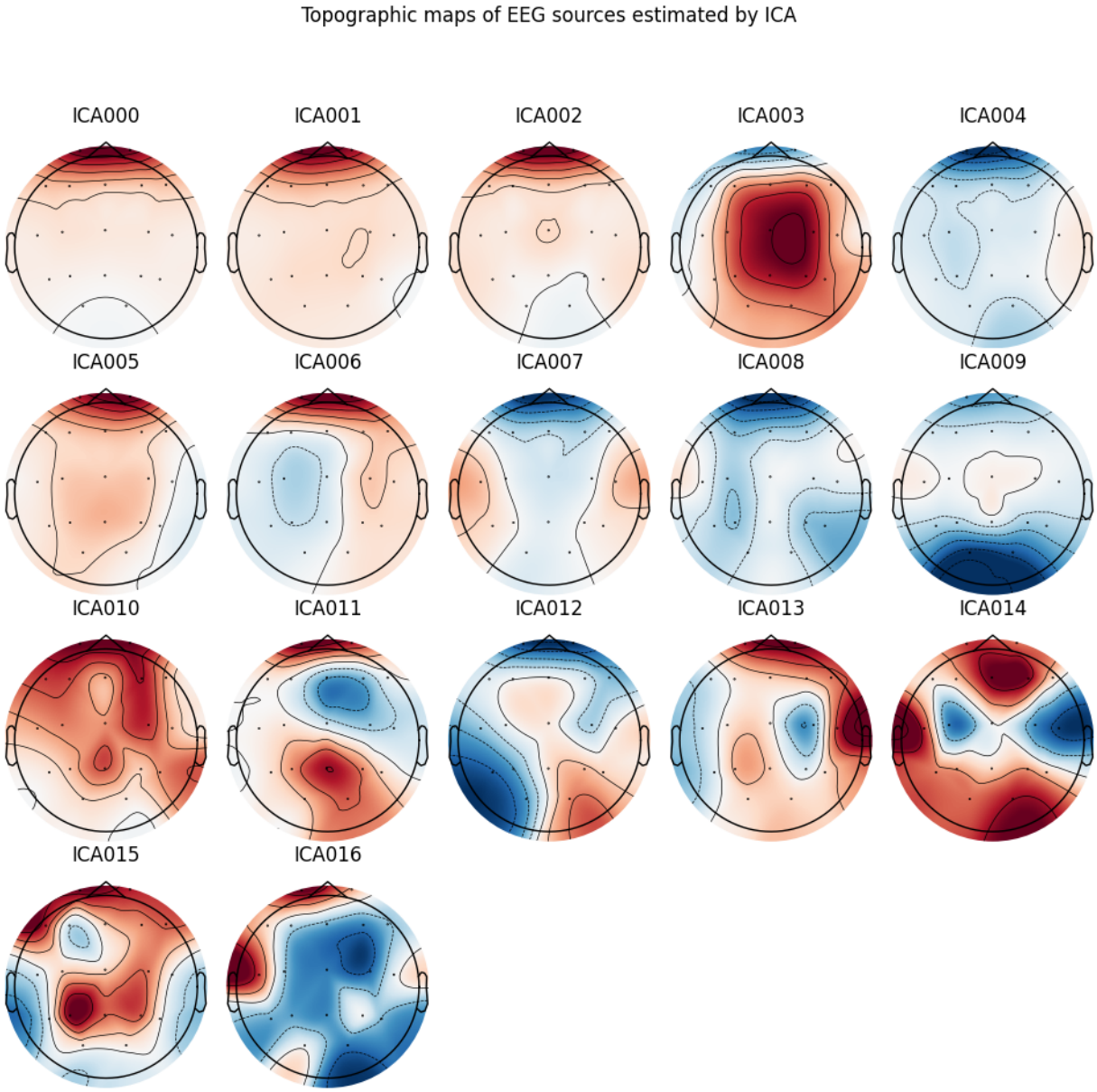

```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/examples/
↪artifacts/plot_correct_ajdc_EEG.py:162: RuntimeWarning: The data has not been high-
↪pass filtered. For good ICA performance, it should be high-pass filtered (e.g., with a␣
↪1.0 Hz lower bound) before fitting ICA.
  ica.fit(signal, picks='eeg')
Fitting ICA to data using 19 channels (please be patient, this may take a while)
Selecting by number: 17 components

Fitting ICA took 0.3s.
```

(continues on next page)

(continued from previous page)

```
Creating RawArray with float64 data, n_channels=17, n_times=1408
    Range : 0 ... 1407 =      0.000 ...    10.992 secs
Ready.

<MNEBrowseFigure size 800x800 with 4 Axes>
```

```
# Plot topographic maps of sources separated by ICA
ica.plot_components(title='Topographic maps of EEG sources estimated by ICA')
```




```
[<MNEFigure size 975x967 with 17 Axes>]
```

References

Total running time of the script: (0 minutes 8.210 seconds)

Online Artifact Detection with Riemannian Potato Field

Example of Riemannian Potato Field (RPF)[1] applied on EEG time-series to detect artifacts in online processing. It is compared to the Riemannian Potato (RP)[2].

```
# Authors: Quentin Barthélemy
#
# License: BSD (3-clause)

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation

from mne.datasets import eegbci
from mne.io import read_raw_edf
from mne.channels import make_standard_montage
from mne import make_fixed_length_epochs

from pyriemann.estimation import Covariances
from pyriemann.utils.covariance import normalize
from pyriemann.clustering import Potato, PotatoField
```

```
def filter_bandpass(signal, low_freq, high_freq, channels=None, method="iir"):
    """Filter signal on specific channels and in a specific frequency band"""
    sig = signal.copy()
    if channels is not None:
        sig.pick_channels(channels)
    sig.filter(l_freq=low_freq, h_freq=high_freq, method=method, verbose=False)
    return sig


def plot_detection(ax, rp_label, rpf_label):
    labels = []
    ylims = ax.get_ylim()
    height = ylims[1] - ylims[0]
    if not rp_label:
        r1 = ax.axhspan(
            ylims[0] + 0.06 * height, ylims[1] - 0.05 * height,
            edgecolor='r', facecolor='none',
            xmin=-test_time_start / test_duration - 0.005,
            xmax=(duration - test_time_start) / test_duration - 0.005)
        labels.append(r1)
        ax.text(0.25, 0.95, 'RP', color='r', size=16, transform=ax.transAxes)
    if not rpf_label:
```

(continues on next page)

[1] The Riemannian Potato Field: A Tool for Online Signal Quality Index of EEG Q. Barthélemy, L. Mayaud, D. Ojeda, and M. Congedo. IEEE Transactions on Neural Systems and Rehabilitation Engineering, IEEE Institute of Electrical and Electronics Engineers, 2019, 27 (2), pp.244-255

[2] The Riemannian Potato: an automatic and adaptive artifact detection method for online experiments using Riemannian geometry A. Barachant, A Andreev, and M. Congedo. TOBI Workshop lV, Jan 2013, Sion, Switzerland. pp.19-20.

(continued from previous page)

```
        r2 = ax.axhspan(
            ylims[0] + 0.05 * height, ylims[1] - 0.06 * height,
            edgecolor='m', facecolor='none',
            xmin=-test_time_start / test_duration + 0.005,
            xmax=(duration - test_time_start) / test_duration + 0.005)
        labels.append(r2)
        ax.text(0.65, 0.95, 'RPF', color='m', size=16, transform=ax.transAxes)
    if rp_label and rpf_label:
        r3 = ax.axhspan(
            ylims[0] + 0.05 * height, ylims[1] - 0.05 * height,
            edgecolor='k', facecolor='none',
            xmin=-test_time_start / test_duration,
            xmax=(duration - test_time_start) / test_duration)
        labels.append(r3)
    return labels
```

Load EEG data

```
# Load motor imagery data
raw = read_raw_edf(eegbci.load_data(2, [5])[0], preload=True, verbose=False)
eegbci.standardize(raw)
raw.set_montage(make_standard_montage('standard_1005'))
sfreq = int(raw.info['sfreq'])  # 160 Hz

# Select the 21 channels of the 10-20 montage
raw.pick_channels(
    ['Fp1', 'Fpz', 'Fp2', 'F7', 'F3', 'Fz', 'F4', 'F8', 'T7', 'C3', 'Cz', 'C4',
     'T8', 'P7', 'P3', 'Pz', 'P4', 'P8', 'O1', 'Oz', 'O2'], ordered=True)
ch_names = raw.ch_names
ch_count = len(ch_names)

# Define time-series epoching with a sliding window
duration = 2.5    # duration of epochs
interval = 0.2    # interval between epochs
```

Riemannian potato

Riemannian potato (RP)[Page 61, 2] selects all channels and filter between 1 and 35 Hz.

```
# RP definition
z_th = 2.0           # z-score threshold
low_freq, high_freq = 1., 35.
rp = Potato(metric='riemann', threshold=z_th)

# EEG processing for RP
rp_sig = filter_bandpass(raw, low_freq, high_freq)  # band-pass filter
rp_epochs = make_fixed_length_epochs(  # epoch time-series
    rp_sig, duration=duration, overlap=duration - interval, verbose=False)
rp_covs = Covariances(estimator='scm').transform(rp_epochs.get_data())
```

(continues on next page)

(continued from previous page)

```
# RP training
train_covs = 45      # nb of matrices for training
train_set = range(train_covs)
rp.fit(rp_covs[train_set])
```

```
Using data from preloaded Raw for 603 events and 400 original time points ...
0 bad epochs dropped

Potato(threshold=2.0)
```

Riemannian potato field

Riemannian potato field (RPF)[Page 61, 1] combines several potatoes of low dimensionality, each one designed to capture a different kind of artifact typically affecting some specific spatial area (i.e. subsets of channels) and/or specific frequency bands.

BCI or NFB applications aim at the modulation specific brain oscillations, it is thus advisable to exclude such frequencies from potatoes so as to prevent desirable brain modulations to be detected as artifactual.

```
# RPF definition
p_th = 0.01          # probability threshold
rpf_config = {
    'RPF eye_blinks': {  # for eye-blinks
        'ch_names': ['Fp1', 'Fpz', 'Fp2'],
        'low_freq': 1.,
        'high_freq': 20.},
    'RPF occipital': {  # for high-frequency artifacts in occipital area
        'ch_names': ['O1', 'Oz', 'O2'],
        'low_freq': 25.,
        'high_freq': 45.,
        'cov_normalization': 'trace'},  # trace-norm to be insensitive to power
    'RPF global_lf': {  # for low-frequency artifacts in all channels
        'ch_names': None,
        'low_freq': 0.5,
        'high_freq': 3.}
}
rpf = PotatoField(metric='riemann', z_threshold=z_th, p_threshold=p_th,
                  n_potatoes=len(rpf_config))

# EEG processing for RPF
rpf_covs = []
for p in rpf_config.values():  # loop on potatoes
    rpf_sig = filter_bandpass(raw, p.get('low_freq'), p.get('high_freq'),
                              channels=p.get('ch_names'))
    rpf_epochs = make_fixed_length_epochs(
        rpf_sig, duration=duration, overlap=duration - interval, verbose=False)
    covs_ = Covariances(estimator='scm').transform(rpf_epochs.get_data())
    if p.get('cov_normalization'):
        covs_ = normalize(covs_, p.get('cov_normalization'))
    rpf_covs.append(covs_)
```

(continues on next page)

(continued from previous page)

```
# RPF training
rpf.fit([c[train_set] for c in rpf_covs])
```

```
Using data from preloaded Raw for 603 events and 400 original time points ...
0 bad epochs dropped
Using data from preloaded Raw for 603 events and 400 original time points ...
0 bad epochs dropped
Using data from preloaded Raw for 603 events and 400 original time points ...
0 bad epochs dropped
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')

PotatoField(n_potatoes=3, z_threshold=2.0)
```

Online Artifact Detection with Potatoes

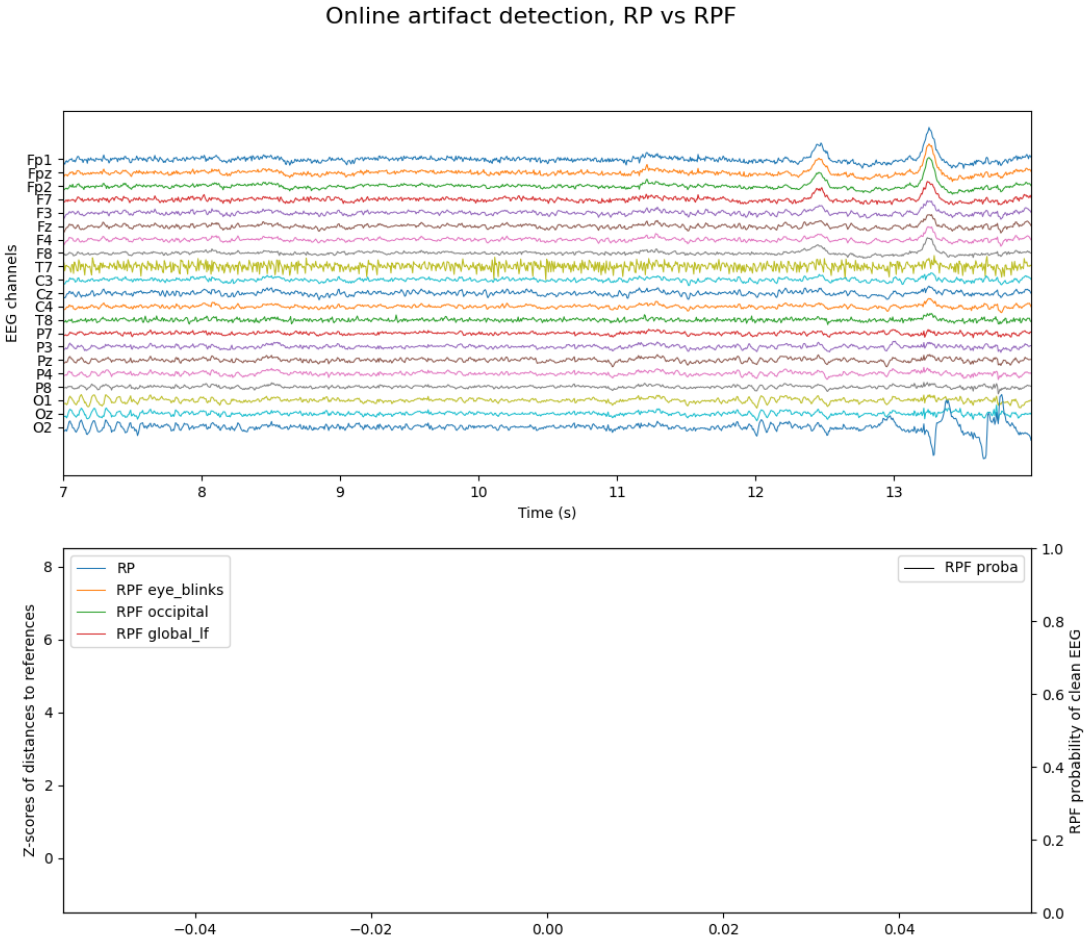
Detect artifacts/outliers on test set, with an animation to imitate an online acquisition, processing and artifact detection of EEG time-series. Remark that all these potatoes are semi-dynamic: they are updated when EEG is not artifacted[Page 61, 1].

```
# Prepare data for online detection
test_covs_max = 400     # nb of epochs to visualize in this example
test_covs_visu = 100    # nb of z-scores/proba to display simultaneously
test_time_start = -2    # start time to display signal
test_time_end = 5       # end time to display signal

test_duration = test_time_end - test_time_start
time_start = train_covs * interval + test_time_start
time_end = train_covs * interval + test_time_end
time = np.linspace(time_start, time_end, int((time_end - time_start) * sfreq),
                   endpoint=False)
raw.filter(l_freq=0.5, h_freq=75., method='iir', verbose=False)
eeg_data = 1e5 * raw.get_data()
sig = eeg_data[:, int(time_start * sfreq):int(time_end * sfreq)]
eeg_offset = - 15 * np.linspace(1, ch_count, ch_count, endpoint=False)
covs_t, covs_z = np.empty([0]), np.empty([len(rpf_config) + 1, 0])
covs_p = np.empty([0])

fig, ax = plt.subplots(figsize=(12, 10), nrows=2, ncols=1)
fig.suptitle('Online artifact detection, RP vs RPF', fontsize=16)
ax[0].set(xlabel='Time (s)', ylabel='EEG channels')
ax[0].set_xlim([time[0], time[-1]])
ax[0].set_yticks(eeg_offset)
ax[0].set_yticklabels(ch_names)
pl = ax[0].plot(time, sig.T + eeg_offset.T, lw=0.75)
labels = []

ax[1].set(ylabel='Z-scores of distances to references')
```

(continues on next page)

(continued from previous page)

```
pl2 = ax[1].plot(covs_t, covs_z.T, lw=0.75)
for c, l in enumerate(['RP'] + [*rpf_config]):
    pl2[c].set_label(l)
ax[1].set_ylim([-1.5, 8.5])
ax[1].legend(loc='upper left')
axp = ax[1].twinx()
axp.set(ylabel='RPF probability of clean EEG')
pl3 = axp.plot(covs_t, covs_p, lw=0.75, c='k', label='RPF proba')
axp.set_ylim([0, 1])
axp.legend(loc='upper right')
```




```
<matplotlib.legend.Legend object at 0x7f4cd335f9d0>
```

```
# Prepare animation for online detection
def online_detect(t):
    global time, sig, labels, covs_t, covs_z, covs_p

    # Online artifact detection
```

(continues on next page)

(continued from previous page)

```
    rp_label = rp.predict(rp_covs[np.newaxis, t])[0]
    rp_zscore = rp.transform(rp_covs[np.newaxis, t])
    rpf_label = rpf.predict([c[np.newaxis, t] for c in rpf_covs])[0]
    rpf_zscores = rpf.transform([c[np.newaxis, t] for c in rpf_covs])
    rpf_proba = rpf.predict_proba([c[np.newaxis, t] for c in rpf_covs])
    if rp_label == 1:
        rp.partial_fit(rp_covs[np.newaxis, t], alpha=1 / t)
    if rpf_label == 1:
        rpf.partial_fit([c[np.newaxis, t] for c in rpf_covs], alpha=1 / t)

    # Update data
    time_start = t * interval + test_time_end
    time_end = (t + 1) * interval + test_time_end
    time_ = np.linspace(time_start, time_end, int(interval * sfreq),
                        endpoint=False)
    time = np.r_[time[int(interval * sfreq):], time_]
    sig = np.hstack((sig[:, int(interval * sfreq):],
                     eeg_data[:, int(time_start*sfreq):int(time_end*sfreq)]))
    covs_t = np.r_[covs_t, time_start]
    covs_z = np.hstack((covs_z,
                        np.vstack((rp_zscore[np.newaxis], rpf_zscores.T))))
    covs_p = np.r_[covs_p, rpf_proba]
    if len(covs_p) > test_covs_visu:
        covs_t, covs_z, covs_p = covs_t[1:], covs_z[:, 1:], covs_p[1:]

    # Update plot
    for c in range(ch_count):
        pl[c].set_data(time, sig[c] + eeg_offset[c])
        pl[c].axes.set_xlim(time[0], time[-1])
    for lbl in labels:
        lbl.remove()
    for txt in ax[0].texts:
        txt.set_visible(False)
    labels = plot_detection(ax[0], rp_label, rpf_label)
    for c in range(len(pl2)):
        pl2[c].set_data(covs_t, covs_z[c])
        pl2[c].axes.set_xlim(covs_t[0] - 0.1, covs_t[-1])
    pl3[0].set_data(covs_t, covs_p)
    return pl, pl2, pl3


interval_display = 1.0  # can be changed for a slower display

potato = FuncAnimation(fig, online_detect,
                       frames=range(train_covs, test_covs_max),
                       interval=interval_display, blit=False, repeat=False)
```

```
# Plot online detection

# Plot complete visu: a dynamic display is required
plt.show()
```

(continues on next page)

(continued from previous page)

```
# Plot only 10s, for animated documentation
try:
    from IPython.display import HTML
except ImportError:
    raise ImportError("Install IPython to plot animation in documentation")

plt.rcParams["animation.embed_limit"] = 10
HTML(potato.to_jshtml(fps=5, default_mode='loop'))
```

```
Animation size has reached 10958720 bytes, exceeding the limit of 10485760.0. If you're␣
↪sure you want a larger animation embedded, set the animation.embed_limit rc parameter␣
↪to a larger value (in MB). This and further frames will be dropped.
```

References

Total running time of the script: (1 minutes 8.752 seconds)

Online Artifact Detection with Riemannian Potato

Example of Riemannian Potato[1] applied on EEG time-series to detect artifacts in online processing. It is computed only for two channels to display intuitive visualizations.

```
# Authors: Quentin Barthélemy & David Ojeda
#
# License: BSD (3-clause)

from functools import partial

import os

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation

from mne.datasets import sample
from mne.io import read_raw_fif
from mne import make_fixed_length_epochs

from pyriemann.estimation import Covariances
from pyriemann.clustering import Potato
from pyriemann.utils.viz import _add_alpha
```

```
@partial(np.vectorize, excluded=['potato'])
def get_zscores(cov_00, cov_01, cov_11, potato):
    cov = np.array([[cov_00, cov_01], [cov_01, cov_11]])
    with np.testing.suppress_warnings() as sup:
        sup.filter(RuntimeWarning)
```

(continues on next page)

[1] The Riemannian Potato: an automatic and adaptive artifact detection method for online experiments using Riemannian geometry A. Barachant, A Andreev, and M. Congedo. TOBI Workshop lV, Jan 2013, Sion, Switzerland. pp.19-20

(continued from previous page)

```
        return potato.transform(cov[np.newaxis, ...])


def plot_potato_2D(ax, cax, X, Y, p_zscores, p_center, covs, p_colors, clabel):
    qcs = ax.contourf(X, Y, p_zscores, levels=20, vmin=p_zscores.min(),
                      vmax=p_zscores.max(), cmap='RdYlBu_r', alpha=0.5)
    ax.contour(X, Y, p_zscores, levels=[z_th], colors='k')
    sc = ax.scatter(covs[:, 0, 0], covs[:, 1, 1], c=p_colors)
    ax.scatter(p_center[0, 0], p_center[1, 1], c='k', s=100)
    if cax:
        cbar = fig.colorbar(qcs, cax=cax)
    else:
        cbar = fig.colorbar(qcs, ax=ax)
    cbar.ax.set_ylabel(clabel)
    return sc


def plot_sig(ax, time, sig):
    ax.axis((time[0], time[-1], -15, 15))
    pl, = ax.plot(time, sig, lw=0.75)
    ax.axhspan(
        -14, 14, edgecolor='k', facecolor='none',
        xmin=-test_time_start / (test_time_end - test_time_start),
        xmax=(duration - test_time_start) / (test_time_end - test_time_start))
    return pl
```

Load EEG data

```
raw_fname = os.path.join(sample.data_path(), 'MEG', 'sample',
                         'sample_audvis_filt-0-40_raw.fif')
raw = read_raw_fif(raw_fname, preload=True, verbose=False)
sfreq = int(raw.info['sfreq'])  # 150 Hz
```

Offline processing of EEG data

```
# Apply common average reference on EEG channels
raw.pick_types(meg=False, eeg=True).apply_proj()

# Select two EEG channels for the example, preferably without artifact at the
# beginning, to have a reliable calibration
ch_names = ['EEG 010', 'EEG 015']

# Apply band-pass filter between 1 and 35 Hz
raw.filter(1., 35., method='iir', picks=ch_names)

# Epoch time-series with a sliding window
duration = 2.5    # duration of epochs
interval = 0.2    # interval between successive epochs
```

(continues on next page)

(continued from previous page)

```
epochs = make_fixed_length_epochs(
    raw, duration=duration, overlap=duration - interval, verbose=False)
epochs_data = 5e5 * epochs.get_data(picks=ch_names)

# Estimate spatial covariance matrices
covs = Covariances(estimator='lwf').transform(epochs_data)
```

```
Removing projector <Projection | PCA-v1, active : False, n_channels : 102>
Removing projector <Projection | PCA-v2, active : False, n_channels : 102>
Removing projector <Projection | PCA-v3, active : False, n_channels : 102>
Created an SSP operator (subspace dimension = 1)
1 projection items activated
SSP projectors applied...
Filtering a subset of channels. The highpass and lowpass values in the measurement info␣
↪will not be updated.
Filtering raw data in 1 contiguous segment
Setting up band-pass filter from 1 - 35 Hz

IIR filter parameters
---------------------
Butterworth bandpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 16 (effective, after forward-backward)
- Cutoffs at 1.00, 35.00 Hz: -6.02, -6.02 dB

Using data from preloaded Raw for 1377 events and 375 original time points ...
0 bad epochs dropped
```

Offline Calibration of Potato

2D projection of the z-score map of the Riemannian potato, for 2x2 covariance matrices (in blue if clean, in red if artifacted) and their reference matrix (in black). The colormap defines the z-score and a chosen isocontour defines the potato. It reproduces Fig 1 of reference[2].

```
z_th = 2.0       # z-score threshold
train_covs = 40  # nb of matrices to train the potato
```

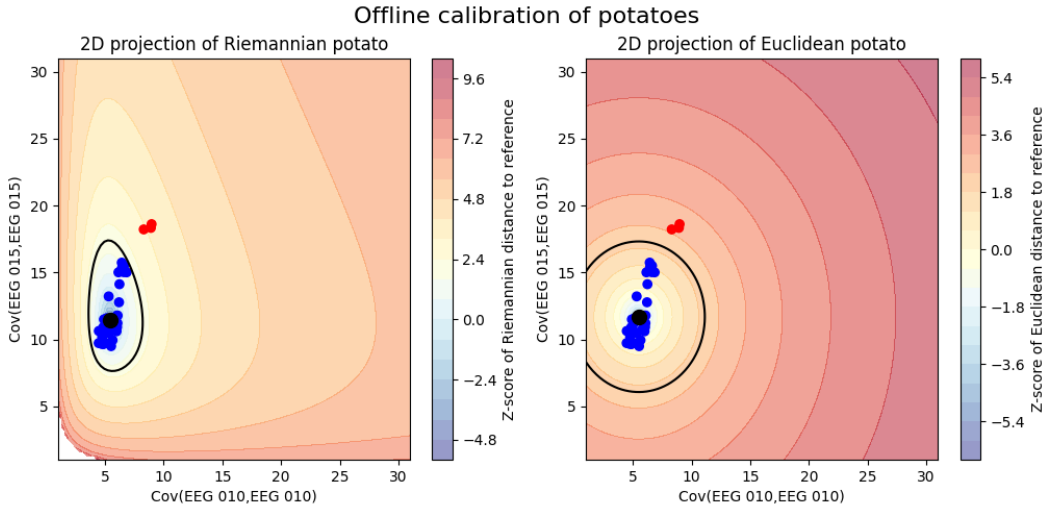
```
# Calibrate potato by unsupervised training on first matrices: compute a
# reference matrix, mean and standard deviation of distances to this reference.
train_set = range(train_covs)
rpotato = Potato(metric='riemann', threshold=z_th).fit(covs[train_set])
rp_center = rpotato._mdm.covmeans_[0]
epotato = Potato(metric='euclid', threshold=z_th).fit(covs[train_set])
ep_center = epotato._mdm.covmeans_[0]

rp_labels = rpotato.predict(covs[train_set])
rp_colors = ['b' if ll == 1 else 'r' for ll in rp_labels.tolist()]
ep_labels = epotato.predict(covs[train_set])
ep_colors = ['b' if ll == 1 else 'r' for ll in ep_labels.tolist()]
```

(continues on next page)

[2] The Riemannian Potato Field: A Tool for Online Signal Quality Index of EEG Q. Barthélemy, L. Mayaud, D. Ojeda, and M. Congedo. IEEE Transactions on Neural Systems and Rehabilitation Engineering, IEEE Institute of Electrical and Electronics Engineers, 2019, 27 (2), pp.244-255

(continued from previous page)

```
# Zscores in the horizontal 2D plane going through the reference
X, Y = np.meshgrid(np.linspace(1, 31, 100), np.linspace(1, 31, 100))
rp_zscores = get_zscores(X, np.full_like(X, rp_center[0, 1]), Y,
                         potato=rpotato)
rp_mzscores = np.ma.masked_where(~np.isfinite(rp_zscores), rp_zscores)
ep_zscores = get_zscores(X, np.full_like(X, ep_center[0, 1]), Y,
                         potato=epotato)

# Plot calibration
xlabel = 'Cov({},{})'.format(ch_names[0], ch_names[0])
ylabel = 'Cov({},{})'.format(ch_names[1], ch_names[1])

fig, axs = plt.subplots(figsize=(12, 5), nrows=1, ncols=2)
fig.suptitle('Offline calibration of potatoes', fontsize=16)
axs[0].set(xlabel=xlabel, ylabel=ylabel,
           title='2D projection of Riemannian potato')
plot_potato_2D(axs[0], None, X, Y, rp_mzscores, rp_center, covs[train_set],
               rp_colors, 'Z-score of Riemannian distance to reference')
axs[1].set(xlabel=xlabel, ylabel=ylabel,
           title='2D projection of Euclidean potato')
plot_potato_2D(axs[1], None, X, Y, ep_zscores, ep_center, covs[train_set],
               ep_colors, 'Z-score of Euclidean distance to reference')
plt.show()
```



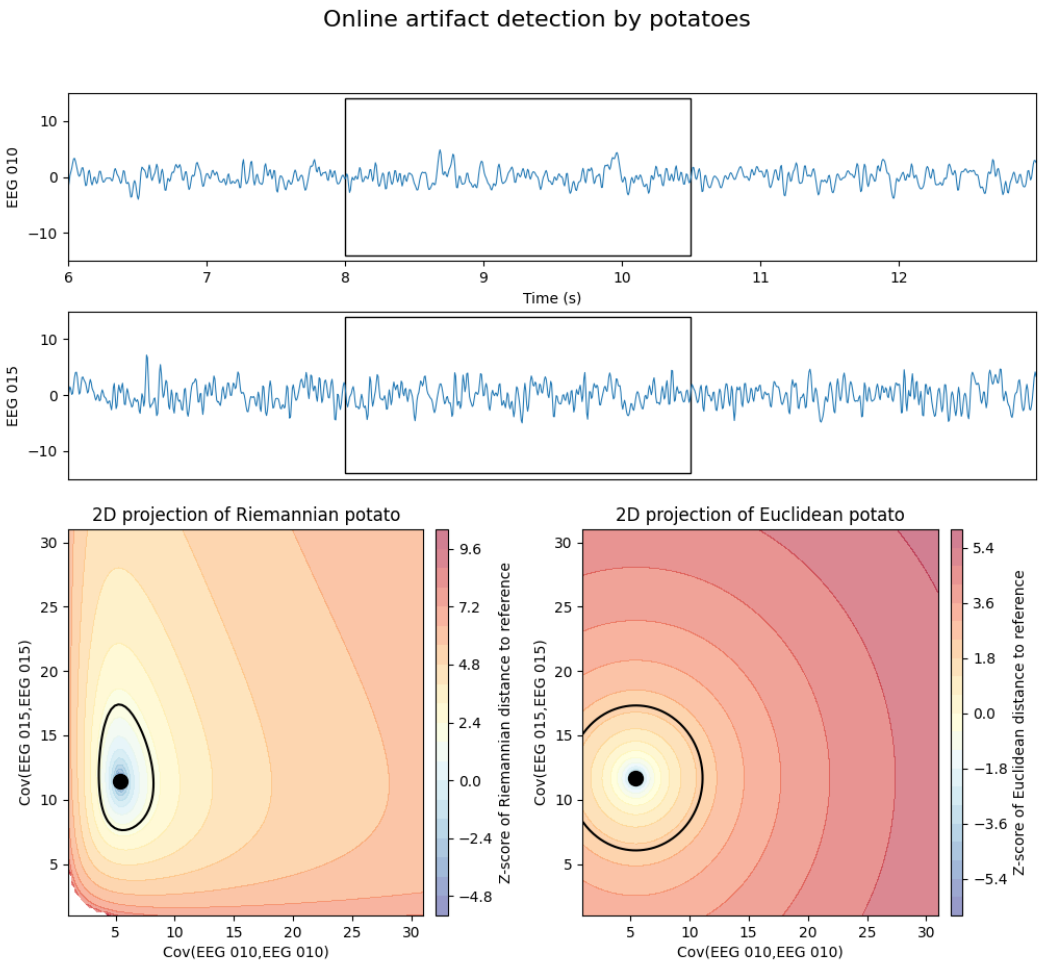
Online Artifact Detection with Potato

Detect artifacts/outliers on test set, with an animation to imitate an online acquisition, processing and artifact detection of EEG time-series. Initialized with an offline calibration, the online potato can be[Page 69, 2]:

- static: it is never updated, damaging its efficiency over time;
- semi-dynamic: it is updated when EEG is not artifacted.

```
is_static = False       # static or semi-dynamic mode
```

```
# Prepare data for online detection
test_covs_max = 300     # nb of matrices to visualize in this example
test_covs_visu = 30     # nb of matrices to display simultaneously
test_time_start = -2    # start time to display signal
test_time_end = 5       # end time to display signal

time_start = train_covs * interval + test_time_start
time_end = train_covs * interval + test_time_end
time = np.linspace(time_start, time_end, int((time_end - time_start) * sfreq),
                   endpoint=False)
eeg_data = 3e5 * raw.get_data(picks=ch_names)
sig = eeg_data[:, int(time_start * sfreq):int(time_end * sfreq)]
covs_visu, rp_colors, ep_colors = np.empty([0, 2, 2]), [], []
alphas = np.linspace(0, 1, test_covs_visu)

fig = plt.figure(figsize=(12, 10), constrained_layout=False)
fig.suptitle('Online artifact detection by potatoes', fontsize=16)
gs = fig.add_gridspec(nrows=4, ncols=40, top=0.90, hspace=0.3, wspace=1.0)
ax_sig0 = fig.add_subplot(gs[0, :], xlabel='Time (s)', ylabel=ch_names[0])
pl_sig0 = plot_sig(ax_sig0, time, sig[0])
ax_sig1 = fig.add_subplot(gs[1, :], ylabel=ch_names[1])
pl_sig1 = plot_sig(ax_sig1, time, sig[1])
ax_sig1.set_xticks([])
ax_rp = fig.add_subplot(gs[2:4, 0:15], xlabel=xlabel, ylabel=ylabel,
                        title='2D projection of Riemannian potato')
cax_rp = fig.add_subplot(gs[2:4, 15])
p_rp = plot_potato_2D(ax_rp, cax_rp, X, Y, rp_mzscores, rp_center, covs_visu,
                      rp_colors, 'Z-score of Riemannian distance to reference')
ax_ep = fig.add_subplot(gs[2:4, 21:36], xlabel=xlabel, ylabel=ylabel,
                        title='2D projection of Euclidean potato')
cax_ep = fig.add_subplot(gs[2:4, 36])
p_ep = plot_potato_2D(ax_ep, cax_ep, X, Y, ep_zscores, ep_center, covs_visu,
                      ep_colors, 'Z-score of Euclidean distance to reference')
```




```
# Prepare animation for online detection
def online_detect(t):
    global time, sig, covs_visu

    # Online artifact detection
    rp_label = rpotato.predict(covs[np.newaxis, t])[0]
    ep_label = epotato.predict(covs[np.newaxis, t])[0]
    if not is_static:
        if rp_label == 1:
            rpotato.partial_fit(covs[np.newaxis, t], alpha=1 / t)
        if ep_label == 1:
            epotato.partial_fit(covs[np.newaxis, t], alpha=1 / t)

    # Update data
    time_start = t * interval + test_time_end
    time_end = (t + 1) * interval + test_time_end
    time_ = np.linspace(time_start, time_end, int(interval * sfreq),
                        endpoint=False)
    time = np.r_[time[int(interval * sfreq):], time_]
    sig = np.hstack((sig[:, int(interval*sfreq):],
```

(continues on next page)

(continued from previous page)

```
                    eeg_data[:, int(time_start*sfreq):int(time_end*sfreq)]))
    covs_visu = np.vstack((covs_visu, covs[np.newaxis, t]))
    rp_colors.append('b' if rp_label == 1 else 'r')
    ep_colors.append('b' if ep_label == 1 else 'r')
    if len(covs_visu) > test_covs_visu:
        covs_visu = covs_visu[1:]
        rp_colors.pop(0)
        ep_colors.pop(0)
    rp_colors_ = _add_alpha(rp_colors, alphas)
    ep_colors_ = _add_alpha(ep_colors, alphas)

    # Update plot
    pl_sig0.set_data(time, sig[0])
    pl_sig0.axes.set_xlim(time[0], time[-1])
    pl_sig1.set_data(time, sig[1])
    pl_sig1.axes.set_xlim(time[0], time[-1])
    p_rp.set_offsets(np.c_[covs_visu[:, 0, 0], covs_visu[:, 1, 1]])
    p_rp.set_color(rp_colors_)
    p_ep.set_offsets(np.c_[covs_visu[:, 0, 0], covs_visu[:, 1, 1]])
    p_ep.set_color(ep_colors_)
    return pl_sig0, pl_sig1, p_rp, p_ep


interval_display = 1.0  # can be changed for a slower display

potato = FuncAnimation(fig, online_detect,
                       frames=range(train_covs, test_covs_max),
                       interval=interval_display, blit=False, repeat=False)
```

```
# Plot online detection

# Plot complete visu: a dynamic display is required
plt.show()

# Plot only 10s, for animated documentation
try:
    from IPython.display import HTML
except ImportError:
    raise ImportError("Install IPython to plot animation in documentation")

plt.rcParams["animation.embed_limit"] = 10
HTML(potato.to_jshtml(fps=5, default_mode='loop'))
```

```
Animation size has reached 10608001 bytes, exceeding the limit of 10485760.0. If you're␣
↪sure you want a larger animation embedded, set the animation.embed_limit rc parameter␣
↪to a larger value (in MB). This and further frames will be dropped.
```

References

Total running time of the script: (0 minutes 41.502 seconds)

4.8.4 Classification of motor imagery

Using Riemannian geometry for classifying motor imagery.

Motor imagery classification

Classify motor imagery data with Riemannian geometry[1].

```
# generic import
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt

# mne import
from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci
from mne.decoding import CSP

# pyriemann import
from pyriemann.classification import MDM, TSclassifier
from pyriemann.estimation import Covariances

# sklearn imports
from sklearn.model_selection import cross_val_score, KFold
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
```

Set parameters and read data

```
# avoid classification of evoked responses by using epochs that start 1s after
# cue onset.
tmin, tmax = 1., 2.
event_id = dict(hands=2, feet=3)
subject = 7
runs = [6, 10, 14]  # motor imagery: hands vs feet

raw_files = [
    read_raw_edf(f, preload=True) for f in eegbci.load_data(subject, runs)
]
raw = concatenate_raws(raw_files)

picks = pick_types(
```

(continues on next page)

[1] Multiclass Brain-Computer Interface Classification by Riemannian Geometry A. Barachant, S. Bonnet, M. Congedo, and C. Jutten. IEEE Transactions on Biomedical Engineering, vol. 59, no. 4, p. 920-928, 2012.

(continued from previous page)

```
    raw.info, meg=False, eeg=True, stim=False, eog=False, exclude='bads')
# subsample elecs
picks = picks[::2]

# Apply band-pass filter
raw.filter(7., 35., method='iir', picks=picks)

events, _ = events_from_annotations(raw, event_id=dict(T1=2, T2=3))

# Read epochs (train will be done only between 1 and 2s)
# Testing will be done with a running classifier
epochs = Epochs(
    raw,
    events,
    event_id,
    tmin,
    tmax,
    proj=True,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False)
labels = epochs.events[:, -1] - 2

# cross validation
cv = KFold(n_splits=10, shuffle=True, random_state=42)
# get epochs
epochs_data_train = 1e6 * epochs.get_data()

# compute covariance matrices
cov_data_train = Covariances().transform(epochs_data_train)
```

```
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S007/S007R06.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S007/S007R10.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S007/S007R14.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Filtering a subset of channels. The highpass and lowpass values in the measurement info␣
↪will not be updated.
```

(continues on next page)

(continued from previous page)

```
Filtering raw data in 3 contiguous segments
Setting up band-pass filter from 7 - 35 Hz

IIR filter parameters
---------------------
Butterworth bandpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 16 (effective, after forward-backward)
- Cutoffs at 7.00, 35.00 Hz: -6.02, -6.02 dB

Used Annotations descriptions: ['T1', 'T2']
```

Classification with Minimum distance to mean

```
mdm = MDM(metric=dict(mean='riemann', distance='riemann'))

# Use scikit-learn Pipeline with cross_val_score function
scores = cross_val_score(mdm, cov_data_train, labels, cv=cv, n_jobs=1)

# Printing the results
class_balance = np.mean(labels == labels[0])
class_balance = max(class_balance, 1. - class_balance)
print("MDM Classification accuracy: %f / Chance level: %f" % (np.mean(scores),
                                                              class_balance))
```

```
MDM Classification accuracy: 0.850000 / Chance level: 0.511111
```

Classification with Tangent Space Logistic Regression

```
clf = TSclassifier()
# Use scikit-learn Pipeline with cross_val_score function
scores = cross_val_score(clf, cov_data_train, labels, cv=cv, n_jobs=1)

# Printing the results
class_balance = np.mean(labels == labels[0])
class_balance = max(class_balance, 1. - class_balance)
print("Tangent space Classification accuracy: %f / Chance level: %f" %
      (np.mean(scores), class_balance))
```

```
Tangent space Classification accuracy: 0.960000 / Chance level: 0.511111
```

Classification with CSP + logistic regression

```
# Assemble a classifier
lr = LogisticRegression()
csp = CSP(n_components=4, reg='ledoit_wolf', log=True)

clf = Pipeline([('CSP', csp), ('LogisticRegression', lr)])
scores = cross_val_score(clf, epochs_data_train, labels, cv=cv, n_jobs=1)

# Printing the results
class_balance = np.mean(labels == labels[0])
class_balance = max(class_balance, 1. - class_balance)
```

(continues on next page)

(continued from previous page)

```
print("CSP + LDA Classification accuracy: %f / Chance level: %f" %
      (np.mean(scores), class_balance))
```

```
Computing rank from data with rank=None
    Using tolerance 33 (2.2e-16 eps * 32 dim * 4.7e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 29 (2.2e-16 eps * 32 dim * 4.1e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 33 (2.2e-16 eps * 32 dim * 4.6e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 28 (2.2e-16 eps * 32 dim * 4e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 34 (2.2e-16 eps * 32 dim * 4.7e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 27 (2.2e-16 eps * 32 dim * 3.8e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 33 (2.2e-16 eps * 32 dim * 4.7e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
```

(continues on next page)

(continued from previous page)

```
Computing rank from data with rank=None
    Using tolerance 29 (2.2e-16 eps * 32 dim * 4e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 31 (2.2e-16 eps * 32 dim * 4.4e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 29 (2.2e-16 eps * 32 dim * 4.1e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 30 (2.2e-16 eps * 32 dim * 4.2e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 29 (2.2e-16 eps * 32 dim * 4.1e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 31 (2.2e-16 eps * 32 dim * 4.4e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 28 (2.2e-16 eps * 32 dim * 4e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 34 (2.2e-16 eps * 32 dim * 4.7e+15  max singular value)
    Estimated rank (mag): 32
```

(continues on next page)

(continued from previous page)

```
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 28 (2.2e-16 eps * 32 dim * 3.9e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 32 (2.2e-16 eps * 32 dim * 4.5e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 29 (2.2e-16 eps * 32 dim * 4.1e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 32 (2.2e-16 eps * 32 dim * 4.5e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
Computing rank from data with rank=None
    Using tolerance 26 (2.2e-16 eps * 32 dim * 3.7e+15  max singular value)
    Estimated rank (mag): 32
    MAG: rank 32 computed from 32 data channels with 0 projectors
Reducing data rank from 32 -> 32
Estimating covariance using LEDOIT_WOLF
Done.
CSP + LDA Classification accuracy: 0.900000 / Chance level: 0.511111
```

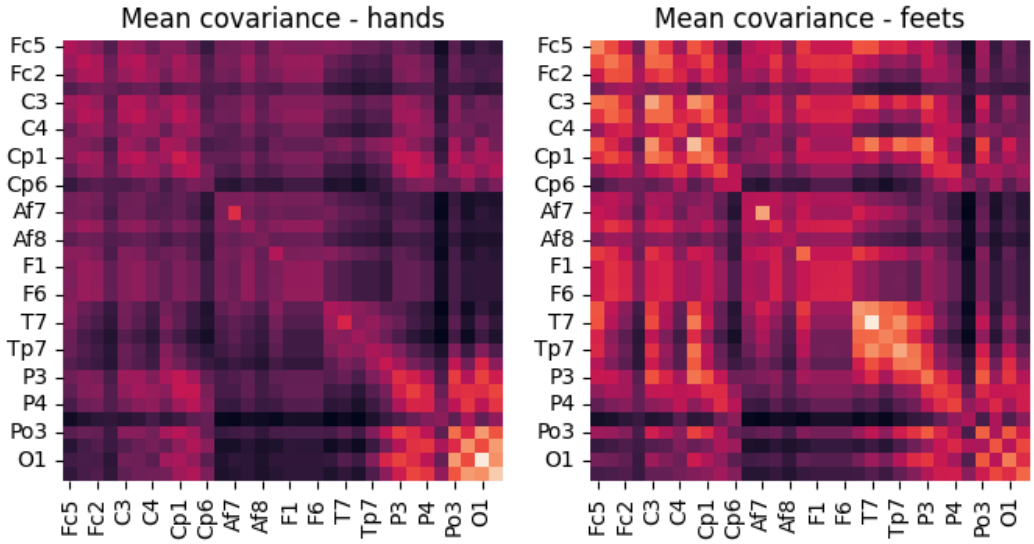
Display MDM centroid

```
mdm = MDM()
mdm.fit(cov_data_train, labels)

fig, axes = plt.subplots(1, 2, figsize=[8, 4])
ch_names = [ch.replace('.', '') for ch in epochs.ch_names]

df = pd.DataFrame(data=mdm.covmeans_[0], index=ch_names, columns=ch_names)
g = sns.heatmap(
    df, ax=axes[0], square=True, cbar=False, xticklabels=2, yticklabels=2)
```

(continues on next page)

(continued from previous page)

```
g.set_title('Mean covariance - hands')

df = pd.DataFrame(data=mdm.covmeans_[1], index=ch_names, columns=ch_names)
g = sns.heatmap(
    df, ax=axes[1], square=True, cbar=False, xticklabels=2, yticklabels=2)
plt.xticks(rotation='vertical')
plt.yticks(rotation='horizontal')
g.set_title('Mean covariance - feets')

# dirty fix
plt.sca(axes[0])
plt.xticks(rotation='vertical')
plt.yticks(rotation='horizontal')
plt.show()
```



References

Total running time of the script: (0 minutes 10.635 seconds)

Ensemble learning on functional connectivity

This example shows how to compute SPD matrices from functional connectivity estimators and how to combine classification with ensemble learning[1].

```
# Authors: Sylvain Chevallier <sylvain.chevallier@universite-paris-saclay.fr>,
#          Marie-Constance Corsi <marie.constance.corsi@gmail.com>
#
```

(continues on next page)

[1] Functional connectivity ensemble method to enhance BCI performance (FUCONE) Corsi, M.-C., Chevallier, S., De Vico Fallani, F. & Yger, F. IEEE TBME, 2022

(continued from previous page)

```
# License: BSD (3-clause)

import matplotlib.pyplot as plt

from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci

import numpy as np
import pandas as pd
import seaborn as sns

from pyriemann.classification import FgMDM
from pyriemann.estimation import Covariances, Coherences
from pyriemann.spatialfilters import CSP
from pyriemann.tangentspace import TangentSpace

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC

from helpers.coherence_helpers import (
    NearestSPD,
    get_results,
)
```

Define connectivity transformer

This estimator computes the functional connectivity from input signal using *pyriemann.estimation.Coherences*

```
class Connectivities(TransformerMixin, BaseEstimator):
    """Getting connectivity features from epoch"""

    def __init__(self, method="ordinary", fmin=8, fmax=35, fs=None):
        self.method = method
        self.fmin = fmin
        self.fmax = fmax
        self.fs = fs

    def fit(self, X, y=None):
        self._coh = Coherences(
            coh=self.method,
            fmin=self.fmin,
            fmax=self.fmax,
            fs=self.fs,
        )
        return self
```

(continues on next page)

(continued from previous page)

```
    def transform(self, X):
        X_coh = self._coh.fit_transform(X)
        X_con = np.mean(X_coh, axis=-1, keepdims=False)
        return X_con
```

Load EEG data

```
# avoid classification of evoked responses by using epochs that start 1s after
# cue onset.
tmin, tmax = 1.0, 2.0
event_id = dict(hands=2, feet=3)
subject = 7
runs = [4, 8]  # motor imagery: left vs right hand

raw_files = [
    read_raw_edf(f, preload=True) for f in eegbci.load_data(subject, runs)
]
raw = concatenate_raws(raw_files)

picks = pick_types(
    raw.info, meg=False, eeg=True, stim=False, eog=False, exclude="bads"
)
# subsample elecs
picks = picks[::2]

# Apply band-pass filter
raw.filter(7.0, 35.0, method="iir", picks=picks)

events, _ = events_from_annotations(raw, event_id=dict(T1=2, T2=3))

# Read epochs (train will be done only between 1 and 2s)
epochs = Epochs(
    raw,
    events,
    event_id,
    tmin,
    tmax,
    proj=True,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False,
)
labels = epochs.events[:, -1] - 2
fs = epochs.info["sfreq"]
X = 1e6 * epochs.get_data()
```

```
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S007/S007R04.edf...
```

(continues on next page)

(continued from previous page)

```
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S007/S007R08.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Filtering a subset of channels. The highpass and lowpass values in the measurement info␣
↪will not be updated.
Filtering raw data in 2 contiguous segments
Setting up band-pass filter from 7 - 35 Hz

IIR filter parameters
---------------------
Butterworth bandpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 16 (effective, after forward-backward)
- Cutoffs at 7.00, 35.00 Hz: -6.02, -6.02 dB

Used Annotations descriptions: ['T1', 'T2']
```

Defining pipelines

Compare CSP+SVM, FgMDM on covariance, tangent space logistic regression with covariance, lag coherence, and instantaneous coherence, along with ensemble method

```
ppl_baseline, ppl_fc, ppl_ens = {}, {}, {}
```

Baseline algorithms are CSP with optimal SVM and FgMDM based on covariances

```
param_svm = {"kernel": ("linear", "rbf"), "C": [0.1, 1, 10]}
step_csp = [
    ("cov", Covariances(estimator="lwf")),
    ("csp", CSP(nfilter=6)),
    ("optsvm", GridSearchCV(SVC(), param_svm, cv=3)),
]
ppl_baseline["CSP+optSVM"] = Pipeline(steps=step_csp)

step_mdm = [
    ("cov", Covariances(estimator="lwf")),
    ("fgmdm", FgMDM(metric="riemann", tsupdate=False)),
]
ppl_baseline["FgMDM"] = Pipeline(steps=step_mdm)
```

Functional connectivity pipelines use logistic regression in tangent space. They will be estimated from covariance, lagged coherence and instantaneous coherence.

```
spectral_met = ["cov", "lagged", "instantaneous"]
fmin, fmax = 8, 35
```

(continues on next page)

(continued from previous page)

```
param_lr = {
    "penalty": "elasticnet",
    "l1_ratio": 0.15,
    "intercept_scaling": 1000.0,
    "solver": "saga",
}
param_ft = {"fmin": fmin, "fmax": fmax, "fs": fs}
step_fc = [
    ("spd", NearestSPD()),
    ("tg", TangentSpace(metric="riemann")),
    ("LogistReg", LogisticRegression(**param_lr)),
]
for sm in spectral_met:
    pname = sm + "+elasticnet"
    if sm == "cov":
        ppl_fc[pname] = Pipeline(
            steps=[("cov", Covariances(estimator="lwf"))] + step_fc
        )
    else:
        ft = Connectivities(**param_ft, method=sm)
        ppl_fc[pname] = Pipeline(steps=[("ft", ft)] + step_fc)
```

The ensemble classifier stacks a logistic regression on top of the three functional connectivity pipelines to make a global prediction

```
fc_estim = [(n, ppl_fc[n]) for n in ppl_fc]
cvkf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

lr = LogisticRegression(**param_lr)
ppl_ens["ensemble"] = StackingClassifier(
    estimators=fc_estim,
    cv=cvkf,
    n_jobs=1,
    final_estimator=lr,
    stack_method="predict_proba",
)
```

Evaluation

```
dataset_res = list()
all_ppl = {**ppl_baseline, **ppl_ens}

# Compute results
results = get_results(X, labels, all_ppl)
results = pd.DataFrame(results)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
```

(continues on next page)

(continued from previous page)

```
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
```

(continues on next page)

(continued from previous page)

```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
```

(continues on next page)

(continued from previous page)

```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
```

(continues on next page)

(continued from previous page)

```
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
↪reached which means the coef_ did not converge
  ConvergenceWarning,
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/linear_model/_sag.py:354: ConvergenceWarning: The max_iter was␣
```

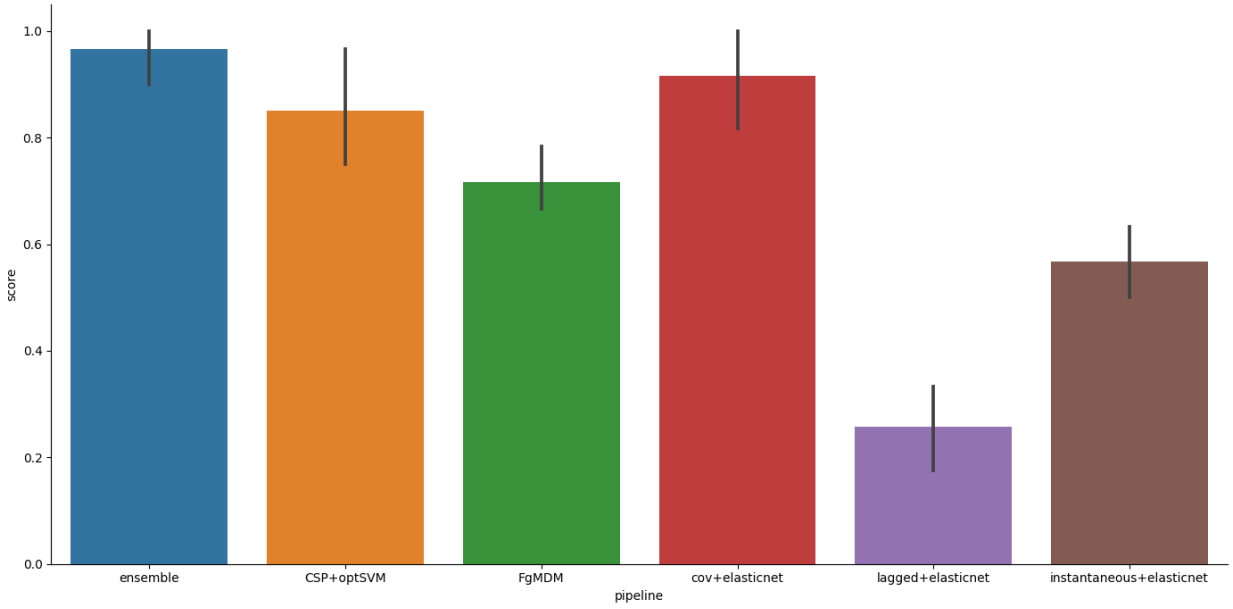
(continues on next page)

(continued from previous page)

```
↪reached which means the coef_ did not converge
  ConvergenceWarning,
```

Plot

```
list_fc_ens = ["ensemble", "CSP+optSVM", "FgMDM"] + \
    [sm + "+elasticnet" for sm in spectral_met]

g = sns.catplot(
    data=results,
    x="pipeline",
    y="score",
    kind="bar",
    order=list_fc_ens,
    height=7,
    aspect=2,
)
plt.show()
```



References

Total running time of the script: (0 minutes 42.124 seconds)

Frequency band selection on the manifold for motor imagery classification

Find optimal frequency band using class distinctiveness measure on the manifold and compare classification performance for motor imagery data to the baseline with no frequency band selection[1].

```
# Authors: Maria Sayu Yamamoto <maria-sayu.yamamoto@universite-paris-saclay.fr>
#
# License: BSD (3-clause)


import numpy as np
from time import time
from matplotlib import pyplot as plt

from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci

from sklearn.model_selection import cross_val_score, ShuffleSplit

from pyriemann.classification import MDM
from pyriemann.estimation import Covariances
from helpers.frequencybandselection_helpers import freq_selection_class_dis
```

Set basic parameters and read data

```
tmin, tmax = 0.5, 2.5
event_id = dict(T1=2, T2=3)
subject = 1
runs = [4, 8, 12]  # motor imagery: left hand vs right hand

raw_files = [
    read_raw_edf(f, preload=True) for f in eegbci.load_data(subject, runs)
]
raw = concatenate_raws(raw_files)
picks = pick_types(
    raw.info, meg=False, eeg=True, stim=False, eog=False, exclude='bads')
# subsample elecs
picks = picks[::2]

# cross validation
cv = ShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
```

[1] Class-distinctiveness-based frequency band selection on the Riemannian manifold for oscillatory activity-based BCIs: preliminary results M. S. Yamamoto, F. Lotte, F. Yger, and S. Chevallier. 44th Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC2022), 2022.

```
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S001/S001R04.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S001/S001R08.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S001/S001R12.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
```

Baseline pipeline without frequency band selection

Apply band-pass filter using a wide frequency band, 5-35 Hz. Train and evaluate classifier.

```
t0 = time()
raw_filter = raw.copy().filter(5., 35., method='iir', picks=picks,
                               verbose=False)

events, _ = events_from_annotations(raw_filter, event_id,
                                    verbose=False)

# Read epochs (train will be done only between 0.5 and 2.5 s)
epochs = Epochs(
    raw_filter,
    events,
    event_id,
    tmin,
    tmax,
    proj=True,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False)
labels = epochs.events[:, -1] - 2

# Get epochs
epochs_data_baseline = epochs.get_data(units="uV")

# Compute covariance matrices
cov_data_baseline = Covariances().transform(epochs_data_baseline)

# Set classifier
```

(continues on next page)

(continued from previous page)

```
model = MDM(metric=dict(mean='riemann', distance='riemann'))

# Classification with minimum distance to mean
acc_baseline = cross_val_score(model, cov_data_baseline, labels,
                               cv=cv, n_jobs=1)
t1 = time() - t0
```

Pipeline with a frequency band selection based on the class distinctiveness

Step1: Select frequency band maximizing class distinctiveness on training set.

Define parameters for frequency band selection

```
t2 = time()
freq_band = [5., 35.]
sub_band_width = 4.
sub_band_step = 2.
alpha = 0.4

# Select frequency band using training set
best_freq, all_class_dis = \
    freq_selection_class_dis(raw, freq_band, sub_band_width,
                             sub_band_step, alpha,
                             tmin, tmax,
                             picks, event_id,
                             cv,
                             return_class_dis=True, verbose=False)

print(f'Selected frequency band : {best_freq[0][0]} - {best_freq[0][1]} Hz')
```

```
Selected frequency band : 9.0 - 15.0 Hz
```

Step2: Train classifier using selected frequency band and evaluate performance using test set

```
# Apply band-pass filter using the best frequency band
best_raw_filter = raw.copy().filter(best_freq[0][0], best_freq[0][1],
                                    method='iir', picks=picks,
                                    verbose=False)

events, _ = events_from_annotations(best_raw_filter, event_id,
                                    verbose=False)

# Read epochs (train will be done only between 0.5 and 2.5s)
epochs = Epochs(
    best_raw_filter,
    events,
    event_id,
    tmin,
    tmax,
    proj=True,
    picks=picks,
```

(continues on next page)

(continued from previous page)

```
    baseline=None,
    preload=True,
    verbose=False)

# Get epochs
epochs_data_train = epochs.get_data(units="uV")

# Estimate covariance matrices
cov_data = Covariances().transform(epochs_data_train)

# Classification with minimum distance to mean
acc = cross_val_score(model, cov_data, labels, cv=cv, n_jobs=1)
t3 = time() - t2
```

Compare pipelines: accuracies and training times

```
print("Classification accuracy without frequency band selection: "
      + f"{acc_baseline[0]:.02f}")
print("Total computational time without frequency band selection: "
      + f"{t1:.5f} s")
print("Classification accuracy with frequency band selection: "
      + f"{acc[0]:.02f}")
print("Total computational time with frequency band selection: "
      + f"{t3:.5f} s")
```

```
Classification accuracy without frequency band selection: 0.56
Total computational time without frequency band selection: 0.32525 s
Classification accuracy with frequency band selection: 0.67
Total computational time with frequency band selection: 12.09287 s
```

Plot selected frequency bands

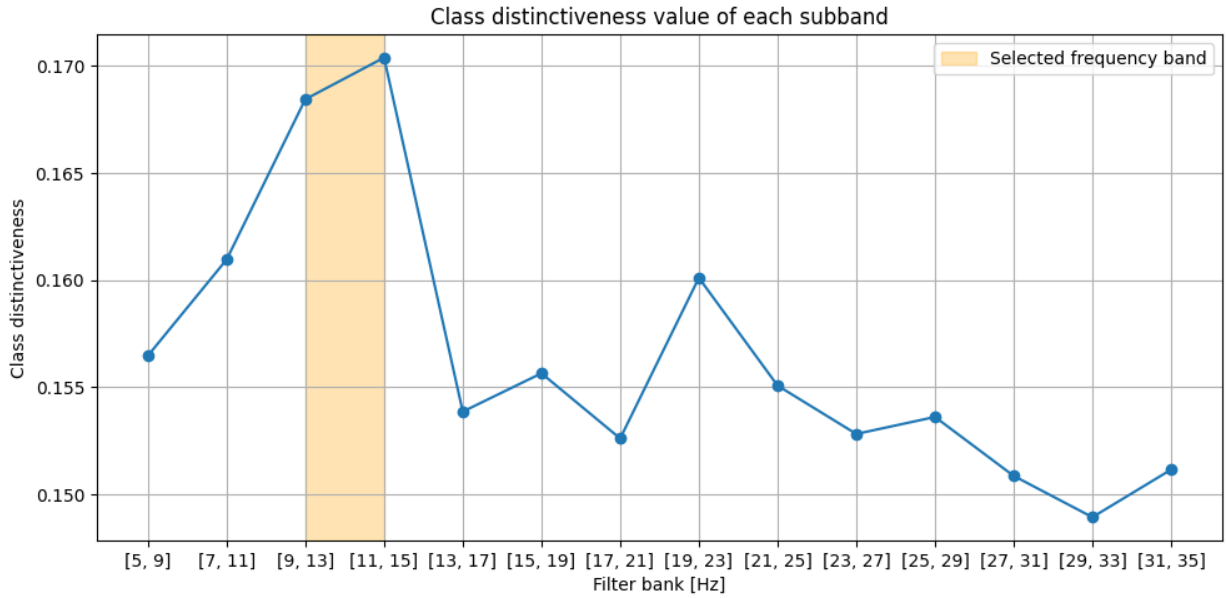
Plot the class distinctiveness values for each sub_band, along with the highlight of the finally selected frequency band.

```
subband_fmin = list(np.arange(freq_band[0],
                              freq_band[1] - sub_band_width + 1.,
                              sub_band_step))
subband_fmax = list(np.arange(freq_band[0] + sub_band_width,
                              freq_band[1] + 1., sub_band_step))
n_subband = len(subband_fmin)

x = list(range(0, n_subband, 1))
fig = plt.figure(figsize=(10, 5))

freq_start = subband_fmin.index(best_freq[0][0])
freq_end = subband_fmax.index(best_freq[0][1])

plt.subplot(1, 1, 1)
plt.grid()
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, all_class_dis[0], marker='o')
plt.xticks(list(range(0, 14, 1)),
           [[int(i), int(j)] for i, j in zip(subband_fmin, subband_fmax)])

plt.axvspan(freq_start, freq_end, color="orange", alpha=0.3,
            label='Selected frequency band')
plt.ylabel('Class distinctiveness')
plt.xlabel('Filter bank [Hz]')
plt.title('Class distinctiveness value of each subband')
plt.legend(loc='upper right')

fig.tight_layout()
plt.show()

print(f'Optimal frequency band for this subject is '
      f'{best_freq[0][0]} - {best_freq[0][1]} Hz')
```




```
Optimal frequency band for this subject is 9.0 - 15.0 Hz
```

References

Total running time of the script: (0 minutes 16.359 seconds)

4.8.5 Covariance estimation

Examples for covariance matrix estimation.

Robust covariance estimation

Comparison of robustness of different covariance estimators on a corrupted low-dimensional dataset. See also[1].

```
# Author: Quentin Barthélemy
#
# License: BSD (3-clause)

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.patches import Ellipse
import matplotlib.transforms as transforms

from pyriemann.estimation import Covariances
```

```
def plot_cov_ellipse(ax, cov, n_std=2.5, **kwargs):
    """Inspired by
    https://matplotlib.org/stable/gallery/statistics/confidence_ellipse.html
    """
    pearson = cov[0, 1] / np.sqrt(cov[0, 0] * cov[1, 1])
    ell_radius_x = np.sqrt(1 + pearson)
    ell_radius_y = np.sqrt(1 - pearson)
    ellipse = Ellipse((0, 0), width=ell_radius_x * 2, height=ell_radius_y * 2,
                      facecolor='none', **kwargs)
    scale_x = np.sqrt(cov[0, 0]) * n_std
    scale_y = np.sqrt(cov[1, 1]) * n_std
    transf = transforms.Affine2D().rotate_deg(45).scale(scale_x, scale_y)
    ellipse.set_transform(transf + ax.transData)
    return ax.add_patch(ellipse)


def plot_cov_estimators(ax, X, estimators):
    plot_cov_ellipse(ax, C_ref, edgecolor="C0", label='Reference')
    for i, est in enumerate(estimators):
        C = Covariances(estimator=est).transform(X[np.newaxis, ...])[0]
        plot_cov_ellipse(ax, C, edgecolor=f"C{i+2}", label=est)
    ax.legend(loc='upper left')
    return ax
```

[1] https://scikit-learn.org/stable/auto_examples/covariance/plot_mahalanobis_distances.html # noqa

Generate a Gaussian dataset

Input samples are generated from a centered 2D Gaussian distribution considered as the reference.

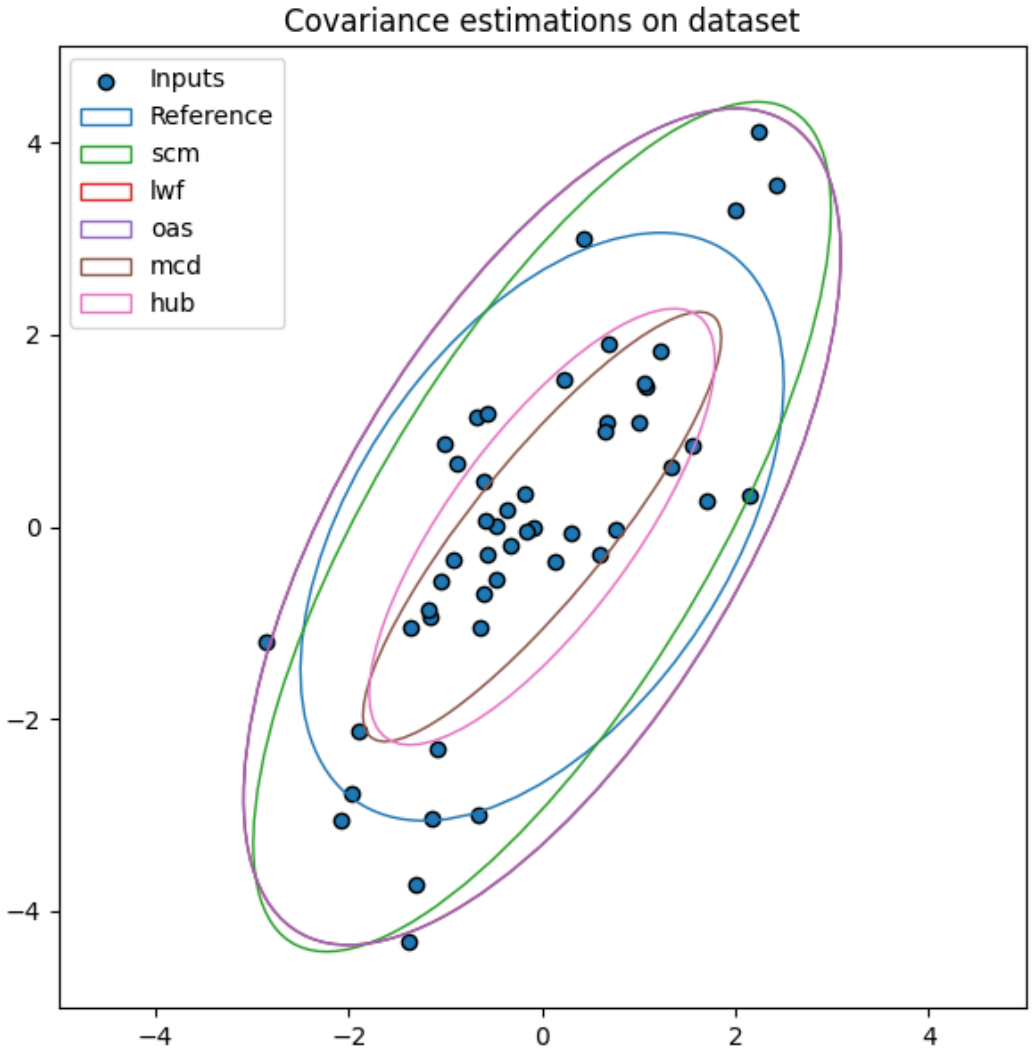
```
rs = np.random.RandomState(2023)

n_channels, n_inliers = 2, 50
C_ref = np.array([[1, 0.6], [0.6, 1.5]])
X = C_ref @ rs.randn(n_channels, n_inliers)
```

Estimate covariance matrices on dataset

Compare reference covariance matrix to different estimators:

- sample covariance matrix (scm),
- Ledoit-Wolf shrunk covariance matrix (lwf),
- oracle approximating shrunk covariance matrix (oas),
- minimum covariance determinant matrix (mcd),
- robust Huber's M-estimator based covariance matrix (hub).

```
estimators = ["scm", "lwf", "oas", "mcd", "hub"]

fig, ax = plt.subplots(figsize=(7, 7))
ax.set_title("Covariance estimations on dataset")
ax.scatter(X[0], X[1], c='C0', edgecolors="k", label='Inputs')
ax = plot_cov_estimators(ax, X, estimators)
xlim, ylim = ax.get_xlim(), ax.get_ylim()
min_, max_ = min(xlim[0], ylim[0]), max(xlim[1], ylim[1])
ax.set_xlim(min_, max_)
ax.set_ylim(min_, max_)
plt.show()
```



Add outliers to dataset

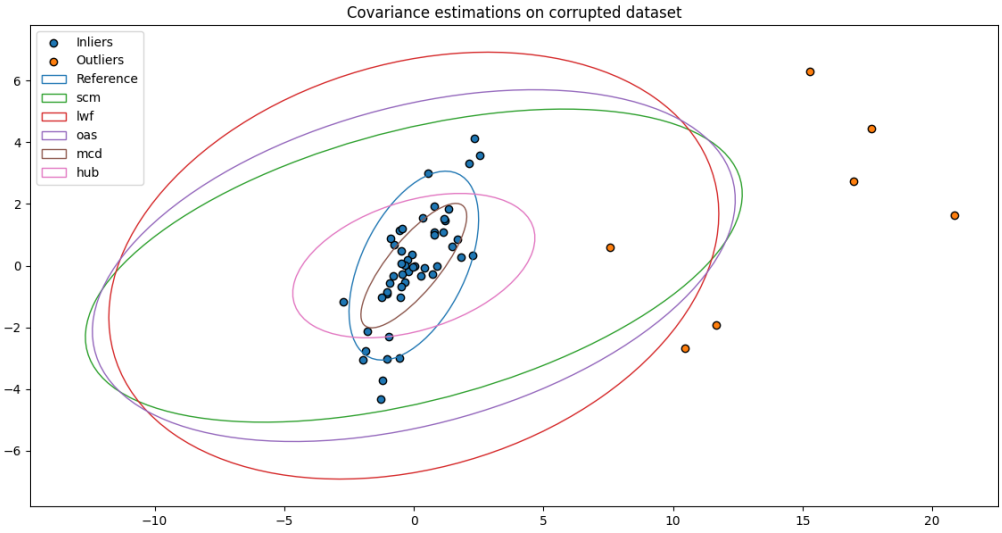
Outliers are added to the dataset.

```
n_outliers = 7
mu, scale = np.array([15, 1]), 5
Xout = mu[:, np.newaxis] + scale * rs.randn(n_channels, n_outliers)
X = np.concatenate((X, Xout), axis=1)
```

Estimate covariance matrices on corrupted dataset

Compare robustness of the different estimators.

```
fig, ax = plt.subplots(figsize=(14, 7))
ax.set_title("Covariance estimations on corrupted dataset")
ax.scatter(X[0, :n_inliers], X[1, :n_inliers], c='C0', edgecolors="k",
           label='Inliers')
ax.scatter(X[0, n_inliers:], X[1, n_inliers:], c='C1', edgecolors="k",
           label='Outliers')
ax = plot_cov_estimators(ax, X, estimators)
plt.show()
```



References

Total running time of the script: (0 minutes 0.411 seconds)

Compare covariance and kernel estimators with different time windows

Comparison of covariance estimators for different EEG signal lengths and their impact on classification[1]. Kernel estimators are also compared[2].

```
# Authors: Sylvain Chevallier and Quentin Barthélemy
#
# License: BSD (3-clause)

import numpy as np
import pandas as pd
```

(continues on next page)

[1] Riemannian classification for SSVEP based BCI: offline versus online implementations S. Chevallier, E. Kalunga, Q. Barthélemy, F. Yger. Brain–Computer Interfaces Handbook: Technological and Theoretical Advances, 2018.

[2] Beyond Covariance: Feature Representation with Nonlinear Kernel Matrices # noqa L. Wang, J. Zhang, L. Zhou, C. Tang, W Li. ICCV, 2015.

(continued from previous page)

```
from matplotlib import pyplot as plt
import seaborn as sns

from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.pipeline import make_pipeline

from pyriemann.estimation import Covariances, Kernels
from pyriemann.utils.distance import distance
from pyriemann.classification import MDM
```

Estimating covariance on synthetic data

Generate synthetic data, sampled from a distribution considered as the groundtruth.

```
rs = np.random.RandomState(42)
n_matrices, n_channels, n_times = 10, 5, 1000
var = 2.0 + 0.1 * rs.randn(n_matrices, n_channels)
A = 2 * rs.rand(n_channels, n_channels) - 1
A /= np.linalg.norm(A, axis=1)[:, np.newaxis]
true_covs = np.empty(shape=(n_matrices, n_channels, n_channels))
X = np.empty(shape=(n_matrices, n_channels, n_times))
for i in range(n_matrices):
    true_covs[i] = A @ np.diag(var[i]) @ A.T
    X[i] = rs.multivariate_normal(
        np.array([0.0] * n_channels), true_covs[i], size=n_times
    ).T
```

Covariances() class offers several estimators:

- sample covariance matrix (SCM),
- Ledoit-Wolf (LWF),
- Schaefer-Strimmer (SCH),
- oracle approximating shrunk (OAS) covariance,
- minimum covariance determinant (MCD),
- and others.

We will compare the distance of LWF, OAS and SCH estimators with the groundtruth, while increasing epoch length.

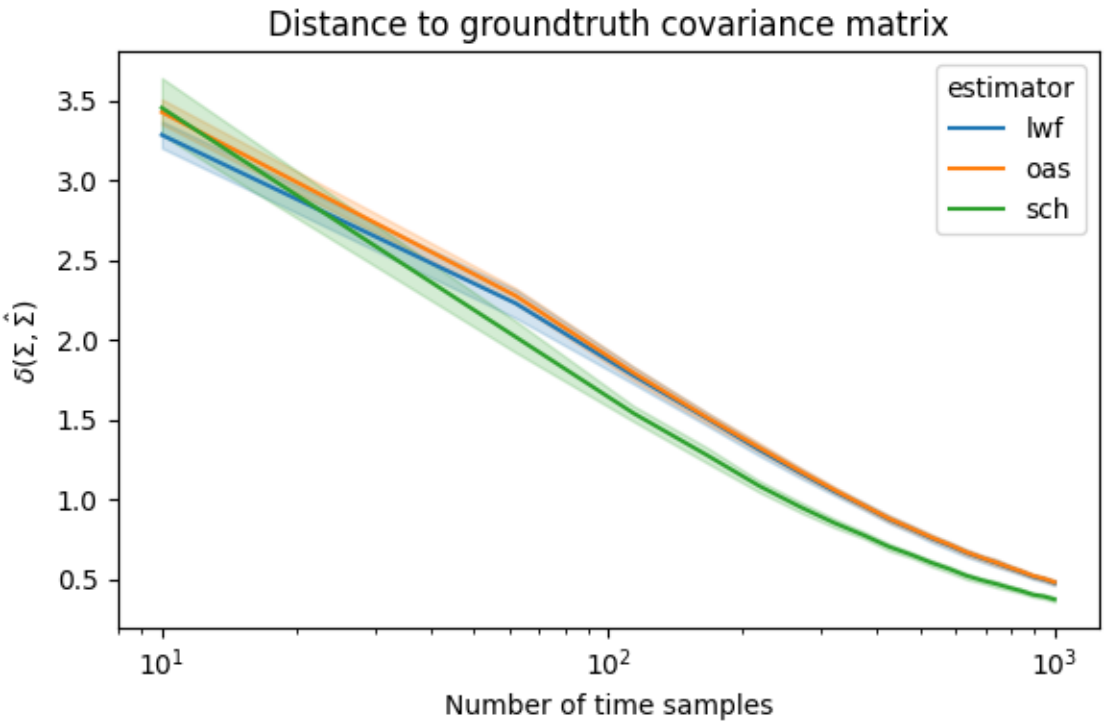
```
estimators = ["lwf", "oas", "sch"]
w_len = np.linspace(10, n_times, 20, dtype=int)
dfd = list()
for est in estimators:
    for wl in w_len:
        est_covs = Covariances(estimator=est).transform(X[:, :, :wl])
        dists = distance(est_covs, true_covs, metric="riemann")
```

(continues on next page)

(continued from previous page)

```
        dfd.extend([dict(estimator=est, wlen=wl, dist=d) for d in dists])
dfd = pd.DataFrame(dfd)
```

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.set(xscale="log")
sns.lineplot(data=dfd, x="wlen", y="dist", hue="estimator", ax=ax)
ax.set_title("Distance to groundtruth covariance matrix")
ax.set_xlabel("Number of time samples")
ax.set_ylabel(r"$\delta(\Sigma, \hat{\Sigma})$")
plt.tight_layout()
plt.show()
```



Choice of estimator for motor imagery data

Loading data from PhysioNet MI dataset, for subject 1.

```
event_id = dict(hands=2, feet=3)
subject = 1
runs = [6, 10, 14]  # motor imagery: hands vs feet
raw_files = [
    read_raw_edf(f, preload=True, stim_channel="auto")
    for f in eegbci.load_data(subject, runs)
]
raw = concatenate_raws(raw_files)
picks = pick_types(raw.info, eeg=True, exclude="bads")
```

(continues on next page)

(continued from previous page)

```
# subsample elecs
picks = picks[::2]
# Apply band-pass filter
raw.filter(7.0, 35.0, method="iir", picks=picks, skip_by_annotation="edge")
events, _ = events_from_annotations(raw, event_id=dict(T1=2, T2=3))
event_ids = dict(hands=2, feet=3)
```

```
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S001/S001R06.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S001/S001R10.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Extracting EDF parameters from /home/docs/mne_data/MNE-eegbci-data/files/eegmmidb/1.0.0/
↪S001/S001R14.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 19999  =      0.000 ...   124.994 secs...
Filtering a subset of channels. The highpass and lowpass values in the measurement info␣
↪will not be updated.
Filtering raw data in 3 contiguous segments
Setting up band-pass filter from 7 - 35 Hz

IIR filter parameters
---------------------
Butterworth bandpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 16 (effective, after forward-backward)
- Cutoffs at 7.00, 35.00 Hz: -6.02, -6.02 dB

Used Annotations descriptions: ['T1', 'T2']
```

Influence of shrinkage to estimate matrices

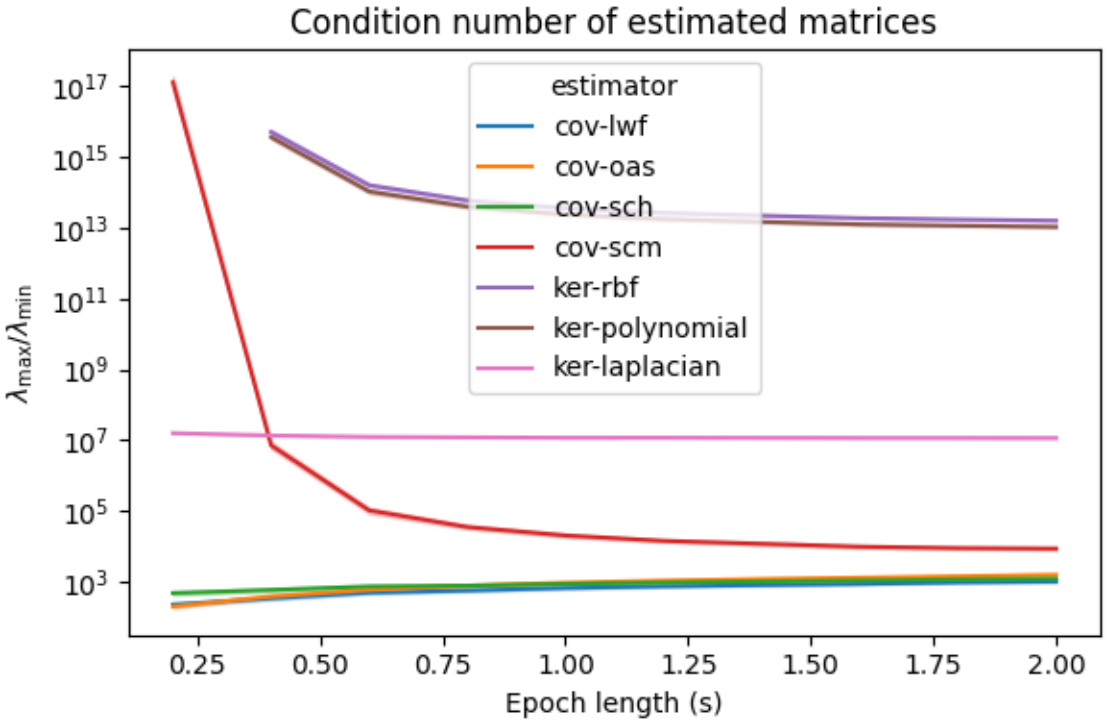
Sample covariance matrix (SCM) estimation could lead to ill-conditionned matrices depending on the quality and quantity of EEG data available. Matrix condition number is the ratio between the highest and lowest eigenvalues: high values indicates ill-conditionned matrices that are not suitable for classification. A common approach to mitigate this issue is to regularize covariance matrices by shrinkage, like in Ledoit-Wolf, Schaefer-Strimmer or oracle estimators.

In addition to covariance matrices, kernel matrices are computed for three kernel functions:

- radial basis function (RBF),
- polynomial,
- Laplacian.

```
estimators = [
    "cov-lwf", "cov-oas", "cov-sch", "cov-scm",
    "ker-rbf", "ker-polynomial", "ker-laplacian",
]
tmin = -0.2
w_len = np.linspace(0.2, 2, 10)
n_matrices = 45
dfc = list()

for wl in w_len:
    X = Epochs(
        raw,
        events,
        event_ids,
        tmin,
        tmin + wl,
        picks=picks,
        preload=True,
        verbose=False,
    ).get_data()
    for est in estimators:
        est_class, est_param = est.split('-')
        if est_class == "ker":
            covs = Kernels(metric=est_param).transform(X)
        else:
            covs = Covariances(estimator=est_param).transform(X)
        evals, _ = np.linalg.eigh(covs)
        dfc.extend([dict(estimator=est, wlen=wl, cond=e[-1] / e[0])
                    for e in evals])
dfc = pd.DataFrame(dfc)
```

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.set(yscale="log")
sns.lineplot(data=dfc, x="wlen", y="cond", hue="estimator", ax=ax)
ax.set_title("Condition number of estimated matrices")
ax.set_xlabel("Epoch length (s)")
ax.set_ylabel(r"$\lambda_{\max}$/$\lambda_{\min}$")
plt.tight_layout()
plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/pandas/core/arraylike.py:364: RuntimeWarning: invalid value encountered in␣
↪log10
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

Picking a good estimator for classification

The choice of estimator have an impact on classification, especially when the matrices are estimated on short time windows.

```
tmin = 0.0
w_len = np.linspace(0.2, 2.0, 5)
n_matrices, n_splits = 45, 5
dfa = list()
sc = "balanced_accuracy"

cv = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=123)
for wl in w_len:
    epochs = Epochs(
        raw,
        events,
        event_ids,
        tmin,
        tmin + wl,
        proj=True,
        picks=picks,
        preload=True,
```

(continues on next page)

(continued from previous page)

```
        baseline=None,
        verbose=False,
    )
    X = epochs.get_data()
    y = np.array([0 if ev == 2 else 1 for ev in epochs.events[:, -1]])
    for est in estimators:
        est_class, est_param = est.split('-')
        if est_class == "ker":
            clf = make_pipeline(Kernels(metric=est_param), MDM())
        else:
            clf = make_pipeline(Covariances(estimator=est_param), MDM())
        try:
            score = cross_val_score(clf, X, y, cv=cv, scoring=sc)
            dfa += [dict(estimator=est, wlen=wl, accuracy=sc) for sc in score]
        except ValueError:
            print(f"{est}: {wl} is not sufficent to estimate a SPD matrix")
            dfa += [dict(estimator=est, wlen=wl, accuracy=np.nan)] * n_splits
dfa = pd.DataFrame(dfa)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/model_selection/_validation.py:372: FitFailedWarning:
5 fits failed out of a total of 5.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_score=
↪'raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
5 fits failed with the following error:
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.
↪7/site-packages/sklearn/model_selection/_validation.py", line 680, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.
↪7/site-packages/sklearn/pipeline.py", line 394, in fit
    self._final_estimator.fit(Xt, y, **fit_params_last_step)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
```

(continues on next page)

(continued from previous page)

```
↪pyriemann/classification.py", line 124, in fit
    for ll in self.classes_]
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/classification.py", line 124, in <listcomp>
    for ll in self.classes_]
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/mean.py", line 591, in mean_covariance
    C = mean_methods[metric](covmats, sample_weight=sample_weight, *args)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/mean.py", line 458, in mean_riemann
    C = C12 @ expm(nu * J) @ C12
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/base.py", line 95, in expm
    return _matrix_operator(C, np.exp)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/base.py", line 15, in _matrix_operator
    "Matrices must be positive definite. Add "
ValueError: Matrices must be positive definite. Add regularization to avoid this error.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/model_selection/_validation.py:372: FitFailedWarning:
5 fits failed out of a total of 5.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_score=
↪'raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
5 fits failed with the following error:
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.
↪7/site-packages/sklearn/model_selection/_validation.py", line 680, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.
↪7/site-packages/sklearn/pipeline.py", line 394, in fit
    self._final_estimator.fit(Xt, y, **fit_params_last_step)
```

(continues on next page)

(continued from previous page)

```
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/classification.py", line 124, in fit
    for ll in self.classes_]
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/classification.py", line 124, in <listcomp>
    for ll in self.classes_]
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/mean.py", line 591, in mean_covariance
    C = mean_methods[metric](covmats, sample_weight=sample_weight, *args)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/mean.py", line 458, in mean_riemann
    C = C12 @ expm(nu * J) @ C12
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/base.py", line 95, in expm
    return _matrix_operator(C, np.exp)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/base.py", line 15, in _matrix_operator
    "Matrices must be positive definite. Add "
ValueError: Matrices must be positive definite. Add regularization to avoid this error.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/base.py:18: RuntimeWarning: invalid value encountered in log
  eigvals = operator(eigvals)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/sklearn/model_selection/_validation.py:372: FitFailedWarning:
5 fits failed out of a total of 5.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_score=
↪'raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
5 fits failed with the following error:
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.
↪7/site-packages/sklearn/model_selection/_validation.py", line 680, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.
↪7/site-packages/sklearn/pipeline.py", line 394, in fit
```

(continues on next page)

(continued from previous page)

```
    self._final_estimator.fit(Xt, y, **fit_params_last_step)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/classification.py", line 124, in fit
    for ll in self.classes_]
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/classification.py", line 124, in <listcomp>
    for ll in self.classes_]
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/mean.py", line 591, in mean_covariance
    C = mean_methods[metric](covmats, sample_weight=sample_weight, *args)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/mean.py", line 458, in mean_riemann
    C = C12 @ expm(nu * J) @ C12
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/base.py", line 95, in expm
    return _matrix_operator(C, np.exp)
  File "/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/
↪pyriemann/utils/base.py", line 15, in _matrix_operator
    "Matrices must be positive definite. Add "
ValueError: Matrices must be positive definite. Add regularization to avoid this error.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
```

(continues on next page)

(continued from previous page)

```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
```

(continues on next page)

(continued from previous page)

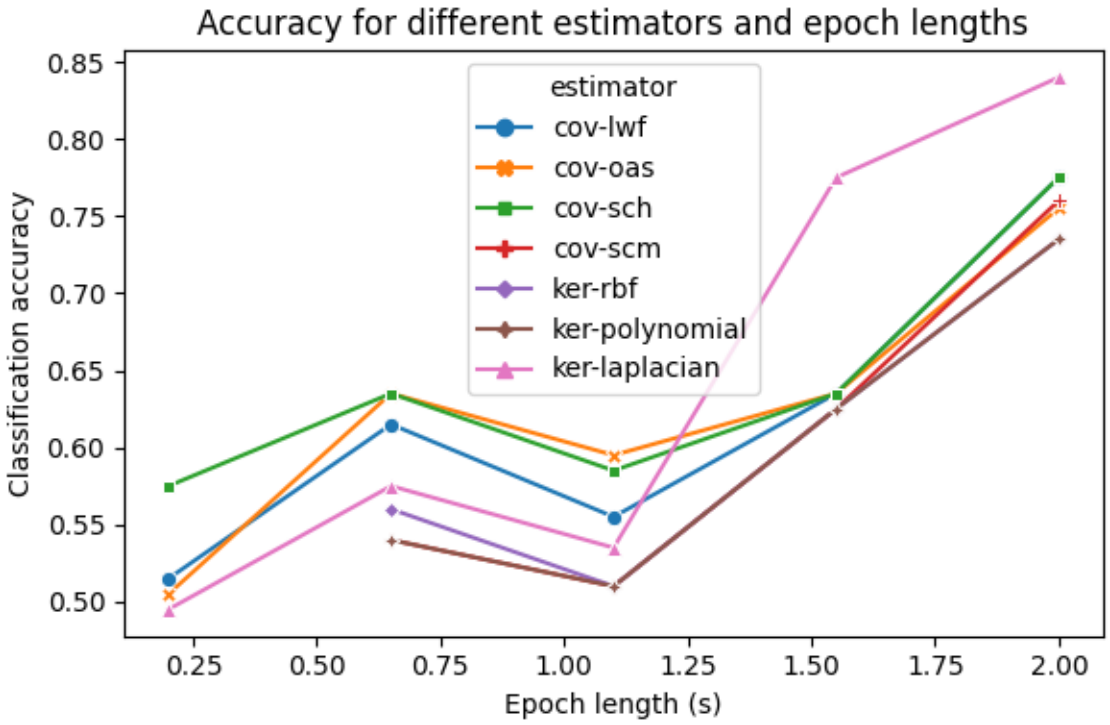
```
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:470: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
```

```
fig, ax = plt.subplots(figsize=(6, 4))
sns.lineplot(
    data=dfa,
    x="wlen",
    y="accuracy",
    hue="estimator",
    style="estimator",
    ax=ax,
    ci=None,
    markers=True,
    dashes=False,
)
ax.set_title("Accuracy for different estimators and epoch lengths")
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Epoch length (s)")
ax.set_ylabel("Classification accuracy")
plt.tight_layout()
plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/examples/
↪signal/plot_covariance_estimation.py:223: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=None` for the same effect.

  dashes=False,
```

References

Total running time of the script: (1 minutes 17.503 seconds)

4.8.6 Simulated data

Examples using datasets sampled from known probability distributions.

Sample from the Riemannian Gaussian distribution in the SPD manifold

Spectral embedding of samples from the Riemannian Gaussian distribution with different centerings and dispersions.

```
# Authors: Pedro Rodrigues <pedro.rodrigues@melix.org>
#
# License: BSD (3-clause)

import numpy as np
import matplotlib.pyplot as plt

from pyriemann.embedding import SpectralEmbedding
from pyriemann.datasets import sample_gaussian_spd, generate_random_spd_matrix


print(__doc__)
```

Set parameters for sampling from the Riemannian Gaussian distribution

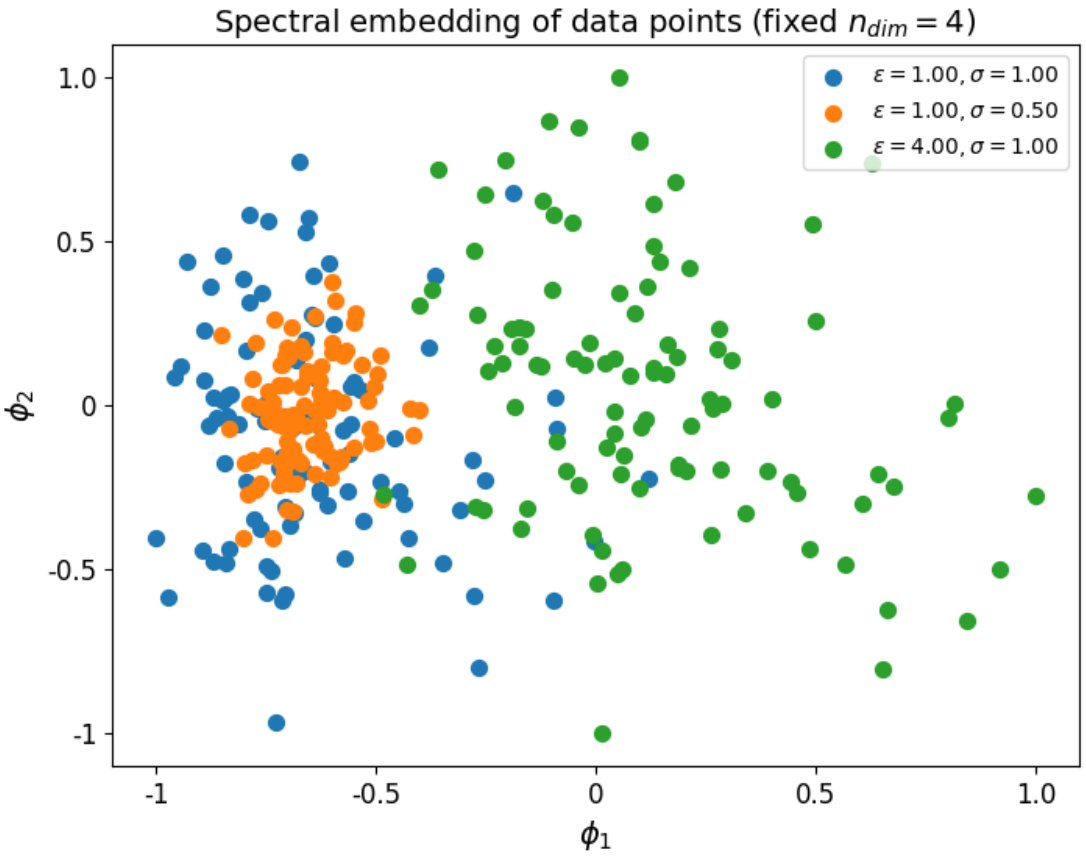
```
n_matrices = 100  # how many SPD matrices to generate
n_dim = 2  # number of dimensions of the SPD matrices
sigma = 1.0  # dispersion of the Gaussian distribution
epsilon = 4.0  # parameter for controlling the distance between centers
random_state = 42  # ensure reproducibility

# Generate the samples on three different conditions
mean = generate_random_spd_matrix(n_dim)  # random reference point

samples_1 = sample_gaussian_spd(n_matrices=n_matrices,
                                mean=mean,
                                sigma=sigma,
                                random_state=random_state)
samples_2 = sample_gaussian_spd(n_matrices=n_matrices,
                                mean=mean,
                                sigma=sigma/2,
                                random_state=random_state)
samples_3 = sample_gaussian_spd(n_matrices=n_matrices,
                                mean=epsilon*mean,
                                sigma=sigma,
                                random_state=random_state)

# Stack all of the samples into one data array for the embedding
samples = np.concatenate([samples_1, samples_2, samples_3])
labels = np.array(n_matrices*[1] + n_matrices*[2] + n_matrices*[3])
```

Apply the spectral embedding over the SPD matrices

```
lapl = SpectralEmbedding(metric='riemann', n_components=2)
embd = lapl.fit_transform(X=samples)
```

Plot the results

```
fig, ax = plt.subplots(figsize=(8, 6))

colors = {1: 'C0', 2: 'C1', 3: 'C2'}
for i in range(len(samples)):
    ax.scatter(embd[i, 0], embd[i, 1], c=colors[labels[i]], s=50)
ax.scatter([], [], c='C0', s=50, label=r'$\varepsilon = 1.00, \sigma = 1.00$')
ax.scatter([], [], c='C1', s=50, label=r'$\varepsilon = 1.00, \sigma = 0.50$')
ax.scatter([], [], c='C2', s=50, label=r'$\varepsilon = 4.00, \sigma = 1.00$')
ax.set_xticks([-1, -0.5, 0, 0.5, 1.0])
ax.set_xticklabels([-1, -0.5, 0, 0.5, 1.0], fontsize=12)
ax.set_yticks([-1, -0.5, 0, 0.5, 1.0])
ax.set_yticklabels([-1, -0.5, 0, 0.5, 1.0], fontsize=12)
ax.set_title(r'Spectral embedding of data points (fixed $n_{dim} = 4$)',
             fontsize=14)
ax.set_xlabel(r'$\phi_1$', fontsize=14)
ax.set_ylabel(r'$\phi_2$', fontsize=14)
ax.legend()

plt.show()
```



Total running time of the script: (0 minutes 3.889 seconds)

Classification accuracy vs class distinctiveness vs class separability

Generate several datasets containing data points from two-classes. Each class is generated with a Riemannian Gaussian distribution centered at the class mean and with the same dispersion sigma. The distance between the class means is parametrized by Delta, which we make vary between zero and 5*sigma. We illustrate how the accuracy of the MDM classifier and the value of the class distinctiveness[1] vary when Delta increases.

```
# Authors: Pedro Rodrigues <pedro.rodrigues@melix.org>
#          Maria Sayu Yamamoto <maria-sayu.yamamoto@universite-paris-saclay.fr>
#
# License: BSD (3-clause)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score, StratifiedKFold

from pyriemann.classification import MDM
from pyriemann.datasets import make_gaussian_blobs
from pyriemann.classification import class_distinctiveness
```

Set general parameters for the illustrations

```
n_matrices = 100  # how many matrices to sample on each class
n_dim = 4  # dimensionality of the data points
sigma = 1.0  # dispersion of the Gaussian distributions
random_state = 42  # ensure reproducibility
```

Loop over different levels of separability between the classes

```
scores_array = []
class_dis_array = []
deltas_array = np.linspace(0, 3*sigma, 10)

for delta in deltas_array:
    # generate data points for a classification problem
    X, y = make_gaussian_blobs(n_matrices=n_matrices,
                               n_dim=n_dim,
                               class_sep=delta,
                               class_disp=sigma,
                               random_state=random_state,
                               n_jobs=4)

    # measure class distinctiveness of training data for each split
    skf = StratifiedKFold(n_splits=5)
    all_class_dis = []
    for train_ind, _ in skf.split(X, y):
        class_dis = class_distinctiveness(X[train_ind], y[train_ind],
                                          exponent=1, metric='riemann',
                                          return_num_denom=False)
        all_class_dis.append(class_dis)
```

(continues on next page)

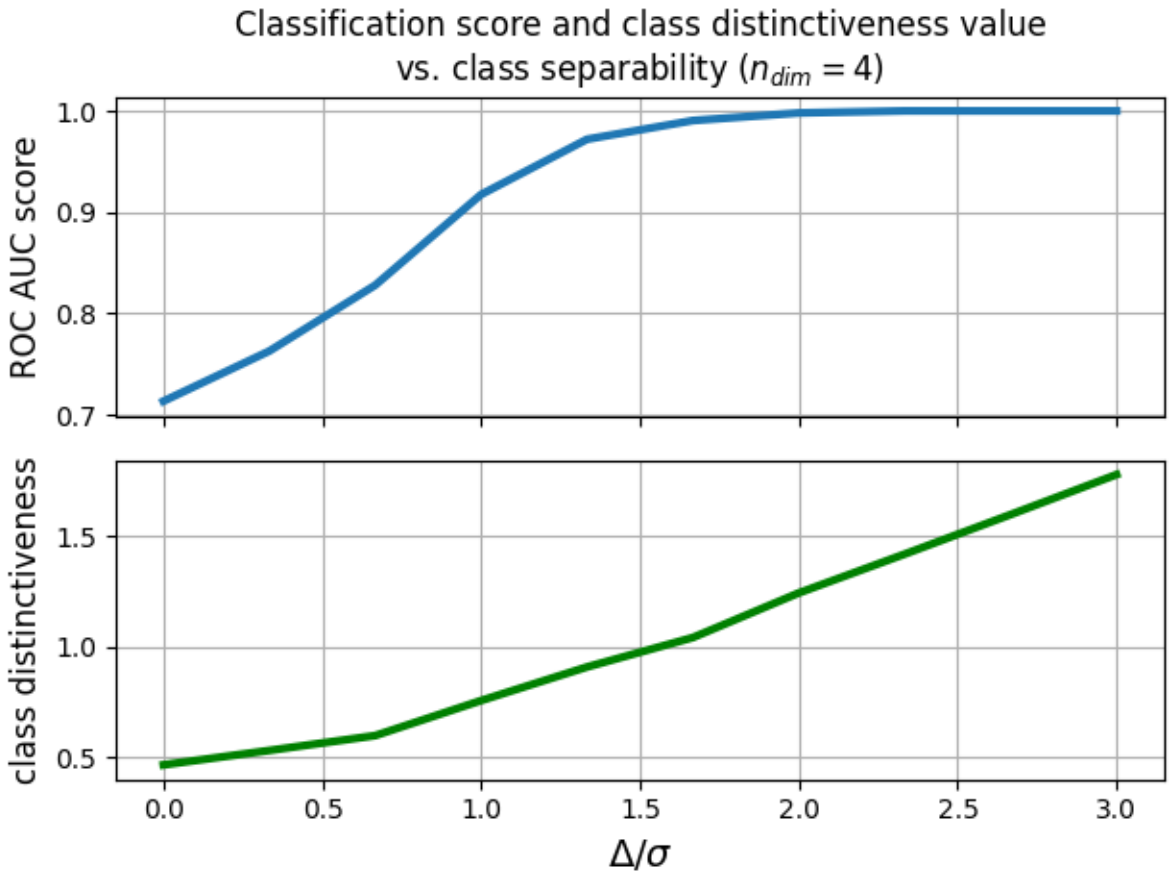
[1] Class-distinctiveness-based frequency band selection on the Riemannian manifold for oscillatory activity-based BCIs: preliminary results M. S. Yamamoto, F. Lotte, F. Yger, and S. Chevallier. 44th Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC2022), 2022.

(continued from previous page)

```
    # average class distinctiveness across splits
    mean_class_dis = np.mean(all_class_dis)
    class_dis_array.append(mean_class_dis)

    # Now let's train a MDM classifier and measure its performance
    clf = MDM()

    # get the classification score for this setup
    scores_array.append(
        cross_val_score(clf, X, y, cv=skf, scoring='roc_auc').mean())

scores_array = np.array(scores_array)
class_dis_array = np.array(class_dis_array)
```

Plot the results

```
fig, (ax1, ax2) = plt.subplots(sharex=True, nrows=2)

ax1.plot(deltas_array, scores_array, lw=3.0, label=r'ROC AUC score')
ax2.plot(deltas_array, class_dis_array, lw=3.0, color='g',
         label='Class Distinctiveness')

ax2.set_xlabel(r'$\Delta/\sigma$', fontsize=14)
ax1.set_ylabel(r'ROC AUC score', fontsize=12)
ax2.set_ylabel(r'class distinctiveness', fontsize=12)
ax1.set_title('Classification score and class distinctiveness value\n'
              r'vs. class separability ($n_{dim} = 4$)',
              fontsize=12)

ax1.grid(True)
ax2.grid(True)
fig.tight_layout()
plt.show()
```



References

Total running time of the script: (0 minutes 40.213 seconds)

Mean and median comparison

A comparison between Euclidean and Riemannian means[1], and Euclidean and Riemannian geometric medians[2], on low-dimensional synthetic datasets.

```
# Authors: Quentin Barthélemy
#
# License: BSD (3-clause)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs
from pyriemann.datasets import make_outliers
```

(continues on next page)

[1] Review of Riemannian distances and divergences, applied to SSVEP-based BCI S. Chevallier, E. K. Kalunga, Q. Barthélemy, E. Monacelli. Neuroinformatics, Springer, 2021, 19 (1), pp.93-106

[2] The geometric median on Riemannian manifolds with application to robust atlas estimation PT. Fletcher, S. Venkatasubramanian S and S. Joshi. NeuroImage, 2009, 45(1), S143-S152

(continued from previous page)

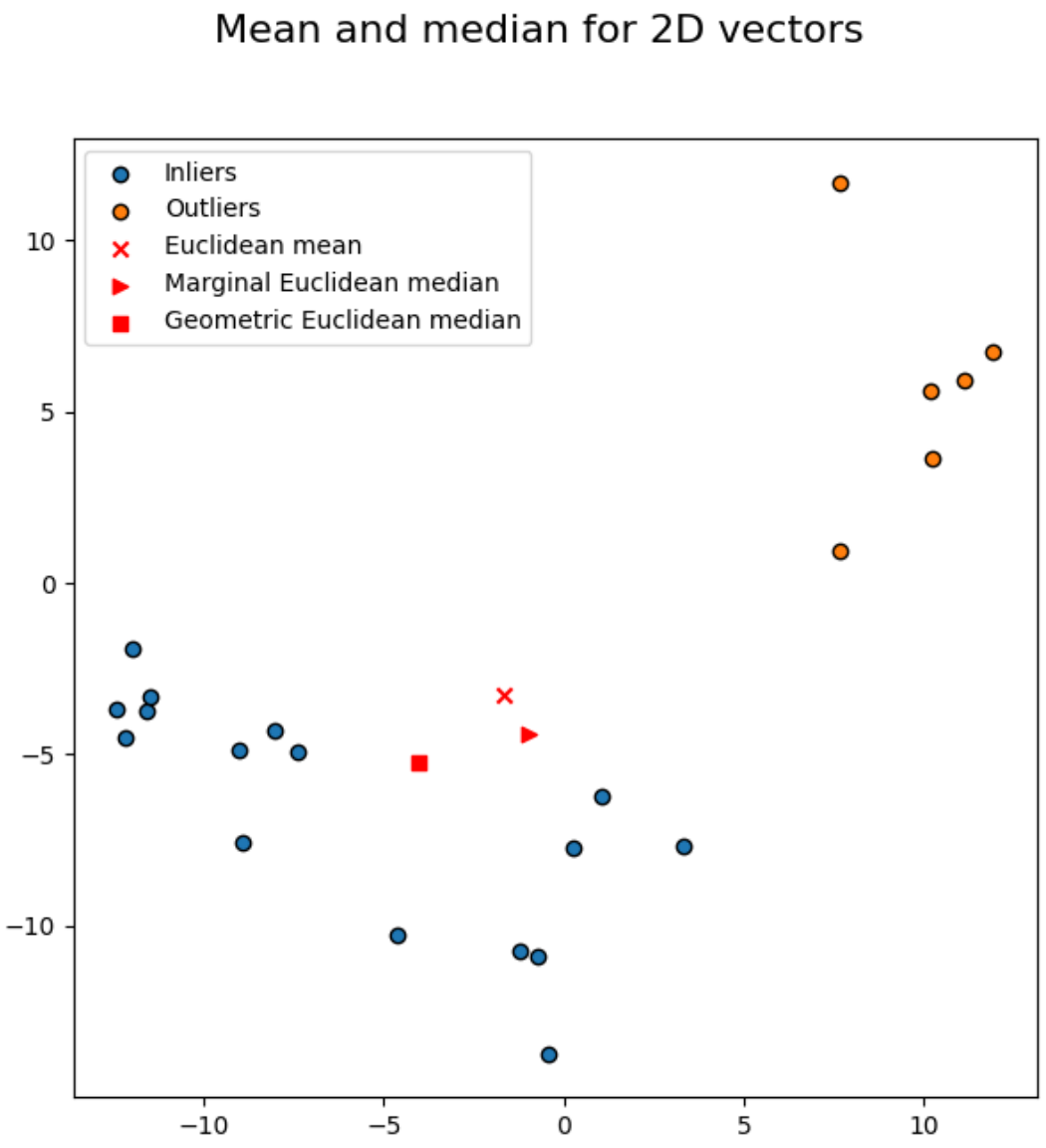
```
from pyriemann.utils import mean_euclid, mean_riemann
from pyriemann.utils import median_euclid, median_riemann
from pyriemann.clustering import Potato

rs = np.random.RandomState(17)
```

Data in vector space

Dataset of 2D vectors, reproducing Fig 1 of reference[Page 115, 2].

Notice how the few outliers at the top right of the picture have forced the mean away from the points, whereas the geometric median remains centrally located.

```
X, y = make_blobs(
    n_samples=[7, 9, 6],
    n_features=2,
    centers=np.array([[-1, -10], [-10, -4], [10, 5]]),
    cluster_std=[2, 2, 2],
    random_state=rs
)
is_inlier = (y <= 1)

C_mean = mean_euclid(X[..., np.newaxis])
C_mmed = np.median(X, axis=0)
C_gmed = median_euclid(X[..., np.newaxis])

fig, ax = plt.subplots(figsize=(7, 7))
fig.suptitle("Mean and median for 2D vectors", fontsize=16)
ax.scatter(X[is_inlier, 0], X[is_inlier, 1], c='C0', edgecolors="k",
           label='Inliers')
ax.scatter(X[~is_inlier, 0], X[~is_inlier, 1], c='C1', edgecolors="k",
           label='Outliers')
ax.scatter(C_mean[0], C_mean[1], c='r', marker="x", label='Euclidean mean')
ax.scatter(C_mmed[0], C_mmed[1], c='r', marker=">",
           label='Marginal Euclidean median')
ax.scatter(C_gmed[0], C_gmed[1], c='r', marker="s",
           label='Geometric Euclidean median')
ax.legend(loc='upper left')
plt.show()
```



Data in manifold of SPD matrices

Dataset of 2x2 SPD matrices.

A dynamic display is required if you want to rotate or zoom the 3D figure. This 3D plot can be tricky to interpret. 2x2 SPD matrices can be viewed as spatial coordinates contained in a hyper-cone[3]. In Euclidean geometry, null matrix is the center of space. In Riemannian geometry, identity matrix is the center of the unbounded and non-linear manifold[Page 117, 3]: due to log(.)^2 in the affine-invariant distance, an eigenvalue of 10 contributes to the distance from the identity as much as an eigenvalue 0.1.

[3] EEG source analysis M. Congedo. HdR, 2013, Chap IX

```
n_channels, n_inliers, n_outliers = 2, 16, 6

Cin = 0.2 * np.eye(n_channels)
Xin = make_outliers(n_inliers, Cin, 0.5, outlier_coeff=1, random_state=rs)
Xout = make_outliers(
    n_outliers, 4 * np.eye(n_channels), 0.5, outlier_coeff=1, random_state=rs)
X = np.concatenate([Xin, Xout])

C_emean = mean_euclid(X)
C_rmean = mean_riemann(X)
C_emed = median_euclid(X)
C_rmed = median_riemann(X)

fig2 = plt.figure(figsize=(7, 7))
fig2.suptitle("Means and medians for 2x2 SPD matrices", fontsize=16)
ax2 = plt.subplot(111, projection='3d')
ax2.scatter(1, 0, 1, c="k", marker="+", s=50, label='Identity')
ax2.scatter(Xin[:, 0, 0], Xin[:, 0, 1], Xin[:, 1, 1], c="C0", edgecolors="k",
            label='Inliers')
ax2.scatter(Xout[:, 0, 0], Xout[:, 0, 1], Xout[:, 1, 1], c="C1",
            edgecolors="k", label='Outliers')
ax2.scatter(C_emean[0, 0], C_emean[0, 1], C_emean[1, 1], c="r", marker="x",
            label='Euclidean mean')
ax2.scatter(C_rmean[0, 0], C_rmean[0, 1], C_rmean[1, 1], c="m", marker="x",
            label='Riemannian mean')
ax2.scatter(C_emed[0, 0], C_emed[0, 1], C_emed[1, 1], c="r", marker="s",
            label='Euclidean median')
ax2.scatter(C_rmed[0, 0], C_rmed[0, 1], C_rmed[1, 1], c="m", marker="s",
            label='Riemannian median')
ax2.legend(loc='center left', bbox_to_anchor=(0.7, 0.6))
plt.show()
```

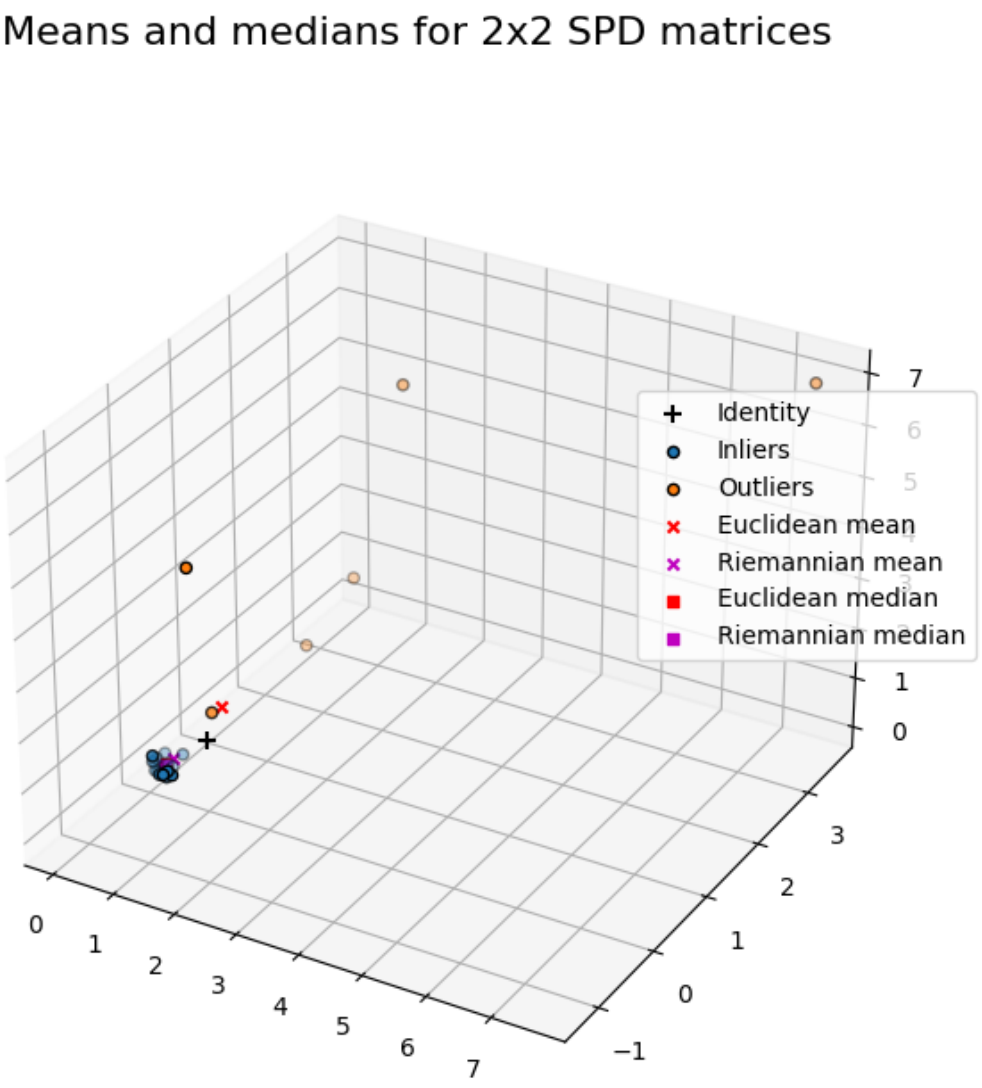


Photo finish

Specific zoom on means and medians.

Surprise guest: Riemannian potato is fitted with an offline iterative outlier removal, providing a robust mean[4].

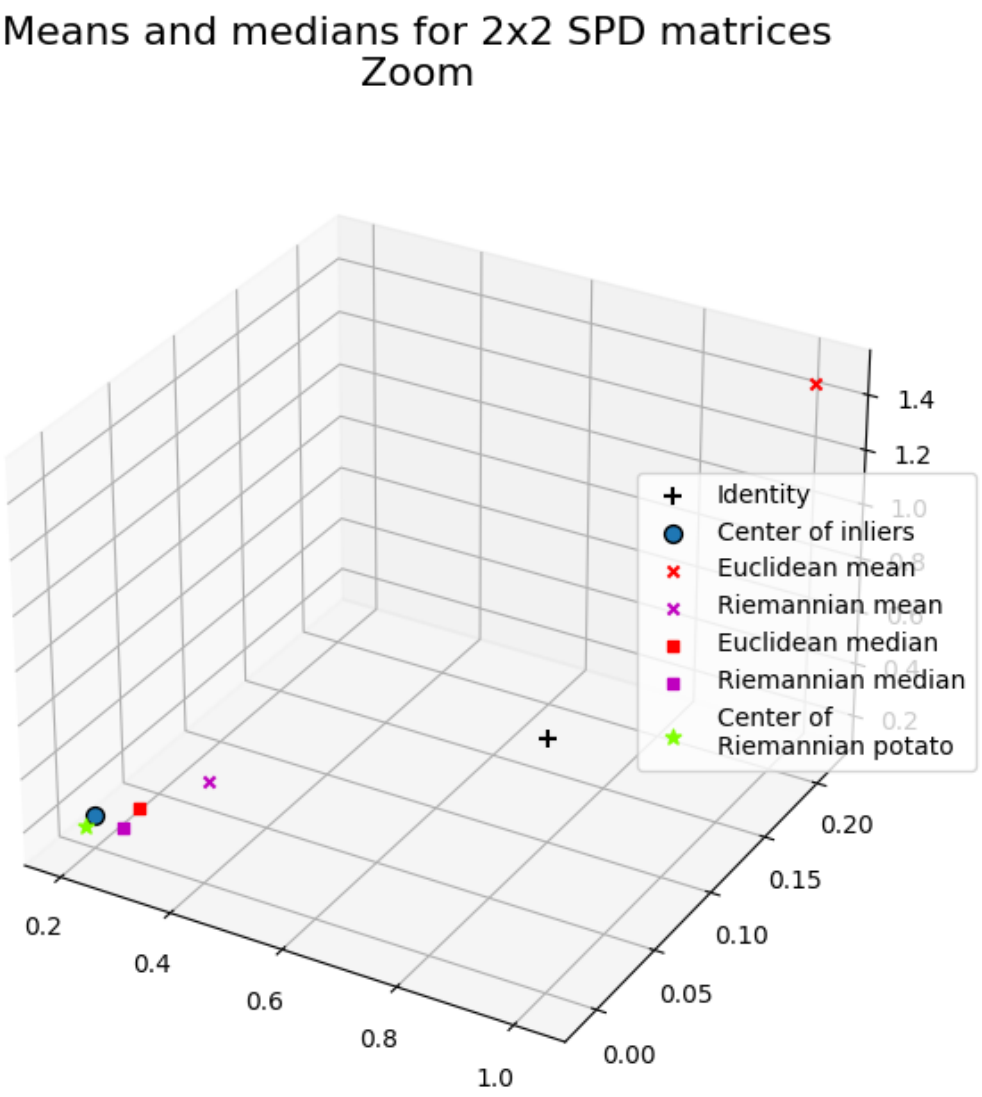
```
C_rp = Potato(metric='riemann', threshold=1.5).fit(X).covmean_

fig3 = plt.figure(figsize=(7, 7))
fig3.suptitle("Means and medians for 2x2 SPD matrices\nZoom", fontsize=16)
```

(continues on next page)

[4] The Riemannian Potato Field: A Tool for Online Signal Quality Index of EEG Q. Barthélemy, L. Mayaud, D. Ojeda, and M. Congedo. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2019, 27 (2), pp.244-255

(continued from previous page)

```
ax3 = plt.subplot(111, projection='3d')
ax3.scatter(1, 0, 1, c="k", marker="+", s=50, label='Identity')
ax3.scatter(Cin[0, 0], Cin[0, 1], Cin[1, 1], c="C0", edgecolors="k", s=50,
            label='Center of inliers')
ax3.scatter(C_emean[0, 0], C_emean[0, 1], C_emean[1, 1], c="r", marker="x",
            label='Euclidean mean')
ax3.scatter(C_rmean[0, 0], C_rmean[0, 1], C_rmean[1, 1], c="m", marker="x",
            label='Riemannian mean')
ax3.scatter(C_emed[0, 0], C_emed[0, 1], C_emed[1, 1], c="r", marker="s",
            label='Euclidean median')
ax3.scatter(C_rmed[0, 0], C_rmed[0, 1], C_rmed[1, 1], c="m", marker="s",
            label='Riemannian median')
ax3.scatter(C_rp[0, 0], C_rp[0, 1], C_rp[1, 1], c="chartreuse", marker="*",
            s=40, label='Center of\nRiemannian potato')
ax3.legend(loc='center left', bbox_to_anchor=(0.7, 0.5))
plt.show()
```



References

Total running time of the script: (0 minutes 0.682 seconds)

Estimate mean of SPD matrices with NaN values

Estimate the mean of SPD matrices corrupted by NaN values[1].

```
# Author: Quentin Barthélemy, Sylvain Chevallier and Florian Yger
#
# License: BSD (3-clause)

import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
import seaborn as sns

from pyriemann.datasets import make_covariances
from pyriemann.utils.mean import mean_riemann, nanmean_riemann
from pyriemann.utils.distance import distance_riemann
```

```
def corrupt(covmats, n_corrup_channels_max, rs):
    n_matrices, n_channels, _ = covmats.shape
    all_n_corrup_channels, all_corrup_channels = np.zeros(n_matrices), []
    for i_matrix in range(n_matrices):
        n_corrupt_channels = rs.randint(n_corrup_channels_max + 1, size=1)
        corrup_channels = rs.choice(
            np.arange(0, n_channels), size=n_corrupt_channels, replace=False)
        for i_channel in corrup_channels:
            covmats[i_matrix, i_channel] = np.nan
            covmats[i_matrix, :, i_channel] = np.nan
            all_corrup_channels.append(i_channel)
        all_n_corrup_channels[i_matrix] = n_corrupt_channels
    return covmats, all_n_corrup_channels, all_corrup_channels
```

Generate data

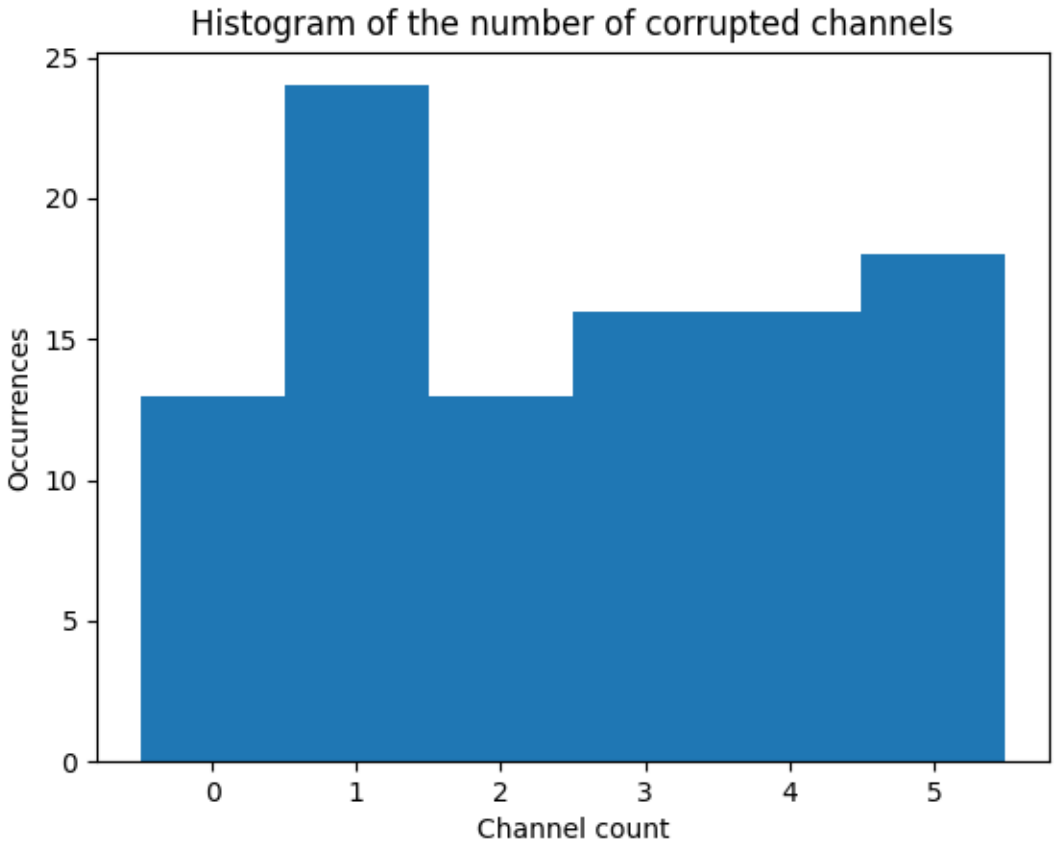
```
rs = np.random.RandomState(42)
n_matrices, n_channels = 100, 10
covmats = make_covariances(
    n_matrices, n_channels, rs, evals_mean=100., evals_std=20.)

# Compute the reference, the Riemannian mean on all SPD matrices
C_ref = mean_riemann(covmats)

# Corrupt data randomly
n_corrup_channels_max = n_channels // 2
print("Maximum number of corrupted channels: {} over {}".format(
    n_corrup_channels_max, n_channels))

covmats, all_n_corrup_channels, all_corrup_channels = corrupt(
    covmats, n_corrup_channels_max, rs)
```

(continues on next page)

[1] Geodesically-convex optimization for averaging partially observed covariance matrices F. Yger, S. Chevallier, Q. Barthélemy, and S. Sra. Asian Conference on Machine Learning (ACML), Nov 2020, Bangkok, Thailand. pp.417 - 432.

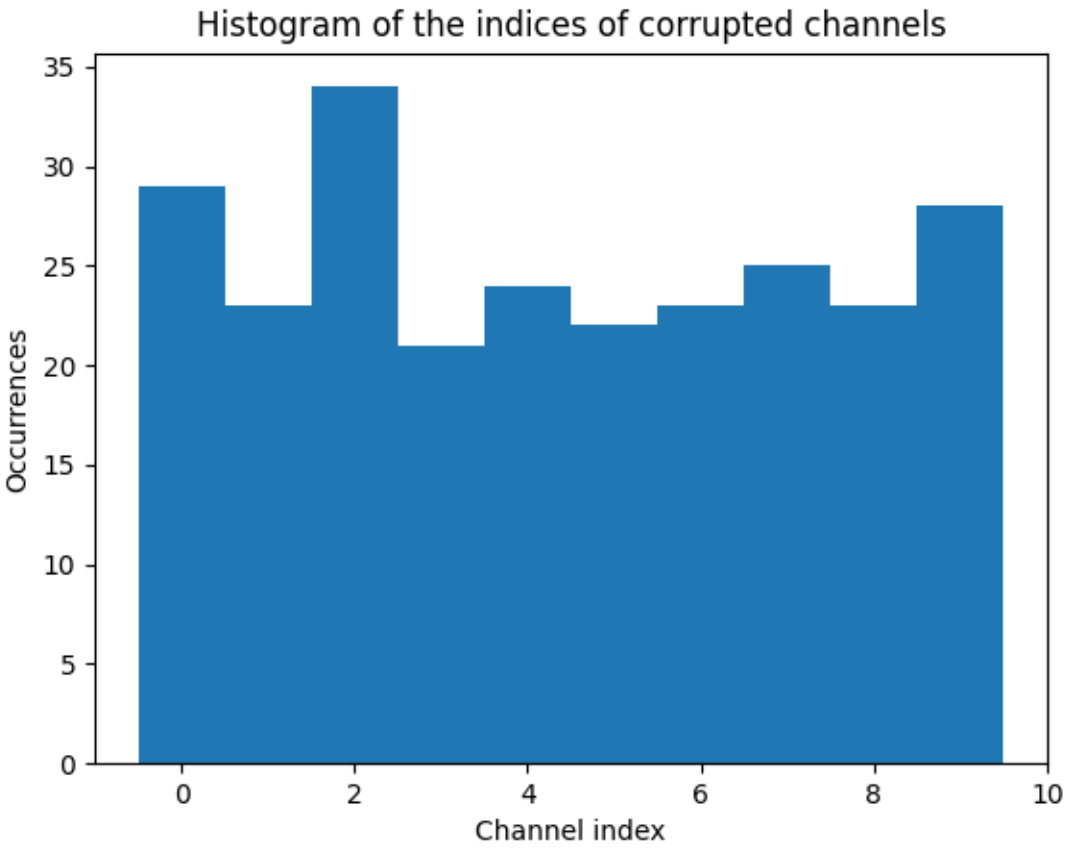
(continued from previous page)

```
fig, ax = plt.subplots(nrows=1, ncols=1)
ax.set(title='Histogram of the number of corrupted channels',
       xlabel='Channel count', ylabel='Occurrences')
plt.hist(all_n_corrup_channels, bins=np.arange(n_corrup_channels_max + 2) - .5)
plt.show()
```




```
Maximum number of corrupted channels: 5 over 10
```

```
fig, ax = plt.subplots(nrows=1, ncols=1)
ax.set(title='Histogram of the indices of corrupted channels',
       xlabel='Channel index', ylabel='Occurrences')
plt.hist(all_corrup_channels, bins=np.arange(n_channels + 1) - .5)
plt.show()
```



Estimate means of SPD matrices

Riemannian mean could only be computed on full-rank matrices. A common strategy is called matrix deletion, that is removing all matrices with corrupted channels before computing mean. This results in discarding useful information as uncorrupted channels are removed from the computation of the mean. Nan-mean uses as much information as possible to estimate the mean[Page 122, 1].

```
# Euclidean NaN-mean
C_naneucl = np.nanmean(covmats, axis=0)

# Riemannian NaN-mean
C_nanriem = nanmean_riemann(covmats)

# Riemannian mean, after matrix deletion: average only uncorrupted matrices
isnan = np.isnan(np.sum(covmats, axis=(1, 2)))
covmats_ = np.delete(covmats, np.where(isnan), axis=0)
perc = len(covmats_) / n_matrices * 100
print("Percentage of uncorrupted matrices: {:.2f} %".format(perc))
C_mdriem = mean_riemann(covmats_)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:700: UserWarning: Convergence not reached
```

(continues on next page)

(continued from previous page)

```
  warnings.warn('Convergence not reached')
Percentage of uncorrupted matrices: 13.00 %
```

Compare means

Compare distances between the different means and the reference.

```
d_naneucl = distance_riemann(C_ref, C_naneucl)
print(f"Euclidean NaN-mean, distance to ref = {d_naneucl:.3f}")

d_nanriem = distance_riemann(C_ref, C_nanriem)
print(f"Riemannian NaN-mean, distance to ref = {d_nanriem:.3f}")

d_mdriem = distance_riemann(C_ref, C_mdriem)
print(f"Riemannian mean after deletion, distance to ref = {d_mdriem:.3f}")

# Riemannian NaN-mean gives the best result, and Riemannian mean after matrix
# deletion is worst than Euclidean NaN-mean.
```

```
Euclidean NaN-mean, distance to ref = 0.081
Riemannian NaN-mean, distance to ref = 0.051
Riemannian mean after deletion, distance to ref = 0.127
```

Evaluate influence of corrupted channels

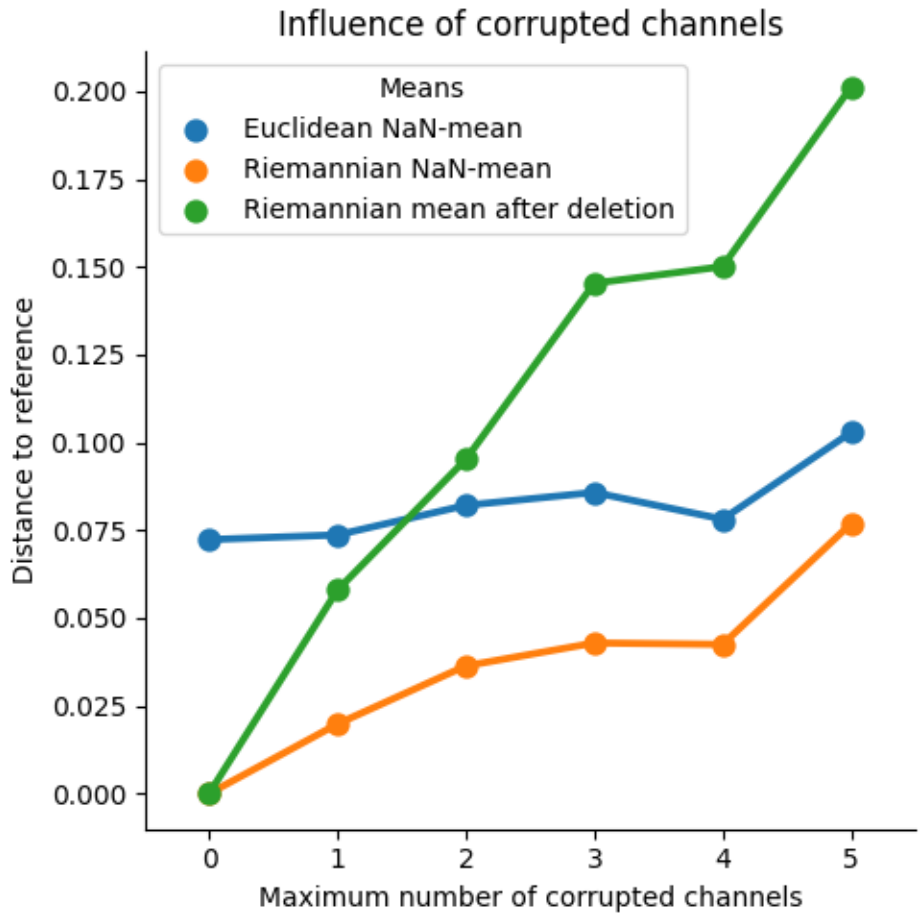
Repeat the previous experiment, varying the maximum number of corrupted channels[Page 122, 1].

```
covmats_orig = make_covariances(
    n_matrices, n_channels, rs, evals_mean=100., evals_std=20.)
C_ref = mean_riemann(covmats_orig)

df = []
for n_corrup_channels_max in range(0, n_channels // 2 + 1):
    covmats = np.copy(covmats_orig)
    covmats, _, _ = corrupt(covmats, n_corrup_channels_max, rs)

    C_naneucl = np.nanmean(covmats, axis=0)
    C_nanriem = nanmean_riemann(covmats)
    isnan = np.isnan(np.sum(covmats, axis=(1, 2)))
    covmats_ = np.delete(covmats, np.where(isnan), axis=0)
    C_mdriem = mean_riemann(covmats_)

    res_naneucl = {'n_corrupt': n_corrup_channels_max,
                   'dist': distance_riemann(C_ref, C_naneucl),
                   'Means': 'Euclidean NaN-mean'}
    res_nanriem = {'n_corrupt': n_corrup_channels_max,
                   'dist': distance_riemann(C_ref, C_nanriem),
                   'Means': 'Riemannian NaN-mean'}
    res_mdriem = {'n_corrupt': n_corrup_channels_max,
```

(continues on next page)

(continued from previous page)

```
                    'dist': distance_riemann(C_ref, C_mdriem),
                    'Means': 'Riemannian mean after deletion'}
    df += [res_naneucl, res_nanriem, res_mdriem]
df = pd.DataFrame(df)

g = sns.catplot(data=df, x="n_corrupt", y="dist", hue="Means", kind="point",
                legend_out=False)
g.set(title="Influence of corrupted channels")
g.set_axis_labels("Maximum number of corrupted channels",
                  "Distance to reference")
plt.tight_layout()
plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪utils/mean.py:700: UserWarning: Convergence not reached
  warnings.warn('Convergence not reached')
```

References

Total running time of the script: (0 minutes 17.557 seconds)

Classifier comparison

A comparison of several classifiers on low-dimensional synthetic datasets, adapted to SPD matrices from[1]. The point of this example is to illustrate the nature of decision boundaries of different classifiers, used with different metrics[2]. This should be taken with a grain of salt, as the intuition conveyed by these examples does not necessarily carry over to real datasets.

The 3D plots show training matrices in solid colors and testing matrices semi-transparent. The lower right shows the classification accuracy on the test set.

```
# Authors: Quentin Barthélemy
#
# License: BSD (3-clause)

from functools import partial
from time import time

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split

from pyriemann.datasets import make_covariances, make_gaussian_blobs
from pyriemann.classification import (
    MDM,
    KNearestNeighbor,
    SVC,
    MeanField,
)
```

```
@partial(np.vectorize, excluded=['clf'])
def get_proba(cov_00, cov_01, cov_11, clf):
    cov = np.array([[cov_00, cov_01], [cov_01, cov_11]])
    with np.testing.suppress_warnings() as sup:
        sup.filter(RuntimeWarning)
        return clf.predict_proba(cov[np.newaxis, ...])[0, 1]


def plot_classifiers(metric):
    figure = plt.figure(figsize=(12, 10))
    figure.suptitle(f"Compare classifiers with metric='{metric}'", fontsize=16)
    i = 1

    # iterate over datasets
    for ds_cnt, (X, y) in enumerate(datasets):
```

(continues on next page)

[1] https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html # noqa

[2] Review of Riemannian distances and divergences, applied to SSVEP-based BCI S. Chevallier, E. K. Kalunga, Q. Barthélemy, E. Monacelli. Neuroinformatics, Springer, 2021, 19 (1), pp.93-106

(continued from previous page)

```
        # split dataset into training and test part
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.4, random_state=42
        )

        x_min, x_max = X[:, 0, 0].min(), X[:, 0, 0].max()
        y_min, y_max = X[:, 0, 1].min(), X[:, 0, 1].max()
        z_min, z_max = X[:, 1, 1].min(), X[:, 1, 1].max()

        # just plot the dataset first
        ax = plt.subplot(n_datasets, n_classifs + 1, i, projection='3d')
        if ds_cnt == 0:
            ax.set_title("Input data")
        # Plot the training points
        ax.scatter(
            X_train[:, 0, 0],
            X_train[:, 0, 1],
            X_train[:, 1, 1],
            c=y_train,
            cmap=cm_bright,
            edgecolors="k"
        )
        # Plot the testing points
        ax.scatter(
            X_test[:, 0, 0],
            X_test[:, 0, 1],
            X_test[:, 1, 1],
            c=y_test,
            cmap=cm_bright,
            alpha=0.6,
            edgecolors="k"
        )
        ax.set_xlim(x_min, x_max)
        ax.set_ylim(y_min, y_max)
        ax.set_zlim(z_min, z_max)
        ax.set_xticklabels(())
        ax.set_yticklabels(())
        ax.set_zticklabels(())
        i += 1

        rx = np.arange(x_min, x_max, (x_max - x_min) / 50)
        ry = np.arange(y_min, y_max, (y_max - y_min) / 50)
        rz = np.arange(z_min, z_max, (z_max - z_min) / 50)

        print(f"Dataset n°{ds_cnt+1}")

        # iterate over classifiers
        for name, clf in zip(names, classifiers):
            ax = plt.subplot(n_datasets, n_classifs + 1, i, projection='3d')

            clf.set_params(**{'metric': metric})
            t0 = time()
```

(continues on next page)

(continued from previous page)

```
            clf.fit(X_train, y_train)
            t1 = time() - t0

            t0 = time()
            score = clf.score(X_test, y_test)
            t2 = time() - t0

            print(
                f" {name}:\n  training time={t1:.5f}\n  test time    ={t2:.5f}"
            )

            # Plot the decision boundaries for horizontal 2D planes going
            # through the mean value of the third coordinates
            xx, yy = np.meshgrid(rx, ry)
            zz = get_proba(xx, yy, X[:, 1, 1].mean()*np.ones_like(xx), clf=clf)
            zz = np.ma.masked_where(~np.isfinite(zz), zz)
            ax.contourf(xx, yy, zz, zdir='z', offset=z_min, cmap=cm, alpha=0.5)

            xx, zz = np.meshgrid(rx, rz)
            yy = get_proba(xx, X[:, 0, 1].mean()*np.ones_like(xx), zz, clf=clf)
            yy = np.ma.masked_where(~np.isfinite(yy), yy)
            ax.contourf(xx, yy, zz, zdir='y', offset=y_max, cmap=cm, alpha=0.5)

            yy, zz = np.meshgrid(ry, rz)
            xx = get_proba(X[:, 0, 0].mean()*np.ones_like(yy), yy, zz, clf=clf)
            xx = np.ma.masked_where(~np.isfinite(xx), xx)
            ax.contourf(xx, yy, zz, zdir='x', offset=x_min, cmap=cm, alpha=0.5)

            # Plot the training points
            ax.scatter(
                X_train[:, 0, 0],
                X_train[:, 0, 1],
                X_train[:, 1, 1],
                c=y_train,
                cmap=cm_bright,
                edgecolors="k"
            )
            # Plot the testing points
            ax.scatter(
                X_test[:, 0, 0],
                X_test[:, 0, 1],
                X_test[:, 1, 1],
                c=y_test,
                cmap=cm_bright,
                edgecolors="k",
                alpha=0.6
            )

            if ds_cnt == 0:
                ax.set_title(name)
            ax.text(
                1.3 * x_max,
```

(continues on next page)

(continued from previous page)

```
                y_min,
                z_min,
                ("%.2f" % score).lstrip("0"),
                size=15,
                horizontalalignment="right",
                verticalalignment="bottom"
            )
            ax.set_xlim(x_min, x_max)
            ax.set_ylim(y_min, y_max)
            ax.set_zlim(z_min, z_max)
            ax.set_xticks(())
            ax.set_yticks(())
            ax.set_zticks(())

            i += 1

    plt.show()
```

Classifiers and Datasets

```
names = [
    "MDM",
    "k-NN",
    "SVC",
    "MeanField",
]
classifiers = [
    MDM(),
    KNearestNeighbor(n_neighbors=3),
    SVC(probability=True),
    MeanField(power_list=[-1, 0, 1]),
]
n_classifs = len(classifiers)

rs = np.random.RandomState(2022)
n_matrices, n_channels = 50, 2
y = np.concatenate([np.zeros(n_matrices), np.ones(n_matrices)])

datasets = [
    (
        np.concatenate([
            make_covariances(
                n_matrices, n_channels, rs, evals_mean=10, evals_std=1
            ),
            make_covariances(
                n_matrices, n_channels, rs, evals_mean=15, evals_std=1
            )
        ]),
        y
    ),
```

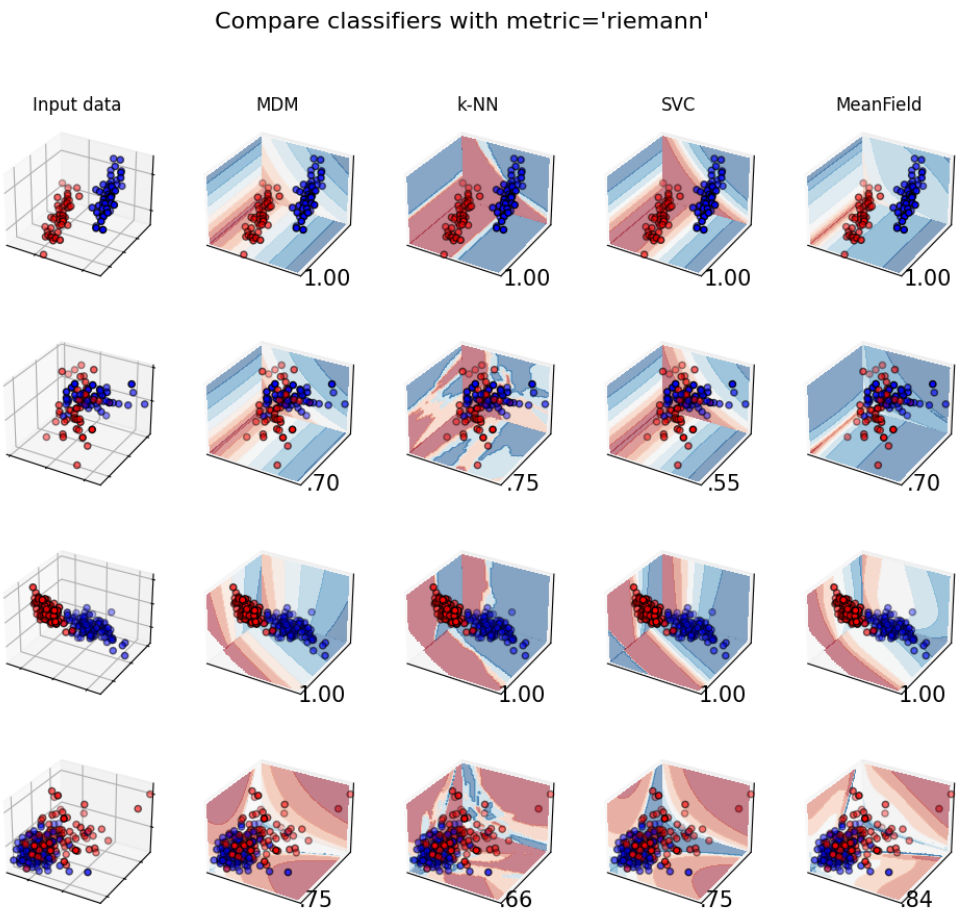
(continues on next page)

(continued from previous page)

```
    (
        np.concatenate([
            make_covariances(
                n_matrices, n_channels, rs, evals_mean=10, evals_std=2
            ),
            make_covariances(
                n_matrices, n_channels, rs, evals_mean=12, evals_std=2
            )
        ]),
        y
    ),
    make_gaussian_blobs(
        2*n_matrices, n_channels, random_state=rs, class_sep=1., class_disp=.2,
        n_jobs=4
    ),
    make_gaussian_blobs(
        2*n_matrices, n_channels, random_state=rs, class_sep=.5, class_disp=.5,
        n_jobs=4
    )
]
n_datasets = len(datasets)

cm = plt.cm.RdBu
cm_bright = ListedColormap(["#FF0000", "#0000FF"])
```

Classifiers with Riemannian metric

```
plot_classifiers("riemann")
```




```
Dataset n°1
 MDM:
  training time=0.00224
  test time    =0.00698
 k-NN:
  training time=0.00007
  test time    =0.15169
 SVC:
  training time=0.00378
  test time    =0.00158
 MeanField:
  training time=0.01857
  test time    =0.02697
Dataset n°2
 MDM:
  training time=0.00219
  test time    =0.00697
 k-NN:
  training time=0.00006
  test time    =0.15051
```

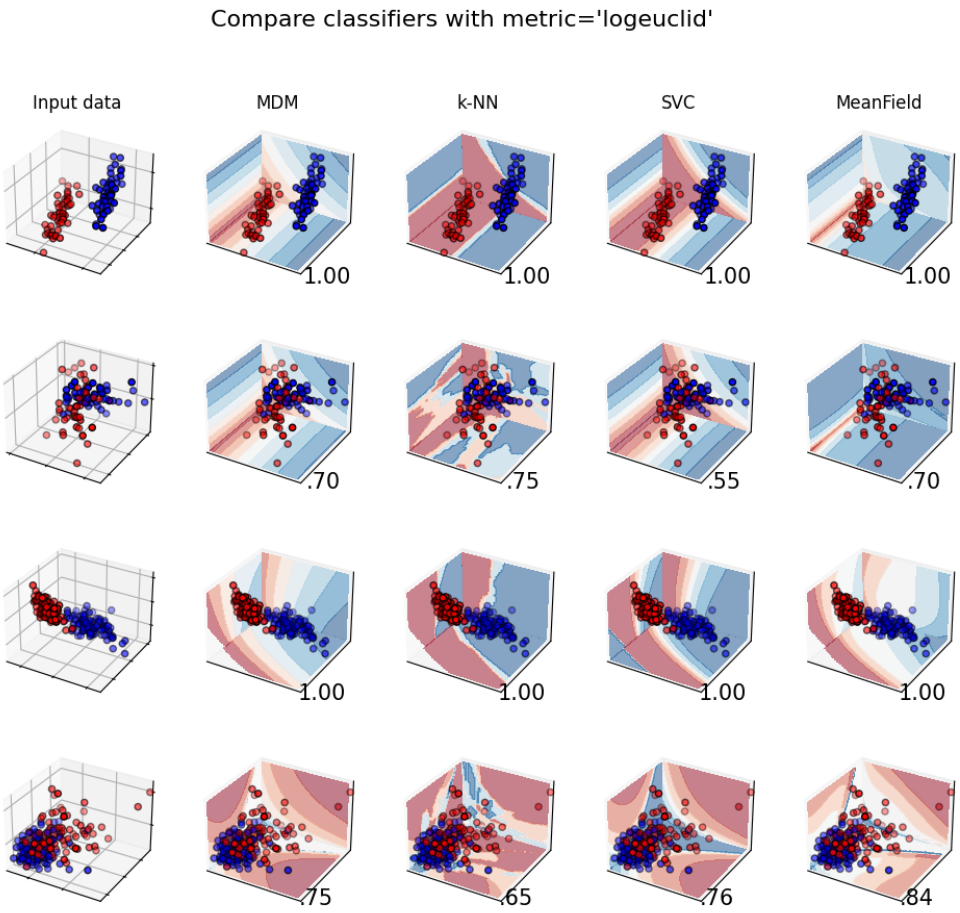
(continues on next page)

(continued from previous page)

```
 SVC:
  training time=0.00430
  test time    =0.00155
 MeanField:
  training time=0.01853
  test time    =0.02723
Dataset n°3
 MDM:
  training time=0.00534
  test time    =0.01304
 k-NN:
  training time=0.00005
  test time    =0.44538
 SVC:
  training time=0.00709
  test time    =0.00180
 MeanField:
  training time=0.02700
  test time    =0.05292
Dataset n°4
 MDM:
  training time=0.00662
  test time    =0.01321
 k-NN:
  training time=0.00005
  test time    =0.44173
 SVC:
  training time=0.00820
  test time    =0.00195
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/numpy/lib/function_base.py:2246: RuntimeWarning: invalid value encountered in␣
↪func (vectorized)
  outputs = ufunc(*inputs)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/numpy/lib/function_base.py:2246: RuntimeWarning: invalid value encountered in␣
↪func (vectorized)
  outputs = ufunc(*inputs)
 MeanField:
  training time=0.02767
  test time    =0.05270
```

Classifiers with Log-Euclidean metric

```
plot_classifiers("logeuclid")
```




```
Dataset n°1
 MDM:
  training time=0.00088
  test time    =0.01053
 k-NN:
  training time=0.00007
  test time    =0.20826
 SVC:
  training time=0.00215
  test time    =0.00135
 MeanField:
  training time=0.01852
  test time    =0.03861
Dataset n°2
 MDM:
```

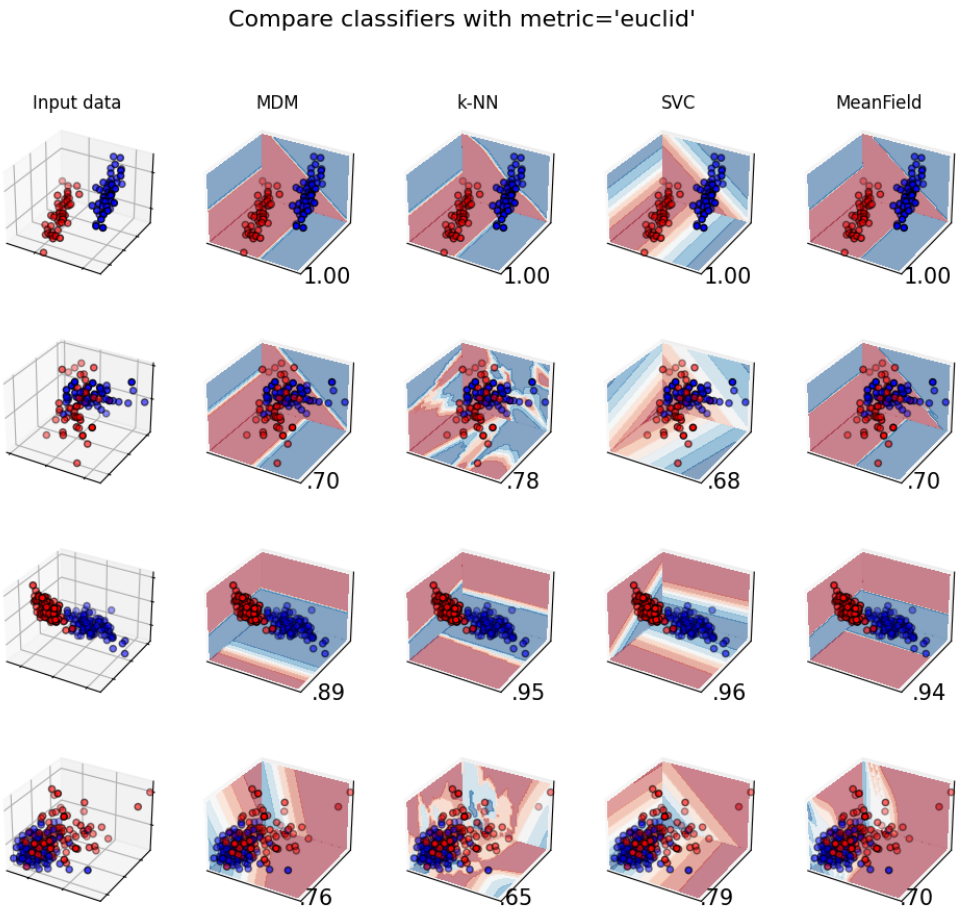
(continues on next page)

(continued from previous page)

```
  training time=0.00093
  test time    =0.01069
 k-NN:
  training time=0.00006
  test time    =0.20739
 SVC:
  training time=0.00237
  test time    =0.00138
 MeanField:
  training time=0.01840
  test time    =0.03863
Dataset n°3
 MDM:
  training time=0.00116
  test time    =0.02075
 k-NN:
  training time=0.00006
  test time    =0.67098
 SVC:
  training time=0.00279
  test time    =0.00149
 MeanField:
  training time=0.02781
  test time    =0.07671
Dataset n°4
 MDM:
  training time=0.00118
  test time    =0.02058
 k-NN:
  training time=0.00005
  test time    =0.67740
 SVC:
  training time=0.00400
  test time    =0.00160
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/numpy/lib/function_base.py:2246: RuntimeWarning: invalid value encountered in␣
↪func (vectorized)
  outputs = ufunc(*inputs)
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/envs/v0.4/lib/python3.7/site-
↪packages/numpy/lib/function_base.py:2246: RuntimeWarning: invalid value encountered in␣
↪func (vectorized)
  outputs = ufunc(*inputs)
 MeanField:
  training time=0.02854
  test time    =0.07650
```

Classifiers with Euclidean metric

```
plot_classifiers("euclid")
```




```
Dataset n°1
 MDM:
  training time=0.00041
  test time    =0.00202
 k-NN:
  training time=0.00007
  test time    =0.04523
 SVC:
  training time=0.00148
  test time    =0.00079
 MeanField:
  training time=0.01832
  test time    =0.01050
Dataset n°2
 MDM:
```

(continues on next page)

(continued from previous page)

```
  training time=0.00037
  test time    =0.00201
 k-NN:
  training time=0.00006
  test time    =0.04581
 SVC:
  training time=0.00258
  test time    =0.00082
 MeanField:
  training time=0.02015
  test time    =0.01046
Dataset n°3
 MDM:
  training time=0.00037
  test time    =0.00331
 k-NN:
  training time=0.00005
  test time    =0.13844
 SVC:
  training time=0.00179
  test time    =0.00085
 MeanField:
  training time=0.02678
  test time    =0.01953
Dataset n°4
 MDM:
  training time=0.00036
  test time    =0.00392
 k-NN:
  training time=0.00005
  test time    =0.13824
 SVC:
  training time=0.00298
  test time    =0.00093
 MeanField:
  training time=0.02838
  test time    =0.01951
```

References

Total running time of the script: (7 minutes 36.739 seconds)

4.8.7 Permutation test

Permutation test with pyRiemann.

One Way manova time

One way manova to compare Left vs Right in time.

```
import numpy as np
import seaborn as sns

from time import time
from pylab import plt

from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci

from pyriemann.stats import PermutationDistance
from pyriemann.estimation import Covariances

sns.set_style('whitegrid')
```

Set parameters and read data

```
# avoid classification of evoked responses by using epochs that start 1s after
# cue onset.
tmin, tmax = -2., 6.
event_id = dict(hands=2, feet=3)
subject = 1
runs = [6, 10, 14]  # motor imagery: hands vs feet

raw_files = [
    read_raw_edf(f, preload=True, verbose=False)
    for f in eegbci.load_data(subject, runs)
]
raw = concatenate_raws(raw_files)

events, _ = events_from_annotations(raw, event_id=dict(T1=2, T2=3))
picks = pick_types(
    raw.info, meg=False, eeg=True, stim=False, eog=False, exclude='bads')

raw.filter(7., 35., method='iir', picks=picks)

epochs = Epochs(
    raw,
    events,
    event_id,
    tmin,
    tmax,
```

(continues on next page)

(continued from previous page)

```
    proj=True,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False)
labels = epochs.events[:, -1] - 2

# get epochs
epochs_data = epochs.get_data()
```

```
Used Annotations descriptions: ['T1', 'T2']
Filtering raw data in 3 contiguous segments
Setting up band-pass filter from 7 - 35 Hz

IIR filter parameters
---------------------
Butterworth bandpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 16 (effective, after forward-backward)
- Cutoffs at 7.00, 35.00 Hz: -6.02, -6.02 dB
```

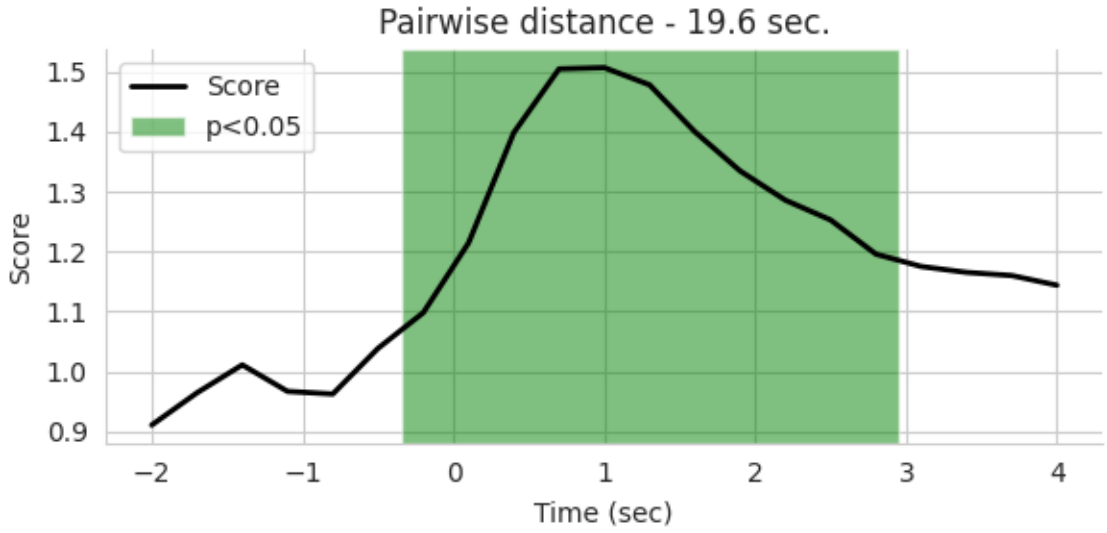
Pairwise distance based permutation test

```
covest = Covariances()

Fs = 160
window = 2 * Fs
Nwindow = 20
Ns = epochs_data.shape[2]
step = int((Ns - window) / Nwindow)
time_bins = range(0, Ns - window, step)

pv = []
Fv = []
# For each frequency bin, estimate the stats
t_init = time()
for t in time_bins:
    covmats = covest.fit_transform(epochs_data[:, ::1, t:(t + window)])
    p_test = PermutationDistance(1000, metric='riemann', mode='pairwise')
    p, F = p_test.test(covmats, labels, verbose=False)
    pv.append(p)
    Fv.append(F[0])
duration = time() - t_init
# plot result
fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
sig = 0.05
times = np.array(time_bins) / float(Fs) + tmin

axes.plot(times, Fv, lw=2, c='k')
plt.xlabel('Time (sec)')
plt.ylabel('Score')
```

(continues on next page)

(continued from previous page)

```
a = np.where(np.diff(np.array(pv) < sig))[0]
a = a.reshape(int(len(a) / 2), 2)
st = (times[1] - times[0]) / 2.0
for p in a:
    axes.axvspan(times[p[0]] - st, times[p[1]] + st, facecolor='g', alpha=0.5)
axes.legend(['Score', 'p<%.2f' % sig])
axes.set_title('Pairwise distance - %.1f sec.' % duration)

sns.despine()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 20.226 seconds)

One Way manova with Frequenty

One way manova to compare Left vs Right for each frequency.

```
import numpy as np
import seaborn as sns

from time import time
from pylab import plt
from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci

from pyriemann.stats import PermutationDistance
from pyriemann.estimation import CospCovariances

sns.set_style('whitegrid')
```

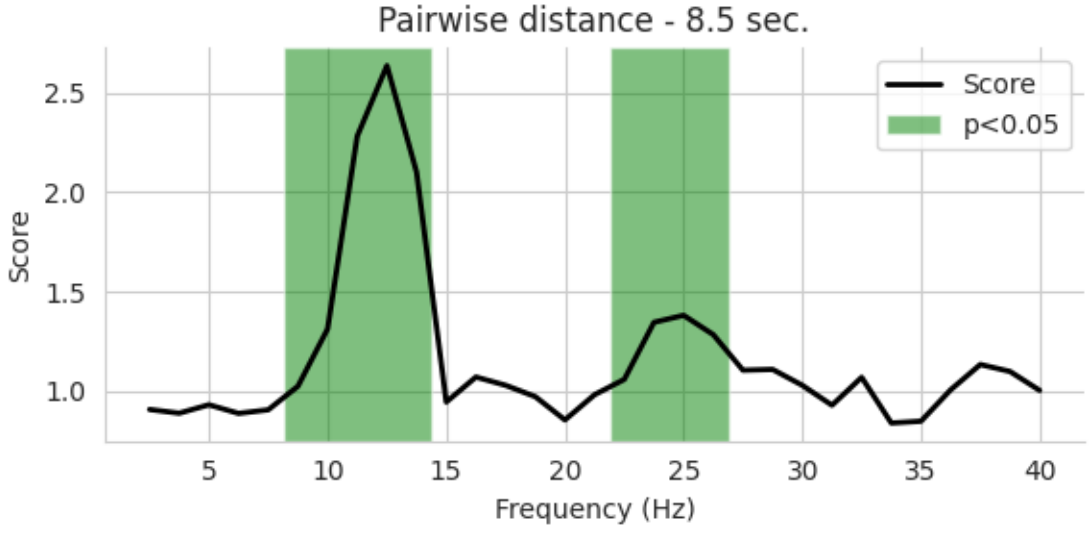
Set parameters and read data

```
# avoid classification of evoked responses by using epochs that start 1s after
# cue onset.
tmin, tmax = 1., 3.
event_id = dict(hands=2, feet=3)
subject = 1
runs = [6, 10, 14]  # motor imagery: hands vs feet

raw_files = [
    read_raw_edf(f, preload=True, verbose=False)
    for f in eegbci.load_data(subject, runs)
]
raw = concatenate_raws(raw_files)

events, _ = events_from_annotations(raw, event_id=dict(T1=2, T2=3))
picks = pick_types(
    raw.info, meg=False, eeg=True, stim=False, eog=False, exclude='bads')

# Read epochs (train will be done only between 1 and 2s)
# Testing will be done with a running classifier
epochs = Epochs(
    raw,
    events,
    event_id,
    tmin,
    tmax,
    proj=True,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False)
labels = epochs.events[:, -1] - 2

# get epochs
epochs_data = epochs.get_data()

# compute cospectral covariance matrices
fmin = 2.0
fmax = 40.0
cosp = CospCovariances(
    window=128, overlap=0.98, fmin=fmin, fmax=fmax, fs=160.0)
covmats = cosp.fit_transform(epochs_data[:, ::4, :])

fr = np.fft.fftfreq(128)[0:64] * 160
fr = fr[(fr >= fmin) & (fr <= fmax)]
```

```
Used Annotations descriptions: ['T1', 'T2']
```

Pairwise distance based permutation test

```
pv = []
Fv = []
# For each frequency bin, estimate the stats
t_init = time()
for i in range(covmats.shape[3]):
    p_test = PermutationDistance(1000, metric='riemann', mode='pairwise')
    p, F = p_test.test(covmats[:, :, :, i], labels, verbose=False)
    pv.append(p)
    Fv.append(F[0])
duration = time() - t_init

# plot result
fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
sig = 0.05
axes.plot(fr, Fv, lw=2, c='k')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Score')

a = np.where(np.diff(np.array(pv) < sig))[0]
a = a.reshape(int(len(a) / 2), 2)
st = (fr[1] - fr[0]) / 2.0
for p in a:
    axes.axvspan(fr[p[0]] - st, fr[p[1]] + st, facecolor='g', alpha=0.5)
axes.legend(['Score', 'p<%.2f' % sig])
axes.set_title('Pairwise distance - %.1f sec.' % duration)

sns.despine()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 8.974 seconds)

One Way manova

One way manova to compare Left vs Right.

```
import seaborn as sns

from time import time
from matplotlib import pyplot as plt

from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci

from pyriemann.stats import PermutationDistance, PermutationModel
from pyriemann.estimation import Covariances
from pyriemann.spatialfilters import CSP

from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression

sns.set_style('whitegrid')
```

Set parameters and read data

```
# avoid classification of evoked responses by using epochs that start 1s after
# cue onset.
tmin, tmax = 1., 3.
event_id = dict(hands=2, feet=3)
subject = 1
runs = [6, 10, 14]  # motor imagery: hands vs feet

raw_files = [
    read_raw_edf(f, preload=True, verbose=False)
    for f in eegbci.load_data(subject, runs)
]
raw = concatenate_raws(raw_files)

# Apply band-pass filter
raw.filter(7., 35., method='iir')

events, _ = events_from_annotations(raw, event_id=dict(T1=2, T2=3))

picks = pick_types(
    raw.info, meg=False, eeg=True, stim=False, eog=False, exclude='bads')
picks = picks[::4]

# Read epochs (train will be done only between 1 and 2s)
# Testing will be done with a running classifier
epochs = Epochs(
    raw,
    events,
    event_id,
    tmin,
```

(continues on next page)

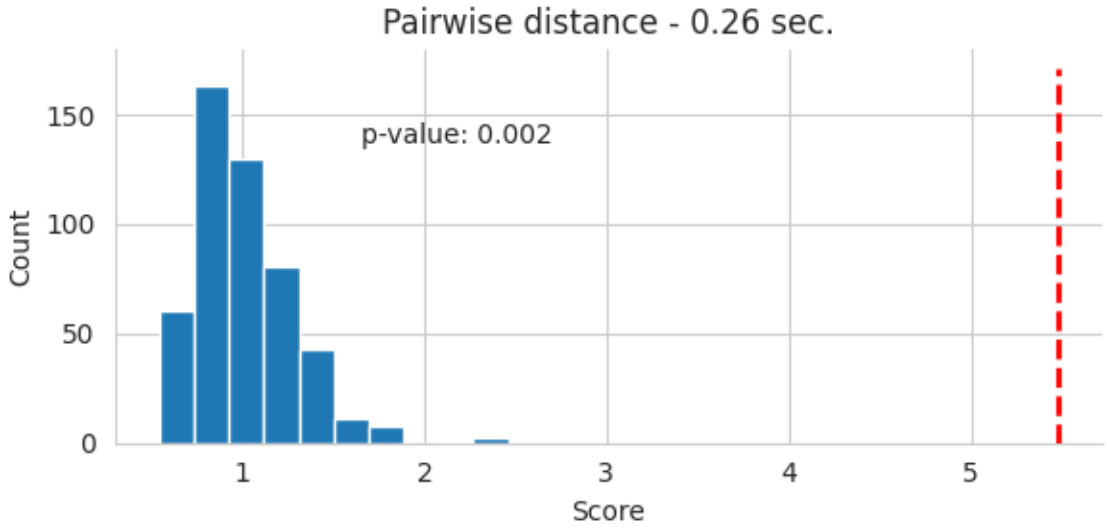
(continued from previous page)

```
    tmax,
    proj=True,
    picks=picks,
    baseline=None,
    preload=True,
    verbose=False)
labels = epochs.events[:, -1] - 2

# get epochs
epochs_data = epochs.get_data()

# compute covariance matrices
covmats = Covariances().fit_transform(epochs_data)

n_perms = 500
```

```
Filtering raw data in 3 contiguous segments
Setting up band-pass filter from 7 - 35 Hz

IIR filter parameters
---------------------
Butterworth bandpass zero-phase (two-pass forward and reverse) non-causal filter:
- Filter order 16 (effective, after forward-backward)
- Cutoffs at 7.00, 35.00 Hz: -6.02, -6.02 dB

Used Annotations descriptions: ['T1', 'T2']
```

Pairwise distance based permutation test

```
t_init = time()
p_test = PermutationDistance(n_perms, metric='riemann', mode='pairwise')
p, F = p_test.test(covmats, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('Pairwise distance - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
plt.tight_layout()
plt.show()
```




```
Performing permutations : [0.2%]
Performing permutations : [0.4%]
Performing permutations : [0.6%]
Performing permutations : [0.8%]
Performing permutations : [1.0%]
Performing permutations : [1.2%]
Performing permutations : [1.4%]
Performing permutations : [1.6%]
Performing permutations : [1.8%]
Performing permutations : [2.0%]
Performing permutations : [2.2%]
Performing permutations : [2.4%]
Performing permutations : [2.6%]
Performing permutations : [2.8%]
Performing permutations : [3.0%]
Performing permutations : [3.2%]
Performing permutations : [3.4%]
Performing permutations : [3.6%]
Performing permutations : [3.8%]
Performing permutations : [4.0%]
Performing permutations : [4.2%]
Performing permutations : [4.4%]
Performing permutations : [4.6%]
Performing permutations : [4.8%]
Performing permutations : [5.0%]
Performing permutations : [5.2%]
Performing permutations : [5.4%]
Performing permutations : [5.6%]
Performing permutations : [5.8%]
Performing permutations : [6.0%]
Performing permutations : [6.2%]
Performing permutations : [6.4%]
Performing permutations : [6.6%]
Performing permutations : [6.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [7.0%]
Performing permutations : [7.2%]
Performing permutations : [7.4%]
Performing permutations : [7.6%]
Performing permutations : [7.8%]
Performing permutations : [8.0%]
Performing permutations : [8.2%]
Performing permutations : [8.4%]
Performing permutations : [8.6%]
Performing permutations : [8.8%]
Performing permutations : [9.0%]
Performing permutations : [9.2%]
Performing permutations : [9.4%]
Performing permutations : [9.6%]
Performing permutations : [9.8%]
Performing permutations : [10.0%]
Performing permutations : [10.2%]
Performing permutations : [10.4%]
Performing permutations : [10.6%]
Performing permutations : [10.8%]
Performing permutations : [11.0%]
Performing permutations : [11.2%]
Performing permutations : [11.4%]
Performing permutations : [11.6%]
Performing permutations : [11.8%]
Performing permutations : [12.0%]
Performing permutations : [12.2%]
Performing permutations : [12.4%]
Performing permutations : [12.6%]
Performing permutations : [12.8%]
Performing permutations : [13.0%]
Performing permutations : [13.2%]
Performing permutations : [13.4%]
Performing permutations : [13.6%]
Performing permutations : [13.8%]
Performing permutations : [14.0%]
Performing permutations : [14.2%]
Performing permutations : [14.4%]
Performing permutations : [14.6%]
Performing permutations : [14.8%]
Performing permutations : [15.0%]
Performing permutations : [15.2%]
Performing permutations : [15.4%]
Performing permutations : [15.6%]
Performing permutations : [15.8%]
Performing permutations : [16.0%]
Performing permutations : [16.2%]
Performing permutations : [16.4%]
Performing permutations : [16.6%]
Performing permutations : [16.8%]
Performing permutations : [17.0%]
Performing permutations : [17.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [17.4%]
Performing permutations : [17.6%]
Performing permutations : [17.8%]
Performing permutations : [18.0%]
Performing permutations : [18.2%]
Performing permutations : [18.4%]
Performing permutations : [18.6%]
Performing permutations : [18.8%]
Performing permutations : [19.0%]
Performing permutations : [19.2%]
Performing permutations : [19.4%]
Performing permutations : [19.6%]
Performing permutations : [19.8%]
Performing permutations : [20.0%]
Performing permutations : [20.2%]
Performing permutations : [20.4%]
Performing permutations : [20.6%]
Performing permutations : [20.8%]
Performing permutations : [21.0%]
Performing permutations : [21.2%]
Performing permutations : [21.4%]
Performing permutations : [21.6%]
Performing permutations : [21.8%]
Performing permutations : [22.0%]
Performing permutations : [22.2%]
Performing permutations : [22.4%]
Performing permutations : [22.6%]
Performing permutations : [22.8%]
Performing permutations : [23.0%]
Performing permutations : [23.2%]
Performing permutations : [23.4%]
Performing permutations : [23.6%]
Performing permutations : [23.8%]
Performing permutations : [24.0%]
Performing permutations : [24.2%]
Performing permutations : [24.4%]
Performing permutations : [24.6%]
Performing permutations : [24.8%]
Performing permutations : [25.0%]
Performing permutations : [25.2%]
Performing permutations : [25.4%]
Performing permutations : [25.6%]
Performing permutations : [25.8%]
Performing permutations : [26.0%]
Performing permutations : [26.2%]
Performing permutations : [26.4%]
Performing permutations : [26.6%]
Performing permutations : [26.8%]
Performing permutations : [27.0%]
Performing permutations : [27.2%]
Performing permutations : [27.4%]
Performing permutations : [27.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [27.8%]
Performing permutations : [28.0%]
Performing permutations : [28.2%]
Performing permutations : [28.4%]
Performing permutations : [28.6%]
Performing permutations : [28.8%]
Performing permutations : [29.0%]
Performing permutations : [29.2%]
Performing permutations : [29.4%]
Performing permutations : [29.6%]
Performing permutations : [29.8%]
Performing permutations : [30.0%]
Performing permutations : [30.2%]
Performing permutations : [30.4%]
Performing permutations : [30.6%]
Performing permutations : [30.8%]
Performing permutations : [31.0%]
Performing permutations : [31.2%]
Performing permutations : [31.4%]
Performing permutations : [31.6%]
Performing permutations : [31.8%]
Performing permutations : [32.0%]
Performing permutations : [32.2%]
Performing permutations : [32.4%]
Performing permutations : [32.6%]
Performing permutations : [32.8%]
Performing permutations : [33.0%]
Performing permutations : [33.2%]
Performing permutations : [33.4%]
Performing permutations : [33.6%]
Performing permutations : [33.8%]
Performing permutations : [34.0%]
Performing permutations : [34.2%]
Performing permutations : [34.4%]
Performing permutations : [34.6%]
Performing permutations : [34.8%]
Performing permutations : [35.0%]
Performing permutations : [35.2%]
Performing permutations : [35.4%]
Performing permutations : [35.6%]
Performing permutations : [35.8%]
Performing permutations : [36.0%]
Performing permutations : [36.2%]
Performing permutations : [36.4%]
Performing permutations : [36.6%]
Performing permutations : [36.8%]
Performing permutations : [37.0%]
Performing permutations : [37.2%]
Performing permutations : [37.4%]
Performing permutations : [37.6%]
Performing permutations : [37.8%]
Performing permutations : [38.0%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [38.2%]
Performing permutations : [38.4%]
Performing permutations : [38.6%]
Performing permutations : [38.8%]
Performing permutations : [39.0%]
Performing permutations : [39.2%]
Performing permutations : [39.4%]
Performing permutations : [39.6%]
Performing permutations : [39.8%]
Performing permutations : [40.0%]
Performing permutations : [40.2%]
Performing permutations : [40.4%]
Performing permutations : [40.6%]
Performing permutations : [40.8%]
Performing permutations : [41.0%]
Performing permutations : [41.2%]
Performing permutations : [41.4%]
Performing permutations : [41.6%]
Performing permutations : [41.8%]
Performing permutations : [42.0%]
Performing permutations : [42.2%]
Performing permutations : [42.4%]
Performing permutations : [42.6%]
Performing permutations : [42.8%]
Performing permutations : [43.0%]
Performing permutations : [43.2%]
Performing permutations : [43.4%]
Performing permutations : [43.6%]
Performing permutations : [43.8%]
Performing permutations : [44.0%]
Performing permutations : [44.2%]
Performing permutations : [44.4%]
Performing permutations : [44.6%]
Performing permutations : [44.8%]
Performing permutations : [45.0%]
Performing permutations : [45.2%]
Performing permutations : [45.4%]
Performing permutations : [45.6%]
Performing permutations : [45.8%]
Performing permutations : [46.0%]
Performing permutations : [46.2%]
Performing permutations : [46.4%]
Performing permutations : [46.6%]
Performing permutations : [46.8%]
Performing permutations : [47.0%]
Performing permutations : [47.2%]
Performing permutations : [47.4%]
Performing permutations : [47.6%]
Performing permutations : [47.8%]
Performing permutations : [48.0%]
Performing permutations : [48.2%]
Performing permutations : [48.4%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [48.6%]
Performing permutations : [48.8%]
Performing permutations : [49.0%]
Performing permutations : [49.2%]
Performing permutations : [49.4%]
Performing permutations : [49.6%]
Performing permutations : [49.8%]
Performing permutations : [50.0%]
Performing permutations : [50.2%]
Performing permutations : [50.4%]
Performing permutations : [50.6%]
Performing permutations : [50.8%]
Performing permutations : [51.0%]
Performing permutations : [51.2%]
Performing permutations : [51.4%]
Performing permutations : [51.6%]
Performing permutations : [51.8%]
Performing permutations : [52.0%]
Performing permutations : [52.2%]
Performing permutations : [52.4%]
Performing permutations : [52.6%]
Performing permutations : [52.8%]
Performing permutations : [53.0%]
Performing permutations : [53.2%]
Performing permutations : [53.4%]
Performing permutations : [53.6%]
Performing permutations : [53.8%]
Performing permutations : [54.0%]
Performing permutations : [54.2%]
Performing permutations : [54.4%]
Performing permutations : [54.6%]
Performing permutations : [54.8%]
Performing permutations : [55.0%]
Performing permutations : [55.2%]
Performing permutations : [55.4%]
Performing permutations : [55.6%]
Performing permutations : [55.8%]
Performing permutations : [56.0%]
Performing permutations : [56.2%]
Performing permutations : [56.4%]
Performing permutations : [56.6%]
Performing permutations : [56.8%]
Performing permutations : [57.0%]
Performing permutations : [57.2%]
Performing permutations : [57.4%]
Performing permutations : [57.6%]
Performing permutations : [57.8%]
Performing permutations : [58.0%]
Performing permutations : [58.2%]
Performing permutations : [58.4%]
Performing permutations : [58.6%]
Performing permutations : [58.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [59.0%]
Performing permutations : [59.2%]
Performing permutations : [59.4%]
Performing permutations : [59.6%]
Performing permutations : [59.8%]
Performing permutations : [60.0%]
Performing permutations : [60.2%]
Performing permutations : [60.4%]
Performing permutations : [60.6%]
Performing permutations : [60.8%]
Performing permutations : [61.0%]
Performing permutations : [61.2%]
Performing permutations : [61.4%]
Performing permutations : [61.6%]
Performing permutations : [61.8%]
Performing permutations : [62.0%]
Performing permutations : [62.2%]
Performing permutations : [62.4%]
Performing permutations : [62.6%]
Performing permutations : [62.8%]
Performing permutations : [63.0%]
Performing permutations : [63.2%]
Performing permutations : [63.4%]
Performing permutations : [63.6%]
Performing permutations : [63.8%]
Performing permutations : [64.0%]
Performing permutations : [64.2%]
Performing permutations : [64.4%]
Performing permutations : [64.6%]
Performing permutations : [64.8%]
Performing permutations : [65.0%]
Performing permutations : [65.2%]
Performing permutations : [65.4%]
Performing permutations : [65.6%]
Performing permutations : [65.8%]
Performing permutations : [66.0%]
Performing permutations : [66.2%]
Performing permutations : [66.4%]
Performing permutations : [66.6%]
Performing permutations : [66.8%]
Performing permutations : [67.0%]
Performing permutations : [67.2%]
Performing permutations : [67.4%]
Performing permutations : [67.6%]
Performing permutations : [67.8%]
Performing permutations : [68.0%]
Performing permutations : [68.2%]
Performing permutations : [68.4%]
Performing permutations : [68.6%]
Performing permutations : [68.8%]
Performing permutations : [69.0%]
Performing permutations : [69.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [69.4%]
Performing permutations : [69.6%]
Performing permutations : [69.8%]
Performing permutations : [70.0%]
Performing permutations : [70.2%]
Performing permutations : [70.4%]
Performing permutations : [70.6%]
Performing permutations : [70.8%]
Performing permutations : [71.0%]
Performing permutations : [71.2%]
Performing permutations : [71.4%]
Performing permutations : [71.6%]
Performing permutations : [71.8%]
Performing permutations : [72.0%]
Performing permutations : [72.2%]
Performing permutations : [72.4%]
Performing permutations : [72.6%]
Performing permutations : [72.8%]
Performing permutations : [73.0%]
Performing permutations : [73.2%]
Performing permutations : [73.4%]
Performing permutations : [73.6%]
Performing permutations : [73.8%]
Performing permutations : [74.0%]
Performing permutations : [74.2%]
Performing permutations : [74.4%]
Performing permutations : [74.6%]
Performing permutations : [74.8%]
Performing permutations : [75.0%]
Performing permutations : [75.2%]
Performing permutations : [75.4%]
Performing permutations : [75.6%]
Performing permutations : [75.8%]
Performing permutations : [76.0%]
Performing permutations : [76.2%]
Performing permutations : [76.4%]
Performing permutations : [76.6%]
Performing permutations : [76.8%]
Performing permutations : [77.0%]
Performing permutations : [77.2%]
Performing permutations : [77.4%]
Performing permutations : [77.6%]
Performing permutations : [77.8%]
Performing permutations : [78.0%]
Performing permutations : [78.2%]
Performing permutations : [78.4%]
Performing permutations : [78.6%]
Performing permutations : [78.8%]
Performing permutations : [79.0%]
Performing permutations : [79.2%]
Performing permutations : [79.4%]
Performing permutations : [79.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [79.8%]
Performing permutations : [80.0%]
Performing permutations : [80.2%]
Performing permutations : [80.4%]
Performing permutations : [80.6%]
Performing permutations : [80.8%]
Performing permutations : [81.0%]
Performing permutations : [81.2%]
Performing permutations : [81.4%]
Performing permutations : [81.6%]
Performing permutations : [81.8%]
Performing permutations : [82.0%]
Performing permutations : [82.2%]
Performing permutations : [82.4%]
Performing permutations : [82.6%]
Performing permutations : [82.8%]
Performing permutations : [83.0%]
Performing permutations : [83.2%]
Performing permutations : [83.4%]
Performing permutations : [83.6%]
Performing permutations : [83.8%]
Performing permutations : [84.0%]
Performing permutations : [84.2%]
Performing permutations : [84.4%]
Performing permutations : [84.6%]
Performing permutations : [84.8%]
Performing permutations : [85.0%]
Performing permutations : [85.2%]
Performing permutations : [85.4%]
Performing permutations : [85.6%]
Performing permutations : [85.8%]
Performing permutations : [86.0%]
Performing permutations : [86.2%]
Performing permutations : [86.4%]
Performing permutations : [86.6%]
Performing permutations : [86.8%]
Performing permutations : [87.0%]
Performing permutations : [87.2%]
Performing permutations : [87.4%]
Performing permutations : [87.6%]
Performing permutations : [87.8%]
Performing permutations : [88.0%]
Performing permutations : [88.2%]
Performing permutations : [88.4%]
Performing permutations : [88.6%]
Performing permutations : [88.8%]
Performing permutations : [89.0%]
Performing permutations : [89.2%]
Performing permutations : [89.4%]
Performing permutations : [89.6%]
Performing permutations : [89.8%]
Performing permutations : [90.0%]
```

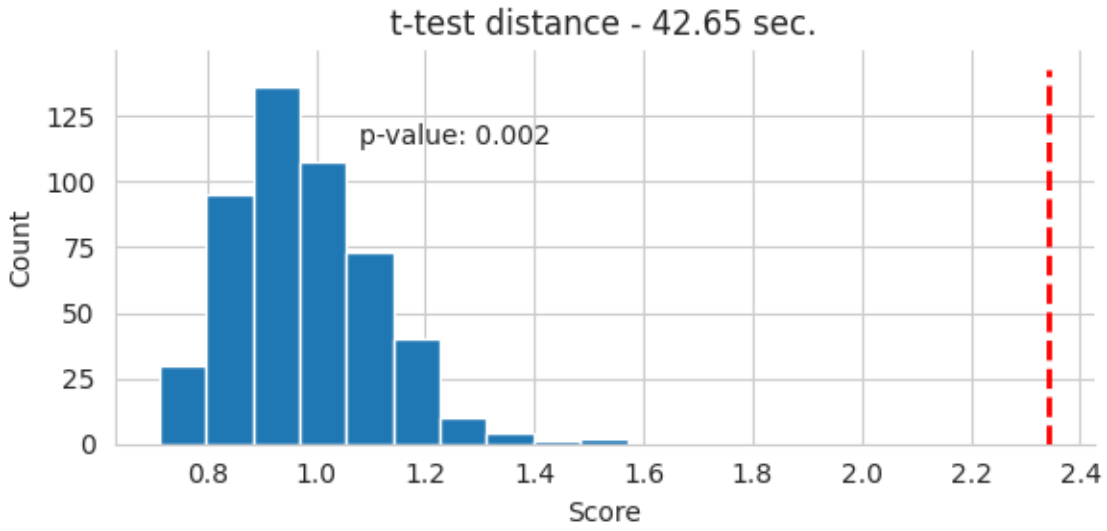
(continues on next page)

(continued from previous page)

```
Performing permutations : [90.2%]
Performing permutations : [90.4%]
Performing permutations : [90.6%]
Performing permutations : [90.8%]
Performing permutations : [91.0%]
Performing permutations : [91.2%]
Performing permutations : [91.4%]
Performing permutations : [91.6%]
Performing permutations : [91.8%]
Performing permutations : [92.0%]
Performing permutations : [92.2%]
Performing permutations : [92.4%]
Performing permutations : [92.6%]
Performing permutations : [92.8%]
Performing permutations : [93.0%]
Performing permutations : [93.2%]
Performing permutations : [93.4%]
Performing permutations : [93.6%]
Performing permutations : [93.8%]
Performing permutations : [94.0%]
Performing permutations : [94.2%]
Performing permutations : [94.4%]
Performing permutations : [94.6%]
Performing permutations : [94.8%]
Performing permutations : [95.0%]
Performing permutations : [95.2%]
Performing permutations : [95.4%]
Performing permutations : [95.6%]
Performing permutations : [95.8%]
Performing permutations : [96.0%]
Performing permutations : [96.2%]
Performing permutations : [96.4%]
Performing permutations : [96.6%]
Performing permutations : [96.8%]
Performing permutations : [97.0%]
Performing permutations : [97.2%]
Performing permutations : [97.4%]
Performing permutations : [97.6%]
Performing permutations : [97.8%]
Performing permutations : [98.0%]
Performing permutations : [98.2%]
Performing permutations : [98.4%]
Performing permutations : [98.6%]
Performing permutations : [98.8%]
Performing permutations : [99.0%]
Performing permutations : [99.2%]
Performing permutations : [99.4%]
Performing permutations : [99.6%]
Performing permutations : [99.8%]
p-value: 0.002
```

t-test distance based permutation test

```
t_init = time()
p_test = PermutationDistance(n_perms, metric='riemann', mode='ttest')
p, F = p_test.test(covmats, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('t-test distance - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
plt.tight_layout()
plt.show()
```




```
Performing permutations : [0.2%]
Performing permutations : [0.4%]
Performing permutations : [0.6%]
Performing permutations : [0.8%]
Performing permutations : [1.0%]
Performing permutations : [1.2%]
Performing permutations : [1.4%]
Performing permutations : [1.6%]
Performing permutations : [1.8%]
Performing permutations : [2.0%]
Performing permutations : [2.2%]
Performing permutations : [2.4%]
Performing permutations : [2.6%]
Performing permutations : [2.8%]
Performing permutations : [3.0%]
Performing permutations : [3.2%]
Performing permutations : [3.4%]
Performing permutations : [3.6%]
Performing permutations : [3.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [4.0%]
Performing permutations : [4.2%]
Performing permutations : [4.4%]
Performing permutations : [4.6%]
Performing permutations : [4.8%]
Performing permutations : [5.0%]
Performing permutations : [5.2%]
Performing permutations : [5.4%]
Performing permutations : [5.6%]
Performing permutations : [5.8%]
Performing permutations : [6.0%]
Performing permutations : [6.2%]
Performing permutations : [6.4%]
Performing permutations : [6.6%]
Performing permutations : [6.8%]
Performing permutations : [7.0%]
Performing permutations : [7.2%]
Performing permutations : [7.4%]
Performing permutations : [7.6%]
Performing permutations : [7.8%]
Performing permutations : [8.0%]
Performing permutations : [8.2%]
Performing permutations : [8.4%]
Performing permutations : [8.6%]
Performing permutations : [8.8%]
Performing permutations : [9.0%]
Performing permutations : [9.2%]
Performing permutations : [9.4%]
Performing permutations : [9.6%]
Performing permutations : [9.8%]
Performing permutations : [10.0%]
Performing permutations : [10.2%]
Performing permutations : [10.4%]
Performing permutations : [10.6%]
Performing permutations : [10.8%]
Performing permutations : [11.0%]
Performing permutations : [11.2%]
Performing permutations : [11.4%]
Performing permutations : [11.6%]
Performing permutations : [11.8%]
Performing permutations : [12.0%]
Performing permutations : [12.2%]
Performing permutations : [12.4%]
Performing permutations : [12.6%]
Performing permutations : [12.8%]
Performing permutations : [13.0%]
Performing permutations : [13.2%]
Performing permutations : [13.4%]
Performing permutations : [13.6%]
Performing permutations : [13.8%]
Performing permutations : [14.0%]
Performing permutations : [14.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [14.4%]
Performing permutations : [14.6%]
Performing permutations : [14.8%]
Performing permutations : [15.0%]
Performing permutations : [15.2%]
Performing permutations : [15.4%]
Performing permutations : [15.6%]
Performing permutations : [15.8%]
Performing permutations : [16.0%]
Performing permutations : [16.2%]
Performing permutations : [16.4%]
Performing permutations : [16.6%]
Performing permutations : [16.8%]
Performing permutations : [17.0%]
Performing permutations : [17.2%]
Performing permutations : [17.4%]
Performing permutations : [17.6%]
Performing permutations : [17.8%]
Performing permutations : [18.0%]
Performing permutations : [18.2%]
Performing permutations : [18.4%]
Performing permutations : [18.6%]
Performing permutations : [18.8%]
Performing permutations : [19.0%]
Performing permutations : [19.2%]
Performing permutations : [19.4%]
Performing permutations : [19.6%]
Performing permutations : [19.8%]
Performing permutations : [20.0%]
Performing permutations : [20.2%]
Performing permutations : [20.4%]
Performing permutations : [20.6%]
Performing permutations : [20.8%]
Performing permutations : [21.0%]
Performing permutations : [21.2%]
Performing permutations : [21.4%]
Performing permutations : [21.6%]
Performing permutations : [21.8%]
Performing permutations : [22.0%]
Performing permutations : [22.2%]
Performing permutations : [22.4%]
Performing permutations : [22.6%]
Performing permutations : [22.8%]
Performing permutations : [23.0%]
Performing permutations : [23.2%]
Performing permutations : [23.4%]
Performing permutations : [23.6%]
Performing permutations : [23.8%]
Performing permutations : [24.0%]
Performing permutations : [24.2%]
Performing permutations : [24.4%]
Performing permutations : [24.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [24.8%]
Performing permutations : [25.0%]
Performing permutations : [25.2%]
Performing permutations : [25.4%]
Performing permutations : [25.6%]
Performing permutations : [25.8%]
Performing permutations : [26.0%]
Performing permutations : [26.2%]
Performing permutations : [26.4%]
Performing permutations : [26.6%]
Performing permutations : [26.8%]
Performing permutations : [27.0%]
Performing permutations : [27.2%]
Performing permutations : [27.4%]
Performing permutations : [27.6%]
Performing permutations : [27.8%]
Performing permutations : [28.0%]
Performing permutations : [28.2%]
Performing permutations : [28.4%]
Performing permutations : [28.6%]
Performing permutations : [28.8%]
Performing permutations : [29.0%]
Performing permutations : [29.2%]
Performing permutations : [29.4%]
Performing permutations : [29.6%]
Performing permutations : [29.8%]
Performing permutations : [30.0%]
Performing permutations : [30.2%]
Performing permutations : [30.4%]
Performing permutations : [30.6%]
Performing permutations : [30.8%]
Performing permutations : [31.0%]
Performing permutations : [31.2%]
Performing permutations : [31.4%]
Performing permutations : [31.6%]
Performing permutations : [31.8%]
Performing permutations : [32.0%]
Performing permutations : [32.2%]
Performing permutations : [32.4%]
Performing permutations : [32.6%]
Performing permutations : [32.8%]
Performing permutations : [33.0%]
Performing permutations : [33.2%]
Performing permutations : [33.4%]
Performing permutations : [33.6%]
Performing permutations : [33.8%]
Performing permutations : [34.0%]
Performing permutations : [34.2%]
Performing permutations : [34.4%]
Performing permutations : [34.6%]
Performing permutations : [34.8%]
Performing permutations : [35.0%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [35.2%]
Performing permutations : [35.4%]
Performing permutations : [35.6%]
Performing permutations : [35.8%]
Performing permutations : [36.0%]
Performing permutations : [36.2%]
Performing permutations : [36.4%]
Performing permutations : [36.6%]
Performing permutations : [36.8%]
Performing permutations : [37.0%]
Performing permutations : [37.2%]
Performing permutations : [37.4%]
Performing permutations : [37.6%]
Performing permutations : [37.8%]
Performing permutations : [38.0%]
Performing permutations : [38.2%]
Performing permutations : [38.4%]
Performing permutations : [38.6%]
Performing permutations : [38.8%]
Performing permutations : [39.0%]
Performing permutations : [39.2%]
Performing permutations : [39.4%]
Performing permutations : [39.6%]
Performing permutations : [39.8%]
Performing permutations : [40.0%]
Performing permutations : [40.2%]
Performing permutations : [40.4%]
Performing permutations : [40.6%]
Performing permutations : [40.8%]
Performing permutations : [41.0%]
Performing permutations : [41.2%]
Performing permutations : [41.4%]
Performing permutations : [41.6%]
Performing permutations : [41.8%]
Performing permutations : [42.0%]
Performing permutations : [42.2%]
Performing permutations : [42.4%]
Performing permutations : [42.6%]
Performing permutations : [42.8%]
Performing permutations : [43.0%]
Performing permutations : [43.2%]
Performing permutations : [43.4%]
Performing permutations : [43.6%]
Performing permutations : [43.8%]
Performing permutations : [44.0%]
Performing permutations : [44.2%]
Performing permutations : [44.4%]
Performing permutations : [44.6%]
Performing permutations : [44.8%]
Performing permutations : [45.0%]
Performing permutations : [45.2%]
Performing permutations : [45.4%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [45.6%]
Performing permutations : [45.8%]
Performing permutations : [46.0%]
Performing permutations : [46.2%]
Performing permutations : [46.4%]
Performing permutations : [46.6%]
Performing permutations : [46.8%]
Performing permutations : [47.0%]
Performing permutations : [47.2%]
Performing permutations : [47.4%]
Performing permutations : [47.6%]
Performing permutations : [47.8%]
Performing permutations : [48.0%]
Performing permutations : [48.2%]
Performing permutations : [48.4%]
Performing permutations : [48.6%]
Performing permutations : [48.8%]
Performing permutations : [49.0%]
Performing permutations : [49.2%]
Performing permutations : [49.4%]
Performing permutations : [49.6%]
Performing permutations : [49.8%]
Performing permutations : [50.0%]
Performing permutations : [50.2%]
Performing permutations : [50.4%]
Performing permutations : [50.6%]
Performing permutations : [50.8%]
Performing permutations : [51.0%]
Performing permutations : [51.2%]
Performing permutations : [51.4%]
Performing permutations : [51.6%]
Performing permutations : [51.8%]
Performing permutations : [52.0%]
Performing permutations : [52.2%]
Performing permutations : [52.4%]
Performing permutations : [52.6%]
Performing permutations : [52.8%]
Performing permutations : [53.0%]
Performing permutations : [53.2%]
Performing permutations : [53.4%]
Performing permutations : [53.6%]
Performing permutations : [53.8%]
Performing permutations : [54.0%]
Performing permutations : [54.2%]
Performing permutations : [54.4%]
Performing permutations : [54.6%]
Performing permutations : [54.8%]
Performing permutations : [55.0%]
Performing permutations : [55.2%]
Performing permutations : [55.4%]
Performing permutations : [55.6%]
Performing permutations : [55.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [56.0%]
Performing permutations : [56.2%]
Performing permutations : [56.4%]
Performing permutations : [56.6%]
Performing permutations : [56.8%]
Performing permutations : [57.0%]
Performing permutations : [57.2%]
Performing permutations : [57.4%]
Performing permutations : [57.6%]
Performing permutations : [57.8%]
Performing permutations : [58.0%]
Performing permutations : [58.2%]
Performing permutations : [58.4%]
Performing permutations : [58.6%]
Performing permutations : [58.8%]
Performing permutations : [59.0%]
Performing permutations : [59.2%]
Performing permutations : [59.4%]
Performing permutations : [59.6%]
Performing permutations : [59.8%]
Performing permutations : [60.0%]
Performing permutations : [60.2%]
Performing permutations : [60.4%]
Performing permutations : [60.6%]
Performing permutations : [60.8%]
Performing permutations : [61.0%]
Performing permutations : [61.2%]
Performing permutations : [61.4%]
Performing permutations : [61.6%]
Performing permutations : [61.8%]
Performing permutations : [62.0%]
Performing permutations : [62.2%]
Performing permutations : [62.4%]
Performing permutations : [62.6%]
Performing permutations : [62.8%]
Performing permutations : [63.0%]
Performing permutations : [63.2%]
Performing permutations : [63.4%]
Performing permutations : [63.6%]
Performing permutations : [63.8%]
Performing permutations : [64.0%]
Performing permutations : [64.2%]
Performing permutations : [64.4%]
Performing permutations : [64.6%]
Performing permutations : [64.8%]
Performing permutations : [65.0%]
Performing permutations : [65.2%]
Performing permutations : [65.4%]
Performing permutations : [65.6%]
Performing permutations : [65.8%]
Performing permutations : [66.0%]
Performing permutations : [66.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [66.4%]
Performing permutations : [66.6%]
Performing permutations : [66.8%]
Performing permutations : [67.0%]
Performing permutations : [67.2%]
Performing permutations : [67.4%]
Performing permutations : [67.6%]
Performing permutations : [67.8%]
Performing permutations : [68.0%]
Performing permutations : [68.2%]
Performing permutations : [68.4%]
Performing permutations : [68.6%]
Performing permutations : [68.8%]
Performing permutations : [69.0%]
Performing permutations : [69.2%]
Performing permutations : [69.4%]
Performing permutations : [69.6%]
Performing permutations : [69.8%]
Performing permutations : [70.0%]
Performing permutations : [70.2%]
Performing permutations : [70.4%]
Performing permutations : [70.6%]
Performing permutations : [70.8%]
Performing permutations : [71.0%]
Performing permutations : [71.2%]
Performing permutations : [71.4%]
Performing permutations : [71.6%]
Performing permutations : [71.8%]
Performing permutations : [72.0%]
Performing permutations : [72.2%]
Performing permutations : [72.4%]
Performing permutations : [72.6%]
Performing permutations : [72.8%]
Performing permutations : [73.0%]
Performing permutations : [73.2%]
Performing permutations : [73.4%]
Performing permutations : [73.6%]
Performing permutations : [73.8%]
Performing permutations : [74.0%]
Performing permutations : [74.2%]
Performing permutations : [74.4%]
Performing permutations : [74.6%]
Performing permutations : [74.8%]
Performing permutations : [75.0%]
Performing permutations : [75.2%]
Performing permutations : [75.4%]
Performing permutations : [75.6%]
Performing permutations : [75.8%]
Performing permutations : [76.0%]
Performing permutations : [76.2%]
Performing permutations : [76.4%]
Performing permutations : [76.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [76.8%]
Performing permutations : [77.0%]
Performing permutations : [77.2%]
Performing permutations : [77.4%]
Performing permutations : [77.6%]
Performing permutations : [77.8%]
Performing permutations : [78.0%]
Performing permutations : [78.2%]
Performing permutations : [78.4%]
Performing permutations : [78.6%]
Performing permutations : [78.8%]
Performing permutations : [79.0%]
Performing permutations : [79.2%]
Performing permutations : [79.4%]
Performing permutations : [79.6%]
Performing permutations : [79.8%]
Performing permutations : [80.0%]
Performing permutations : [80.2%]
Performing permutations : [80.4%]
Performing permutations : [80.6%]
Performing permutations : [80.8%]
Performing permutations : [81.0%]
Performing permutations : [81.2%]
Performing permutations : [81.4%]
Performing permutations : [81.6%]
Performing permutations : [81.8%]
Performing permutations : [82.0%]
Performing permutations : [82.2%]
Performing permutations : [82.4%]
Performing permutations : [82.6%]
Performing permutations : [82.8%]
Performing permutations : [83.0%]
Performing permutations : [83.2%]
Performing permutations : [83.4%]
Performing permutations : [83.6%]
Performing permutations : [83.8%]
Performing permutations : [84.0%]
Performing permutations : [84.2%]
Performing permutations : [84.4%]
Performing permutations : [84.6%]
Performing permutations : [84.8%]
Performing permutations : [85.0%]
Performing permutations : [85.2%]
Performing permutations : [85.4%]
Performing permutations : [85.6%]
Performing permutations : [85.8%]
Performing permutations : [86.0%]
Performing permutations : [86.2%]
Performing permutations : [86.4%]
Performing permutations : [86.6%]
Performing permutations : [86.8%]
Performing permutations : [87.0%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [87.2%]
Performing permutations : [87.4%]
Performing permutations : [87.6%]
Performing permutations : [87.8%]
Performing permutations : [88.0%]
Performing permutations : [88.2%]
Performing permutations : [88.4%]
Performing permutations : [88.6%]
Performing permutations : [88.8%]
Performing permutations : [89.0%]
Performing permutations : [89.2%]
Performing permutations : [89.4%]
Performing permutations : [89.6%]
Performing permutations : [89.8%]
Performing permutations : [90.0%]
Performing permutations : [90.2%]
Performing permutations : [90.4%]
Performing permutations : [90.6%]
Performing permutations : [90.8%]
Performing permutations : [91.0%]
Performing permutations : [91.2%]
Performing permutations : [91.4%]
Performing permutations : [91.6%]
Performing permutations : [91.8%]
Performing permutations : [92.0%]
Performing permutations : [92.2%]
Performing permutations : [92.4%]
Performing permutations : [92.6%]
Performing permutations : [92.8%]
Performing permutations : [93.0%]
Performing permutations : [93.2%]
Performing permutations : [93.4%]
Performing permutations : [93.6%]
Performing permutations : [93.8%]
Performing permutations : [94.0%]
Performing permutations : [94.2%]
Performing permutations : [94.4%]
Performing permutations : [94.6%]
Performing permutations : [94.8%]
Performing permutations : [95.0%]
Performing permutations : [95.2%]
Performing permutations : [95.4%]
Performing permutations : [95.6%]
Performing permutations : [95.8%]
Performing permutations : [96.0%]
Performing permutations : [96.2%]
Performing permutations : [96.4%]
Performing permutations : [96.6%]
Performing permutations : [96.8%]
Performing permutations : [97.0%]
Performing permutations : [97.2%]
Performing permutations : [97.4%]
```

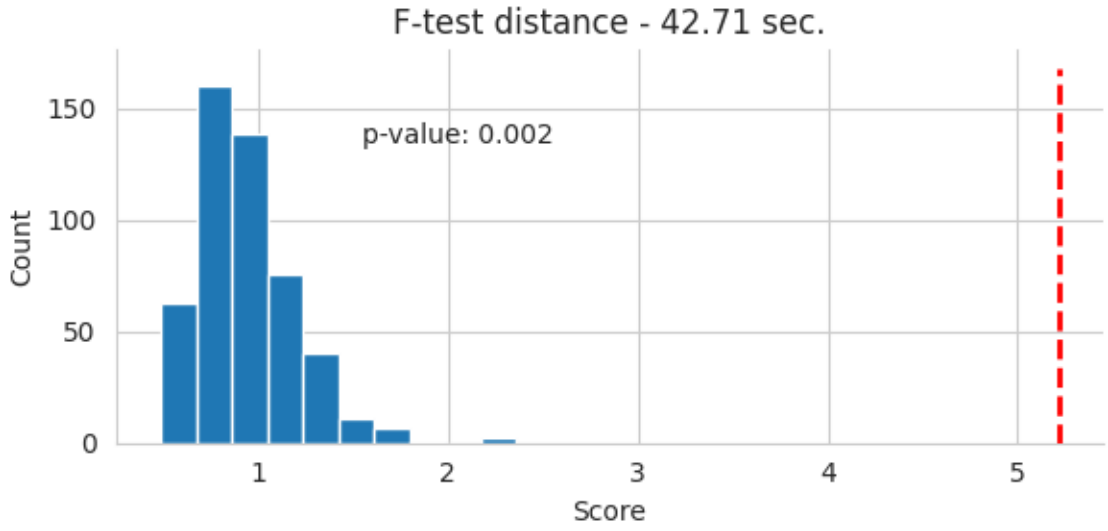
(continues on next page)

(continued from previous page)

```
Performing permutations : [97.6%]
Performing permutations : [97.8%]
Performing permutations : [98.0%]
Performing permutations : [98.2%]
Performing permutations : [98.4%]
Performing permutations : [98.6%]
Performing permutations : [98.8%]
Performing permutations : [99.0%]
Performing permutations : [99.2%]
Performing permutations : [99.4%]
Performing permutations : [99.6%]
Performing permutations : [99.8%]
p-value: 0.002
```

F-test distance based permutation test

```
t_init = time()
p_test = PermutationDistance(n_perms, metric='riemann', mode='ftest')
p, F = p_test.test(covmats, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('F-test distance - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
plt.tight_layout()
plt.show()
```




```
Performing permutations : [0.2%]
Performing permutations : [0.4%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [0.6%]
Performing permutations : [0.8%]
Performing permutations : [1.0%]
Performing permutations : [1.2%]
Performing permutations : [1.4%]
Performing permutations : [1.6%]
Performing permutations : [1.8%]
Performing permutations : [2.0%]
Performing permutations : [2.2%]
Performing permutations : [2.4%]
Performing permutations : [2.6%]
Performing permutations : [2.8%]
Performing permutations : [3.0%]
Performing permutations : [3.2%]
Performing permutations : [3.4%]
Performing permutations : [3.6%]
Performing permutations : [3.8%]
Performing permutations : [4.0%]
Performing permutations : [4.2%]
Performing permutations : [4.4%]
Performing permutations : [4.6%]
Performing permutations : [4.8%]
Performing permutations : [5.0%]
Performing permutations : [5.2%]
Performing permutations : [5.4%]
Performing permutations : [5.6%]
Performing permutations : [5.8%]
Performing permutations : [6.0%]
Performing permutations : [6.2%]
Performing permutations : [6.4%]
Performing permutations : [6.6%]
Performing permutations : [6.8%]
Performing permutations : [7.0%]
Performing permutations : [7.2%]
Performing permutations : [7.4%]
Performing permutations : [7.6%]
Performing permutations : [7.8%]
Performing permutations : [8.0%]
Performing permutations : [8.2%]
Performing permutations : [8.4%]
Performing permutations : [8.6%]
Performing permutations : [8.8%]
Performing permutations : [9.0%]
Performing permutations : [9.2%]
Performing permutations : [9.4%]
Performing permutations : [9.6%]
Performing permutations : [9.8%]
Performing permutations : [10.0%]
Performing permutations : [10.2%]
Performing permutations : [10.4%]
Performing permutations : [10.6%]
Performing permutations : [10.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [11.0%]
Performing permutations : [11.2%]
Performing permutations : [11.4%]
Performing permutations : [11.6%]
Performing permutations : [11.8%]
Performing permutations : [12.0%]
Performing permutations : [12.2%]
Performing permutations : [12.4%]
Performing permutations : [12.6%]
Performing permutations : [12.8%]
Performing permutations : [13.0%]
Performing permutations : [13.2%]
Performing permutations : [13.4%]
Performing permutations : [13.6%]
Performing permutations : [13.8%]
Performing permutations : [14.0%]
Performing permutations : [14.2%]
Performing permutations : [14.4%]
Performing permutations : [14.6%]
Performing permutations : [14.8%]
Performing permutations : [15.0%]
Performing permutations : [15.2%]
Performing permutations : [15.4%]
Performing permutations : [15.6%]
Performing permutations : [15.8%]
Performing permutations : [16.0%]
Performing permutations : [16.2%]
Performing permutations : [16.4%]
Performing permutations : [16.6%]
Performing permutations : [16.8%]
Performing permutations : [17.0%]
Performing permutations : [17.2%]
Performing permutations : [17.4%]
Performing permutations : [17.6%]
Performing permutations : [17.8%]
Performing permutations : [18.0%]
Performing permutations : [18.2%]
Performing permutations : [18.4%]
Performing permutations : [18.6%]
Performing permutations : [18.8%]
Performing permutations : [19.0%]
Performing permutations : [19.2%]
Performing permutations : [19.4%]
Performing permutations : [19.6%]
Performing permutations : [19.8%]
Performing permutations : [20.0%]
Performing permutations : [20.2%]
Performing permutations : [20.4%]
Performing permutations : [20.6%]
Performing permutations : [20.8%]
Performing permutations : [21.0%]
Performing permutations : [21.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [21.4%]
Performing permutations : [21.6%]
Performing permutations : [21.8%]
Performing permutations : [22.0%]
Performing permutations : [22.2%]
Performing permutations : [22.4%]
Performing permutations : [22.6%]
Performing permutations : [22.8%]
Performing permutations : [23.0%]
Performing permutations : [23.2%]
Performing permutations : [23.4%]
Performing permutations : [23.6%]
Performing permutations : [23.8%]
Performing permutations : [24.0%]
Performing permutations : [24.2%]
Performing permutations : [24.4%]
Performing permutations : [24.6%]
Performing permutations : [24.8%]
Performing permutations : [25.0%]
Performing permutations : [25.2%]
Performing permutations : [25.4%]
Performing permutations : [25.6%]
Performing permutations : [25.8%]
Performing permutations : [26.0%]
Performing permutations : [26.2%]
Performing permutations : [26.4%]
Performing permutations : [26.6%]
Performing permutations : [26.8%]
Performing permutations : [27.0%]
Performing permutations : [27.2%]
Performing permutations : [27.4%]
Performing permutations : [27.6%]
Performing permutations : [27.8%]
Performing permutations : [28.0%]
Performing permutations : [28.2%]
Performing permutations : [28.4%]
Performing permutations : [28.6%]
Performing permutations : [28.8%]
Performing permutations : [29.0%]
Performing permutations : [29.2%]
Performing permutations : [29.4%]
Performing permutations : [29.6%]
Performing permutations : [29.8%]
Performing permutations : [30.0%]
Performing permutations : [30.2%]
Performing permutations : [30.4%]
Performing permutations : [30.6%]
Performing permutations : [30.8%]
Performing permutations : [31.0%]
Performing permutations : [31.2%]
Performing permutations : [31.4%]
Performing permutations : [31.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [31.8%]
Performing permutations : [32.0%]
Performing permutations : [32.2%]
Performing permutations : [32.4%]
Performing permutations : [32.6%]
Performing permutations : [32.8%]
Performing permutations : [33.0%]
Performing permutations : [33.2%]
Performing permutations : [33.4%]
Performing permutations : [33.6%]
Performing permutations : [33.8%]
Performing permutations : [34.0%]
Performing permutations : [34.2%]
Performing permutations : [34.4%]
Performing permutations : [34.6%]
Performing permutations : [34.8%]
Performing permutations : [35.0%]
Performing permutations : [35.2%]
Performing permutations : [35.4%]
Performing permutations : [35.6%]
Performing permutations : [35.8%]
Performing permutations : [36.0%]
Performing permutations : [36.2%]
Performing permutations : [36.4%]
Performing permutations : [36.6%]
Performing permutations : [36.8%]
Performing permutations : [37.0%]
Performing permutations : [37.2%]
Performing permutations : [37.4%]
Performing permutations : [37.6%]
Performing permutations : [37.8%]
Performing permutations : [38.0%]
Performing permutations : [38.2%]
Performing permutations : [38.4%]
Performing permutations : [38.6%]
Performing permutations : [38.8%]
Performing permutations : [39.0%]
Performing permutations : [39.2%]
Performing permutations : [39.4%]
Performing permutations : [39.6%]
Performing permutations : [39.8%]
Performing permutations : [40.0%]
Performing permutations : [40.2%]
Performing permutations : [40.4%]
Performing permutations : [40.6%]
Performing permutations : [40.8%]
Performing permutations : [41.0%]
Performing permutations : [41.2%]
Performing permutations : [41.4%]
Performing permutations : [41.6%]
Performing permutations : [41.8%]
Performing permutations : [42.0%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [42.2%]
Performing permutations : [42.4%]
Performing permutations : [42.6%]
Performing permutations : [42.8%]
Performing permutations : [43.0%]
Performing permutations : [43.2%]
Performing permutations : [43.4%]
Performing permutations : [43.6%]
Performing permutations : [43.8%]
Performing permutations : [44.0%]
Performing permutations : [44.2%]
Performing permutations : [44.4%]
Performing permutations : [44.6%]
Performing permutations : [44.8%]
Performing permutations : [45.0%]
Performing permutations : [45.2%]
Performing permutations : [45.4%]
Performing permutations : [45.6%]
Performing permutations : [45.8%]
Performing permutations : [46.0%]
Performing permutations : [46.2%]
Performing permutations : [46.4%]
Performing permutations : [46.6%]
Performing permutations : [46.8%]
Performing permutations : [47.0%]
Performing permutations : [47.2%]
Performing permutations : [47.4%]
Performing permutations : [47.6%]
Performing permutations : [47.8%]
Performing permutations : [48.0%]
Performing permutations : [48.2%]
Performing permutations : [48.4%]
Performing permutations : [48.6%]
Performing permutations : [48.8%]
Performing permutations : [49.0%]
Performing permutations : [49.2%]
Performing permutations : [49.4%]
Performing permutations : [49.6%]
Performing permutations : [49.8%]
Performing permutations : [50.0%]
Performing permutations : [50.2%]
Performing permutations : [50.4%]
Performing permutations : [50.6%]
Performing permutations : [50.8%]
Performing permutations : [51.0%]
Performing permutations : [51.2%]
Performing permutations : [51.4%]
Performing permutations : [51.6%]
Performing permutations : [51.8%]
Performing permutations : [52.0%]
Performing permutations : [52.2%]
Performing permutations : [52.4%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [52.6%]
Performing permutations : [52.8%]
Performing permutations : [53.0%]
Performing permutations : [53.2%]
Performing permutations : [53.4%]
Performing permutations : [53.6%]
Performing permutations : [53.8%]
Performing permutations : [54.0%]
Performing permutations : [54.2%]
Performing permutations : [54.4%]
Performing permutations : [54.6%]
Performing permutations : [54.8%]
Performing permutations : [55.0%]
Performing permutations : [55.2%]
Performing permutations : [55.4%]
Performing permutations : [55.6%]
Performing permutations : [55.8%]
Performing permutations : [56.0%]
Performing permutations : [56.2%]
Performing permutations : [56.4%]
Performing permutations : [56.6%]
Performing permutations : [56.8%]
Performing permutations : [57.0%]
Performing permutations : [57.2%]
Performing permutations : [57.4%]
Performing permutations : [57.6%]
Performing permutations : [57.8%]
Performing permutations : [58.0%]
Performing permutations : [58.2%]
Performing permutations : [58.4%]
Performing permutations : [58.6%]
Performing permutations : [58.8%]
Performing permutations : [59.0%]
Performing permutations : [59.2%]
Performing permutations : [59.4%]
Performing permutations : [59.6%]
Performing permutations : [59.8%]
Performing permutations : [60.0%]
Performing permutations : [60.2%]
Performing permutations : [60.4%]
Performing permutations : [60.6%]
Performing permutations : [60.8%]
Performing permutations : [61.0%]
Performing permutations : [61.2%]
Performing permutations : [61.4%]
Performing permutations : [61.6%]
Performing permutations : [61.8%]
Performing permutations : [62.0%]
Performing permutations : [62.2%]
Performing permutations : [62.4%]
Performing permutations : [62.6%]
Performing permutations : [62.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [63.0%]
Performing permutations : [63.2%]
Performing permutations : [63.4%]
Performing permutations : [63.6%]
Performing permutations : [63.8%]
Performing permutations : [64.0%]
Performing permutations : [64.2%]
Performing permutations : [64.4%]
Performing permutations : [64.6%]
Performing permutations : [64.8%]
Performing permutations : [65.0%]
Performing permutations : [65.2%]
Performing permutations : [65.4%]
Performing permutations : [65.6%]
Performing permutations : [65.8%]
Performing permutations : [66.0%]
Performing permutations : [66.2%]
Performing permutations : [66.4%]
Performing permutations : [66.6%]
Performing permutations : [66.8%]
Performing permutations : [67.0%]
Performing permutations : [67.2%]
Performing permutations : [67.4%]
Performing permutations : [67.6%]
Performing permutations : [67.8%]
Performing permutations : [68.0%]
Performing permutations : [68.2%]
Performing permutations : [68.4%]
Performing permutations : [68.6%]
Performing permutations : [68.8%]
Performing permutations : [69.0%]
Performing permutations : [69.2%]
Performing permutations : [69.4%]
Performing permutations : [69.6%]
Performing permutations : [69.8%]
Performing permutations : [70.0%]
Performing permutations : [70.2%]
Performing permutations : [70.4%]
Performing permutations : [70.6%]
Performing permutations : [70.8%]
Performing permutations : [71.0%]
Performing permutations : [71.2%]
Performing permutations : [71.4%]
Performing permutations : [71.6%]
Performing permutations : [71.8%]
Performing permutations : [72.0%]
Performing permutations : [72.2%]
Performing permutations : [72.4%]
Performing permutations : [72.6%]
Performing permutations : [72.8%]
Performing permutations : [73.0%]
Performing permutations : [73.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [73.4%]
Performing permutations : [73.6%]
Performing permutations : [73.8%]
Performing permutations : [74.0%]
Performing permutations : [74.2%]
Performing permutations : [74.4%]
Performing permutations : [74.6%]
Performing permutations : [74.8%]
Performing permutations : [75.0%]
Performing permutations : [75.2%]
Performing permutations : [75.4%]
Performing permutations : [75.6%]
Performing permutations : [75.8%]
Performing permutations : [76.0%]
Performing permutations : [76.2%]
Performing permutations : [76.4%]
Performing permutations : [76.6%]
Performing permutations : [76.8%]
Performing permutations : [77.0%]
Performing permutations : [77.2%]
Performing permutations : [77.4%]
Performing permutations : [77.6%]
Performing permutations : [77.8%]
Performing permutations : [78.0%]
Performing permutations : [78.2%]
Performing permutations : [78.4%]
Performing permutations : [78.6%]
Performing permutations : [78.8%]
Performing permutations : [79.0%]
Performing permutations : [79.2%]
Performing permutations : [79.4%]
Performing permutations : [79.6%]
Performing permutations : [79.8%]
Performing permutations : [80.0%]
Performing permutations : [80.2%]
Performing permutations : [80.4%]
Performing permutations : [80.6%]
Performing permutations : [80.8%]
Performing permutations : [81.0%]
Performing permutations : [81.2%]
Performing permutations : [81.4%]
Performing permutations : [81.6%]
Performing permutations : [81.8%]
Performing permutations : [82.0%]
Performing permutations : [82.2%]
Performing permutations : [82.4%]
Performing permutations : [82.6%]
Performing permutations : [82.8%]
Performing permutations : [83.0%]
Performing permutations : [83.2%]
Performing permutations : [83.4%]
Performing permutations : [83.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [83.8%]
Performing permutations : [84.0%]
Performing permutations : [84.2%]
Performing permutations : [84.4%]
Performing permutations : [84.6%]
Performing permutations : [84.8%]
Performing permutations : [85.0%]
Performing permutations : [85.2%]
Performing permutations : [85.4%]
Performing permutations : [85.6%]
Performing permutations : [85.8%]
Performing permutations : [86.0%]
Performing permutations : [86.2%]
Performing permutations : [86.4%]
Performing permutations : [86.6%]
Performing permutations : [86.8%]
Performing permutations : [87.0%]
Performing permutations : [87.2%]
Performing permutations : [87.4%]
Performing permutations : [87.6%]
Performing permutations : [87.8%]
Performing permutations : [88.0%]
Performing permutations : [88.2%]
Performing permutations : [88.4%]
Performing permutations : [88.6%]
Performing permutations : [88.8%]
Performing permutations : [89.0%]
Performing permutations : [89.2%]
Performing permutations : [89.4%]
Performing permutations : [89.6%]
Performing permutations : [89.8%]
Performing permutations : [90.0%]
Performing permutations : [90.2%]
Performing permutations : [90.4%]
Performing permutations : [90.6%]
Performing permutations : [90.8%]
Performing permutations : [91.0%]
Performing permutations : [91.2%]
Performing permutations : [91.4%]
Performing permutations : [91.6%]
Performing permutations : [91.8%]
Performing permutations : [92.0%]
Performing permutations : [92.2%]
Performing permutations : [92.4%]
Performing permutations : [92.6%]
Performing permutations : [92.8%]
Performing permutations : [93.0%]
Performing permutations : [93.2%]
Performing permutations : [93.4%]
Performing permutations : [93.6%]
Performing permutations : [93.8%]
Performing permutations : [94.0%]
```

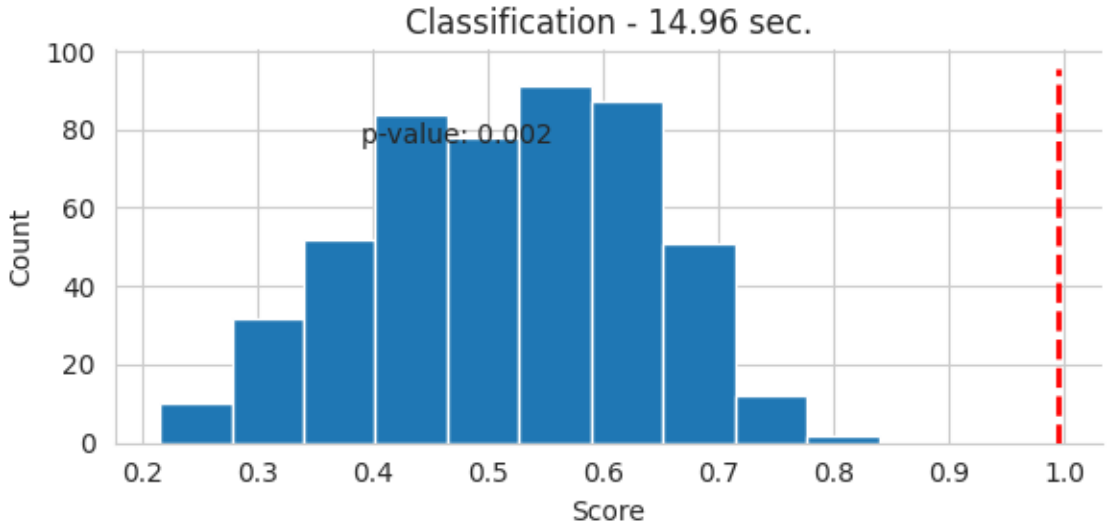
(continues on next page)

(continued from previous page)

```
Performing permutations : [94.2%]
Performing permutations : [94.4%]
Performing permutations : [94.6%]
Performing permutations : [94.8%]
Performing permutations : [95.0%]
Performing permutations : [95.2%]
Performing permutations : [95.4%]
Performing permutations : [95.6%]
Performing permutations : [95.8%]
Performing permutations : [96.0%]
Performing permutations : [96.2%]
Performing permutations : [96.4%]
Performing permutations : [96.6%]
Performing permutations : [96.8%]
Performing permutations : [97.0%]
Performing permutations : [97.2%]
Performing permutations : [97.4%]
Performing permutations : [97.6%]
Performing permutations : [97.8%]
Performing permutations : [98.0%]
Performing permutations : [98.2%]
Performing permutations : [98.4%]
Performing permutations : [98.6%]
Performing permutations : [98.8%]
Performing permutations : [99.0%]
Performing permutations : [99.2%]
Performing permutations : [99.4%]
Performing permutations : [99.6%]
Performing permutations : [99.8%]
p-value: 0.002
```

Classification based permutation test

```
clf = make_pipeline(CSP(4), LogisticRegression())

t_init = time()
p_test = PermutationModel(n_perms, model=clf, cv=3, scoring='roc_auc')
p, F = p_test.test(covmats, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('Classification - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
plt.tight_layout()
plt.show()
```




```
Performing permutations : [0.2%]
Performing permutations : [0.4%]
Performing permutations : [0.6%]
Performing permutations : [0.8%]
Performing permutations : [1.0%]
Performing permutations : [1.2%]
Performing permutations : [1.4%]
Performing permutations : [1.6%]
Performing permutations : [1.8%]
Performing permutations : [2.0%]
Performing permutations : [2.2%]
Performing permutations : [2.4%]
Performing permutations : [2.6%]
Performing permutations : [2.8%]
Performing permutations : [3.0%]
Performing permutations : [3.2%]
Performing permutations : [3.4%]
Performing permutations : [3.6%]
Performing permutations : [3.8%]
Performing permutations : [4.0%]
Performing permutations : [4.2%]
Performing permutations : [4.4%]
Performing permutations : [4.6%]
Performing permutations : [4.8%]
Performing permutations : [5.0%]
Performing permutations : [5.2%]
Performing permutations : [5.4%]
Performing permutations : [5.6%]
Performing permutations : [5.8%]
Performing permutations : [6.0%]
Performing permutations : [6.2%]
Performing permutations : [6.4%]
Performing permutations : [6.6%]
Performing permutations : [6.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [7.0%]
Performing permutations : [7.2%]
Performing permutations : [7.4%]
Performing permutations : [7.6%]
Performing permutations : [7.8%]
Performing permutations : [8.0%]
Performing permutations : [8.2%]
Performing permutations : [8.4%]
Performing permutations : [8.6%]
Performing permutations : [8.8%]
Performing permutations : [9.0%]
Performing permutations : [9.2%]
Performing permutations : [9.4%]
Performing permutations : [9.6%]
Performing permutations : [9.8%]
Performing permutations : [10.0%]
Performing permutations : [10.2%]
Performing permutations : [10.4%]
Performing permutations : [10.6%]
Performing permutations : [10.8%]
Performing permutations : [11.0%]
Performing permutations : [11.2%]
Performing permutations : [11.4%]
Performing permutations : [11.6%]
Performing permutations : [11.8%]
Performing permutations : [12.0%]
Performing permutations : [12.2%]
Performing permutations : [12.4%]
Performing permutations : [12.6%]
Performing permutations : [12.8%]
Performing permutations : [13.0%]
Performing permutations : [13.2%]
Performing permutations : [13.4%]
Performing permutations : [13.6%]
Performing permutations : [13.8%]
Performing permutations : [14.0%]
Performing permutations : [14.2%]
Performing permutations : [14.4%]
Performing permutations : [14.6%]
Performing permutations : [14.8%]
Performing permutations : [15.0%]
Performing permutations : [15.2%]
Performing permutations : [15.4%]
Performing permutations : [15.6%]
Performing permutations : [15.8%]
Performing permutations : [16.0%]
Performing permutations : [16.2%]
Performing permutations : [16.4%]
Performing permutations : [16.6%]
Performing permutations : [16.8%]
Performing permutations : [17.0%]
Performing permutations : [17.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [17.4%]
Performing permutations : [17.6%]
Performing permutations : [17.8%]
Performing permutations : [18.0%]
Performing permutations : [18.2%]
Performing permutations : [18.4%]
Performing permutations : [18.6%]
Performing permutations : [18.8%]
Performing permutations : [19.0%]
Performing permutations : [19.2%]
Performing permutations : [19.4%]
Performing permutations : [19.6%]
Performing permutations : [19.8%]
Performing permutations : [20.0%]
Performing permutations : [20.2%]
Performing permutations : [20.4%]
Performing permutations : [20.6%]
Performing permutations : [20.8%]
Performing permutations : [21.0%]
Performing permutations : [21.2%]
Performing permutations : [21.4%]
Performing permutations : [21.6%]
Performing permutations : [21.8%]
Performing permutations : [22.0%]
Performing permutations : [22.2%]
Performing permutations : [22.4%]
Performing permutations : [22.6%]
Performing permutations : [22.8%]
Performing permutations : [23.0%]
Performing permutations : [23.2%]
Performing permutations : [23.4%]
Performing permutations : [23.6%]
Performing permutations : [23.8%]
Performing permutations : [24.0%]
Performing permutations : [24.2%]
Performing permutations : [24.4%]
Performing permutations : [24.6%]
Performing permutations : [24.8%]
Performing permutations : [25.0%]
Performing permutations : [25.2%]
Performing permutations : [25.4%]
Performing permutations : [25.6%]
Performing permutations : [25.8%]
Performing permutations : [26.0%]
Performing permutations : [26.2%]
Performing permutations : [26.4%]
Performing permutations : [26.6%]
Performing permutations : [26.8%]
Performing permutations : [27.0%]
Performing permutations : [27.2%]
Performing permutations : [27.4%]
Performing permutations : [27.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [27.8%]
Performing permutations : [28.0%]
Performing permutations : [28.2%]
Performing permutations : [28.4%]
Performing permutations : [28.6%]
Performing permutations : [28.8%]
Performing permutations : [29.0%]
Performing permutations : [29.2%]
Performing permutations : [29.4%]
Performing permutations : [29.6%]
Performing permutations : [29.8%]
Performing permutations : [30.0%]
Performing permutations : [30.2%]
Performing permutations : [30.4%]
Performing permutations : [30.6%]
Performing permutations : [30.8%]
Performing permutations : [31.0%]
Performing permutations : [31.2%]
Performing permutations : [31.4%]
Performing permutations : [31.6%]
Performing permutations : [31.8%]
Performing permutations : [32.0%]
Performing permutations : [32.2%]
Performing permutations : [32.4%]
Performing permutations : [32.6%]
Performing permutations : [32.8%]
Performing permutations : [33.0%]
Performing permutations : [33.2%]
Performing permutations : [33.4%]
Performing permutations : [33.6%]
Performing permutations : [33.8%]
Performing permutations : [34.0%]
Performing permutations : [34.2%]
Performing permutations : [34.4%]
Performing permutations : [34.6%]
Performing permutations : [34.8%]
Performing permutations : [35.0%]
Performing permutations : [35.2%]
Performing permutations : [35.4%]
Performing permutations : [35.6%]
Performing permutations : [35.8%]
Performing permutations : [36.0%]
Performing permutations : [36.2%]
Performing permutations : [36.4%]
Performing permutations : [36.6%]
Performing permutations : [36.8%]
Performing permutations : [37.0%]
Performing permutations : [37.2%]
Performing permutations : [37.4%]
Performing permutations : [37.6%]
Performing permutations : [37.8%]
Performing permutations : [38.0%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [38.2%]
Performing permutations : [38.4%]
Performing permutations : [38.6%]
Performing permutations : [38.8%]
Performing permutations : [39.0%]
Performing permutations : [39.2%]
Performing permutations : [39.4%]
Performing permutations : [39.6%]
Performing permutations : [39.8%]
Performing permutations : [40.0%]
Performing permutations : [40.2%]
Performing permutations : [40.4%]
Performing permutations : [40.6%]
Performing permutations : [40.8%]
Performing permutations : [41.0%]
Performing permutations : [41.2%]
Performing permutations : [41.4%]
Performing permutations : [41.6%]
Performing permutations : [41.8%]
Performing permutations : [42.0%]
Performing permutations : [42.2%]
Performing permutations : [42.4%]
Performing permutations : [42.6%]
Performing permutations : [42.8%]
Performing permutations : [43.0%]
Performing permutations : [43.2%]
Performing permutations : [43.4%]
Performing permutations : [43.6%]
Performing permutations : [43.8%]
Performing permutations : [44.0%]
Performing permutations : [44.2%]
Performing permutations : [44.4%]
Performing permutations : [44.6%]
Performing permutations : [44.8%]
Performing permutations : [45.0%]
Performing permutations : [45.2%]
Performing permutations : [45.4%]
Performing permutations : [45.6%]
Performing permutations : [45.8%]
Performing permutations : [46.0%]
Performing permutations : [46.2%]
Performing permutations : [46.4%]
Performing permutations : [46.6%]
Performing permutations : [46.8%]
Performing permutations : [47.0%]
Performing permutations : [47.2%]
Performing permutations : [47.4%]
Performing permutations : [47.6%]
Performing permutations : [47.8%]
Performing permutations : [48.0%]
Performing permutations : [48.2%]
Performing permutations : [48.4%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [48.6%]
Performing permutations : [48.8%]
Performing permutations : [49.0%]
Performing permutations : [49.2%]
Performing permutations : [49.4%]
Performing permutations : [49.6%]
Performing permutations : [49.8%]
Performing permutations : [50.0%]
Performing permutations : [50.2%]
Performing permutations : [50.4%]
Performing permutations : [50.6%]
Performing permutations : [50.8%]
Performing permutations : [51.0%]
Performing permutations : [51.2%]
Performing permutations : [51.4%]
Performing permutations : [51.6%]
Performing permutations : [51.8%]
Performing permutations : [52.0%]
Performing permutations : [52.2%]
Performing permutations : [52.4%]
Performing permutations : [52.6%]
Performing permutations : [52.8%]
Performing permutations : [53.0%]
Performing permutations : [53.2%]
Performing permutations : [53.4%]
Performing permutations : [53.6%]
Performing permutations : [53.8%]
Performing permutations : [54.0%]
Performing permutations : [54.2%]
Performing permutations : [54.4%]
Performing permutations : [54.6%]
Performing permutations : [54.8%]
Performing permutations : [55.0%]
Performing permutations : [55.2%]
Performing permutations : [55.4%]
Performing permutations : [55.6%]
Performing permutations : [55.8%]
Performing permutations : [56.0%]
Performing permutations : [56.2%]
Performing permutations : [56.4%]
Performing permutations : [56.6%]
Performing permutations : [56.8%]
Performing permutations : [57.0%]
Performing permutations : [57.2%]
Performing permutations : [57.4%]
Performing permutations : [57.6%]
Performing permutations : [57.8%]
Performing permutations : [58.0%]
Performing permutations : [58.2%]
Performing permutations : [58.4%]
Performing permutations : [58.6%]
Performing permutations : [58.8%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [59.0%]
Performing permutations : [59.2%]
Performing permutations : [59.4%]
Performing permutations : [59.6%]
Performing permutations : [59.8%]
Performing permutations : [60.0%]
Performing permutations : [60.2%]
Performing permutations : [60.4%]
Performing permutations : [60.6%]
Performing permutations : [60.8%]
Performing permutations : [61.0%]
Performing permutations : [61.2%]
Performing permutations : [61.4%]
Performing permutations : [61.6%]
Performing permutations : [61.8%]
Performing permutations : [62.0%]
Performing permutations : [62.2%]
Performing permutations : [62.4%]
Performing permutations : [62.6%]
Performing permutations : [62.8%]
Performing permutations : [63.0%]
Performing permutations : [63.2%]
Performing permutations : [63.4%]
Performing permutations : [63.6%]
Performing permutations : [63.8%]
Performing permutations : [64.0%]
Performing permutations : [64.2%]
Performing permutations : [64.4%]
Performing permutations : [64.6%]
Performing permutations : [64.8%]
Performing permutations : [65.0%]
Performing permutations : [65.2%]
Performing permutations : [65.4%]
Performing permutations : [65.6%]
Performing permutations : [65.8%]
Performing permutations : [66.0%]
Performing permutations : [66.2%]
Performing permutations : [66.4%]
Performing permutations : [66.6%]
Performing permutations : [66.8%]
Performing permutations : [67.0%]
Performing permutations : [67.2%]
Performing permutations : [67.4%]
Performing permutations : [67.6%]
Performing permutations : [67.8%]
Performing permutations : [68.0%]
Performing permutations : [68.2%]
Performing permutations : [68.4%]
Performing permutations : [68.6%]
Performing permutations : [68.8%]
Performing permutations : [69.0%]
Performing permutations : [69.2%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [69.4%]
Performing permutations : [69.6%]
Performing permutations : [69.8%]
Performing permutations : [70.0%]
Performing permutations : [70.2%]
Performing permutations : [70.4%]
Performing permutations : [70.6%]
Performing permutations : [70.8%]
Performing permutations : [71.0%]
Performing permutations : [71.2%]
Performing permutations : [71.4%]
Performing permutations : [71.6%]
Performing permutations : [71.8%]
Performing permutations : [72.0%]
Performing permutations : [72.2%]
Performing permutations : [72.4%]
Performing permutations : [72.6%]
Performing permutations : [72.8%]
Performing permutations : [73.0%]
Performing permutations : [73.2%]
Performing permutations : [73.4%]
Performing permutations : [73.6%]
Performing permutations : [73.8%]
Performing permutations : [74.0%]
Performing permutations : [74.2%]
Performing permutations : [74.4%]
Performing permutations : [74.6%]
Performing permutations : [74.8%]
Performing permutations : [75.0%]
Performing permutations : [75.2%]
Performing permutations : [75.4%]
Performing permutations : [75.6%]
Performing permutations : [75.8%]
Performing permutations : [76.0%]
Performing permutations : [76.2%]
Performing permutations : [76.4%]
Performing permutations : [76.6%]
Performing permutations : [76.8%]
Performing permutations : [77.0%]
Performing permutations : [77.2%]
Performing permutations : [77.4%]
Performing permutations : [77.6%]
Performing permutations : [77.8%]
Performing permutations : [78.0%]
Performing permutations : [78.2%]
Performing permutations : [78.4%]
Performing permutations : [78.6%]
Performing permutations : [78.8%]
Performing permutations : [79.0%]
Performing permutations : [79.2%]
Performing permutations : [79.4%]
Performing permutations : [79.6%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [79.8%]
Performing permutations : [80.0%]
Performing permutations : [80.2%]
Performing permutations : [80.4%]
Performing permutations : [80.6%]
Performing permutations : [80.8%]
Performing permutations : [81.0%]
Performing permutations : [81.2%]
Performing permutations : [81.4%]
Performing permutations : [81.6%]
Performing permutations : [81.8%]
Performing permutations : [82.0%]
Performing permutations : [82.2%]
Performing permutations : [82.4%]
Performing permutations : [82.6%]
Performing permutations : [82.8%]
Performing permutations : [83.0%]
Performing permutations : [83.2%]
Performing permutations : [83.4%]
Performing permutations : [83.6%]
Performing permutations : [83.8%]
Performing permutations : [84.0%]
Performing permutations : [84.2%]
Performing permutations : [84.4%]
Performing permutations : [84.6%]
Performing permutations : [84.8%]
Performing permutations : [85.0%]
Performing permutations : [85.2%]
Performing permutations : [85.4%]
Performing permutations : [85.6%]
Performing permutations : [85.8%]
Performing permutations : [86.0%]
Performing permutations : [86.2%]
Performing permutations : [86.4%]
Performing permutations : [86.6%]
Performing permutations : [86.8%]
Performing permutations : [87.0%]
Performing permutations : [87.2%]
Performing permutations : [87.4%]
Performing permutations : [87.6%]
Performing permutations : [87.8%]
Performing permutations : [88.0%]
Performing permutations : [88.2%]
Performing permutations : [88.4%]
Performing permutations : [88.6%]
Performing permutations : [88.8%]
Performing permutations : [89.0%]
Performing permutations : [89.2%]
Performing permutations : [89.4%]
Performing permutations : [89.6%]
Performing permutations : [89.8%]
Performing permutations : [90.0%]
```

(continues on next page)

(continued from previous page)

```
Performing permutations : [90.2%]
Performing permutations : [90.4%]
Performing permutations : [90.6%]
Performing permutations : [90.8%]
Performing permutations : [91.0%]
Performing permutations : [91.2%]
Performing permutations : [91.4%]
Performing permutations : [91.6%]
Performing permutations : [91.8%]
Performing permutations : [92.0%]
Performing permutations : [92.2%]
Performing permutations : [92.4%]
Performing permutations : [92.6%]
Performing permutations : [92.8%]
Performing permutations : [93.0%]
Performing permutations : [93.2%]
Performing permutations : [93.4%]
Performing permutations : [93.6%]
Performing permutations : [93.8%]
Performing permutations : [94.0%]
Performing permutations : [94.2%]
Performing permutations : [94.4%]
Performing permutations : [94.6%]
Performing permutations : [94.8%]
Performing permutations : [95.0%]
Performing permutations : [95.2%]
Performing permutations : [95.4%]
Performing permutations : [95.6%]
Performing permutations : [95.8%]
Performing permutations : [96.0%]
Performing permutations : [96.2%]
Performing permutations : [96.4%]
Performing permutations : [96.6%]
Performing permutations : [96.8%]
Performing permutations : [97.0%]
Performing permutations : [97.2%]
Performing permutations : [97.4%]
Performing permutations : [97.6%]
Performing permutations : [97.8%]
Performing permutations : [98.0%]
Performing permutations : [98.2%]
Performing permutations : [98.4%]
Performing permutations : [98.6%]
Performing permutations : [98.8%]
Performing permutations : [99.0%]
Performing permutations : [99.2%]
Performing permutations : [99.4%]
Performing permutations : [99.6%]
Performing permutations : [99.8%]
p-value: 0.002
```

Total running time of the script: (1 minutes 41.706 seconds)

Manova for ERP data

```
# Authors: Alexandre Barachant <alexandre.barachant@gmail.com>
#
# License: BSD (3-clause)

import seaborn as sns

from time import time
from matplotlib import pyplot as plt

import mne
from mne import io
from mne.datasets import sample

from pyriemann.stats import PermutationDistance, PermutationModel
from pyriemann.estimation import XdawnCovariances
from pyriemann.tangentspace import TangentSpace

from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression

print(__doc__)
sns.set_style('whitegrid')
data_path = sample.data_path()
```

Set parameters and read data

```
raw_fname = data_path + '/MEG/sample/sample_audvis_filt-0-40_raw.fif'
event_fname = data_path + '/MEG/sample/sample_audvis_filt-0-40_raw-eve.fif'
tmin, tmax = -0., 1
event_id = dict(aud_l=1, aud_r=2, vis_l=3, vis_r=4)

# Setup for reading the raw data
raw = io.Raw(raw_fname, preload=True)
raw.filter(2, None, method='iir')  # replace baselining with high-pass
events = mne.read_events(event_fname)

raw.info['bads'] = ['MEG 2443']  # set bad channels
picks = mne.pick_types(raw.info, meg='grad', eeg=False, stim=False, eog=False,
                       exclude='bads')

# Read epochs
epochs = mne.Epochs(raw, events, event_id, tmin, tmax, proj=False,
                    picks=picks, baseline=None, preload=True, verbose=False)

labels = epochs.events[::5, -1]

# get epochs
epochs_data = epochs.get_data()[::5]
```

(continues on next page)

(continued from previous page)

```
n_perms = 100
```

Pairwise distance based permutation test

```
t_init = time()
p_test = PermutationDistance(n_perms, metric='riemann', mode='pairwise',
                             estimator=XdawnCovariances(2))
p, F = p_test.test(epochs_data, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('Pairwise distance - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
plt.tight_layout()
plt.show()
```

t-test distance based permutation test

```
t_init = time()
p_test = PermutationDistance(n_perms, metric='riemann', mode='ttest',
                             estimator=XdawnCovariances(2))
p, F = p_test.test(epochs_data, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('Pairwise distance - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
plt.tight_layout()
plt.show()
```

F-test distance based permutation test

```
t_init = time()
p_test = PermutationDistance(n_perms, metric='riemann', mode='ftest',
                             estimator=XdawnCovariances(2))
p, F = p_test.test(epochs_data, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('Pairwise distance - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

Classification based permutation test

```
clf = make_pipeline(XdawnCovariances(2), TangentSpace('logeuclid'),
                    LogisticRegression())

t_init = time()
p_test = PermutationModel(n_perms, model=clf, cv=3)
p, F = p_test.test(epochs_data, labels)
duration = time() - t_init

fig, axes = plt.subplots(1, 1, figsize=[6, 3], sharey=True)
p_test.plot(nbins=10, axes=axes)
plt.title('Classification - %.2f sec.' % duration)
print('p-value: %.3f' % p)
sns.despine()
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

4.8.8 Transfer learning

Using Riemannian geometry for transfer learning and domain adaptation.

Plot the data transformations in the Riemannian Procrustes Analysis

Use the SpectralEmbedding module to plot in 2D the transformations on the data points from source and target domains when applying the Riemannian Procrustes Analysis[1] to match their statistics.

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

from pyriemann.embedding import SpectralEmbedding
from pyriemann.datasets.simulated import make_classification_transfer
from pyriemann.transfer import (
    decode_domains,
    TLCenter,
    TLRotate,
)
```

Fix seed for reproducible results

[1] Riemannian Procrustes analysis: transfer learning for brain-computer interfaces PLC Rodrigues et al, IEEE Transactions on Biomedical Engineering, vol. 66, no. 8, pp. 2390-2401, December, 2018

```
seed = 66

# create source and target datasets
n_matrices = 50
X_enc, y_enc = make_classification_transfer(
    n_matrices=n_matrices,
    class_sep=2.0,
    class_disp=0.25,
    domain_sep=2.0,
    theta=np.pi/4,
    random_state=seed
)

# generate dataset
X_org, y, domain = decode_domains(X_enc, y_enc)

# instantiate object for doing spectral embeddings
emb = SpectralEmbedding(n_components=2, metric='riemann')

# create dict to store the embedding after each step of RPA
embedded_points = {}

# embed the original source and target datasets
points = np.concatenate([X_org, np.eye(2)[None, :, :]])  # stack the identity
embedded_points['origin'] = emb.fit_transform(points)

# embed the source and target datasets after recentering
rct = TLCenter(target_domain='target_domain')
X_rct = rct.fit_transform(X_org, y_enc)
points = np.concatenate([X_rct, np.eye(2)[None, :, :]])  # stack the identity
embedded_points['rct'] = emb.fit_transform(points)

# embed the source and target datasets after recentering
rot = TLRotate(target_domain='target_domain', metric='riemann')
X_rot = rot.fit_transform(X_rct, y_enc)

points = np.concatenate([X_org, X_rct, X_rot, np.eye(2)[None, :, :]])
S = emb.fit_transform(points)
S = S - S[-1]
embedded_points['origin'] = S[:4*n_matrices]
embedded_points['rct'] = S[4*n_matrices:8*n_matrices]
embedded_points['rot'] = S[8*n_matrices:-1]
```

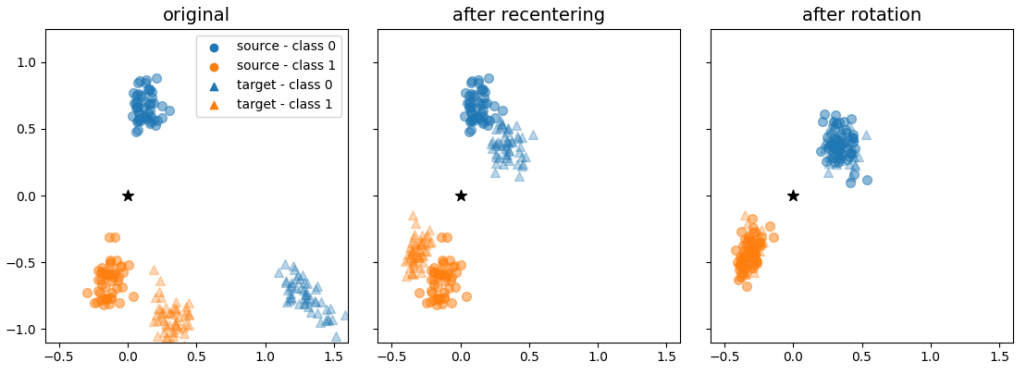
Plot the results, reproducing the Figure 1 of[Page 188, 1].

```
fig, ax = plt.subplots(figsize=(13.5, 4.4), ncols=3, sharey=True)
plt.subplots_adjust(wspace=0.10)
steps = ['origin', 'rct', 'rot']
titles = ['original', 'after recentering', 'after rotation']
for axi, step, title in zip(ax, steps, titles):
    S_step = embedded_points[step]
    S_source = S_step[domain == 'source_domain']
    y_source = y[domain == 'source_domain']
```

(continues on next page)

(continued from previous page)

```
    S_target = S_step[domain == 'target_domain']
    y_target = y[domain == 'target_domain']
    axi.scatter(
        S_source[y_source == '1'][:, 0],
        S_source[y_source == '1'][:, 1],
        c='C0', s=50, alpha=0.50)
    axi.scatter(
        S_source[y_source == '2'][:, 0],
        S_source[y_source == '2'][:, 1],
        c='C1', s=50, alpha=0.50)
    axi.scatter(
        S_target[y_target == '1'][:, 0],
        S_target[y_target == '1'][:, 1],
        c='C0', s=50, alpha=0.30, marker="^")
    axi.scatter(
        S_target[y_target == '2'][:, 0],
        S_target[y_target == '2'][:, 1],
        c='C1', s=50, alpha=0.30, marker="^")
    axi.scatter(S[-1, 0], S[-1, 1], c='k', s=80, marker="*")
    axi.set_xlim(-0.60, +1.60)
    axi.set_ylim(-1.10, +1.25)
    axi.set_xticks([-0.5, 0.0, 0.5, 1.0, 1.5])
    axi.set_yticks([-1.0, -0.5, 0.0, 0.5, 1.0])
    axi.set_title(title, fontsize=14)
ax[0].scatter([], [], c="C0", label="source - class 0")
ax[0].scatter([], [], c="C1", label="source - class 1")
ax[0].scatter([], [], marker="^", c="C0", label="target - class 0")
ax[0].scatter([], [], marker="^", c="C1", label="target - class 1")
ax[0].legend(loc="upper right")

plt.show()
```



References

Total running time of the script: (0 minutes 9.987 seconds)

Motor imagery classification by transfer learning

In this example, we use transfer learning (TL) to classify epochs from a subject using a classifier trained on data from another subject. We consider TL with a pooling strategy: for each target subject of choice, we use the data from several source subjects to train a single classifier using all of their data points pooled together. We compare the results of simply mixing all covariances from all source subjects without any care (dummy) versus transforming the covariances of all subjects so that they are centered around the Identity matrix (recenter)[1]. We use data from the Physionet BCI database and compare the classification performance of MDM with each strategy.

```
import numpy as np
from tqdm import tqdm
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import StratifiedShuffleSplit
import matplotlib.pyplot as plt

from mne import Epochs, pick_types, events_from_annotations
from mne.io import concatenate_raws
from mne.io.edf import read_raw_edf
from mne.datasets import eegbci
from mne import set_log_level

from pyriemann.classification import MDM
from pyriemann.estimation import Covariances
from pyriemann.transfer import (
    encode_domains,
    TLDummy,
    TLCenter,
    TLClassifier,
    TLSplitter,
)

set_log_level(verbose=False)
```

```
def get_subject_dataset(subject):

    # Consider epochs that start 1s after cue onset.
    tmin, tmax = 1., 2.
    event_id = dict(hands=2, feet=3)
    runs = [6, 10, 14]  # motor imagery: hands vs feet

    # Download data with MNE
    raw_files = [
        read_raw_edf(f, preload=True) for f in eegbci.load_data(subject, runs)
    ]
    raw = concatenate_raws(raw_files)
```

(continues on next page)

[1] Transfer Learning: A Riemannian Geometry Framework With Applications to Brain–Computer Interfaces P Zanini et al, IEEE Transactions on Biomedical Engineering, vol. 65, no. 5, pp. 1107-1116, August, 2017

(continued from previous page)

```
    # Select only EEG channels
    picks = pick_types(
        raw.info, meg=False, eeg=True, stim=False, eog=False, exclude='bads')
    # select only nine electrodes: F3, Fz, F4, C3, Cz, C4, P3, Pz, P4
    picks = picks[[31, 33, 35, 8, 10, 12, 48, 50, 52]]

    # Apply band-pass filter
    raw.filter(7., 35., method='iir', picks=picks)

    # Check the events
    events, _ = events_from_annotations(raw, event_id=dict(T1=2, T2=3))

    # Define the epochs
    epochs = Epochs(
        raw,
        events,
        event_id,
        tmin,
        tmax,
        proj=True,
        picks=picks,
        baseline=None,
        preload=True,
        verbose=False)

    # Extract the labels for each event
    labels = epochs.events[:, -1] - 2

    # Compute covariance matrices on scaled data
    covs = Covariances().fit_transform(1e6 * epochs.get_data())

    return covs, labels
```

We will consider subjects from the Physionet EEG database for which the intra-subject classification has been checked to be > 0.70

```
subject_list = [1, 2, 4, 7, 8, 15, 20, 29, 34, 35]

# Load the data from subjects
X, y, d = [], [], []
for i, subject_source in enumerate(subject_list):
    X_source_i, y_source_i = get_subject_dataset(subject=subject_source)
    X.append(X_source_i)
    y.append(y_source_i)
    d = d + [f'subject_{subject_source:02}'] * len(X_source_i)
X = np.concatenate(X)
y = np.concatenate(y)
domains = np.array(d)

# Encode the data for transfer learning purposes
X_enc, y_enc = encode_domains(X, y, domains)
```

(continues on next page)

(continued from previous page)

```
# Object for splitting the datasets into training and validation partitions
n_splits = 5  # How many times to split the target domain into train/test
tl_cv = TLSplitter(
    target_domain='',
    cv=StratifiedShuffleSplit(
        n_splits=n_splits, train_size=0.10, random_state=42))

# We consider two types of pipelines for transfer learning
# dct : no transformation of dataset between the domains
# rct : re-center the data points from each domain to the Identity
scores = {meth: [] for meth in ['dummy', 'rct']}

# Base classifier to be wrapped for transfer learning
clf_base = MDM()

# Consider different subjects as target
for subject_target_idx in tqdm(range(len(subject_list))):

    # Change the target domain
    tl_cv.target_domain = f'subject_{subject_list[subject_target_idx]:02}'

    # Create dict for storing results of this particular CV split
    scores_cv = {meth: [] for meth in scores.keys()}

    # Carry out the cross-validation
    for train_idx, test_idx in tl_cv.split(X_enc, y_enc):

        # Split the dataset into training and testing
        X_enc_train, X_enc_test = X_enc[train_idx], X_enc[test_idx]
        y_enc_train, y_enc_test = y_enc[train_idx], y_enc[test_idx]

        # (1) Dummy pipeline: no transfer learning at all.
        # Classifier is trained only with points from the source domain.
        domain_weight_dummy = {}
        for d in np.unique(domains):
            domain_weight_dummy[d] = 1.0
        domain_weight_dummy[tl_cv.target_domain] = 0.0

        pipeline = make_pipeline(
            TLDummy(),
            TLClassifier(
                target_domain=tl_cv.target_domain,
                estimator=clf_base,
                domain_weight=domain_weight_dummy,
            ),
        )

        # Fit and get accuracy score
        pipeline.fit(X_enc_train, y_enc_train)
        scores_cv['dummy'].append(pipeline.score(X_enc_test, y_enc_test))

        # (2) Recentering pipeline: recenter the data from each domain to
```

(continues on next page)

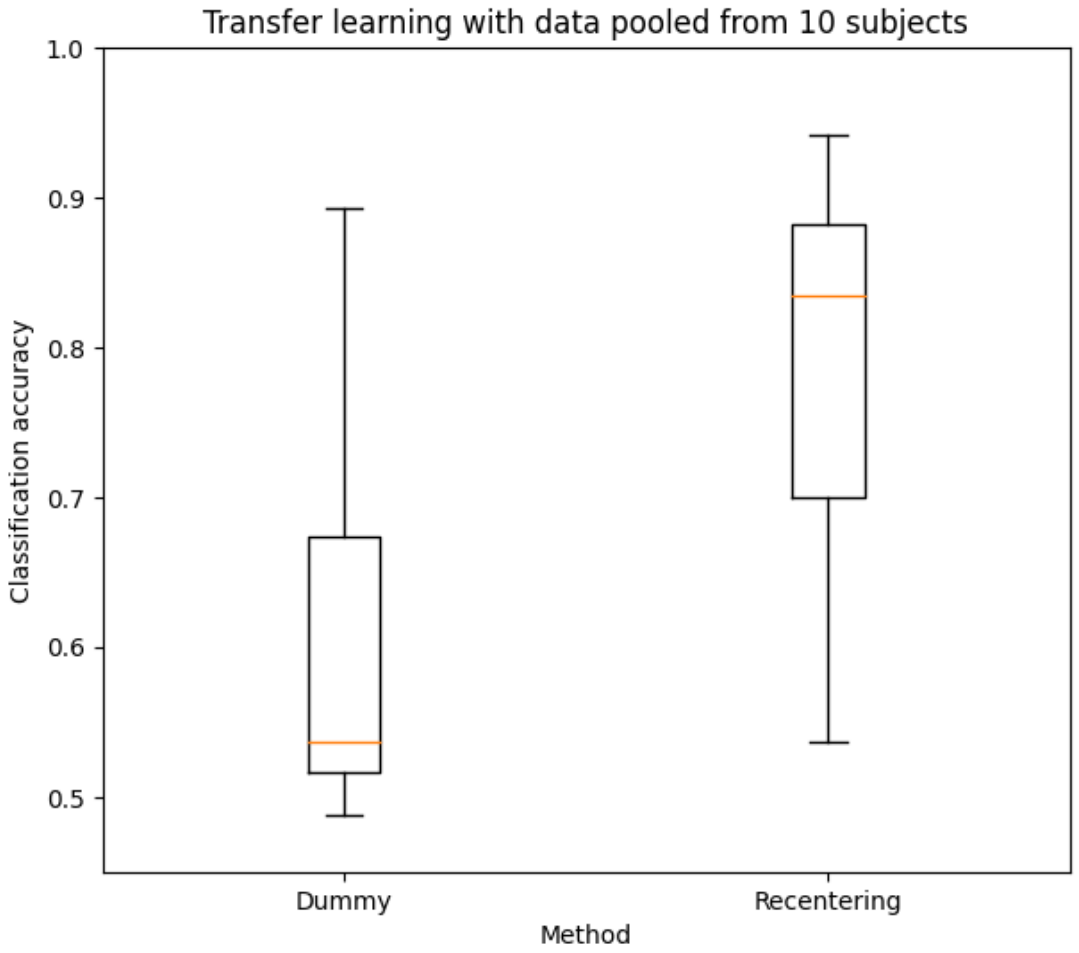
(continued from previous page)

```
        # identity [1]_.
        # Classifier is trained only with points from the source domain.
        domain_weight_rct = {}
        for d in np.unique(domains):
            domain_weight_rct[d] = 1.0
        domain_weight_rct[tl_cv.target_domain] = 0.0

        pipeline = make_pipeline(
            TLCenter(target_domain=tl_cv.target_domain),
            TLClassifier(
                target_domain=tl_cv.target_domain,
                estimator=clf_base,
                domain_weight=domain_weight_rct,
            ),
        )

        pipeline.fit(X_enc_train, y_enc_train)
        scores_cv['rct'].append(pipeline.score(X_enc_test, y_enc_test))

    for meth in scores.keys():
        scores[meth].append(np.mean(scores_cv[meth]))
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
 10%|#         | 1/10 [00:03<00:29,  3.26s/it]
 20%|##        | 2/10 [00:06<00:26,  3.29s/it]
 30%|###       | 3/10 [00:09<00:23,  3.32s/it]
 40%|####      | 4/10 [00:13<00:19,  3.23s/it]
 50%|#####     | 5/10 [00:16<00:16,  3.22s/it]
 60%|######    | 6/10 [00:19<00:13,  3.26s/it]
 70%|#######   | 7/10 [00:22<00:09,  3.27s/it]
 80%|########  | 8/10 [00:26<00:06,  3.28s/it]
 90%|######### | 9/10 [00:29<00:03,  3.30s/it]
100%|##########| 10/10 [00:32<00:00,  3.28s/it]
100%|##########| 10/10 [00:32<00:00,  3.27s/it]
```

Plot results

```
fig, ax = plt.subplots(figsize=(7, 6))
ax.boxplot(x=[scores[meth] for meth in scores.keys()])
ax.set_ylim(0.45, 1.00)
ax.set_xticklabels(['Dummy', 'Recentering'])
ax.set_ylabel('Classification accuracy')
ax.set_xlabel('Method')
ax.set_title('Transfer learning with data pooled from 10 subjects')

plt.show()
```



References

Total running time of the script: (0 minutes 59.180 seconds)

Comparison of pipelines for transfer learning

We compare the classification performance of MDM on five different strategies for transfer learning. These include re-centering the datasets as done in[1], matching the statistical distributions in a semi-supervised way with Riemannian Procrustes Analysis[2], and improving the MDM classifier with a weighting scheme (MDWM)[3]. All data points are simulated from a toy model based on the Riemannian Gaussian distribution and the differences in statistics between source and target distributions are determined by a set of parameters that have control over the distance between the

[1] Transfer Learning: A Riemannian Geometry Framework With Applications to Brain–Computer Interfaces P Zanini et al, IEEE Transactions on Biomedical Engineering, vol. 65, no. 5, pp. 1107-1116, August, 2017

[2] Riemannian Procrustes analysis: transfer learning for brain-computer interfaces PLC Rodrigues et al, IEEE Transactions on Biomedical Engineering, vol. 66, no. 8, pp. 2390-2401, December, 2018

[3] Transfer Learning for SSVEP-based BCI using Riemannian similarities between users E Kalunga et al, 26th European Signal Processing Conference (EUSIPCO 2018) Sep 2018, Rome, Italy, pp.1685-1689

centers of each dataset, the angle of rotation between the means of each class, and the differences in dispersion of the data points from each dataset.

```
from tqdm import tqdm

import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import StratifiedShuffleSplit

from pyriemann.classification import MDM
from pyriemann.datasets.simulated import make_classification_transfer
from pyriemann.transfer import (
    TLSplitter,
    TLDummy,
    TLCenter,
    TLStretch,
    TLRotate,
    TLClassifier,
    MDWM
)
```

Choose seed for reproducible results

```
seed = 100

# We consider several types of pipeline for transfer learning
# dummy : no transformation of dataset between the domains
# rct : re-center the data points from each domain to the Identity
# rpa : re-center, stretch and rotate (Riemannian Procrustes Analysis)
# mdwm : transfer learning with minimum distance to weighted mean
# calibration : use only data from target-train partition for the classifier
scores = {meth: [] for meth in ['dummy', 'rct', 'rpa', 'mdwm', 'calibration']}

# Create a dataset with two domains, each with two classes both datasets
# are generated by the same generative procedure with the SPD Gaussian
# and one of them is transformed by a matrix A, i.e. X <- A @ X @ A.T
X_enc, y_enc = make_classification_transfer(
    n_matrices=100,
    class_sep=0.75,
    class_disp=1.0,
    domain_sep=5.0,
    theta=3*np.pi/5,
    random_state=seed,
)

# Object for splitting the datasets into training and validation partitions
# the training set is composed of all data points from the source domain
# plus a partition of the target domain whose size we can control
target_domain = 'target_domain'
n_splits = 5  # how many times to split the target domain into train/test
tl_cv = TLSplitter(
    target_domain=target_domain,
    cv=StratifiedShuffleSplit(n_splits=n_splits, random_state=seed),
```

(continues on next page)

(continued from previous page)

```
)

# Which base classifier to consider
clf_base = MDM()

# Vary the proportion of the target domain for training
target_train_frac_array = np.linspace(0.01, 0.20, 10)
for target_train_frac in tqdm(target_train_frac_array):

    # Change fraction of the target training partition
    tl_cv.cv.train_size = target_train_frac

    # Create dict for storing results of this particular CV split
    scores_cv = {meth: [] for meth in scores.keys()}

    # Carry out the cross-validation
    for train_idx, test_idx in tl_cv.split(X_enc, y_enc):

        # Split the dataset into training and testing
        X_enc_train, X_enc_test = X_enc[train_idx], X_enc[test_idx]
        y_enc_train, y_enc_test = y_enc[train_idx], y_enc[test_idx]

        # (1) Dummy pipeline: no transfer learning at all.
        # Classifier is trained only with samples from the source dataset.
        pipeline = make_pipeline(
            TLDummy(),
            TLClassifier(
                target_domain=target_domain,
                estimator=clf_base,
                domain_weight={'source_domain': 1.0, 'target_domain': 0.0},
            ),
        )

        # Fit and get accuracy score
        pipeline.fit(X_enc_train, y_enc_train)
        scores_cv['dummy'].append(pipeline.score(X_enc_test, y_enc_test))

        # (2) RCT pipeline: recenter data from each domain to identity [1]_.
        # Classifier is trained only with points from the source domain.
        pipeline = make_pipeline(
            TLCenter(target_domain=target_domain),
            TLClassifier(
                target_domain=target_domain,
                estimator=clf_base,
                domain_weight={'source_domain': 1.0, 'target_domain': 0.0},
            ),
        )

        pipeline.fit(X_enc_train, y_enc_train)
        scores_cv['rct'].append(pipeline.score(X_enc_test, y_enc_test))

        # (3) RPA pipeline: recenter, stretch, and rotate [2]_.
```

(continues on next page)

(continued from previous page)

```
        # Classifier is trained with points from source and target.
        pipeline = make_pipeline(
            TLCenter(target_domain=target_domain),
            TLStretch(
                target_domain=target_domain,
                final_dispersion=1,
                centered_data=True,
            ),
            TLRotate(target_domain=target_domain, metric='euclid'),
            TLClassifier(
                target_domain=target_domain,
                estimator=clf_base,
                domain_weight={'source_domain': 0.5, 'target_domain': 0.5},
            ),
        )

        pipeline.fit(X_enc_train, y_enc_train)
        scores_cv['rpa'].append(pipeline.score(X_enc_test, y_enc_test))

        # (4) MDWM pipeline
        domain_tradeoff = 1 - np.exp(-25*target_train_frac)
        pipeline = MDWM(domain_tradeoff=domain_tradeoff,
                        target_domain=target_domain)
        pipeline.fit(X_enc_train, y_enc_train)
        scores_cv['mdwm'].append(pipeline.score(X_enc_test, y_enc_test))

        # (5) Calibration: use only data from target-train partition.
        # Classifier is trained only with points from the target domain.
        pipeline = make_pipeline(
            TLClassifier(
                target_domain=target_domain,
                estimator=clf_base,
                domain_weight={'source_domain': 0.0, 'target_domain': 1.0},
            ),
        )

        pipeline.fit(X_enc_train, y_enc_train)
        scores_cv['calibration'].append(pipeline.score(X_enc_test, y_enc_test))

    # Get the average score of each pipeline
    for meth in scores.keys():
        scores[meth].append(np.mean(scores_cv[meth]))

# Store the results for each method on this particular seed
for meth in scores.keys():
    scores[meth] = np.array(scores[meth])
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
 10%|#         | 1/10 [00:01<00:11,  1.31s/it]
 20%|##        | 2/10 [00:05<00:22,  2.79s/it]
 30%|###       | 3/10 [00:10<00:28,  4.04s/it]/home/docs/checkouts/readthedocs.org/user_
↪builds/pyriemann/checkouts/v0.4/pyriemann/transfer/_rotate.py:132: UserWarning:␣
```

(continues on next page)

(continued from previous page)

```
↪Convergence not reached.
  warnings.warn('Convergence not reached.')

 40%|####      | 4/10 [00:19<00:35,  5.90s/it]/home/docs/checkouts/readthedocs.org/user_
↪builds/pyriemann/checkouts/v0.4/pyriemann/transfer/_rotate.py:132: UserWarning:␣
↪Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')

 50%|#####     | 5/10 [00:29<00:37,  7.47s/it]/home/docs/checkouts/readthedocs.org/user_
↪builds/pyriemann/checkouts/v0.4/pyriemann/transfer/_rotate.py:132: UserWarning:␣
↪Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')

 60%|######    | 6/10 [00:39<00:32,  8.23s/it]/home/docs/checkouts/readthedocs.org/user_
↪builds/pyriemann/checkouts/v0.4/pyriemann/transfer/_rotate.py:132: UserWarning:␣
↪Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')

 70%|#######   | 7/10 [00:49<00:26,  8.96s/it]/home/docs/checkouts/readthedocs.org/user_
↪builds/pyriemann/checkouts/v0.4/pyriemann/transfer/_rotate.py:132: UserWarning:␣
↪Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
```

(continues on next page)

(continued from previous page)

```
 80%|########  | 8/10 [01:02<00:19,  9.99s/it]/home/docs/checkouts/readthedocs.org/user_
↪builds/pyriemann/checkouts/v0.4/pyriemann/transfer/_rotate.py:132: UserWarning:␣
↪Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')

 90%|######### | 9/10 [01:13<00:10, 10.40s/it]/home/docs/checkouts/readthedocs.org/user_
↪builds/pyriemann/checkouts/v0.4/pyriemann/transfer/_rotate.py:132: UserWarning:␣
↪Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')
/home/docs/checkouts/readthedocs.org/user_builds/pyriemann/checkouts/v0.4/pyriemann/
↪transfer/_rotate.py:132: UserWarning: Convergence not reached.
  warnings.warn('Convergence not reached.')

100%|##########| 10/10 [01:24<00:00, 10.69s/it]
100%|##########| 10/10 [01:24<00:00,  8.47s/it]
```

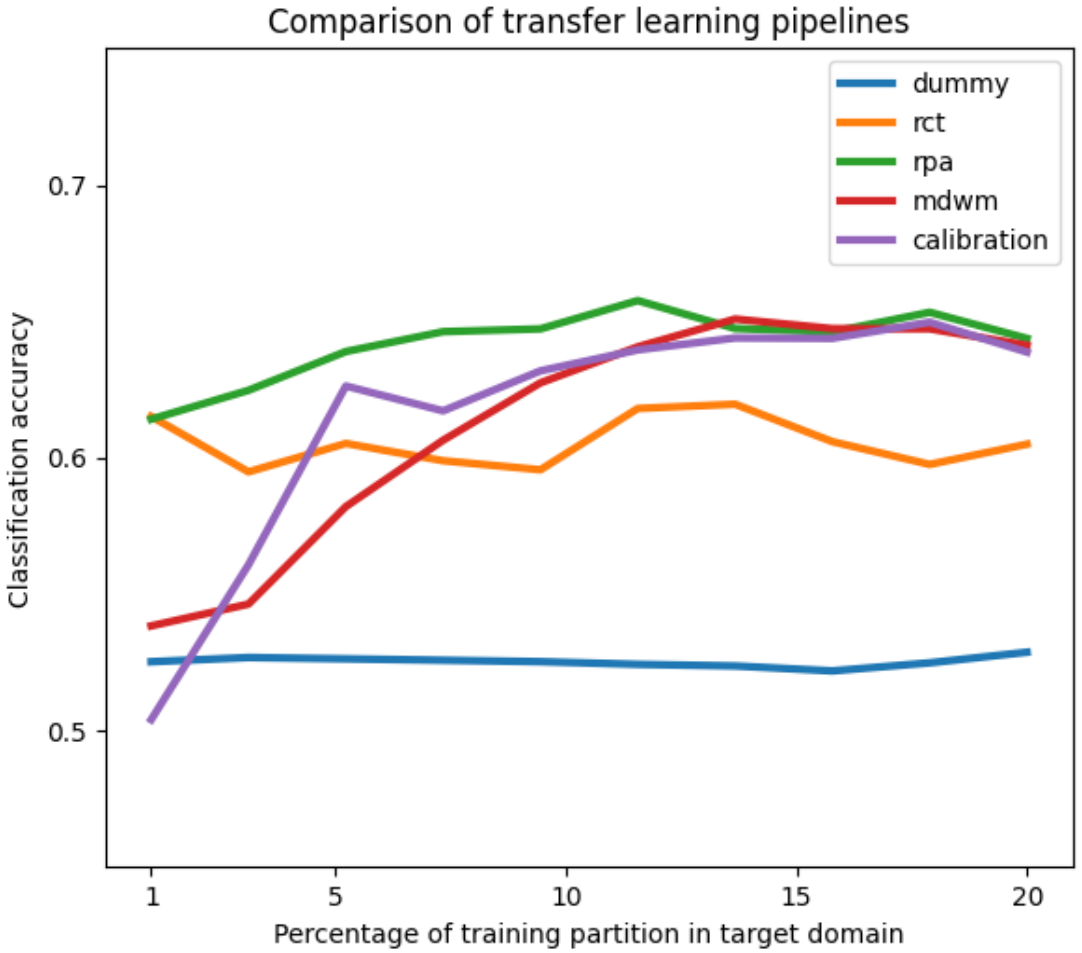
Plot the results, reproducing Figure 2 of[Page 195, 2].

```
fig, ax = plt.subplots(figsize=(6.7, 5.7))
for meth in scores.keys():
    ax.plot(
        target_train_frac_array,
        scores[meth],
        label=meth,
        lw=3.0)
ax.legend(loc='upper right')
ax.set_ylim(0.45, 0.75)
ax.set_yticks([0.5, 0.6, 0.7])
ax.set_xlim(0.00, 0.21)
ax.set_xticks([0.01, 0.05, 0.10, 0.15, 0.20])
ax.set_xticklabels([1, 5, 10, 15, 20])
ax.set_xlabel('Percentage of training partition in target domain')
ax.set_ylabel('Classification accuracy')
```

(continues on next page)

(continued from previous page)

```
ax.set_title('Comparison of transfer learning pipelines')

plt.show()
```



References

Total running time of the script: (1 minutes 25.054 seconds)

CHAPTER

FIVE

API REFERENCE

5.1 SPD Matrices Estimation

Covariances([estimator])	Estimation of covariance matrix.
ERPCovariances([classes, estimator, svd])	Estimate special form covariance matrix for ERP.
XdawnCovariances([nfilter, applyfilters, ...])	Estimate special form covariance matrix for ERP combined with Xdawn.
BlockCovariances(block_size[, estimator])	Estimation of block covariance matrix.
CospCovariances([window, overlap, fmin, ...])	Estimation of cospectral covariance matrix.
Coherences([window, overlap, fmin, fmax, ...])	Estimation of squared coherence matrices.
HankelCovariances([delays, estimator])	Estimation of covariance matrix with time delayed Hankel matrices.
Kernels([metric, n_jobs])	Estimation of kernel matrix between channels of time series.
Shrinkage([shrinkage])	Regularization of SPD matrices by shrinkage.

5.1.1 pyriemann.estimation.Covariances

class pyriemann.estimation.**Covariances**(*estimator='scm'*, ***kwds*)

Estimation of covariance matrix.

Perform a simple covariance matrix estimation for each given input.

Parameters

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

See also:

ERPCovariances
XdawnCovariances
CospCovariances
HankelCovariances

`__init__`(*estimator='scm'*, ***kwds*)

Init.

`fit`(*X*, *y=None*)

Fit.

Do nothing. For compatibility purpose.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[Covariances instance] The Covariances instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate covariance matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Covariance matrices.

5.1.2 pyriemann.estimation.ERPCovariances

`class pyriemann.estimation.ERPCovariances`(*classes=None*, *estimator='scm'*, *svd=None*, ***kwds*)

Estimate special form covariance matrix for ERP.

Estimation of special form covariance matrix dedicated to ERP processing. For each class, a prototyped response is obtained by average across trials:

$$\mathbf{P} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{X}_i$$

and a super trial is built using the concatenation of $\mathbf{P}$ and the trial $\mathbf{X}_i$:

$$\tilde{\mathbf{X}}_i = \begin{bmatrix} \mathbf{P} \\ \mathbf{X}_i \end{bmatrix}$$

This super trial $\tilde{\mathbf{X}}_i$ will be used for covariance estimation. This allows to take into account the spatial structure of the signal, as described in [1].

Parameters

classes
[list of int | None, default=None] List of classes to take into account for prototype estimation. If None, all classes will be accounted.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

svd
[int | None, default=None] If not None, number of components of SVD used to reduce prototype responses.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

See also:

Covariances
XdawnCovariances
CospCovariances
HankelCovariances

References

[1], [2], [3]

Attributes

P_
[ndarray, shape (n_components, n_times)] If fit, prototyped responses for each class, where *n_components* is equal to *n_classes x n_channels* if *svd* is None, and to *n_classes x min(svd, n_channels)* otherwise.

`__init__`(*classes=None*, *estimator='scm'*, *svd=None*, ***kwds*)

Init.

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

Fit.

Estimate the prototyped responses for each class.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

Returns

self
[ERPCovariances instance] The ERPCovariances instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate special form covariance matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

covmats
[ndarray, shape (n_matrices, n_components, n_components)] Covariance matrices for ERP, where the size of matrices *n_components* is equal to *(1 + n_classes) x n_channels* if *svd* is None, and to *n_channels + n_classes x min(svd, n_channels)* otherwise.

5.1.3 pyriemann.estimation.XdawnCovariances

`class` pyriemann.estimation.**XdawnCovariances**(*nfilter=4*, *applyfilters=True*, *classes=None*, *estimator='scm'*, *xdawn_estimator='scm'*, *baseline_cov=None*, ***kwds*)

Estimate special form covariance matrix for ERP combined with Xdawn.

Estimation of special form covariance matrix dedicated to ERP processing combined with *Xdawn* spatial filtering. This is similar to *pyriemann.estimation.ERPCovariances* but data are spatially filtered with *pyriemann.spatialfilters.Xdawn*. A complete description of the method is available in [1].

The advantage of this estimation is to reduce dimensionality of the covariance matrices supervisely.

Parameters

nfilter
[int, default=4] Number of Xdawn filters per class.

applyfilters
[bool, default=True] If set to true, spatial filter are applied to the prototypes and the signals. When set to False, filters are applied only to the ERP prototypes allowing for a better generalization across subject and session at the expense of dimensionality increase. In that case, the estimation is similar to *pyriemann.estimation.ERPCovariances* with *svd=nfilter* but with more compact prototype reduction.

classes
[list of int | None, default=None] list of classes to take into account for prototype estimation. If None, all classes will be accounted.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

xdawn_estimator
[string, default='scm'] Covariance matrix estimator for *Xdawn* spatial filtering. Should be regularized using 'lwf' or 'oas', see *pyriemann.utils.covariance.covariances()*.

baseline_cov
[array, shape (n_channels, n_channels) | None, default=None] Baseline covariance for *Xdawn* spatial filtering, see *pyriemann.spatialfilters.Xdawn*.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

See also:

ERPCovariances
Xdawn

References

[1]

Attributes

P_
[ndarray, shape (n_classes x min(n_channels, n_filters), n_times)] If fit, the evoked response for each event type, concatenated.

__init__(*nfilter=4*, *applyfilters=True*, *classes=None*, *estimator='scm'*, *xdawn_estimator='scm'*, *baseline_cov=None*, ***kwds*)

Init.

fit(*X*, *y*)

Fit.

Estimate spatial filters and prototyped response for each classes.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

Returns

self
[XdawnCovariances instance] The XdawnCovariances instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate Xdawn covariance matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

covmats
[ndarray, shape (n_matrices, n_components, n_components)] Covariance matrices filtered by Xdawn, where n_components is equal to *2 x n_classes x min(n_channels, nfilter)* if *applyfilters* is True, and to *n_channels + n_classes x min(n_channels, nfilter)* otherwise.

5.1.4 pyriemann.estimation.BlockCovariances

class pyriemann.estimation.**BlockCovariances**(*block_size*, *estimator='scm'*, ***kwds*)

Estimation of block covariance matrix.

Perform a block covariance estimation for each given matrix. The resulting matrices are block diagonal matrices.

The blocks on the diagonal are calculated as individual covariance matrices for a subset of channels using the given the estimator. Varying block sized possible by passing a list to allow incorporation of different modalities with different number of channels (e.g. EEG, ECoG, LFP, EMG) with their own respective covariance matrices.

Parameters

block_size
[int | list of int] Sizes of individual blocks given as int for same-size block, or list for varying block sizes.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

See also:

Covariances

Notes

New in version 0.3.

__init__(*block_size*, *estimator='scm'*, ***kwds*)

Init.

fit(*X*, *y=None*)

Fit.

Do nothing. For compatibility purpose.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[BlockCovariances instance] The BlockCovariances instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate block covariance matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Covariance matrices.

5.1.5 pyriemann.estimation.CospCovariances

class pyriemann.estimation.**CospCovariances**(*window=128*, *overlap=0.75*, *fmin=None*, *fmax=None*, *fs=None*)

Estimation of cospectral covariance matrix.

Co-spectral matrices are the real part of complex cross-spectral matrices (see *pyriemann.utils.covariance.cross_spectrum()*), estimated as the spectrum covariance in the frequency domain. This method returns a 4-d array with a cospectral covariance matrix for each input and in each frequency bin of the FFT.

Parameters

window
[int, default=128] The length of the FFT window used for spectral estimation.

overlap
[float, default=0.75] The percentage of overlap between window.

fmin
[float | None, default=None] The minimal frequency to be returned.

fmax
[float | None, default=None] The maximal frequency to be returned.

fs
[float | None, default=None] The sampling frequency of the signal.

See also:

Covariances
HankelCovariances
Coherences

Attributes

freqs_
[ndarray, shape (n_freqs,)] If transformed, the frequencies associated to cospectra. None if `fs` is None.

__init__(*window=128*, *overlap=0.75*, *fmin=None*, *fmax=None*, *fs=None*)

Init.

fit(*X*, *y=None*)

Fit.

Do nothing. For compatibility purpose.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[CospCovariances instance] The CospCovariances instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate the cospectral covariance matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

covmats
[ndarray, shape (n_matrices, n_channels, n_channels, n_freqs)] Covariance matrices for each input and for each frequency bin.

5.1.6 pyriemann.estimation.Coherences

class pyriemann.estimation.**Coherences**(*window=128*, *overlap=0.75*, *fmin=None*, *fmax=None*, *fs=None*, *coh='ordinary'*)

Estimation of squared coherence matrices.

Squared coherence matrices estimation [1]. This method will return a 4-d array with a squared coherence matrix estimation for each input and in each frequency bin of the FFT.

Parameters

window
[int, default=128] The length of the FFT window used for spectral estimation.

overlap
[float, default=0.75] The percentage of overlap between window.

fmin
[float | None, default=None] the minimal frequency to be returned.

fmax
[float | None, default=None] The maximal frequency to be returned.

fs
[float | None, default=None] The sampling frequency of the signal.

coh
[{'ordinary', 'instantaneous', 'lagged', 'imaginary'}, default='ordinary'] The coherence type:

- 'ordinary' for the ordinary coherence, defined in Eq.(22) of [1]; this normalization of cross-spectral matrices captures both in-phase and out-of-phase correlations. However it is inflated by the artificial in-phase (zero-lag) correlation engendered by volume conduction.
- 'instantaneous' for the instantaneous coherence, Eq.(26) of [1], capturing only in-phase correlation.
- 'lagged' for the lagged-coherence, Eq.(28) of [1], capturing only out-of-phase correlation (not defined for DC and Nyquist bins).
- 'imaginary' for the imaginary coherence [2], Eq.(0.16) of [3], capturing out-of-phase correlation but still affected by in-phase correlation.

See also:

Covariances
HankelCovariances
CospCovariances

Notes

New in version 0.3.

References

[1], [2], [3]

Attributes

freqs_
[ndarray, shape (n_freqs,)] If transformed, the frequencies associated to cospectra. None if `fs` is None.

`__init__`(*window=128*, *overlap=0.75*, *fmin=None*, *fmax=None*, *fs=None*, *coh='ordinary'*)

Init.

`fit`(*X*, *y=None*)

Fit.

Do nothing. For compatibility purpose.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[CospCovariances instance] The CospCovariances instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate the squared coherences matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

covmats
[ndarray, shape (n_matrices, n_channels, n_channels, n_freqs)] Squared coherence matrices for each input and for each frequency bin.

5.1.7 pyriemann.estimation.HankelCovariances

`class` pyriemann.estimation.`HankelCovariances`(*delays=4*, *estimator='scm'*, ***kwds*)

Estimation of covariance matrix with time delayed Hankel matrices.

Hankel covariance matrices [1] are useful to catch spectral dynamics of the signal, similarly to the CSSP method [2]. It is done by concatenating time delayed version of the signal before covariance estimation.

Parameters

delays
[int | list of int, default=4] The delays to apply for the Hankel matrices. If *int*, it use a range of delays up to the given value. A list of int can be given.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

See also:

Covariances
ERPCovariances
CospCovariances

References

[1], [2]

__init__(*delays=4*, *estimator='scm'*, ***kwds*)

Init.

fit(*X*, *y=None*)

Fit.

Do nothing. For compatibility purpose.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[HankelCovariances instance] The HankelCovariances instance.

fit_transform(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate the Hankel covariance matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

covmats
[ndarray, shape (n_matrices, n_delays x n_channels, n_delays x n_channels)] Hankel covariance matrices, where n_delays is equal to *delays* when it is a int, and to *1 + len(delays)* when it is a list.

5.1.8 pyriemann.estimation.Kernels

`class pyriemann.estimation.Kernels`(*metric='linear'*, *n_jobs=None*, ***kwds*)

Estimation of kernel matrix between channels of time series.

Perform a kernel matrix estimation for each given time series, evaluating a kernel function between each pair of channels (rather than between pairs of time samples) and allowing to extract nonlinear channel relationship [1].

For an input time series $X \in \mathbb{R}^{c \times t}$, composed of c channels and t time samples, kernel function $\kappa()$ is computed between channels i and j:

$$K_{i,j} = \kappa\left(X[i], X[j]\right)$$

Linear kernel is related to *pyriemann.estimation.Covariances* [1], but this class allows to generalize to nonlinear relationships.

Parameters

metric
[string, default='linear'] The metric to use when computing kernel function between channels [2]: 'linear', 'poly', 'polynomial', 'rbf', 'laplacian', 'cosine'.

n_jobs
[int, default=None] The number of jobs to use for the computation [2]. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the kernel function [2].

See also:

Covariances

Notes

New in version 0.3.1.

References

[1], [2]

__init__(*metric='linear'*, *n_jobs=None*, ***kwds*)
Init.

fit(*X*, *y=None*)
Fit.

Do nothing. For compatibility purpose.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[Kernels instance] The Kernels instance.

fit_transform(*X*, *y=None*, ***fit_params*)
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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Estimate kernel matrices from time series.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

K
[ndarray, shape (n_matrices, n_channels, n_channels)] Kernel matrices.

5.1.9 pyriemann.estimation.Shrinkage

`class` pyriemann.estimation.**Shrinkage**(*shrinkage=0.1*)

Regularization of SPD matrices by shrinkage.

This transformer applies a shrinkage regularization to any SPD matrix. It directly uses the *shrunk_covariance* function from scikit-learn [1], applied on each input.

Parameters

shrinkage
[float, default=0.1] Coefficient in the convex combination used for the computation of the shrunk estimate. Must be between 0 and 1.

Notes

New in version 0.2.5.

References

[1]

__init__(*shrinkage=0.1*)

Init.

fit(*X*, *y=None*)

Fit.

Do nothing. For compatibility purpose.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[Shrinkage instance] The Shrinkage instance.

fit_transform(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Shrink and return the SPD matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of shrunk SPD matrices.

5.2 Embedding

locally_linear_embedding(X, *[, ...])	Perform a Locally Linear Embedding (LLE) of SPD matrices.
barycenter_weights(X, Y, indices[, metric, reg])	Compute Riemannian barycenter weights of X from Y along the first axis.
SpectralEmbedding([n_components, metric, eps])	Spectral embedding of SPD matrices into an Euclidean space.
LocallyLinearEmbedding([n_components, ...])	Locally Linear Embedding (LLE) of SPD matrices.

5.2.1 pyriemann.embedding.locally_linear_embedding

pyriemann.embedding.**locally_linear_embedding**(*X*, ***, *n_components=2*, *n_neighbors=5*, *metric='riemann'*, *reg=0.001*)

Perform a Locally Linear Embedding (LLE) of SPD matrices.

As proposed in [1], Locally Linear Embedding (LLE) is a non-linear, neighborhood-preserving dimensionality reduction algorithm which consists of three main steps. For each point xi,

1. find its k nearest neighbors KNN(xi),
2. calculate the best reconstruction of xi based on its k-nearest neighbors (Eq.9 in [1]),
3. calculate a low-dimensional embedding for all points based on the weights in step 2.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

n_components
[int, default=2] Dimensionality of projected space.

n_neighbors
[int, default=5] Number of neighbors for reconstruction of each point.

metric
[{'riemann', 'logeuclid', 'euclid'}, default: 'riemann'] Metric used for KNN and Kernel estimation.

reg
[float, default=1e-3] Regularization parameter.

Returns

embd
[ndarray, shape (n_matrices, n_components)] Locally linear embedding of matrices in X.

error
[float] Error of the projected embedding.

Notes

New in version 0.3.

References

[1]

5.2.2 pyriemann.embedding.barycenter_weights

pyriemann.embedding.**barycenter_weights**(*X*, *Y*, *indices*, *metric='riemann'*, *reg=0.001*)

Compute Riemannian barycenter weights of X from Y along the first axis.

Estimates the weights to assign to each point in Y[indices] to recover the point X[i] by geodesic interpolation. The barycenter weights sum to 1.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Y
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

indices
[ndarray, shape (n_matrices, n_neighbors)] Indices of the points in Y used to compute the barycenter

metric
[{'riemann', 'logeuclid', 'euclid'}, default='riemann'] Kernel metric.

reg
[float, default=1e-3] Amount of regularization to add for the problem to be well-posed in the case of n_neighbors > n_channels.

Returns

B
[ndarray, shape (n_matrices, n_neighbors)] Interpolation weights.

Notes

New in version 0.3.

5.2.3 pyriemann.embedding.SpectralEmbedding

class pyriemann.embedding.**SpectralEmbedding**(*n_components=2*, *metric='riemann'*, *eps=None*)

Spectral embedding of SPD matrices into an Euclidean space.

It uses Laplacian Eigenmaps [1] to embed SPD matrices into an Euclidean space of smaller dimension. The basic hypothesis is that high-dimensional data lives in a low-dimensional manifold, whose intrinsic geometry can be described via the Laplacian matrix of a graph. The vertices of this graph are the SPD matrices and the weights of the links are determined by the Riemannian distance between each pair of them.

Parameters

n_components
[integer, default=2] The dimension of the projected subspace.

metric
[string | dict, default='riemann'] The type of metric to be used for defining pairwise distance between SPD matrices.

eps
[None | float, default=None] The scaling of the Gaussian kernel. If none is given it will use the square of the median of pairwise distances between points.

References

[1]

__init__(*n_components=2*, *metric='riemann'*, *eps=None*)

Init.

fit(*X*, *y=None*)

Fit the model from data in X.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[object] Returns the instance itself.

`fit_transform`(*X*, *y=None*)

Calculate the coordinates of the embedded points.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

Returns

X_new
[ndarray, shape (n_matrices, n_components)] Coordinates of embedded matrices.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

5.2.4 pyriemann.embedding.LocallyLinearEmbedding

class pyriemann.embedding.**LocallyLinearEmbedding**(*n_components=2*, *n_neighbors=5*, *metric='riemann'*, *reg=0.001*)

Locally Linear Embedding (LLE) of SPD matrices.

As proposed in [1], Locally Linear Embedding (LLE) is a non-linear, neighborhood-preserving dimensionality reduction algorithm which consists of three main steps. For each point x,

1. find its k nearest neighbors KNN(x) and
2. calculate the best reconstruction of x based on its KNN.
3. Then calculate a low-dimensional embedding for all points based on the weights in step 2.

This implementation using SPD matrices is based on [2].

Parameters

n_components
[int, default=2] Dimensionality of projected space.

n_neighbors
[int, default=5] Number of neighbors for reconstruction of each point.

metric
[{'riemann', 'logeuclid', 'euclid'}, default: 'riemann'] Metric used for KNN and Kernel estimation.

reg
[float, default=1e-3] Regularization parameter.

Notes

New in version 0.3.

References

[1], [2]

Attributes

embedding_
[ndarray, shape (n_matrices, n_components)] Stores the embedding vectors.

error_
[float] Reconstruction error associated with *embedding_*.

data_
[ndarray, shape (n_matrices, n_channels, n_channels)] Training data.

`__init__`(*n_components=2*, *n_neighbors=5*, *metric='riemann'*, *reg=0.001*)

`fit`(*X*, *y=None*)
Fit the model from data in X.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

Returns

self
[object] Returns the instance itself.

`fit_transform`(*X*, *y=None*)
Calculate the coordinates of the embedded points.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

Returns

X_new
[ndarray, shape (n_matrices, n_components)] Coordinates of embedded matrices.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*, *y=None*)

Calculate embedding coordinates.

Calculate embedding coordinates for new data points based on fitted points.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

Returns

X_new
[ndarray, shape (n_matrices, n_components)] Coordinates of embedded matrices.

5.3 Classification

MDM([metric, n_jobs])	Classification by Minimum Distance to Mean.
FgMDM([metric, tsupdate, n_jobs])	Classification by Minimum Distance to Mean with geodesic filtering.
TSclassifier([metric, tsupdate, clf])	Classification in the tangent space.
KNearestNeighbor([n_neighbors, metric, n_jobs])	Classification by k-nearest neighbors.
SVC(*[, metric, kernel_fct, Cref, C, ...])	Classification by support-vector machine.
MeanField([power_list, method_label, ...])	Classification by Minimum Distance to Mean Field.

5.3.1 pyriemann.classification.MDM

class pyriemann.classification.**MDM**(*metric='riemann', n_jobs=1*)

Classification by Minimum Distance to Mean.

Classification by nearest centroid. For each of the given classes, a centroid is estimated according to the chosen metric. Then, for each new point, the class is affected according to the nearest centroid.

Parameters

metric
[string | dict, default='riemann'] The type of metric used for centroid and distance estimation. see *mean_covariance* for the list of supported metric. the metric could be a dict with two keys, *mean* and *distance* in order to pass different metrics for the centroid estimation and the distance estimation. Typical usecase is to pass 'logeuclid' metric for the mean in order to boost the computional speed and 'riemann' for the distance in order to keep the good sensitivity for the classification.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the class centroid in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

See also:

`Kmeans`
FgMDM
KNearestNeighbor

References

[1], [2]

Attributes

classes_
[ndarray, shape (n_classes,)] Labels for each class.

covmeans_
[list of `n_classes` ndarrays of shape (n_channels, n_channels)] Centroids for each class.

__init__(*metric='riemann', n_jobs=1*)

Init.

`fit`(*X*, *y*, *sample_weight=None*)

Fit (estimates) the centroids.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

self
[MDM instance] The MDM instance.

`fit_predict`(*X*, *y*)

Fit and predict in one function.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of int, shape (n_matrices,)] Predictions for each matrix according to the closest centroid.

`predict_proba`(*X*)

Predict proba using softmax.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

prob
[ndarray, shape (n_matrices, n_classes)] Probabilities for each class.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Get the distance to each centroid.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

dist
[ndarray, shape (n_matrices, n_classes)] The distance to each centroid according to the metric.

5.3.2 pyriemann.classification.FgMDM

class pyriemann.classification.**FgMDM**(*metric='riemann'*, *tsupdate=False*, *n_jobs=1*)

Classification by Minimum Distance to Mean with geodesic filtering.

Apply geodesic filtering described in [1], and classify using MDM. The geodesic filtering is achieved in tangent space with a Linear Discriminant Analysis, then data are projected back to the manifold and classifier with a regular MDM. This is basically a pipeline of FGDA and MDM.

Parameters

metric
[string | dict, default='riemann'] The type of metric used for reference matrix estimation (see *mean_covariance* for the list of supported metric), for distance estimation, and for tangent space map (see *tangent_space* for the list of supported metric). The metric could be a dict with three keys, *mean*, *dist* and *map* in order to pass different metrics for the reference matrix estimation, the distance estimation, and the tangent space mapping.

tsupdate
[bool, default=False] Activate tangent space update for covariante shift correction between training and test, as described in [2]. This is not compatible with online implementation. Performance are better when the number of matrices for prediction is higher.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the class centroid in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

See also:

MDM
FGDA
TangentSpace

References

[1], [2]

Attributes

classes_
[ndarray, shape (n_classes,)] Labels for each class.

__init__(*metric='riemann'*, *tsupdate=False*, *n_jobs=1*)

Init.

`fit`(*X*, *y*, *sample_weight=None*)

Fit FgMDM.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

self
[FgMDM instance] The FgMDM instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions after FGDA filtering.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of int, shape (n_matrices,)] Predictions for each matrix according to the closest centroid.

`predict_proba`(*X*)

Predict proba using softmax after FGDA filtering.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

prob
[ndarray, shape (n_matrices, n_classes)] The softmax probabilities for each class.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Get the distance to each centroid after FGDA filtering.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

dist
[ndarray, shape (n_matrices, n_cluster)] The distance to each centroid according to the metric.

5.3.3 pyriemann.classification.TSclassifier

class pyriemann.classification.**TSclassifier**(*metric='riemann'*, *tsupdate=False*, *clf=LogisticRegression()*)

Classification in the tangent space.

Project data in the tangent space and apply a classifier on the projected data. This is a simple helper to pipeline the tangent space projection and a classifier. Default classifier is LogisticRegression

Parameters

metric
[string | dict, default='riemann'] The type of metric used for reference matrix estimation (see *mean_covariance* for the list of supported metric) and for tangent space map (see *tangent_space* for the list of supported metric). The metric could be a dict with two keys, *mean* and *map* in order to pass different metrics for the reference matrix estimation and the tangent space mapping.

tsupdate
[bool, default=False] Activate tangent space update for covariate shift correction between training and test, as described in [2]. This is not compatible with online implementation. Performance are better when the number of matrices for prediction is higher.

clf
[sklearn classifier, default=LogisticRegression()] The classifier to apply in the tangent space.

See also:

`TangentSpace`

Notes

New in version 0.2.4.

Attributes

classes_
[ndarray, shape (n_classes,)] Labels for each class.

__init__(*metric='riemann'*, *tsupdate=False*, *clf=LogisticRegression()*)

Init.

fit(*X*, *y*, *sample_weight=None*)

Fit TSclassifier.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

self
[TSclassifier instance] The TSclassifier instance.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of int, shape (n_matrices,)] Predictions for each matrix according to the closest centroid.

`predict_proba`(*X*)

Get the probability.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of ifloat, shape (n_matrices, n_classes)] Predictions for each matrix according to the closest centroid.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form <`component`>__<`parameter`> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

5.3.4 pyriemann.classification.KNearestNeighbor

`class` pyriemann.classification.`KNearestNeighbor`(*n_neighbors=5*, *metric='riemann'*, *n_jobs=1*)

Classification by k-nearest neighbors.

Classification by k-nearest neighbors (k-NN). For each point of the test set, the pairwise distance to each element of the training set is estimated. The class is affected according to the majority class of the k-nearest neighbors.

Parameters

n_neighbors
[int, default=5] Number of neighbors.

metric
[string | dict, default='riemann'] The type of metric used for distance estimation. see *distance* for the list of supported metric.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the distance to the training set in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

See also:

`Kmeans`
MDM

Attributes

classes_
[ndarray, shape (n_classes,)] Labels for each class.

covmeans_
[ndarray, shape (n_matrices, n_channels, n_channels)] Matrices of training set.

classmeans_
[ndarray, shape (n_matrices,)] Labels of training set.

`__init__`(*n_neighbors=5*, *metric='riemann'*, *n_jobs=1*)

Init.

`fit`(*X*, *y*, *sample_weight=None*)

Fit (store the training data).

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

sample_weight
[None] Not used, here for compatibility with sklearn API.

Returns

self
[NearestNeighbor instance] The NearestNeighbor instance.

`fit_predict`(*X*, *y*)

Fit and predict in one function.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*covtest*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of int, shape (n_matrices,)] Predictions for each matrix according to the closest centroid.

`predict_proba`(*X*)

Predict proba using softmax.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

prob
[ndarray, shape (n_matrices, n_classes)] Probabilities for each class.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Get the distance to each centroid.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

dist
[ndarray, shape (n_matrices, n_classes)] The distance to each centroid according to the metric.

5.3.5 pyriemann.classification.SVC

`class pyriemann.classification.SVC`(*, *metric='riemann'*, *kernel_fct=None*, *Cref=None*, *C=1.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)

Classification by support-vector machine.

Support-vector machine (SVM) classification with precomputed Riemannian kernel matrix according to different metrics as described in [1].

Parameters

metric
[{'riemann', 'euclid', 'logeuclid'}, default='riemann'] Metric for kernel matrix computation.

Cref
[None | callable | ndarray, shape (n_channels, n_channels)] Reference point for kernel matrix computation. If None, the mean of the training data according to the metric is used. If callable, the function is called on the training data to calculate Cref.

kernel_fct
[None | 'precomputed' | callable] If 'precomputed', the kernel matrix for datasets X and Y is estimated according to *pyriemann.utils.kernel(X, Y, Cref, metric)*. If callable, the callable is passed as the kernel parameter to *sklearn.svm.SVC()* [2]. The callable has to be of the form *kernel(X, Y, Cref, metric)*.

C
[float, default=1.0] Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

shrinking
[bool, default=True] Whether to use the shrinking heuristic.

probability
[bool, default=False] Whether to enable probability estimates. This must be enabled prior to calling *fit*, will slow down that method as it internally uses 5-fold cross-validation, and *predict_proba* may be inconsistent with *predict*.

tol
[float, default=1e-3] Tolerance for stopping criterion.

cache_size
[float, default=200] Specify the size of the kernel cache (in MB).

class_weight
[dict or 'balanced', default=None] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_matrices / (n_classes * np.bincount(y))`.

verbose
[bool, default=False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter
[int, default=-1] Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape
[{'ovo', 'ovr'}, default='ovr'] Whether to return a one-vs-rest ('ovr') decision function of shape (n_matrices, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_matrices, n_classes * (n_classes - 1) / 2). However, note that internally, one-vs-one ('ovo') is always used as a multi-class strategy to train models; an ovr matrix is only constructed from the ovo matrix. The parameter is ignored for binary classification.

break_ties
[bool, default=False] If true, `decision_function_shape='ovr'`, and number of classes > 2, *predict* will break ties according to the confidence values of *decision_function*; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict.

random_state
[int, RandomState instance or None, default=None] Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when *probability* is False. Pass an int for reproducible output across multiple function calls.

Notes

New in version 0.3.

References

[1], [2]

`__init__`(*, *metric='riemann'*, *kernel_fct=None*, *Cref=None*, *C=1.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)

Init.

`property coef_`

Weights assigned to the features when *kernel="linear"*.

Returns

ndarray of shape (n_features, n_classes)

`decision_function`(*X*)

Evaluate the decision function for the samples in X.

Parameters

X
[array-like of shape (n_samples, n_features)] The input samples.

Returns

X
[ndarray of shape (n_samples, n_classes * (n_classes-1) / 2)] Returns the decision function of the sample for each class in the model. If decision_function_shape='ovr', the shape is (n_samples, n_classes).

Notes

If decision_function_shape='ovo', the function values are proportional to the distance of the samples X to the separating hyperplane. If the exact distances are required, divide the function values by the norm of the weight vector (`coef_`). See also this question for further details. If decision_function_shape='ovr', the decision function is a monotonic transformation of ovo decision function.

`fit`(*X*, *y*, *sample_weight=None*)

Fit.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. Rescale C per matrix. Higher weights force the classifier to put more emphasis on these matrices. If None, it uses equal weights.

Returns

self
[SVC instance] The SVC instance.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`property n_support_`

Number of support vectors for each class.

`predict`(*X*)

Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.

Parameters

X
[{array-like, sparse matrix} of shape (n_samples, n_features) or (n_samples_test, n_samples_train)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

y_pred
[ndarray of shape (n_samples,)] Class labels for samples in X.

`predict_log_proba`(*X*)

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute *probability* set to True.

Parameters

X
[array-like of shape (n_samples, n_features) or (n_samples_test, n_samples_train)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

T
[ndarray of shape (n_samples, n_classes)] Returns the log-probabilities of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute classes_.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

`predict_proba`(*X*)

Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute *probability* set to True.

Parameters

X
[array-like of shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

T
[ndarray of shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute classes_.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

`property probA_`

Parameter learned in Platt scaling when *probability=True*.

Returns

ndarray of shape (n_classes * (n_classes - 1) / 2)

`property probB_`

Parameter learned in Platt scaling when *probability=True*.

Returns

ndarray of shape (n_classes * (n_classes - 1) / 2)

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

5.3.6 pyriemann.classification.MeanField

class pyriemann.classification.**MeanField**(*power_list=[-1, 0, 1]*, *method_label='sum_means'*, *metric='riemann'*, *n_jobs=1*)

Classification by Minimum Distance to Mean Field.

Classification by Minimum Distance to Mean Field [1], defining several power means for each class.

Parameters

power_list
[list of float, default=[-1,0,+1]] Exponents of power means.

method_label
[{'sum_means', 'inf_means'}, default='sum_means'] Method to combine labels:

- sum_means: it assigns the covariance to the class whom the sum of distances to means of the field is the lowest;
- inf_means: it assigns the covariance to the class of the closest mean of the field.

metric
[string | dict, default='riemann'] The type of metric used for distance estimation during prediction. See *distance* for the list of supported metric.

See also:

MDM

Notes

New in version 0.3.

References

[1]

Attributes

classes_
[ndarray, shape (n_classes,)] Labels for each class.

covmeans_
[dict of `n_powers` lists of `n_classes` ndarrays of shape (n_channels, n_channels)] Centroids for each power and each class.

__init__(*power_list=[-1, 0, 1]*, *method_label='sum_means'*, *metric='riemann'*, *n_jobs=1*)

Init.

fit(*X*, *y*, *sample_weight=None*)

Fit (estimates) the centroids.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

sample_weight
[None | ndarray shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

self
[MeanField instance] The MeanField instance.

`fit_predict`(*X*, *y*)

Fit and predict in one function.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of int, shape (n_matrices,)] Predictions for each matrix according to the closest means field.

`predict_proba`(*X*)

Predict proba using softmax.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

prob
[ndarray, shape (n_matrices, n_classes)] Probabilities for each class.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Get the distance to each means field.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

dist
[ndarray, shape (n_matrices, n_classes)] Distance to each means field according to the metric.

class_distinctiveness(X, y[, exponent, ...])	Measure class distinctiveness between classes of SPD matrices.

5.3.7 pyriemann.classification.class_distinctiveness

pyriemann.classification.**class_distinctiveness**(*X*, *y*, *exponent=1*, *metric='riemann'*, *return_num_denom=False*)

Measure class distinctiveness between classes of SPD matrices.

For two class problem, the class distinctiveness between class A and B on the manifold of SPD matrices is quantified as [1]:

$$\text{classDis}(A, B, p) = \frac{d\left(\bar{X}^A, \bar{X}^B\right)^p}{\frac{1}{2}\left(\sigma^p_{X^A} + \sigma^p_{X^B}\right)}$$

where $\bar{X}^K$ is the center of class K, ie the mean of matrices from class K (see pyriemann.utils.mean.mean_covariance()) and σ_{X^K} is the class dispersion, ie the mean of distances between matrices from class K and their center of class $\bar{X}^K$:

$$\sigma^p_{X^K} = \frac{1}{m}\sum_{i=1}^{m} d\left(X_i, \bar{X}^K\right)^p$$

For more than two classes, it is quantified as:

$$\text{classDis}\left(\{K_j\}, p\right) = \frac{\sum_{j=1}^{c} d\left(\bar{X}^{K_j}, \tilde{X}\right)^p}{\sum_{j=1}^{c} \sigma^p_{X^{K_j}}}$$

where $\tilde{X}$ is the mean of centers of class of all c classes and p is the exponentiation of the distance measure named exponent at the input of this function.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

exponent
[int, default=1] Parameter for exponentiation of distances, corresponding to p in the above equations:

- exponent = 1 gives the formula originally defined in [1];
- exponent = 2 gives the Fisher criterion generalized on the manifold, ie the ratio of the variance between the classes to the variance within the classes.

metric
[string | dict, default='riemann'] The type of metric used for centroid and distance estimation. See *mean_covariance* for the list of supported metric. The metric could be a dict with two keys, *mean* and *distance* in order to pass different metrics for the centroid estimation and the distance estimation. The original equation of class distinctiveness in [1] uses 'riemann' for both the centroid estimation and the distance estimation but you can customize other metrics with your interests.

return_num_denom
[bool, default=False] Whether to return numerator and denominator of class_dis.

Returns

class_dis
[float] Class distinctiveness value.

num
[float] Numerator value of class_dis. Returned only if return_num_denom is True.

denom
[float] Denominator value of class_dis. Returned only if return_num_denom is True.

Notes

New in version 0.3.1.

References

[1]

5.4 Regression

KNearestNeighborRegressor([n_neighbors, metric])	Regression by k-nearest-neighbors.
SVR(*[, metric, kernel_fct, Cref, tol, C, ...])	Regression by support-vector machine.

5.4.1 pyriemann.regression.KNearestNeighborRegressor

class pyriemann.regression.**KNearestNeighborRegressor**(*n_neighbors=5*, *metric='riemann'*)

Regression by k-nearest-neighbors.

Regression by k-nearest neighbors (k-NN). For each point of the test set, the pairwise distance to each element of the training set is estimated. The value is calculated according to the softmax average w.r.t. distance of the k-nearest neighbors.

DISCLAIMER: This is an unpublished algorithm.

Parameters

n_neighbors
[int, default=5] Number of neighbors.

metric
[string | dict, default='riemann'] The type of metric used for distance estimation. See *distance* for the list of supported metric.

Notes

New in version 0.3.

Attributes

values_
[ndarray, shape (n_matrices,)] Training target values.

covmeans_
[ndarray, shape (n_matrices, n_channels, n_channels)] Training set of SPD matrices.

`__init__`(*n_neighbors=5*, *metric='riemann'*)

Init.

`fit`(*X*, *y*, *sample_weight=None*)

Fit (store the training data).

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Target values for each matrix.

sample_weight
[None] Not used, here for compatibility with sklearn API.

Returns

self
[KNearestNeighborRegressor instance] The KNearestNeighborRegressor instance.

`fit_predict`(*X*, *y*)

Fit and predict in one function.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray, shape (n_matrices,)] Predictions for each matrix according to the closest neighbors.

`predict_proba`(*X*)

Predict proba using softmax.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

prob
[ndarray, shape (n_matrices, n_classes)] Probabilities for each class.

`score`(*X*, *y*)

Return the coefficient of determination of the prediction.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Test set of SPD matrices.

y
[ndarray, shape (n_matrices,)] True values for each matrix.

Returns

score
[float] R2 of self.predict(X) wrt. y.

Notes

New in version 0.3.1.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Get the distance to each centroid.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

dist
[ndarray, shape (n_matrices, n_classes)] The distance to each centroid according to the metric.

5.4.2 pyriemann.regression.SVR

`class` pyriemann.regression.`SVR`(**, metric='riemann', kernel_fct=None, Cref=None, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=-1*)

Regression by support-vector machine.

Support-vector machine (SVM) regression with precomputed Riemannian kernel matrix according to different metrics, extending the idea described in [1] to regression.

Parameters

metric
[{'riemann', 'euclid', 'logeuclid'}, default='riemann'] Metric for kernel matrix computation.

Cref
[None | ndarray, shape (n_channels, n_channels)] Reference point for kernel matrix computation. If None, the mean of the training data according to the metric is used.

kernel_fct
['precomputed' | callable] If 'precomputed', the kernel matrix for datasets X and Y is estimated according to *pyriemann.utils.kernel(X, Y, Cref, metric)*. If callable, the callable is passed as the kernel parameter to *sklearn.svm.SVC()* [2]. The callable has to be of the form *kernel(X, Y, Cref, metric)*.

tol
[float, default=1e-3] Tolerance for stopping criterion.

C
[float, default=1.0] Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

epsilon
[float, default=0.1] Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

shrinking
[bool, default=True] Whether to use the shrinking heuristic.

cache_size
[float, default=200] Specify the size of the kernel cache (in MB).

verbose
[bool, default=False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter
[int, default=-1] Hard limit on iterations within solver, or -1 for no limit.

Notes

New in version 0.3.

References

[1], [2]

Attributes

data_
[ndarray, shape (n_matrices, n_channels, n_channels)] If fitted, training data.

`__init__`(*, *metric='riemann'*, *kernel_fct=None*, *Cref=None*, *tol=0.001*, *C=1.0*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

Init.

`property coef_`

Weights assigned to the features when *kernel="linear"*.

Returns

ndarray of shape (n_features, n_classes)

`fit`(*X*, *y*, *sample_weight=None*)

Fit.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Target values for each matrix.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. Rescale C per matrix. Higher weights force the classifier to put more emphasis on these matrices. If None, it uses equal weights.

Returns

self
[SVR instance] The SVR instance.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`property n_support_`

Number of support vectors for each class.

`predict`(*X*)

Perform regression on samples in X.

For an one-class model, +1 (inlier) or -1 (outlier) is returned.

Parameters

X
[{array-like, sparse matrix} of shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

y_pred
[ndarray of shape (n_samples,)] The predicted values.

`score`(*X*, *y*, *sample_weight=None*)

Return the coefficient of determination of the prediction.

The coefficient of determination R^2 is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] R^2 of `self.predict(X)` wrt. y.

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

5.5 Clustering

Kmeans([n_clusters, max_iter, metric, ...])	Clustering by k-means with SPD matrices as inputs.
KmeansPerClassTransform([n_clusters])	Clustering by k-means for each class with SPD matrices as inputs.
Potato([metric, threshold, n_iter_max, ...])	Artefact detection with the Riemannian Potato.
PotatoField([n_potatoes, p_threshold, ...])	Artefact detection with the Riemannian Potato Field.

5.5.1 pyriemann.clustering.Kmeans

class pyriemann.clustering.**Kmeans**(*n_clusters=2*, *max_iter=100*, *metric='riemann'*, *random_state=None*, *init='random'*, *n_init=10*, *n_jobs=1*, *tol=0.0001*)

Clustering by k-means with SPD matrices as inputs.

Find clusters that minimize the sum of squared distance to their centroids. This is a direct implementation of the k-means algorithm with a Riemannian metric.

Parameters

n_cluster
[int, default=2] Number of clusters.

max_iter
[int, default=100] The maximum number of iteration to reach convergence.

metric
[string, default='riemann'] The type of metric used for centroid and distance estimation.

random_state
[integer or np.RandomState, optional] The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

init
['random' or ndarray, shape (n_clusters, n_channels, n_channels), default='random'] Method for initialization of centers. 'random': choose k observations (rows) at random from data for the initial centroids. If an ndarray is passed, it should be of shape (n_clusters, n_channels, n_channels) and gives the initial centers.

n_init
[int, default=10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

tol
[float, default=1e-4] The stopping criterion to stop convergence, representing the minimum amount of change in labels between two iterations.

See also:

Kmeans
MDM

Notes

New in version 0.2.2.

Attributes

mdm_
[MDM instance.] MDM instance containing the centroids.

labels_
Labels of each point.

inertia_
[float] Sum of distances of samples to their closest cluster center.

__init__(*n_clusters=2*, *max_iter=100*, *metric='riemann'*, *random_state=None*, *init='random'*, *n_init=10*, *n_jobs=1*, *tol=0.0001*)

Init.

centroids()

Helper for fast access to the centroid.

Returns

centroids
[list of SPD matrices, len (n_cluster)] Return a list containing the centroid of each cluster.

`fit`(*X*, *y=None*)

Fit (estimates) the clusters.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,) | None, default=None] Not used, here for compatibility with sklearn API.

Returns

self
[Kmeans instance] The Kmeans instance.

`fit_predict`(*X*, *y=None*)

Perform clustering on *X* and returns cluster labels.

Parameters

X
[array-like of shape (n_samples, n_features)] Input data.

y
[Ignored] Not used, present for API consistency by convention.

Returns

labels
[ndarray of shape (n_samples,), dtype=np.int64] Cluster labels.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of int, shape (n_matrices,)] Prediction for each matrix according to the closest centroid.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Get the distance to each centroid.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

dist
[ndarray, shape (n_matrices, n_cluster)] The distance to each centroid according to the metric.

5.5.2 pyriemann.clustering.KmeansPerClassTransform

class pyriemann.clustering.**KmeansPerClassTransform**(*n_clusters=2*, ***params*)

Clustering by k-means for each class with SPD matrices as inputs.

See also:

Kmeans

__init__(*n_clusters=2*, ***params*)

Init.

fit(*X*, *y*)

Fit.

fit_transform(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform.

5.5.3 pyriemann.clustering.Potato

`class pyriemann.clustering.Potato`(*metric='riemann'*, *threshold=3*, *n_iter_max=100*, *pos_label=1*, *neg_label=0*)

Artefact detection with the Riemannian Potato.

The Riemannian Potato [1] is a clustering method used to detect artifact in EEG signals. The algorithm iteratively estimates the centroid of clean signal by rejecting every trial that is too far from it.

Parameters

metric
[string, default='riemann'] The type of metric used for centroid and distance estimation.

threshold
[float, default=3] Threshold on z-score of distance to reject artifacts. It is the number of standard deviations from the mean of distances to the centroid.

n_iter_max
[int, default=100] The maximum number of iteration to reach convergence.

pos_label
[int, default=1] The positive label corresponding to clean data.

neg_label
[int, default=0] The negative label corresponding to artifact data.

See also:

Kmeans
MDM

Notes

New in version 0.2.3.

References

[1], [2]

Attributes

covmean_
[ndarray, shape (n_channels, n_channels)] Centroid of potato.

`__init__`(*metric='riemann'*, *threshold=3*, *n_iter_max=100*, *pos_label=1*, *neg_label=0*)

Init.

`fit`(*X*, *y=None*)

Fit the potato from covariance matrices.

Fit the potato from covariance matrices, with an iterative outlier removal to obtain a reliable potato.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None | ndarray, shape (n_matrices,), default=None] Labels corresponding to each matrix: positive (resp. negative) label corresponds to a clean (resp. artifact) matrix. If None, all matrices are considered as clean.

Returns

self
[Potato instance] The Potato instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`partial_fit`(*X*, *y=None*, *alpha=0.1*)

Partially fit the potato from covariance matrices.

This partial fit can be used to update dynamic or semi-dymanic online potatoes with clean EEG.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None | ndarray, shape (n_matrices,), default=None] Labels corresponding to each matrix: positive (resp. negative) label corresponds to a clean (resp. artifact) matrix. If None, all matrices are considered as clean.

alpha
[float, default=0.1] Update rate in [0, 1] for the centroid, and mean and standard deviation of log-distances: 0 for no update, 1 for full update.

Returns

self
[Potato instance] The Potato instance.

`predict`(*X*)

Predict artefact from data.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of bool, shape (n_matrices,)] The artefact detection: True if the matrix is clean, and False if the matrix contain an artefact.

`predict_proba`(*X*)

Return probability of belonging to the potato / being clean.

It is the probability to reject the null hypothesis "clean data", computing the right-tailed probability from z-score.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

proba
[ndarray, shape (n_matrices,)] Matrix is considered as normal/clean for high value of proba. It is considered as abnormal/artifacted for low value of proba.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return the normalized log-distance to the centroid (z-score).

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

z
[ndarray, shape (n_matrices,)] the normalized log-distance to the centroid.

5.5.4 pyriemann.clustering.PotatoField

class pyriemann.clustering.**PotatoField**(*n_potatoes=1*, *p_threshold=0.01*, *z_threshold=3*, *metric='riemann'*, *n_iter_max=10*, *pos_label=1*, *neg_label=0*)

Artefact detection with the Riemannian Potato Field.

The Riemannian Potato Field [1] is a clustering method used to detect artifact in EEG signals. The algorithm combines several potatoes of low dimension, each one being designed to capture specific artifact typically affecting specific subsets of channels and/or specific frequency bands.

Parameters

n_potatoes
[int, default=1] Number of potatoes in the field.

p_threshold
[float, default=0.01] Threshold on probability to being clean, in (0, 1), combining probabilities of potatoes using Fisher's method.

z_threshold
[float, default=3] Threshold on z-score of distance to reject artifacts. It is the number of standard deviations from the mean of distances to the centroid.

metric
[string, default='riemann'] The type of metric used for centroid and distance estimation.

n_iter_max
[int, default=10] The maximum number of iteration to reach convergence.

pos_label: int, default=1
The positive label corresponding to clean data.

neg_label: int, default=0
The negative label corresponding to artifact data.

See also:

Potato

Notes

New in version 0.3.

References

[1]

__init__(*n_potatoes=1*, *p_threshold=0.01*, *z_threshold=3*, *metric='riemann'*, *n_iter_max=10*, *pos_label=1*, *neg_label=0*)

Init.

fit(*X*, *y=None*)

Fit the potato field from covariance matrices.

Fit the potato field from covariance matrices, with iterative outlier removal to obtain reliable potatoes.

Parameters

X
[list of n_potatoes ndarrays of shape (n_matrices, n_channels, n_channels) with same n_matrices but potentially different n_channels] List of sets of SPD matrices, each corresponding to a different subset of EEG channels and/or filtering with a specific frequency band.

y
[None | ndarray, shape (n_matrices,), default=None] Labels corresponding to each matrix: positive (resp. negative) label corresponds to a clean (resp. artifact) matrix. If None, all matrices are considered as clean.

Returns

self
[PotatoField instance] The PotatoField instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`partial_fit`(*X*, *y=None*, *alpha=0.1*)

Partially fit the potato field from covariance matrices.

This partial fit can be used to update dynamic or semi-dymanic online potatoes with clean EEG.

Parameters

X
[list of n_potatoes ndarrays of shape (n_matrices, n_channels, n_channels) with same

n_matrices but potentially different n_channels] List of sets of SPD matrices, each corresponding to a different subset of EEG channels and/or filtering with a specific frequency band.

y
[None | ndarray, shape (n_matrices,), default=None] Labels corresponding to each matrix: positive (resp. negative) label corresponds to a clean (resp. artifact) matrix. If None, all matrices are considered as clean.

alpha
[float, default=0.1] Update rate in [0, 1] for the centroid, and mean and standard deviation of log-distances: 0 for no update, 1 for full update.

Returns

self
[PotatoField instance] The PotatoField instance.

`predict`(*X*)

Predict artefact from data.

Parameters

X
[list of n_potatoes ndarrays of shape (n_matrices, n_channels, n_channels) with same n_matrices but potentially different n_channels] List of sets of SPD matrices, each corresponding to a different subset of EEG channels and/or filtering with a specific frequency band.

Returns

pred
[ndarray of bool, shape (n_matrices,)] The artefact detection: True if the matrix is clean, and False if the matrix contain an artefact.

`predict_proba`(*X*)

Predict probability obtained combining probabilities of potatoes.

Predict probability obtained combining probabilities of potatoes using Fisher's method. A threshold of 0.01 can be used.

Parameters

X
[list of n_potatoes ndarrays of shape (n_matrices, n_channels, n_channels) with same n_matrices but potentially different n_channels] List of sets of SPD matrices, each corresponding to a different subset of EEG channels and/or filtering with a specific frequency band.

Returns

proba
[ndarray, shape (n_matrices,)] Matrix is considered as normal/clean for high value of proba. It is considered as abnormal/artifacted for low value of proba.

`score`(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

X
[array-like of shape (n_samples, n_features)] Test samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights.

Returns

score
[float] Mean accuracy of `self.predict(X)` wrt. *y*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return the normalized log-distances to the centroids.

Return the normalized log-distances to the centroids, ie geometric z-scores of distances.

Parameters

X
[list of n_potatoes ndarrays of shape (n_matrices, n_channels, n_channels) with same n_matrices but potentially different n_channels] List of sets of SPD matrices, each corresponding to a different subset of EEG channels and/or filtering with a specific frequency band.

Returns

z
[ndarray, shape (n_matrices, n_potatoes)] The normalized log-distances to the centroids.

5.6 Tangent Space

TangentSpace([metric, tsupdate])	Tangent space project TransformerMixin.
FGDA([metric, tsupdate])	Fisher Geodesic Discriminant analysis.

5.6.1 pyriemann.tangentspace.TangentSpace

class pyriemann.tangentspace.**TangentSpace**(*metric='riemann'*, *tsupdate=False*)

Tangent space project TransformerMixin.

Tangent space projection map a set of SPD matrices to their tangent space according to [1]. The Tangent space projection can be seen as a kernel operation, cf [2]. After projection, each matrix is represented as a vector of size $n(n+1)/2$, where n is the dimension of the SPD matrices.

Tangent space projection is useful to convert SPD matrices in Euclidean vectors while conserving the inner structure of the manifold. After projection, standard processing and vector-based classification can be applied.

Tangent space projection is a local approximation of the manifold. it takes one parameter, the reference point, that is usually estimated using the geometric mean of the SPD matrices set you project. If the function *fit* is not called, the identity matrix will be used as reference point. This can lead to serious degradation of performances. The approximation will be bigger if the matrices in the set are scattered in the manifold, and lower if they are grouped in a small region of the manifold.

After projection, it is possible to go back in the manifold using the inverse transform.

Parameters

metric
[string | dict, default='riemann'] The type of metric used for reference matrix estimation (see *mean_covariance* for the list of supported metric) and for tangent space map (see *tangent_space* for the list of supported metric). The metric could be a dict with two keys, *mean* and *map* in order to pass different metrics for the reference matrix estimation and the tangent space mapping.

tsupdate
[bool, default=False] Activate tangent space update for covariante shift correction between training and test, as described in [2]. This is not compatible with online implementation. Performance are better when the number of matrices for prediction is higher.

See also:

FgMDM
FGDA

References

[1], [2]

Attributes

reference_
[ndarray] If fit, the reference point for tangent space mapping.

__init__(*metric='riemann'*, *tsupdate=False*)

Init.

fit(*X*, *y=None*, *sample_weight=None*)

Fit (estimates) the reference point.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

self
[TangentSpace instance] The TangentSpace instance.

`fit_transform`(*X*, *y=None*, *sample_weight=None*)

Fit and transform in a single function.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

ts
[ndarray, shape (n_matrices, n_ts)] Tangent space projections of SPD matrices.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*, *y=None*)

Inverse transform.

Project back a set of tangent space vector in the manifold.

Parameters

X
[ndarray, shape (n_matrices, n_ts)] Set of tangent space projections of the matrices.

y
[None] Not used, here for compatibility with sklearn API.

Returns

cov
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices corresponding to each of tangent vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Tangent space projection.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

ts
[ndarray, shape (n_matrices, n_ts)] Tangent space projections of SPD matrices.

5.6.2 pyriemann.tangentspace.FGDA

`class pyriemann.tangentspace.FGDA`(*metric='riemann'*, *tsupdate=False*)

Fisher Geodesic Discriminant analysis.

Project data in Tangent space, apply a FLDA to reduce dimention, and project filtered data back in the manifold. For a complete description of the algorithm, see [1].

Parameters

metric
[string | dict, default='riemann'] The type of metric used for reference matrix estimation (see *mean_covariance* for the list of supported metric) and for tangent space map (see *tangent_space* for the list of supported metric). The metric could be a dict with two keys, *mean* and *map* in order to pass different metrics for the reference matrix estimation and the tangent space mapping.

tsupdate
[bool, default=False] Activate tangent space update for covariante shift correction between training and test, as described in [2]. This is not compatible with online implementation. Performance are better when the number of matrices for prediction is higher.

See also:

FgMDM
TangentSpace

References

[1], [2]

`__init__`(*metric='riemann'*, *tsupdate=False*)

Init.

`fit`(*X*, *y=None*, *sample_weight=None*)

Fit (estimates) the reference point and the FLDA.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

self
[FGDA instance] The FGDA instance.

`fit_transform`(*X*, *y=None*, *sample_weight=None*)

Fit and transform in a single function.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Not used, here for compatibility with sklearn API.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

covs
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices after filtering.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Filtering operation.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

covs
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices after filtering.

5.7 Spatial Filtering

Xdawn([nfilter, classes, estimator, ...])	Xdawn algorithm.
CSP([nfilter, metric, log])	CSP spatial filtering with covariance matrices as inputs.
SPoC([nfilter, metric, log])	SPoC spatial filtering with covariance matrices as inputs.
BilinearFilter(filters[, log])	Bilinear spatial filter.
AJDC([window, overlap, fmin, fmax, fs, ...])	AJDC algorithm.

5.7.1 pyriemann.spatialfilters.Xdawn

class pyriemann.spatialfilters.**Xdawn**(*nfilter=4*, *classes=None*, *estimator='scm'*, *baseline_cov=None*)

Xdawn algorithm.

Xdawn [1] is a spatial filtering method designed to improve the signal to signal + noise ratio (SSNR) of the ERP responses. Xdawn was originaly designed for P300 evoked potential by enhancing the target response with respect to the non-target response [2]. This implementation is a generalization to any type of ERP.

Parameters

nfilter
[int, default=4] The number of components to decompose M/EEG signals.

classes
[list of int | None, default=None] List of classes to take into account for Xdawn. If None, all classes will be accounted.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

baseline_cov
[None | array, shape(n_channels, n_channels), default=None] Covariance matrix to which the average signals are compared. If None, the baseline covariance is computed across all trials and time samples.

See also:

`XdawnCovariances`

References

[1], [2]

Attributes

classes_
[ndarray, shape (n_classes,)] Labels for each class.

filters_
[ndarray, shape (n_classes x min(n_channels, n_filters), n_channels)] If fit, the Xdawn components used to decompose the data for each event type, concatenated.

patterns_
[ndarray, shape (n_classes x min(n_channels, n_filters), n_channels)] If fit, the Xdawn patterns used to restore M/EEG signals for each event type, concatenated.

evokeds_
[ndarray, shape (n_classes x min(n_channels, n_filters), n_times)] If fit, the evoked response for each event type, concatenated.

`__init__`(*nfilter=4*, *classes=None*, *estimator='scm'*, *baseline_cov=None*)

Init.

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

Train Xdawn spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_times)] Set of trials.

y
[ndarray, shape (n_trials,)] Labels for each trial.

Returns

self
[Xdawn instance] The Xdawn instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_times)] Set of trials.

Returns

Xf
[ndarray, shape (n_trials, n_classes x min(n_channels, n_filters), n_times)] Set of spatialy filtered trials.

5.7.2 pyriemann.spatialfilters.CSP

class pyriemann.spatialfilters.**CSP**(*nfilter=4*, *metric='euclid'*, *log=True*)

CSP spatial filtering with covariance matrices as inputs.

Implementation of the famous Common Spatial Pattern algorithm [1] [2], but with covariance matrices as input. In addition, the implementation allows different metric for the estimation of the class-related mean covariance matrices, as described in [3].

This implementation support multiclass CSP by means of approximate joint diagonalization. In this case, the spatial filter selection is achieved according to [4].

Parameters

nfilter
[int, default=4] The number of components to decompose M/EEG signals.

metric
[str, default='euclid'] The metric for the estimation of mean covariance matrices.

log
[bool, default=True] If true, return the log variance, otherwise return the spatially filtered covariance matrices.

See also:

MDM, *SPoC*

References

[1], [2], [3], [4]

Attributes

filters_
[ndarray, shape (min(n_channels, n_filters), n_channels)] If fit, the CSP spatial filters.

patterns_
[ndarray, shape (min(n_channels, n_filters), n_channels)] If fit, the CSP spatial patterns.

__init__(*nfilter=4*, *metric='euclid'*, *log=True*)

Init.

fit(*X*, *y*)

Train CSP spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_channels)] Set of covariance matrices.

y
[ndarray, shape (n_trials,)] Labels for each trial.

Returns

self
[CSP instance] The CSP instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_channels)] Set of covariance matrices.

Returns

Xf
[ndarray, shape (n_trials, n_filters) or ndarray, shape (n_trials, n_filters, n_filters)] Set of spatialy filtered log-variance or covariance, depending on the 'log' input parameter.

5.7.3 pyriemann.spatialfilters.SPoC

class pyriemann.spatialfilters.**SPoC**(*nfilter=4*, *metric='euclid'*, *log=True*)

SPoC spatial filtering with covariance matrices as inputs.

Source Power Comodulation (SPoC) [1] allows to extract spatial filters and patterns by using a target (continuous) variable in the decomposition process in order to give preference to components whose power comodulates with the target variable.

SPoC can be seen as an extension of the *pyriemann.spatialfilters.CSP* driven by a continuous variable rather than a discrete (often binary) variable. Typical applications include extraction of motor patterns using EMG power or audio paterns using sound envelope.

Parameters

nfilter
[int, default=4] The number of components to decompose M/EEG signals.

metric
[str, default='euclid'] The metric for the estimation of mean covariance matrices.

log
[bool, default=True] If true, return the log variance, otherwise return the spatially filtered covariance matrices.

See also:

CSP

Notes

New in version 0.2.4.

References

[1]

Attributes

filters_
[ndarray, shape (min(n_channels, n_filters), n_channels)] If fit, the SPoC spatial filters.

patterns_
[ndarray, shape (min(n_channels, n_filters), n_channels)] If fit, the SPoC spatial patterns.

__init__(*nfilter=4*, *metric='euclid'*, *log=True*)

Init.

fit(*X*, *y*)

Train spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_channels)] Set of covariance matrices.

y
[ndarray, shape (n_trials,)] Target variable for each trial.

Returns

self
[SPoC instance] The SPoC instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_channels)] Set of covariance matrices.

Returns

Xf
[ndarray, shape (n_trials, n_filters) or ndarray, shape (n_trials, n_filters, n_filters)] Set of spatialy filtered log-variance or covariance, depending on the 'log' input parameter.

5.7.4 pyriemann.spatialfilters.BilinearFilter

`class pyriemann.spatialfilters.BilinearFilter`(*filters*, *log=False*)

Bilinear spatial filter.

Bilinear spatial filter for SPD matrices allows to define a custom spatial filter for bilinear projection of the data:

$$\mathbf{Cf}_i = \mathbf{V}\mathbf{C}_i\mathbf{V}^T$$

If log parameter is set to true, will return the log of the diagonal:

$$\mathbf{cf}_i = \log[\mathrm{diag}(\mathbf{Cf}_i)]$$

Parameters

filters
[ndarray, shape (n_filters, n_channels)] The filters for bilinear transform.

log
[bool, default=False] If true, return the log variance, otherwise return the spatially filtered covariance matrices.

Attributes

filters_
[ndarray, shape (n_filters, n_channels)] If fit, the filter components used to decompose the data for each event type, concatenated.

`__init__`(*filters*, *log=False*)

Init.

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

Train BilinearFilter spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_channels)] Set of covariance matrices.

y
[ndarray, shape (n_trials,)] Labels for each trial.

Returns

self
[BilinearFilter instance] The BilinearFilter instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply spatial filters.

Parameters

X
[ndarray, shape (n_trials, n_channels, n_channels)] Set of covariance matrices.

Returns

Xf
[ndarray, shape (n_trials, n_filters) or ndarray, shape (n_trials, n_filters, n_filters)] Set of spatialy filtered log-variance or covariance, depending on the 'log' input parameter.

5.7.5 pyriemann.spatialfilters.AJDC

class pyriemann.spatialfilters.**AJDC**(*window=128*, *overlap=0.5*, *fmin=None*, *fmax=None*, *fs=None*, *dim_red=None*, *verbose=True*)

AJDC algorithm.

The approximate joint diagonalization of Fourier cospectral matrices (AJDC) [1] is a versatile tool for blind source separation (BSS) tasks based on Second-Order Statistics (SOS), estimating spectrally uncorrelated sources.

It can be applied:

- on a single subject, to solve the classical BSS problem [1],
- on several subjects, to solve the group BSS (gBSS) problem [2],
- on several experimental conditions (for eg, baseline versus task), to exploit the diversity of source energy between conditions in addition to generic coloration and time-varying energy [1].

AJDC estimates Fourier cospectral matrices by the Welch's method, and applies a trace-normalization. If necessary, it averages cospectra across subjects, and concatenates them along experimental conditions. Then, a dimension reduction and a whitening are applied on cospectra. An approximate joint diagonalization (AJD) [3] allows to estimate the joint diagonalizer, not constrained to be orthogonal. Finally, forward and backward spatial filters are computed.

Parameters

window
[int, default=128] The length of the FFT window used for spectral estimation.

overlap
[float, default=0.5] The percentage of overlap between window.

fmin
[float | None, default=None] The minimal frequency to be returned. Since BSS models assume zero-mean processes, the first cospectrum (0 Hz) must be excluded.

fmax
[float | None, default=None] The maximal frequency to be returned.

fs
[float | None, default=None] The sampling frequency of the signal.

dim_red
[None | dict, default=None] Parameter for dimension reduction of cospectra, because Pham's AJD is sensitive to matrices conditioning.

If `None` :
no dimension reduction during whitening.

If `{'n_components': val}` :
dimension reduction defining the number of components; `val` must be an integer superior to 1.

If `{'expl_var': val}` :
dimension reduction selecting the number of components such that the amount of variance

that needs to be explained is greater than the percentage specified by `val`. `val` must be a float in (0,1], typically `0.99`.

If `{'max_cond': val}`:
dimension reduction selecting the number of components such that the condition number of the mean matrix is lower than `val`. This threshold has a physiological interpretation, because it can be viewed as the ratio between the power of the strongest component (usually, eye-blink source) and the power of the lowest component you don't want to keep (acquisition sensor noise). `val` must be a float strictly superior to 1, typically 100.

If `{'warm_restart': val}`:
dimension reduction defining the number of components from an initial joint diagonalizer, and then run AJD from this solution. `val` must be a square ndarray.

verbose
[bool, default=True] Verbose flag.

See also:

`CospCovariances`

Notes

New in version 0.2.7.

References

[1], [2], [3]

Attributes

n_channels_
[int] If fit, the number of channels of the signal.

freqs_
[ndarray, shape (n_freqs,)] If fit, the frequencies associated to cospectra.

n_sources_
[int] If fit, the number of components of the source space.

diag_filters_
[ndarray, shape `(n_sources_, n_sources_)`] If fit, the diagonalization filters, also called joint diagonalizer.

forward_filters_
[ndarray, shape `(n_sources_, n_channels_)`] If fit, the spatial filters used to transform signal into source, also called deximing or separating matrix.

backward_filters_
[ndarray, shape `(n_channels_, n_sources_)`] If fit, the spatial filters used to transform source into signal, also called mixing matrix.

`__init__`(*window=128*, *overlap=0.5*, *fmin=None*, *fmax=None*, *fs=None*, *dim_red=None*, *verbose=True*)

Init.

`fit`(*X*, *y=None*)

Fit.

Compute and diagonalize cospectra, to estimate forward and backward spatial filters.

Parameters

X
[ndarray, shape (n_subjects, n_conditions, n_channels, n_times) | list of n_subjects of list of n_conditions ndarray of shape (n_channels, n_times), with same n_conditions and n_channels but different n_times] Multi-channel time-series in channel space, acquired for different subjects and under different experimental conditions.

y
[None] Currently not used, here for compatibility with sklearn API.

Returns

self
[AJDC instance] The AJDC instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_src_expl_var`(*X*)

Estimate explained variances of sources.

Estimate explained variances of sources, see Appendix D in [1].

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series in channel space.

Returns

src_var
[ndarray, shape (n_matrices, n_sources)] Explained variance for each source.

`inverse_transform`(*X*, *supp=None*)

Transform source space to channel space.

Transform source space to channel space, applying backward spatial filters, with the possibility to suppress some sources, like in BSS filtering/denoising.

Parameters

X
[ndarray, shape (n_matrices, n_sources, n_times)] Multi-channel time-series in source space.

supp
[list of int | None, default=None] Indices of sources to suppress. If None, no source suppression.

Returns

signal
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series in channel space.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform channel space to source space.

Transform channel space to source space, applying forward spatial filters.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series in channel space.

Returns

source
[ndarray, shape (n_matrices, n_sources, n_times)] Multi-channel time-series in source space.

5.8 Preprocessing

Whitening([metric, dim_red, verbose])	Whitening, and optional unsupervised dimension reduction.

5.8.1 pyriemann.preprocessing.Whitening

class pyriemann.preprocessing.**Whitening**(*metric='euclid'*, *dim_red=None*, *verbose=False*)

Whitening, and optional unsupervised dimension reduction.

Implementation of the whitening, and an optional unsupervised dimension reduction, with SPD matrices as inputs.

Parameters

metric
[str, default='euclid'] The metric for the estimation of mean matrix used for whitening and dimension reduction.

dim_red
[None | dict, default=None]

If `None` :
no dimension reduction during whitening.

If `{'n_components': val}` :
dimension reduction defining the number of components; `val` must be an integer superior to 1.

If `{'expl_var': val}` :
dimension reduction selecting the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by `val`. `val` must be a float in (0,1], typically `0.99`.

If `{'max_cond': val}` :
dimension reduction selecting the number of components such that the condition number of the mean matrix is lower than `val`. This threshold has a physiological interpretation, because it can be viewed as the ratio between the power of the strongest component (usually, eye-blink source) and the power of the lowest component you don't want to keep (acquisition sensor noise). `val` must be a float strictly superior to 1, typically 100.

verbose
[bool, default=False] Verbose flag.

Notes

New in version 0.2.7.

Attributes

n_components_
[int] If fit, the number of components after dimension reduction.

filters_
[ndarray, shape `(n_channels_, n_components_)`] If fit, the spatial filters to whiten SPD matrices.

inv_filters_
[ndarray, shape (n_components_, n_channels_)] If fit, the spatial filters to unwhiten SPD matrices.

`__init__`(*metric='euclid'*, *dim_red=None*, *verbose=False*)

Init.

`fit`(*X*, *y=None*, *sample_weight=None*)

Train whitening spatial filters.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None] Ignored as unsupervised.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weight of each matrix, to compute the weighted mean matrix used for whitening and dimension reduction. If None, it uses equal weights.

Returns

self
[Whitening instance] The Whitening instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Apply inverse whitening spatial filters.

Parameters

X
[ndarray, shape (n_matrices, n_components, n_components)] Set of whitened, and optionally reduced, SPD matrices.

Returns

Xiw
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of unwhitened, and optionally unreduced, SPD matrices.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply whitening spatial filters.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

Xw
[ndarray, shape (n_matrices, n_components, n_components)] Set of whitened, and optionally reduced, SPD matrices.

5.9 Channel selection

ElectrodeSelection([nelec, metric, n_jobs])	Channel selection based on a Riemannian geometry criterion.
FlatChannelRemover()	Finds and removes flat channels.

5.9.1 pyriemann.channelselection.ElectrodeSelection

class pyriemann.channelselection.**ElectrodeSelection**(*nelec=16*, *metric='riemann'*, *n_jobs=1*)

Channel selection based on a Riemannian geometry criterion.

For each class, a centroid is estimated, and the channel selection is based on the maximization of the distance between centroids. This is done by a backward elimination where the electrode that carries the less distance is removed from the subset at each iteration. This algorithm is described in [1].

Parameters

nelec
[int, default=16] The number of electrode to keep in the final subset.

metric
[string | dict, default='riemann'] The type of metric used for centroid and distance estimation. see *mean_covariance* for the list of supported metric. the metric could be a dict with two keys, *mean* and *distance* in order to pass different metric for the centroid estimation and the distance estimation. Typical usecase is to pass 'logeuclid' metric for the mean in order to boost the computional speed and 'riemann' for the distance in order to keep the good sensitivity for the selection.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the class centroid in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

See also:

Kmeans
FgMDM

References

[1]

Attributes

covmeans_
[list] The class centroids.

dist_
[list] List of distance at each interation.

__init__(*nelec=16*, *metric='riemann'*, *n_jobs=1*)

Init.

fit(*X*, *y=None*, *sample_weight=None*)

Find the optimal subset of electrodes.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None | ndarray, shape (n_matrices,), default=None] Labels for each matrix.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

self
[ElectrodeSelection instance] The ElectrodeSelection instance.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return reduced matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

covs
[ndarray, shape (n_matrices, n_elec, n_elec)] Set of SPD matrices after reduction of the number of channels.

5.9.2 pyriemann.channelselection.FlatChannelRemover

`class pyriemann.channelselection.FlatChannelRemover`

Finds and removes flat channels.

Attributes

channels_
[ndarray, shape (n_good_channels)] The indices of the non-flat channels.

`__init__`(**args*, ***kwargs*)

`fit`(*X*, *y=None*)

Find flat channels.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

X
[ndarray, shape (n_matrices, n_good_channels, n_times)] Multi-channel time-series without flat channels.

`fit_transform`(*X*, *y=None*)

Find and remove flat channels.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

y
[None] Not used, here for compatibility with sklearn API.

Returns

X
[ndarray, shape (n_matrices, n_good_channels, n_times)] Multi-channel time-series without flat channels.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Remove flat channels.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

Returns

X
[ndarray, shape (n_matrices, n_good_channels, n_times)] Multi-channel time-series without flat channels.

5.10 Transfer Learning

encode_domains(X, y, domain)	Encode the domains of the matrices in the labels.
decode_domains(X_enc, y_enc)	Decode the domains of the matrices in the labels.
TLSplitter(target_domain, cv)	Class for handling the cross-validation splits of multi-domain data.
TLEstimator(target_domain, estimator[, ...])	Transfer learning wrapper for estimators.
TLClassifier(target_domain, estimator[, ...])	Transfer learning wrapper for classifiers.
TLRegressor(target_domain, estimator[, ...])	Transfer learning wrapper for regressors.
TLDummy()	No transformation on data for transfer learning.
TLCenter(target_domain[, metric])	Recenter data for transfer learning.
TLStretch(target_domain[, final_dispersion, ...])	Stretch data for transfer learning.
TLRotate(target_domain[, weights, metric, ...])	Rotate data for transfer learning.
MDWM(domain_tradeoff, target_domain[, ...])	Classification by Minimum Distance to Weighted Mean.

5.10.1 pyriemann.transfer.encode_domains

pyriemann.transfer.**encode_domains**(*X*, *y*, *domain*)

Encode the domains of the matrices in the labels.

We handle the possibility of having different domains for the datasets by extending the labels of the matrices and including this information to them. For instance, if we have a matrix X with class *left_hand* on the *domain_01* then its extended label will be *domain_01/left_hand*. Note that if the classes were integers at first, they will be converted to strings.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

domain
[ndarray, shape (n_matrices,)] Domains for each matrix.

Returns

X_enc
[ndarray, shape (n_matrices, n_channels, n_channels)] The same set of SPD matrices given as input.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

See also:

decode_domains

Notes

New in version 0.3.1.

5.10.2 pyriemann.transfer.decode_domains

pyriemann.transfer.**decode_domains**(*X_enc*, *y_enc*)

Decode the domains of the matrices in the labels.

We handle the possibility of having different domains for the datasets by encoding the domain information into the labels of the matrices. This method converts the data into its original form, with a separate data structure for labels and for domains.

Parameters

X_enc
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Labels for each matrix.

domain
[ndarray, shape (n_matrices,)] Domains for each matrix.

See also:

encode_domains

Notes

New in version 0.3.1.

5.10.3 pyriemann.transfer.TLSplitter

class pyriemann.transfer.**TLSplitter**(*target_domain*, *cv*)

Class for handling the cross-validation splits of multi-domain data.

This is a wrapper to sklearn's cross-validation iterators [1] which ensures the handling of domain information with the data points. In fact, the data from source domain is always fully available in the training partition whereas the random splits are done on the data points from the target domain.

Parameters

target_domain
[str] Domain considered as target.

cv
[None | BaseCrossValidator | BaseShuffleSplit, default=None] An instance of a cross validation iterator from sklearn.

Notes

New in version 0.3.1.

References

[1]

__init__(*target_domain*, *cv*)

get_n_splits(*X=None*, *y=None*)

Returns the number of splitting iterations in the cross-validator.

Parameters

X
[object] Ignored, exists for compatibility.

y
[object] Ignored, exists for compatibility.

Returns

n_splits
[int] Returns the number of splitting iterations in the cross-validator.

split(*X*, *y*)

Generate indices to split data into training and test set.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Yields

train
[ndarray] The training set indices for that split.

test
[ndarray] The testing set indices for that split.

5.10.4 pyriemann.transfer.TLEstimator

class pyriemann.transfer.**TLEstimator**(*target_domain*, *estimator*, *domain_weight=None*)

Transfer learning wrapper for estimators.

This is a wrapper for any BaseEstimator (i.e. classifier or regressor) that converts extended labels used in Transfer Learning into the usual y array to train a classifier/regressor of choice.

Parameters

target_domain
[str] Domain to consider as target.

estimator
[BaseEstimator] The estimator to apply on matrices. It can be any regressor or classifier from pyRiemann.

domain_weight
[None | dict, default=None] Weights to combine matrices from each domain to train the estimator. The dict contains key=domain_name and value=weight_to_assign. If None, it uses equal weights.

Notes

New in version 0.3.1.

`__init__`(*target_domain*, *estimator*, *domain_weight=None*)
Init.

`fit`(*X*, *y_enc*)
Fit TLEstimator.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

self
[TLEstimator instance] The TLEstimator instance.

`get_params`(*deep=True*)
Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)
Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray, shape (n_matrices,)] Predictions for each matrix according to the estimator.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

5.10.5 pyriemann.transfer.TLClassifier

class pyriemann.transfer.**TLClassifier**(*target_domain*, *estimator*, *domain_weight=None*)

Transfer learning wrapper for classifiers.

This is a wrapper for any classifier that converts extended labels used in Transfer Learning into the usual y array to train a classifier of choice.

Parameters

target_domain
[str] Domain to consider as target.

estimator
[BaseClassifier] The classifier to apply on matrices.

domain_weight
[None | dict, default=None] Weights to combine matrices from each domain to train the classifier. The dict contains key=domain_name and value=weight_to_assign. If None, it uses equal weights.

Notes

New in version 0.3.1.

__init__(*target_domain*, *estimator*, *domain_weight=None*)

Init.

fit(*X*, *y_enc*)

Fit TLClassifier.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

self
[TLClassifier instance] The TLClassifier instance.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray, shape (n_matrices,)] Predictions for each matrix according to the estimator.

`predict_proba`(*X*)

Get the probability.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray, shape (n_matrices, n_classes)] Predictions for each matrix.

`score`(*X*, *y_enc*)

Return the mean accuracy on the given test data and labels.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Test set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended true labels for each matrix.

Returns

score
[float] Mean accuracy of self.predict(X) wrt. y.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

5.10.6 pyriemann.transfer.TLRegressor

class pyriemann.transfer.**TLRegressor**(*target_domain*, *estimator*, *domain_weight=None*)

Transfer learning wrapper for regressors.

This is a wrapper for any regressor that converts extended labels used in Transfer Learning into the usual y array to train a regressor of choice.

Parameters

target_domain
[str] Domain to consider as target.

estimator
[BaseRegressor] The regressor to apply on matrices.

domain_weight
[None | dict, default=None] Weights to combine matrices from each domain to train the regressor. The dict contains key=domain_name and value=weight_to_assign. If None, it uses equal weights.

Notes

New in version 0.3.1.

__init__(*target_domain*, *estimator*, *domain_weight=None*)

Init.

fit(*X*, *y_enc*)

Fit TLRegressor.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

self
[TLRegressor instance] The TLRegressor instance.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray, shape (n_matrices,)] Predictions for each matrix according to the estimator.

`score`(*X*, *y_enc*)

Return the coefficient of determination of the prediction.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Test set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended true values for each matrix.

Returns

score
[float] R2 of self.predict(X) wrt. y.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

5.10.7 pyriemann.transfer.TLDummy

`class` pyriemann.transfer.`TLDummy`

No transformation on data for transfer learning.

No transformation of the data points between the domains. This is what we call the Direct Center Transfer (DCT) method.

Notes

New in version 0.3.1.

`__init__`()

`fit`(*X*, *y_enc*)

Do nothing.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

self
[TLDummy instance] The TLDummy instance.

`fit_transform`(*X*, *y_enc*)

Do nothing.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Set of SPD matrices with mean in the Identity.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*, *y_enc=None*)

Do nothing.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[None] Not used, here for compatibility with sklearn API.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Same set of SPD matrices as in the input.

5.10.8 pyriemann.transfer.TLCenter

`class` pyriemann.transfer.`TLCenter`(*target_domain*, *metric='riemann'*)

Recenter data for transfer learning.

Recenter the data points from each domain to the Identity on manifold, ie make the mean of the datasets become the identity. This operation corresponds to a whitening step if the SPD matrices represent the spatial covariance matrices of multivariate signals.

Note: Using .fit() and then .transform() will give different results than .fit_transform(). In fact, .fit_transform() should be applied on the training dataset (target and source) and .transform() on the test partition of the target dataset.

Parameters

target_domain
[str] Domain to consider as target.

metric
[str, default='riemann'] The metric for mean, can be: 'ale', 'alm', 'euclid', 'harmonic', 'identity', 'kullback_sym', 'logdet', 'logeuclid', 'riemann', 'wasserstein', or a callable function. Note, however, that only when using the 'riemann' metric that we are ensured to re-center the data points precisely to the Identity.

Notes

New in version 0.3.1.

References

[1]

Attributes

recenter_
[dict] Dictionary with key=domain_name and value=domain_mean

`__init__`(*target_domain*, *metric='riemann'*)

Init

`fit`(*X*, *y_enc*)

Fit TLCenter.

Calculate the mean of all matrices in each domain.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

self
[TLCenter instance] The TLCenter instance.

`fit_transform`(*X*, *y_enc*)

Fit TLCenter and then transform data points.

Calculate the mean of all matrices in each domain and then recenter them to Identity.

Note: This method is designed for using at training time. The output for .fit_transform() will be different than using .fit() and .transform() separately.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Set of SPD matrices with mean in the Identity.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*, *y_enc=None*)

Re-center the data points in the target domain to Identity.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[None] Not used, here for compatibility with sklearn API.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Set of SPD matrices with mean in the Identity.

5.10.9 pyriemann.transfer.TLStretch

`class pyriemann.transfer.TLStretch`(*target_domain*, *final_dispersion=1.0*, *centered_data=False*, *metric='riemann'*)

Stretch data for transfer learning.

Change the dispersion of the datapoints around their geometric mean for each dataset so that they all have the same desired value.

Note: Using .fit() and then .transform() will give different results than .fit_transform(). In fact, .fit_transform() should be applied on the training dataset (target and source) and .transform() on the test partition of the target dataset.

Parameters

target_domain
[str] Domain to consider as target.

dispersion
[float, default=1.0] Target value for the dispersion of the data points.

centered_data
[bool, default=False] Whether the data has been re-centered to the Identity beforehand.

metric
[str, default='riemann'] The metric for calculating the dispersion can be: 'ale', 'alm', 'euclid', 'harmonic', 'identity', 'kullback_sym', 'logdet', 'logeuclid', 'riemann', 'wasserstein', or a callable function.

Notes

New in version 0.3.1.

References

[1]

Attributes

dispersions_
[dict] Dictionary with key=domain_name and value=domain_dispersion.

`__init__`(*target_domain*, *final_dispersion=1.0*, *centered_data=False*, *metric='riemann'*)

Init

`fit`(*X*, *y_enc*)

Fit TLStretch.

Calculate the dispersion around the mean for each domain.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

self
[TLStretch instance] The TLStretch instance.

`fit_transform`(*X*, *y_enc*)

Fit TLStretch and then transform data points.

Calculate the dispersion around the mean for each domain and then stretch the data points to the desired final dispersion.

Note: This method is designed for using at training time. The output for .fit_transform() will be different than using .fit() and .transform() separately.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Set of SPD matrices with desired final dispersion.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*, *y_enc=None*)

Stretch the data points in the target domain.

Note: The stretching operation is properly defined only for the riemann metric.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[None] Not used, here for compatibility with sklearn API.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Set of SPD matrices with desired final dispersion.

5.10.10 pyriemann.transfer.TLRotate

class pyriemann.transfer.**TLRotate**(*target_domain*, *weights=None*, *metric='euclid'*, *n_jobs=1*)

Rotate data for transfer learning.

Rotate the data points from each source domain so to match its class means with those from the target domain. The loss function for this matching was first proposed in [1] and the optimization procedure for mininimizing it follows the presentation from [2].

Note: The data points from each domain must have been re-centered to the identity before calculating the rotation.

Note: Using .fit() and then .transform() will give different results than .fit_transform(). In fact, .fit_transform() should be applied on the training dataset (target and source) and .transform() on the test partition of the target dataset.

Parameters

target_domain
[str] Domain to consider as target.

weights
[None | array, shape (n_classes,), default=None] Weights to assign for each class. If None, then give the same weight for each class.

metric
[{'euclid', 'riemann'}, default='euclid'] Metric for the distance to minimize between class means. Options are either the Euclidean ('euclid') or Riemannian ('riemann') distance.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing the rotation matrix for each source domain in parallel. If -1 all CPUs are used.

Notes

New in version 0.3.1.

References

[1], [2]

Attributes

rotations_
[dict] Dictionary with key=domain_name and value=domain_rotation_matrix.

__init__(*target_domain*, *weights=None*, *metric='euclid'*, *n_jobs=1*)

Init

fit(*X*, *y_enc*)

Fit TLRotate.

Calculate the rotations matrices to transform each source domain into the target domain.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

self
[TLRotate instance] The TLRotate instance.

`fit_transform`(*X*, *y_enc*)

Fit TLRotate and then transform data points.

Calculate the rotation matrix for matching each source domain to the target domain.

Note: This method is designed for using at training time. The output for .fit_transform() will be different than using .fit() and .transform() separately.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Set of SPD matrices after rotation step.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*, *y_enc=None*)

Rotate the data points in the target domain.

The rotations are done from source to target, so in this step the data points suffer no transformation at all.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[None] Not used, here for compatibility with sklearn API.

Returns

X
[ndarray, shape (n_matrices, n_classes)] Same set of SPD matrices as in the input.

5.10.11 pyriemann.transfer.MDWM

`class` pyriemann.transfer.`MDWM`(*domain_tradeoff*, *target_domain*, *metric='riemann'*, *n_jobs=1*)

Classification by Minimum Distance to Weighted Mean.

Classification by nearest centroid. For each of the given classes, a centroid is estimated, according to the chosen metric, as a weighted mean of SPD matrices from the source domain, combined with the class centroid of the target domain [1] [2]. For classification, a given new matrix is attibuted to the class whose centroid is the nearest according to the chosen metric.

Parameters

domain_tradeoff
[float] Coefficient in [0,1] controlling the transfer, ie the trade-off between source and target domains. At 0, there is no transfer, only matrices acquired from the source domain are used. At 1, this is a calibration-free system as no matrices are required from the source domain.

target_domain
[string] Name of the target domain in extended labels.

metric
[string | dict, default='riemann'] The type of metric used for centroid and distance estimation. see *mean_covariance* for the list of supported metric. the metric could be a dict with two keys, *mean* and *distance* in order to pass different metric for the centroid estimation and the distance estimation. Typical usecase is to pass 'logeuclid' metric for the mean in order to boost the computional speed and 'riemann' for the distance in order to keep the good sensitivity for the classification.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the class centroid in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

See also:

`MDM`

Notes

New in version 0.3.1.

References

[1], [2]

Attributes

classes_
[ndarray, shape (n_classes,)] Labels for each class.

covmeans_
[list of `n_classes` ndarrays of shape (n_channels, n_channels)] Centroids for each class.

`__init__`(*domain_tradeoff*, *target_domain*, *metric='riemann'*, *n_jobs=1*)

Init.

`fit`(*X*, *y_enc*, *sample_weight=None*)

Fit (estimates) the centroids.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices from source and target domain.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

sample_weight
[None | ndarray, shape (n_matrices_source,), default=None] Weights for each matrix from the source domains. If None, it uses equal weights.

Returns

self
[MDWM instance] The MDWM instance.

`fit_predict`(*X*, *y*)

Fit and predict in one function.

`fit_transform`(*X*, *y=None*, ***fit_params*)

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
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Get the predictions.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

pred
[ndarray of int, shape (n_matrices,)] Predictions for each matrix according to the closest centroid.

`predict_proba`(*X*)

Predict proba using softmax.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

prob
[ndarray, shape (n_matrices, n_classes)] Probabilities for each class.

`score`(*X*, *y_enc*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y_enc
[ndarray, shape (n_matrices,)] Extended labels for each matrix.

Returns

score
[float] Mean accuracy of clf.predict(X) wrt. y_enc.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Get the distance to each centroid.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

Returns

dist
[ndarray, shape (n_matrices, n_classes)] The distance to each centroid according to the metric.

5.11 Stats

PermutationDistance([n_perms, metric, mode, ...])	Permutation test based on distance.
PermutationModel([n_perms, model, cv, ...])	Permutation test using any scikit-learn model for scoring.

5.11.1 pyriemann.stats.PermutationDistance

class pyriemann.stats.**PermutationDistance**(*n_perms=100*, *metric='riemann'*, *mode='pairwise'*, *n_jobs=1*, *random_state=42*, *estimator=None*)

Permutation test based on distance.

Perform a permutation test based on distance. You have the choice of 3 different statistic:

- **'pairwise' :**
  the statistic is based on paiwire distance as descibed in [1]. This is the fastest option for low sample size since the pairwise distance matrix does not need to be estimated for each permutation.
- **'ttest' :**
  t-test based statistic obtained by the ration of the distance between each riemannian centroid and the group dispersion. The means have to be estimated for each permutation, leading to a slower procedure. However, this can be used for high sample size.
- **'ftest':**
  f-test based statistic estimated using the between and within group variability. As for the 'ttest' stats, group centroid are estimated for each permutation.

Parameters

n_perms
[int, default=100] The number of permutation. The minimum should be 20 for a resolution of 0.05 p-value.

metric
[string | dict, default='riemann'] The type of metric used for centroid and distance estimation. see *distance* anb *mean_covariance* for the list of supported metric. the metric could be a dict with two keys, *mean* and *distance* in order to pass different metric for the centroid estimation and the distance estimation. Typical usecase is to pass 'logeuclid' metric for the mean in order to boost the computional speed and 'riemann' for the distance in order to keep the good sensitivity for the classification.

mode
[string, default='pairwise'] Type of statistic to use. could be 'pairwise', 'ttest' of 'ftest'

n_jobs
[integer, default=1] The number of CPUs to use to do the computation. -1 means 'all CPUs'.

random_state
[int, default=42] random state for the permutation test.

estimator
[None or sklearn compatible estimator, default=None] If provided, data are transformed before every permutation. should not be used unless a supervised opperation must be applied on the data. This would be the case for ERP covariance.

See also:

PermutationModel

References

[1]

Attributes

p_value_
[float] the p-value of the test

scores_
[list] contain all scores for all permutations. The fist element is the non-permuted score.

`__init__`(*n_perms=100*, *metric='riemann'*, *mode='pairwise'*, *n_jobs=1*, *random_state=42*, *estimator=None*)
Init.

`plot`(*nbins=10*, *range=None*, *axes=None*)
Plot results of the permutation test.

Parameters

nbins
[integer or array_like or 'auto', default=10] If an integer is given, bins + 1 bin edges are returned, consistently with np.histogram() for numpy version >= 1.3. Unequally spaced bins are supported if bins is a sequence.

range
[tuple or None, default=None] The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, range is (x.min(), x.max()). Range has no effect if

bins is a sequence. If bins is a sequence or range is specified, autoscaling is based on the specified bin range instead of the range of x.

axes
[axes handle, default=None] Axes handle for matplotlib. if None a new figure will be created.

`score`(*X*, *y*, *groups=None*)

Score of a permutation.

Parameters

X
[array-like] The data to fit. Can be, for example a list, or an array at least 2d.

y
[array-like] The target variable to try to predict in the case of supervised learning.

`test`(*X*, *y*, *groups=None*, *verbose=True*)

Performs the permutation test

Parameters

X
[array-like] The data to fit. Can be, for example a list, or an array at least 2d.

y
[array-like] The target variable to try to predict in the case of supervised learning.

verbose
[bool, default=True] If true, print progress.

5.11.2 pyriemann.stats.PermutationModel

`class` pyriemann.stats.`PermutationModel`(*n_perms=100*, *model=MDM()*, *cv=3*, *scoring=None*, *n_jobs=1*, *random_state=42*)

Permutation test using any scikit-learn model for scoring.

Perform a permutation test using the cross-validation score of any scikit-learn compatible model. Score is obtained with *cross_val_score* from scikit-learn. The score should be a "higer is better" metric.

Parameters

n_perms
[int, default=100] The number of permutation. The minimum should be 20 for a resolution of 0.05 p-value.

model
[sklearn compatible model, default=MDM()] The model for scoring.

cv
[int or cross-validation generator or an iterable, default=3] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation;
- integer, to specify the number of folds in a *(Stratified)KFold*;
- an object to be used as a cross-validation generator;
- an iterable yielding train, test splits.

For integer/None inputs, if the estimator is a classifier and *y* is either binary or multiclass, `StratifiedKFold` is used. In all other cases, *KFold* is used.

scoring
[string or callable or None, default=None] A string (see model evaluation documentation) or a scorer callable object / function with signature *scorer(estimator, X, y)*.

n_jobs
[integer, default=1] The number of CPUs to use to do the computation. -1 means 'all CPUs'.

random_state
[int, default=42] random state for the permutation test.

See also:

PermutationDistance

Attributes

p_value_
[float] the p-value of the test

scores_
[list] contain all scores for all permutations. The fist element is the non-permuted score.

`__init__`(*n_perms=100*, *model=MDM()*, *cv=3*, *scoring=None*, *n_jobs=1*, *random_state=42*)

Init.

`plot`(*nbins=10*, *range=None*, *axes=None*)

Plot results of the permutation test.

Parameters

nbins
[integer or array_like or 'auto', default=10] If an integer is given, bins + 1 bin edges are returned, consistently with np.histogram() for numpy version >= 1.3. Unequally spaced bins are supported if bins is a sequence.

range
[tuple or None, default=None] The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, range is (x.min(), x.max()). Range has no effect if bins is a sequence. If bins is a sequence or range is specified, autoscaling is based on the specified bin range instead of the range of x.

axes
[axes handle, default=None] Axes handle for matplotlib. if None a new figure will be created.

`score`(*X*, *y*, *groups=None*)

Score one permutation.

Parameters

X
[array-like] The data to fit. Can be, for example a list, or an array at least 2d.

y
[array-like] The target variable to try to predict in the case of supervised learning.

test(*X*, *y*, *groups=None*, *verbose=True*)

Performs the permutation test

Parameters

X
[array-like] The data to fit. Can be, for example a list, or an array at least 2d.

y
[array-like] The target variable to try to predict in the case of supervised learning.

verbose
[bool, default=True] If true, print progress.

5.12 Datasets

make_gaussian_blobs([n_matrices, n_dim, ...])	Generate SPD dataset with two classes sampled from Riemannian Gaussian.
make_outliers(n_matrices, mean, sigma[, ...])	Generate a set of outlier points.
make_covariances(n_matrices, n_channels[, ...])	Generate a set of covariances matrices, with the same eigenvectors.
make_masks(n_masks, n_dim0, n_dim1_min[, rs])	Generate a set of masks, defined as semi-orthogonal matrices.
sample_gaussian_spd(n_matrices, mean, sigma)	Sample a Riemannian Gaussian distribution.
generate_random_spd_matrix(n_dim[, ...])	Generate a random SPD matrix.
make_classification_transfer(n_matrices[, ...])	Generate source and target toy datasets for transfer learning examples.

5.12.1 pyriemann.datasets.make_gaussian_blobs

pyriemann.datasets.**make_gaussian_blobs**(*n_matrices=100*, *n_dim=2*, *class_sep=1.0*, *class_disp=1.0*, *return_centers=False*, *center_dataset=False*, *random_state=None*, *centers=None*, ***, *n_jobs=1*, *sampling_method='auto'*)

Generate SPD dataset with two classes sampled from Riemannian Gaussian.

Generate a dataset with SPD matrices drawn from two Riemannian Gaussian distributions. The distributions have the same class dispersions and the distance between their centers of mass is an input parameter. Useful for testing classification or clustering methods.

Parameters

n_matrices
[int, default=100] How many matrices to generate for each class.

n_dim
[int, default=2] Dimensionality of the SPD matrices generated by the distributions.

class_sep
[float, default=1.0] Parameter controlling the separability of the classes.

class_disp
[float, default=1.0] Intra dispersion of the points sampled from each class.

centers
[ndarray, shape (2, n_dim, n_dim), default=None] List with the centers of mass for each class. If None, the centers are sampled randomly based on class_sep.

return_centers
[bool, default=False] If True, then return the centers of each cluster

center_dataset
[bool, default=False] If True, re-center the simulated dataset to the Identity. If False, the dataset is centered around a random SPD matrix.

random_state
[int, RandomState instance or None, default=None] Pass an int for reproducible output across multiple function calls.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the class centroid in parallel. If -1 all CPUs are used.

sampling_method
[str, default='auto'] Name of the sampling method used to sample samples_r. It can be 'auto', 'slice' or 'rejection'. If it is 'auto', the sampling_method will be equal to 'slice' for n_dim != 2 and equal to 'rejection' for n_dim = 2.

New in version 0.3.1.

Returns

X
[ndarray, shape (2*n_matrices, n_dim, n_dim)] Set of SPD matrices.

y
[ndarray, shape (2*n_matrices,)] Labels corresponding to each matrix.

centers
[ndarray, shape (2, n_dim, n_dim)] The centers of each class. Only returned if `return_centers=True`.

Notes

New in version 0.3.

5.12.2 pyriemann.datasets.make_outliers

`pyriemann.datasets.`**`make_outliers`**(*n_matrices*, *mean*, *sigma*, *outlier_coeff=10*, *random_state=None*)

Generate a set of outlier points.

Simulate data points that are outliers for a given Riemannian Gaussian distribution with fixed mean and dispersion.

Parameters

n_matrices
[int] How many matrices to generate.

mean
[ndarray, shape (n_dim, n_dim)] Center of the Riemannian Gaussian distribution.

sigma
[float] Dispersion of the Riemannian Gaussian distribution.

outlier_coeff: float, default=10
Coefficient determining how to define an outlier data point, i.e. how many times the sigma parameter its distance to the mean should be.

random_state
[int, RandomState instance or None, default=None] Pass an int for reproducible output across multiple function calls.

Returns

outliers
[ndarray, shape (n_matrices, n_dim, n_dim)] Set of simulated outlier matrix.

Notes

New in version 0.3.

5.12.3 pyriemann.datasets.make_covariances

pyriemann.datasets.**make_covariances**(*n_matrices*, *n_channels*, *rs=None*, *return_params=False*, *evals_mean=2.0*, *evals_std=0.1*)

Generate a set of covariances matrices, with the same eigenvectors.

Parameters

n_matrices
[int] Number of matrices to generate.

n_channels
[int] Number of channels in covariance matrices.

rs
[RandomState instance, default=None] Random state for reproducible output across multiple function calls.

return_params
[bool, default=False] If True, then return parameters.

evals_mean
[float, default=2.0] Mean of eigen values.

evals_std
[float, default=0.1] Standard deviation of eigen values.

Returns

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Covariances matrices.

evals
[ndarray, shape (n_matrices, n_channels)] Eigen values used for each covariance matrix. Only returned if `return_params=True`.

evecs
[ndarray, shape (n_channels, n_channels)] Eigen vectors used for all covariance matrices. Only returned if `return_params=True`.

5.12.4 pyriemann.datasets.make_masks

pyriemann.datasets.**make_masks**(*n_masks*, *n_dim0*, *n_dim1_min*, *rs=None*)

Generate a set of masks, defined as semi-orthogonal matrices.

Parameters

n_masks
[int] Number of masks to generate.

n_dim0
[int] First dimension of masks.

n_dim1_min
[int] Minimal value for second dimension of masks.

rs
[RandomState instance, default=None] Random state for reproducible output across multiple function calls.

Returns

masks
[list of n_masks ndarray of shape (n_dim0, n_dim1_i), with different n_dim1_i, such that n_dim1_min <= n_dim1_i <= n_dim0] Masks.

5.12.5 pyriemann.datasets.sample_gaussian_spd

pyriemann.datasets.**sample_gaussian_spd**(*n_matrices*, *mean*, *sigma*, *random_state=None*, *n_jobs=1*, *sampling_method='auto'*)

Sample a Riemannian Gaussian distribution.

Sample SPD matrices from a Riemannian Gaussian distribution centered at mean and with dispersion parametrized by sigma. This distribution has been defined in [1] and generalizes the notion of a Gaussian distribution to the space of SPD matrices. The sampling is based on a spectral factorization of SPD matrices in terms of their eigenvectors (U-parameters) and the log of the eigenvalues (r-parameters).

Parameters

n_matrices
[int] How many matrices to generate.

mean
[ndarray, shape (n_dim, n_dim)] Center of the Riemannian Gaussian distribution.

sigma
[float] Dispersion of the Riemannian Gaussian distribution.

random_state
[int, RandomState instance or None, default=None] Pass an int for reproducible output across multiple function calls.

n_jobs
[int, default=1] The number of jobs to use for the computation. This works by computing each of the class centroid in parallel. If -1 all CPUs are used.

sampling_method
[str, default='auto'] Sampling method to sample eigenvalues. It can be 'auto', 'slice' or 'rejection'. If it is 'auto', the sampling_method will be equal to 'slice' for n_dim != 2 and equal to 'rejection' for n_dim = 2.

New in version 0.3.1.

Returns

samples
[ndarray, shape (n_matrices, n_dim, n_dim)] Samples of the Riemannian Gaussian distribution.

Notes

New in version 0.3.

References

[1]

5.12.6 pyriemann.datasets.generate_random_spd_matrix

pyriemann.datasets.**generate_random_spd_matrix**(*n_dim*, *random_state=None*, ***, *mat_mean=0.0*, *mat_std=1.0*)

Generate a random SPD matrix.

Parameters

n_dim
[int] Dimensionality of the matrix to sample.

random_state
[int, RandomState instance or None, default=None] Pass an int for reproducible output across multiple function calls.

mat_mean
[float, default=0.0] Mean of random values to generate matrix.

mat_std
[float, default=1.0] Standard deviation of random values to generate matrix.

Returns

C
[ndarray, shape (n_dim, n_dim)] Random SPD matrix.

Notes

New in version 0.3.

5.12.7 pyriemann.datasets.make_classification_transfer

pyriemann.datasets.**make_classification_transfer**(*n_matrices*, *class_sep=3.0*, *class_disp=1.0*, *domain_sep=5.0*, *theta=0.0*, *stretch=1.0*, *random_state=None*, *class_names=[1, 2]*)

Generate source and target toy datasets for transfer learning examples.

Generate a dataset with 2x2 SPD matrices drawn from two Riemannian Gaussian distributions. The distributions have the same class dispersions and the distance between their centers of mass is an input parameter. We can stretch the target dataset and control a rotation matrix that maps the source to the target domains. This function is useful for testing classification or clustering methods on transfer learning applications.

Parameters

n_matrices
[int, default=100] How many 2x2 matrices to generate for each class on each domain.

class_sep
[float, default=3.0] Distance between the centers of the two classes.

class_disp
[float, default=1.0] Dispersion of the data points to be sampled on each class.

domain_sep
[float, default=5.0] Distance between the global means of each source and target datasets.

theta
[float, default=0.0] Angle of the 2x2 rotation matrix from source to target dataset.

stretch
[float, default=1.0] Factor to stretch the data points in target dataset. Note that when it is != 1.0 the class dispersions in target domain will be different than those in source domain (fixed at class_disp).

random_state
[None | int | RandomState instance, default=None] Pass an int for reproducible output across multiple function calls.

class_names
[list, default=[1, 2]] Names of classes.

Returns

X_enc
[ndarray, shape (4*n_matrices, 2, 2)] Set of SPD matrices.

y_enc
[ndarray, shape (4*n_matrices,)] Extended labels for each data point.

Notes

New in version 0.3.1.

5.13 Utils function

Utils functions are low level functions that implement most base components of Riemannian Geometry.

5.13.1 Covariance preprocessing

covariances(X[, estimator])	Estimation of covariance matrix.
covariance_mest(X, m_estimator, *[, init, ...])	Robust M-estimators.
covariance_sch(X)	Schaefer-Strimmer shrunk covariance estimator.
covariances_EP(X, P[, estimator])	Special form covariance matrix, concatenating a prototype P.
covariances_X(X[, estimator, alpha])	Special form covariance matrix, embedding input X.
block_covariances(X, blocks[, estimator])	Compute block diagonal covariance.
cross_spectrum(X[, window, overlap, fmin, ...])	Compute the complex cross-spectral matrices of a real signal X.
cospectrum(X[, window, overlap, fmin, fmax, fs])	Compute co-spectral matrices, the real part of cross-spectra.
coherence(X[, window, overlap, fmin, fmax, ...])	Compute squared coherence.
normalize(X, norm)	Normalize a set of square matrices, using corr, trace or determinant.
get_nondiag_weight(X)	Compute non-diagonality weights of a set of square matrices.

pyriemann.utils.covariance.covariances

pyriemann.utils.covariance.**covariances**(*X*, *estimator='cov'*, ***kwds*)

Estimation of covariance matrix.

Parameters

X

[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

estimator

[{'corr', 'cov', 'hub', 'lwf', 'mcd', 'oas', 'sch', 'scm', 'stu', 'tyl'}, default='scm'] Covariance matrix estimator [est]:

- 'corr' for correlation coefficient matrix [corr] supporting complex inputs,
- 'cov' for numpy based covariance matrix [cov] supporting complex inputs,
- 'hub' for Huber's M-estimator based covariance matrix [mest] supporting complex inputs,
- 'lwf' for Ledoit-Wolf shrunk covariance matrix [lwf],
- 'mcd' for minimum covariance determinant matrix [mcd],
- 'oas' for oracle approximating shrunk covariance matrix [oas],
- 'sch' for Schaefer-Strimmer shrunk covariance matrix [sch],
- 'scm' for sample covariance matrix [scm],
- 'stu' for Student-t's M-estimator based covariance matrix [mest] supporting complex inputs,
- 'tyl' for Tyler's M-estimator based covariance matrix [mest] supporting complex inputs,

- or a callable function.

For regularization, consider 'lwf' or 'oas'. For robustness, consider 'hub', 'mcd', 'stu' or 'tyl'.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

Returns

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Covariance matrices.

References

[est], [corr], [cov], [lwf], [mcd], [mest], [oas], [sch], [scm]

pyriemann.utils.covariance.covariance_mest

pyriemann.utils.covariance.**covariance_mest**(*X*, *m_estimator*, ***, *init=None*, *tol=0.01*, *n_iter_max=50*, *assume_centered=False*, *q=0.9*, *nu=5*, *norm='trace'*)

Robust M-estimators.

Robust M-estimator based covariance matrix [1], computed by fixed point algorithm.

For an input time series $X \in \mathbb{R}^{c \times t}$, composed of c channels and t time samples,

$$C = \frac{1}{t} \sum_i \varphi(X[:,i]^H C^{-1} X[:,i]) X[:,i] X[:,i]^H$$

where $\varphi()$ is a function allowing to weight the squared Mahalanobis distance depending on the M-estimator type: Huber, Student-t or Tyler.

Parameters

X
[ndarray, shape (n_channels, n_times)] Multi-channel time-series, real or complex-valued.

m_estimator
[{'hub', 'stu', 'tyl'}] Type of M-estimator:

- 'hub' for Huber's M-estimator [2];
- 'stu' for Student-t's M-estimator [3];
- 'tyl' for Tyler's M-estimator [4].

init
[None | ndarray, shape (n_channels, n_channels), default=None] A matrix used to initialize the algorithm. If None, the sample covariance matrix is used.

tol
[float, default=10e-3] The tolerance to stop the fixed point estimation.

n_iter_max
[int, default=50] The maximum number of iterations.

assume_centered
[bool, default=False] If *True*, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If *False*, data will be centered before computation.

q
[float, default=0.9] Using Huber's M-estimator, q is the percentage in (0, 1] of inputs deemed uncorrupted, while (1-q) is the percentage of inputs treated as outliers w.r.t a Gaussian distribution. This estimator is a trade-off between Tyler's estimator (q=0) and the sample covariance matrix (q=1).

nu
[int, default=5] Using Student-t's M-estimator, degree of freedom for t-distribution (strictly positive). This estimator is a trade-off between Tyler's estimator (nu->0) and the sample covariance matrix (nu->inf).

norm
[{"trace", "determinant"}, default="trace"] Using Tyler's M-estimator, the type of normalization:

- 'trace': trace of covariance matrix is `n_channels`;
- 'determinant': determinant of covariance matrix is 1.

Returns

cov
[ndarray, shape (n_channels, n_channels)] Robust M-estimator based covariance matrix.

Notes

New in version 0.3.1.

References

[1], [2], [3], [4]

pyriemann.utils.covariance.covariance_sch

pyriemann.utils.covariance.**covariance_sch**(*X*)

Schaefer-Strimmer shrunk covariance estimator.

Shrinkage covariance estimator [1]:

$$C = (1 - \gamma)C_{\text{scm}} + \gamma T$$

where T is the diagonal target matrix:

$$T[i, j] = \{C_{\text{scm}}[i, i] \text{ if} i = j, 0 \text{otherwise}\}$$

Note that the optimal γ is estimated by the authors' method.

Parameters

X
[ndarray, shape (n_channels, n_times)] Multi-channel time-series.

Returns

cov
[ndarray, shape (n_channels, n_channels)] Schaefer-Strimmer shrunk covariance matrix.

Notes

New in version 0.3.

References

[1]

pyriemann.utils.covariance.covariances_EP

pyriemann.utils.covariance.**covariances_EP**(*X*, *P*, *estimator='cov'*, ***kwds*)

Special form covariance matrix, concatenating a prototype P.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

P
[ndarray, shape (n_channels_proto, n_times)] Multi-channel prototype.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

Returns

covmats
[ndarray, shape (n_matrices, n_channels + n_channels_proto, n_channels + n_channels_proto)] Covariance matrices.

pyriemann.utils.covariance.covariances_X

pyriemann.utils.covariance.**covariances_X**(*X*, *estimator='scm'*, *alpha=0.2*, ***kwds*)

Special form covariance matrix, embedding input X.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

alpha
[float, default=0.2] Regularization parameter (strictly positive).

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

Returns

covmats
[ndarray, shape (n_matrices, n_channels + n_times, n_channels + n_times)] Covariance matrices.

References

[1]

pyriemann.utils.covariance.block_covariances

pyriemann.utils.covariance.**block_covariances**(*X*, *blocks*, *estimator='cov'*, ***kwds*)

Compute block diagonal covariance.

Calculates block diagonal matrices where each block is a covariance matrix of a subset of channels. Block sizes are passed as a list of integers and can vary. The sum of block sizes must equal the number of channels in X.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_times)] Multi-channel time-series.

blocks: list of int
List of block sizes.

estimator
[string, default='scm'] Covariance matrix estimator, see *pyriemann.utils.covariance.covariances()*.

****kwds**
[optional keyword parameters] Any further parameters are passed directly to the covariance estimator.

Returns

C
[ndarray, shape (n_matrices, n_channels, n_channels)] Block diagonal covariance matrices.

pyriemann.utils.covariance.cross_spectrum

pyriemann.utils.covariance.**cross_spectrum**(*X*, *window=128*, *overlap=0.75*, *fmin=None*, *fmax=None*, *fs=None*)

Compute the complex cross-spectral matrices of a real signal X.

Parameters

X
[ndarray, shape (n_channels, n_times)] Multi-channel time-series.

window
[int, default=128] The length of the FFT window used for spectral estimation.

overlap
[float, default=0.75] The percentage of overlap between window.

fmin
[float | None, default=None] The minimal frequency to be returned.

fmax
[float | None, default=None] The maximal frequency to be returned.

fs
[float | None, default=None] The sampling frequency of the signal.

Returns

S
[ndarray, shape (n_channels, n_channels, n_freqs)] Cross-spectral matrices, for each frequency bin.

freqs
[ndarray, shape (n_freqs,)] The frequencies associated to cross-spectra.

References

[1]

pyriemann.utils.covariance.cospectrum

pyriemann.utils.covariance.**cospectrum**(*X*, *window=128*, *overlap=0.75*, *fmin=None*, *fmax=None*, *fs=None*)

Compute co-spectral matrices, the real part of cross-spectra.

Parameters

X
[ndarray, shape (n_channels, n_times)] Multi-channel time-series.

window
[int, default=128] The length of the FFT window used for spectral estimation.

overlap
[float, default=0.75] The percentage of overlap between window.

fmin
[float | None, default=None] The minimal frequency to be returned.

fmax
[float | None, default=None] The maximal frequency to be returned.

fs
[float | None, default=None] The sampling frequency of the signal.

Returns

S
[ndarray, shape (n_channels, n_channels, n_freqs)] Co-spectral matrices, for each frequency bin.

freqs
[ndarray, shape (n_freqs,)] The frequencies associated to cospectra.

pyriemann.utils.covariance.coherence

pyriemann.utils.covariance.**coherence**(*X*, *window=128*, *overlap=0.75*, *fmin=None*, *fmax=None*, *fs=None*, *coh='ordinary'*)

Compute squared coherence.

Parameters

X
[ndarray, shape (n_channels, n_times)] Multi-channel time-series.

window
[int, default=128] The length of the FFT window used for spectral estimation.

overlap
[float, default=0.75] The percentage of overlap between window.

fmin
[float | None, default=None] The minimal frequency to be returned.

fmax
[float | None, default=None] The maximal frequency to be returned.

fs
[float | None, default=None] The sampling frequency of the signal.

coh
[{'ordinary', 'instantaneous', 'lagged', 'imaginary'}, default='ordinary'] The coherence type, see *pyriemann.estimation.Coherences*.

Returns

C
[ndarray, shape (n_channels, n_channels, n_freqs)] Squared coherence matrices, for each frequency bin.

freqs
[ndarray, shape (n_freqs,)] The frequencies associated to coherence.

pyriemann.utils.covariance.normalize

pyriemann.utils.covariance.**normalize**(*X*, *norm*)

Normalize a set of square matrices, using corr, trace or determinant.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray. Matrices must be invertible for determinant-normalization.

norm
[{"corr", "trace", "determinant"}] The type of normalization:

- 'corr': normalized matrices are correlation matrices, with values in [-1, 1] and diagonal values equal to 1;
- 'trace': trace of normalized matrices is 1;
- 'determinant': determinant of normalized matrices is +/- 1.

Returns

Xn
[ndarray, shape (..., n, n)] The set of normalized matrices, same dimensions as X.

pyriemann.utils.covariance.get_nondiag_weight

pyriemann.utils.covariance.**get_nondiag_weight**(*X*)

Compute non-diagonality weights of a set of square matrices.

Compute non-diagonality weights of a set of square matrices, following Eq(B.1) in [1].

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

weights
[ndarray, shape (...,)] The non-diagonality weights for matrices.

References

[1]

5.13.2 Distances

distance(A, B[, metric])	Distance between SPD matrices according to a metric.
distance_euclid(A, B)	Euclidean distance between SPD matrices.
distance_harmonic(A, B)	Harmonic distance between SPD matrices.
distance_kullback(A, B)	Kullback-Leibler divergence between SPD matrices.
distance_kullback_sym(A, B)	Symmetrized Kullback-Leibler divergence between SPD matrices.
distance_logdet(A, B)	Log-det distance between SPD matrices.
distance_logeuclid(A, B)	Log-Euclidean distance between SPD matrices.
distance_riemann(A, B)	Affine-invariant Riemannian distance between SPD matrices.
distance_wasserstein(A, B)	Wasserstein distance between SPD matrices.
distance_mahalanobis(X, cov[, mean])	Mahalanobis distance between vectors and a Gaussian distribution.

pyriemann.utils.distance.distance

pyriemann.utils.distance.**distance**(*A*, *B*, *metric='riemann'*)

Distance between SPD matrices according to a metric.

Compute the distance between two SPD matrices A and B according to a metric, or between a set of SPD matrices A and a SPD matrix B.

Parameters

A
[ndarray, shape (n, n) or shape (n_matrices, n, n)] First SPD matrix.

B
[ndarray, shape (n, n)] Second SPD matrix.

metric
[string, default='riemann'] The metric for distance, can be: 'euclid', 'harmonic', 'kullback', 'kullback_right', 'kullback_sym', 'logdet', 'logeuclid', 'riemann', 'wasserstein', or a callable function.

Returns

d
[float or ndarray, shape (n_matrices, 1)] The distance(s) between A and B.

pyriemann.utils.distance.distance_euclid

pyriemann.utils.distance.**distance_euclid**(*A*, *B*)

Euclidean distance between SPD matrices.

The Euclidean distance between two SPD matrices A and B is defined as the Frobenius norm of the difference of the two matrices:

$$d(\mathbf{A}, \mathbf{B}) = \|\mathbf{A} - \mathbf{B}\|_F$$

Parameters

A
[ndarray, shape (..., n, n)] First SPD matrices, at least 2D ndarray.

B
[ndarray, shape (..., n, n)] Second SPD matrices, same dimensions as A.

Returns

d
[ndarray, shape (...,) or float] Euclidean distance between A and B.

pyriemann.utils.distance.distance_harmonic

pyriemann.utils.distance.**distance_harmonic**(*A*, *B*)

Harmonic distance between SPD matrices.

The harmonic distance between two SPD matrices A and B is:

$$d(\mathbf{A}, \mathbf{B}) = \|\mathbf{A}^{-1} - \mathbf{B}^{-1}\|_F$$

Parameters

A
[ndarray, shape (..., n, n)] First SPD matrices, at least 2D ndarray.

B
[ndarray, shape (..., n, n)] Second SPD matrices, same dimensions as A.

Returns

d
[ndarray, shape (...,) or float] Harmonic distance between A and B.

pyriemann.utils.distance.distance_kullback

pyriemann.utils.distance.**distance_kullback**(*A*, *B*)

Kullback-Leibler divergence between SPD matrices.

The left Kullback-Leibler divergence between two SPD matrices A and B is:

$$d(\mathbf{A}, \mathbf{B}) = \frac{1}{2}\left(\mathrm{tr}(\mathbf{B}^{-1}\mathbf{A}) - n + \log(\frac{\det(\mathbf{B})}{\det(\mathbf{A})})\right)$$

Parameters

A
[ndarray, shape (…, n, n)] First SPD matrices, at least 2D ndarray.

B
[ndarray, shape (…, n, n)] Second SPD matrices, same dimensions as A.

Returns

d
[ndarray, shape (…,) or float] Left Kullback-Leibler divergence between A and B.

pyriemann.utils.distance.distance_kullback_sym

pyriemann.utils.distance.**distance_kullback_sym**(*A*, *B*)

Symmetrized Kullback-Leibler divergence between SPD matrices.

The symmetrized Kullback-Leibler divergence between two SPD matrices A and B is the sum of left and right Kullback-Leibler divergences.

pyriemann.utils.distance.distance_logdet

pyriemann.utils.distance.**distance_logdet**(*A*, *B*)

Log-det distance between SPD matrices.

The log-det distance between two SPD matrices A and B is:

$$d(\mathbf{A}, \mathbf{B}) = \sqrt{\log(\det(\frac{\mathbf{A}+\mathbf{B}}{2})) - \frac{1}{2}\log(\det(\mathbf{AB}))}$$

Parameters

A
[ndarray, shape (…, n, n)] First SPD matrices, at least 2D ndarray.

B
[ndarray, shape (…, n, n)] Second SPD matrices, same dimensions as A.

Returns

d
[ndarray, shape (…,) or float] Log-det distance between A and B.

pyriemann.utils.distance.distance_logeuclid

pyriemann.utils.distance.**distance_logeuclid**(*A*, *B*)

Log-Euclidean distance between SPD matrices.

The Log-Euclidean distance between two SPD matrices A and B is:

$$d(\mathbf{A}, \mathbf{B}) = \| \log(\mathbf{A}) - \log(\mathbf{B}) \|_F$$

Parameters

A
[ndarray, shape (…, n, n)] First SPD matrices, at least 2D ndarray.

B
[ndarray, shape (…, n, n)] Second SPD matrices, same dimensions as A.

Returns

d
[ndarray, shape (…,) or float] Log-Euclidean distance between A and B.

pyriemann.utils.distance.distance_riemann

pyriemann.utils.distance.**distance_riemann**(*A*, *B*)

Affine-invariant Riemannian distance between SPD matrices.

The affine-invariant Riemannian distance between two SPD matrices A and B is:

$$d(\mathbf{A}, \mathbf{B}) = \left(\sum_i \log(\lambda_i)^2 \right)^{1/2}$$

where λ_i are the joint eigenvalues of $\mathbf{A}$ and $\mathbf{B}$.

Parameters

A
[ndarray, shape (…, n, n)] First SPD matrices, at least 2D ndarray.

B
[ndarray, shape (…, n, n)] Second SPD matrices, same dimensions as A.

Returns

d
[ndarray, shape (…,) or float] Affine-invariant Riemannian distance between A and B.

pyriemann.utils.distance.distance_wasserstein

pyriemann.utils.distance.**distance_wasserstein**(*A*, *B*)

Wasserstein distance between SPD matrices.

The Wasserstein distance between two SPD matrices A and B is:

$$d(\mathbf{A}, \mathbf{B}) = \sqrt{\text{tr}(A + B - 2(B^{1/2}AB^{1/2})^{1/2})}$$

Parameters

A

[ndarray, shape (..., n, n)] First SPD matrices, at least 2D ndarray.

B

[ndarray, shape (..., n, n)] Second SPD matrices, same dimensions as A.

Returns

d

[ndarray, shape (...,) or float] Wasserstein distance between A and B.

pyriemann.utils.distance.distance_mahalanobis

pyriemann.utils.distance.**distance_mahalanobis**(*X*, *cov*, *mean=None*)

Mahalanobis distance between vectors and a Gaussian distribution.

The Mahalanobis distance between a vector x and a Gaussian distribution $\mathcal{N}(\mu, C)$, with mean μ and covariance matrix C, is:

$$d(x, \mathcal{N}(\mu, C)) = \sqrt{(x-\mu)^H C^{-1} (x-\mu)}$$

Parameters

X

[ndarray, shape (n_channels, n_vectors)] Multi-channel vectors.

cov

[ndarray, shape (n_channels, n_channels)] Covariance matrix of the Gaussian distribution.

mean

[None | ndarray, shape (n_channels, 1), default=None] Mean of the Gaussian distribution. If None, distribution is considered as centered.

Returns

d

[ndarray, shape (n_vectors,)] Mahalanobis distances.

Notes

New in version 0.3.1.

References

[1]

5.13.3 Means

mean_covariance(covmats[, metric, sample_weight])	Mean of SPD matrices according to a metric.
mean_ale(covmats[, tol, maxiter, sample_weight])	AJD-based log-Euclidean (ALE) mean of SPD matrices.
mean_alm(covmats[, tol, maxiter, sample_weight])	Ando-Li-Mathias (ALM) mean of SPD matrices.
mean_euclid(covmats[, sample_weight])	Mean of matrices according to the Euclidean metric.
mean_harmonic(covmats[, sample_weight])	Harmonic mean of SPD matrices.
mean_identity(covmats[, sample_weight])	Identity matrix corresponding to the matrices dimension.
mean_kullback_sym(covmats[, sample_weight])	Mean of SPD matrices according to Kullback-Leibler divergence.
mean_logdet(covmats[, tol, maxiter, init, ...])	Mean of SPD matrices according to the log-det metric.
mean_logeuclid(covmats[, sample_weight])	Mean of SPD matrices according to the log-Euclidean metric.
mean_power(covmats, p, *[, sample_weight, ...])	Power mean of SPD matrices.
mean_riemann(covmats[, tol, maxiter, init, ...])	Mean of SPD matrices according to the Riemannian metric.
mean_wasserstein(covmats[, tol, maxiter, ...])	Mean of SPD matrices according to the Wasserstein metric.
maskedmean_riemann(covmats, masks[, tol, ...])	Masked Riemannian mean of SPD matrices.
nanmean_riemann(covmats[, tol, maxiter, ...])	Riemannian NaN-mean of SPD matrices.

pyriemann.utils.mean.mean_covariance

pyriemann.utils.mean.**mean_covariance**(*covmats*, *metric='riemann'*, *sample_weight=None*, **args*)

Mean of SPD matrices according to a metric.

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

metric
[string, default='riemann'] The metric for mean, can be: 'ale', 'alm', 'euclid', 'harmonic', 'identity', 'kullback_sym', 'logdet', 'logeuclid', 'riemann', 'wasserstein', or a callable function.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

args
[list of params] The arguments passed to the sub function.

Returns

C
[ndarray, shape (n_channels, n_channels)] Mean of SPD matrices.

pyriemann.utils.mean.mean_ale

pyriemann.utils.mean.**mean_ale**(*covmats*, *tol=1e-06*, *maxiter=50*, *sample_weight=None*)

AJD-based log-Euclidean (ALE) mean of SPD matrices.

Return the mean of a set of SPD matrices using the AJD-based log-Euclidean (ALE) mean [1].

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

tol
[float, default=10e-7] The tolerance to stop the gradient descent.

maxiter
[int, default=50] The maximum number of iterations.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] ALE mean.

Notes

New in version 0.2.4.

References

[1]

pyriemann.utils.mean.mean_alm

pyriemann.utils.mean.**mean_alm**(*covmats*, *tol=1e-14*, *maxiter=100*, *sample_weight=None*)

Ando-Li-Mathias (ALM) mean of SPD matrices.

Return the geometric mean recursively [1], generalizing from:

$$\mathbf{C} = A^{\frac{1}{2}}(A^{-\frac{1}{2}}B^{\frac{1}{2}}A^{-\frac{1}{2}})^{\frac{1}{2}}A^{\frac{1}{2}}$$

and requiring a high number of iterations.

This is the adaptation of the Matlab code proposed by Dario Bini and Bruno Iannazzo, http://bezout.dm.unipi.it/software/mmtoolbox/ . Extremely slow, due to the recursive formulation.

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

tol
[float, default=10e-14] The tolerance to stop the gradient descent.

maxiter
[int, default=100] The maximum number of iterations.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] ALM mean.

Notes

New in version 0.3.

References

[1]

pyriemann.utils.mean.mean_euclid

pyriemann.utils.mean.**mean_euclid**(*covmats*, *sample_weight=None*)

Mean of matrices according to the Euclidean metric.

$$\mathbf{C} = \frac{1}{m} \sum_i \mathbf{C}_i$$

This mean is also called arithmetic.

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of matrices.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Euclidean mean.

pyriemann.utils.mean.mean_harmonic

pyriemann.utils.mean.**mean_harmonic**(*covmats*, *sample_weight=None*)

Harmonic mean of SPD matrices.

$$\mathbf{C} = \left(\frac{1}{m} \sum_i {\mathbf{C}_i}^{-1} \right)^{-1}$$

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Harmonic mean.

pyriemann.utils.mean.mean_identity

pyriemann.utils.mean.**mean_identity**(*covmats*, *sample_weight=None*)

Identity matrix corresponding to the matrices dimension.

$$\mathbf{C} = \mathbf{I}_c$$

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

sample_weight
[None] Not used, here for compatibility with other means.

Returns

C
[ndarray, shape (n_channels, n_channels)] Identity matrix.

pyriemann.utils.mean.mean_kullback_sym

pyriemann.utils.mean.**mean_kullback_sym**(*covmats*, *sample_weight=None*)

Mean of SPD matrices according to Kullback-Leibler divergence.

Symmetrized Kullback-Leibler mean is the geometric mean between the Euclidean and the harmonic means, as shown in [1].

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Kullback-Leibler mean.

References

[1]

pyriemann.utils.mean.mean_logdet

pyriemann.utils.mean.**mean_logdet**(*covmats*, *tol=0.0001*, *maxiter=50*, *init=None*, *sample_weight=None*)

Mean of SPD matrices according to the log-det metric.

Log-det mean is obtained by an iterative procedure where the update is:

$$\mathbf{C} = \left(\sum_i (0.5\mathbf{C} + 0.5\mathbf{C}_i)^{-1} \right)^{-1}$$

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

tol
[float, default=10e-5] The tolerance to stop the gradient descent.

maxiter
[int, default=50] The maximum number of iterations.

init
[None | ndarray, shape (n_channels, n_channels), default=None] A SPD matrix used to initialize the gradient descent. If None, the weighted Euclidean mean is used.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Log-det mean.

pyriemann.utils.mean.mean_logeuclid

pyriemann.utils.mean.**mean_logeuclid**(*covmats*, *sample_weight=None*)

Mean of SPD matrices according to the log-Euclidean metric.

Log-Euclidean mean is [1]:

$$\mathbf{C} = \exp\left(\frac{1}{m} \sum_i \log \mathbf{C}_i\right)$$

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Log-Euclidean mean.

References

[1]

pyriemann.utils.mean.mean_power

pyriemann.utils.mean.**mean_power**(*covmats*, *p*, *, *sample_weight=None*, *zeta=1e-09*, *maxiter=100*)

Power mean of SPD matrices.

Power mean is the solution of [1] [2]:

$$\mathbf{C} = \frac{1}{m} \sum_i \mathbf{C} \sharp_p \mathbf{C}_i$$

where $\mathbf{A} \sharp_p \mathbf{B}$ is the geodesic between matrices $\mathbf{A}$ and $\mathbf{B}$.

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

p
[float] Exponent, in [-1,+1]. For p=0, it returns *pyriemann.utils.mean.mean_riemann()*.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

zeta
[float, default=10e-10] Stopping criterion.

maxiter
[int, default=100] The maximum number of iterations.

Returns

C
[ndarray, shape (n_channels, n_channels)] Power mean.

Notes

New in version 0.3.

References

[1], [2]

pyriemann.utils.mean.mean_riemann

pyriemann.utils.mean.**mean_riemann**(*covmats*, *tol=1e-08*, *maxiter=50*, *init=None*, *sample_weight=None*)

Mean of SPD matrices according to the Riemannian metric.

The affine-invariant Riemannian mean minimizes the sum of squared affine-invariant Riemannian distances d_R to all matrices [1]:

$$\arg\min_{\mathbf{C}} \sum_i w_i d_R(\mathbf{C}, \mathbf{C}_i)^2$$

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

tol
[float, default=10e-9] The tolerance to stop the gradient descent.

maxiter
[int, default=50] The maximum number of iterations.

init
[None | ndarray, shape (n_channels, n_channels), default=None] A SPD matrix used to initialize the gradient descent. If None, the weighted Euclidean mean is used.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Affine-invariant Riemannian mean.

References

[1]

pyriemann.utils.mean.mean_wasserstein

pyriemann.utils.mean.**mean_wasserstein**(*covmats*, *tol=0.001*, *maxiter=50*, *init=None*, *sample_weight=None*)

Mean of SPD matrices according to the Wasserstein metric.

Wasserstein mean is obtained by an iterative procedure where the update is [1]:

$$\mathbf{K} = \left(\sum_i (\mathbf{K}\mathbf{C}_i\mathbf{K})^{1/2} \right)^{1/2}$$

with $\mathbf{K} = \mathbf{C}^{1/2}$.

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

tol
[float, default=10e-4] The tolerance to stop the gradient descent.

maxiter
[int, default=50] The maximum number of iterations.

init
[None | ndarray, shape (n_channels, n_channels), default=None] A SPD matrix used to initialize the gradient descent. If None the Euclidean mean is used.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Wasserstein mean.

References

[1]

pyriemann.utils.mean.maskedmean_riemann

pyriemann.utils.mean.**maskedmean_riemann**(*covmats*, *masks*, *tol=1e-08*, *maxiter=100*, *init=None*, *sample_weight=None*)

Masked Riemannian mean of SPD matrices.

Given masks defined as semi-orthogonal matrices, the masked Riemannian mean of SPD matrices is obtained with a gradient descent minimizing the sum of affine-invariant Riemannian distances between masked SPD matrices and the masked mean [1].

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

masks
[list of n_matrices ndarray of shape (n_channels, n_channels_i), with different n_channels_i, such that n_channels_i <= n_channels] Masks, defined as semi-orthogonal matrices. See [1].

tol
[float, default=10e-9] The tolerance to stop the gradient descent.

maxiter
[int, default=100] The maximum number of iteration.

init
[None | ndarray, shape (n_channels, n_channels), default=None] A SPD matrix used to initialize the gradient descent. If None, the Identity is used.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Masked Riemannian mean.

Notes

New in version 0.3.

References

[1]

pyriemann.utils.mean.nanmean_riemann

pyriemann.utils.mean.**nanmean_riemann**(*covmats*, *tol=1e-08*, *maxiter=100*, *init=None*, *sample_weight=None*)

Riemannian NaN-mean of SPD matrices.

The Riemannian NaN-mean is the masked Riemannian mean applied to SPD matrices potentially corrupted by symmetric NaN values [1].

Parameters

covmats
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices, corrupted by symmetric NaN values [1].

tol
[float, default=10e-9] The tolerance to stop the gradient descent.

maxiter
[int, default=100] The maximum number of iteration.

init
[None | ndarray, shape (n_channels, n_channels), default=None] A SPD matrix used to initialize the gradient descent. If None, a regularized Euclidean NaN-mean is used.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

C
[ndarray, shape (n_channels, n_channels)] Riemannian NaN-mean.

Notes

New in version 0.3.

References

[1]

5.13.4 Medians

median_euclid(X, *[, tol, maxiter, init, ...])	Euclidean geometric median of matrices.
median_riemann(X, *[, tol, maxiter, init, ...])	Affine-invariant Riemannian geometric median of SPD matrices.

pyriemann.utils.median_euclid

pyriemann.utils.**median_euclid**(*X*, ***, *tol=1e-05*, *maxiter=50*, *init=None*, *weights=None*)

Euclidean geometric median of matrices.

The Euclidean geometric median minimizes the sum of Euclidean distances d_E to all matrices [1] [2]:

$$\arg\min_{\mathbf{M}} \sum_i w_i d_E(\mathbf{M}, \mathbf{X}_i)$$

It is different from the marginal median provided by NumPy [3].

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of matrices.

tol
[float, default=10e-6] The tolerance to stop the iterative algorithm.

maxiter
[int, default=50] The maximum number of iterations.

init
[None | ndarray, shape (n_channels, n_channels), default=None] A matrix used to initialize the iterative algorithm. If None, the weighted Euclidean mean is used.

weights
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

Returns

M
[ndarray, shape (n_channels, n_channels)] Euclidean geometric median.

Notes

New in version 0.3.1.

References

[1], [2], [3]

pyriemann.utils.median_riemann

pyriemann.utils.**median_riemann**(*X*, ***, *tol=1e-05*, *maxiter=50*, *init=None*, *weights=None*, *step_size=1*)

Affine-invariant Riemannian geometric median of SPD matrices.

The affine-invariant Riemannian geometric median minimizes the sum of affine-invariant Riemannian distances d_R to all SPD matrices [1]:

$$\arg\min_{\mathbf{M}} \sum_i w_i d_R(\mathbf{M}, \mathbf{X}_i)$$

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

tol
[float, default=10e-6] The tolerance to stop the gradient descent.

maxiter
[int, default=50] The maximum number of iterations.

init
[None | ndarray, shape (n_channels, n_channels), default=None] A SPD matrix used to initialize the gradient descent. If None, the weighted Euclidean mean is used.

weights
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix. If None, it uses equal weights.

step_size
[float, default=1.0] The step size of the gradient descent, in (0,2].

Returns

M
[ndarray, shape (n_channels, n_channels)] Affine-invariant Riemannian geometric median.

Notes

New in version 0.3.1.

References

[1], [2]

5.13.5 Geodesics

geodesic(A, B, alpha[, metric])	Geodesic between SPD matrices according to a metric.
geodesic_euclid(A, B[, alpha])	Euclidean geodesic between SPD matrices.
geodesic_logeuclid(A, B[, alpha])	Log-Euclidean geodesic between SPD matrices.
geodesic_riemann(A, B[, alpha])	Affine-invariant Riemannian geodesic between SPD matrices.

pyriemann.utils.geodesic.geodesic

pyriemann.utils.geodesic.**geodesic**(*A*, *B*, *alpha*, *metric='riemann'*)

Geodesic between SPD matrices according to a metric.

Return the matrix at the position alpha on the geodesic between SPD matrices A and B according to a metric.

Parameters

A
[ndarray, shape (..., n, n)] First SPD matrices.

B
[ndarray, shape (..., n, n)] Second SPD matrices.

alpha
[float] The position on the geodesic.

metric
[string, default='riemann'] The metric used for geodesic, can be: 'euclid', 'logeuclid', 'riemann'.

Returns

C
[ndarray, shape (..., n, n)] SPD matrices on the geodesic.

pyriemann.utils.geodesic.geodesic_euclid

pyriemann.utils.geodesic.**geodesic_euclid**(*A*, *B*, *alpha=0.5*)

Euclidean geodesic between SPD matrices.

The matrix at the position alpha on the Euclidean geodesic between two SPD matrices A and B is:

$$\mathbf{C} = (1 - \alpha)\mathbf{A} + \alpha\mathbf{B}$$

C is equal to A if alpha = 0 and B if alpha = 1.

Parameters

A
[ndarray, shape (..., n, n)] First SPD matrices.

B
[ndarray, shape (..., n, n)] Second SPD matrices.

alpha
[float, default=0.5] The position on the geodesic.

Returns

C
[ndarray, shape (..., n, n)] SPD matrices on the Euclidean geodesic.

pyriemann.utils.geodesic.geodesic_logeuclid

pyriemann.utils.geodesic.**geodesic_logeuclid**(*A*, *B*, *alpha=0.5*)

Log-Euclidean geodesic between SPD matrices.

The matrix at the position alpha on the Log-Euclidean geodesic between two SPD matrices A and B is:

$$\mathbf{C} = \exp\left((1-\alpha)\log(\mathbf{A}) + \alpha\log(\mathbf{B})\right)$$

C is equal to A if alpha = 0 and B if alpha = 1.

Parameters

A
[ndarray, shape (..., n, n)] First SPD matrices.

B
[ndarray, shape (..., n, n)] Second SPD matrices.

alpha
[float, default=0.5] The position on the geodesic.

Returns

C
[ndarray, shape (..., n, n)] SPD matrices on the Log-Euclidean geodesic.

pyriemann.utils.geodesic.geodesic_riemann

pyriemann.utils.geodesic.**geodesic_riemann**(*A*, *B*, *alpha=0.5*)

Affine-invariant Riemannian geodesic between SPD matrices.

The matrix at the position alpha on the affine-invariant Riemannian geodesic between two SPD matrices A and B is:

$$\mathbf{C} = \mathbf{A}^{1/2}\left(\mathbf{A}^{-1/2}\mathbf{B}\mathbf{A}^{-1/2}\right)^{\alpha}\mathbf{A}^{1/2}$$

C is equal to A if alpha = 0 and B if alpha = 1.

Parameters

A
[ndarray, shape (..., n, n)] First SPD matrices.

B
[ndarray, shape (..., n, n)] Second SPD matrices.

alpha
[float, default=0.5] The position on the geodesic.

Returns

C
[ndarray, shape (..., n, n)] SPD matrices on the affine-invariant Riemannian geodesic.

5.13.6 Kernels

kernel(X[, Y, Cref, metric, reg])	Kernel matrix between SPD matrices according to a specified metric.
kernel_euclid(X[, Y, reg])	Euclidean kernel between two sets of SPD matrices.
kernel_logeuclid(X[, Y, reg])	Log-Euclidean kernel between two sets of SPD matrices.
kernel_riemann(X[, Y, Cref, reg])	Affine-invariant Riemannian kernel between two sets of SPD matrices.

pyriemann.utils.kernel.kernel

pyriemann.utils.kernel.**kernel**(*X*, *Y=None*, ***, *Cref=None*, *metric='riemann'*, *reg=1e-10*)

Kernel matrix between SPD matrices according to a specified metric.

Calculates the kernel matrix K of inner products of two sets X and Y of SPD matrices on the tangent space of Cref according to a specified metric.

Parameters

X
[ndarray, shape (n_matrices_X, n_channels, n_channels)] First set of SPD matrices.

Y
[None | ndarray, shape (n_matrices_Y, n_channels, n_channels), default=None] Second set of SPD matrices. If None, Y is set to X.

Cref
[None | ndarray, shape (n_channels, n_channels), default=None] Reference point for the tangent space and inner product calculation. Only used if metric='riemann'.

metric
[{'euclid', 'logeuclid', 'riemann'}, default='riemann'] The type of metric used for tangent space and mean estimation.

reg
[float, default=1e-10] Regularization parameter to mitigate numerical errors in kernel matrix estimation, to provide a positive-definite kernel matrix.

Returns

K
[ndarray, shape (n_matrices_X, n_matrices_Y)] The kernel matrix between X and Y.

See also:

kernel_euclid
kernel_logeuclid
kernel_riemann

Notes

New in version 0.3.

pyriemann.utils.kernel.kernel_euclid

pyriemann.utils.kernel.**kernel_euclid**(*X*, *Y=None*, ***, *reg=1e-10*, ***kwargs*)

Euclidean kernel between two sets of SPD matrices.

Calculates the Euclidean kernel matrix K of inner products of two sets X and Y of SPD matrices by calculating pairwise:

$$K_{i,j} = \text{tr}(X_i Y_j)$$

Parameters

X
[ndarray, shape (n_matrices_X, n_channels, n_channels)] First set of SPD matrices.

Y
[None | ndarray, shape (n_matrices_Y, n_channels, n_channels), default=None] Second set of SPD matrices. If None, Y is set to X.

reg
[float, default=1e-10] Regularization parameter to mitigate numerical errors in kernel matrix estimation.

Returns

K
[ndarray, shape (n_matrices_X, n_matrices_Y)] The Euclidean kernel matrix between X and Y.

See also:

kernel

Notes

New in version 0.3.

pyriemann.utils.kernel.kernel_logeuclid

pyriemann.utils.kernel.**kernel_logeuclid**(*X*, *Y=None*, ***, *reg=1e-10*, ***kwargs*)

Log-Euclidean kernel between two sets of SPD matrices.

Calculates the Log-Euclidean kernel matrix K of inner products of two sets X and Y of SPD matrices by calculating pairwise [1]:

$$K_{i,j} = \text{tr}(\log(X_i)\log(Y_j))$$

Parameters

X
[ndarray, shape (n_matrices_X, n_channels, n_channels)] First set of SPD matrices.

Y

[None | ndarray, shape (n_matrices_Y, n_channels, n_channels), default=None] Second set of SPD matrices. If None, Y is set to X.

reg

[float, default=1e-10] Regularization parameter to mitigate numerical errors in kernel matrix estimation.

Returns

K

[ndarray, shape (n_matrices_X, n_matrices_Y)] The Log-Euclidean kernel matrix between X and Y.

See also:

kernel

Notes

New in version 0.3.

References

[1]

pyriemann.utils.kernel.kernel_riemann

pyriemann.utils.kernel.**kernel_riemann**(*X*, *Y=None*, ***, *Cref=None*, *reg=1e-10*)

Affine-invariant Riemannian kernel between two sets of SPD matrices.

Calculates the affine-invariant Riemannian kernel matrix K of inner products of two sets X and Y of SPD matrices on tangent space of Cref by calculating pairwise [1]:

$$K_{i,j} = \text{tr}(\log(C_{\text{ref}}^{-1/2} X_i C_{\text{ref}}^{-1/2}) \log(C_{\text{ref}}^{-1/2} Y_j C_{\text{ref}}^{-1/2}))$$

Parameters

X

[ndarray, shape (n_matrices_X, n_channels, n_channels)] First set of SPD matrices.

Y

[None | ndarray, shape (n_matrices_Y, n_channels, n_channels), default=None] Second set of SPD matrices. If None, Y is set to X.

Cref

[None | ndarray, shape (n_channels, n_channels), default=None] Reference point for the tangent space and inner product calculation. If None, Cref is calculated as the Riemannian mean of X.

reg

[float, default=1e-10] Regularization parameter to mitigate numerical errors in kernel matrix estimation.

Returns

K

[ndarray, shape (n_matrices_X, n_matrices_Y)] The affine-invariant Riemannian kernel matrix between X and Y.

See also:

kernel

Notes

New in version 0.3.

References

[1]

5.13.7 Tangent Space

exp_map_euclid(X, Cref)	Project matrices back to SPD manifold by Euclidean exponential map.
exp_map_logeuclid(X, Cref)	Project matrices back to SPD manifold by Log-Euclidean exponential map.
exp_map_riemann(X, Cref[, Cm12])	Project matrices back to SPD manifold by Riemannian exponential map.
log_map_euclid(X, Cref)	Project SPD matrices in tangent space by Euclidean logarithmic map.
log_map_logeuclid(X, Cref)	Project SPD matrices in tangent space by Log-Euclidean logarithmic map.
log_map_riemann(X, Cref[, C12])	Project SPD matrices in tangent space by Riemannian logarithmic map.
upper(X)	Return the weighted upper triangular part of symmetric matrices.
unupper(T)	Inverse upper function.
tangent_space(X, Cref, *[, metric])	Transform SPD matrices into tangent vectors.
untangent_space(T, Cref, *[, metric])	Transform tangent vectors back to SPD matrices.

pyriemann.utils.tangentspace.exp_map_euclid

pyriemann.utils.tangentspace.**exp_map_euclid**(*X*, *Cref*)

Project matrices back to SPD manifold by Euclidean exponential map.

The projection of a matrix X back to the SPD manifold with Euclidean exponential map according to a reference matrix $\mathbf{C}_{\text{ref}}$ is:

$$\mathbf{X}_{\text{original}} = \mathbf{X} + \mathbf{C}_{\text{ref}}$$

Parameters

X

[ndarray, shape (..., n_channels, n_channels)] Matrices in tangent space.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

Returns

X_original
[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

Notes

New in version 0.3.1.

pyriemann.utils.tangentspace.exp_map_logeuclid

pyriemann.utils.tangentspace.**exp_map_logeuclid**(*X*, *Cref*)

Project matrices back to SPD manifold by Log-Euclidean exponential map.

The projection of a matrix X back to the SPD manifold with Log-Euclidean exponential map according to a reference matrix $\mathbf{C}_{\text{ref}}$ is:

$$\mathbf{X}_{\text{original}} = \exp(\mathbf{X} + \log(\mathbf{C}_{\text{ref}}))$$

Parameters

X
[ndarray, shape (..., n_channels, n_channels)] Matrices in tangent space.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

Returns

X_original
[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

Notes

New in version 0.3.1.

pyriemann.utils.tangentspace.exp_map_riemann

pyriemann.utils.tangentspace.**exp_map_riemann**(*X*, *Cref*, *Cm12=False*)

Project matrices back to SPD manifold by Riemannian exponential map.

The projection of a matrix X back to the SPD manifold with Riemannian exponential map according to a reference matrix $\mathbf{C}_{\text{ref}}$ is:

$$\mathbf{X}_{\text{original}} = \mathbf{C}_{\text{ref}}^{1/2} \exp(\mathbf{X}) \mathbf{C}_{\text{ref}}^{1/2}$$

When Cm12=True, it returns the full Riemannian exponential map:

$$\mathbf{X}_{\text{original}} = \mathbf{C}_{\text{ref}}^{1/2} \exp(\mathbf{C}_{\text{ref}}^{-1/2} \mathbf{X} \mathbf{C}_{\text{ref}}^{-1/2}) \mathbf{C}_{\text{ref}}^{1/2}$$

Parameters

X
[ndarray, shape (..., n_channels, n_channels)] Matrices in tangent space.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

Cm12
[bool, default=False] If True, it returns the full Riemannian exponential map.

Returns

X_original
[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

Notes

New in version 0.3.1.

pyriemann.utils.tangentspace.log_map_euclid

pyriemann.utils.tangentspace.**log_map_euclid**(*X*, *Cref*)

Project SPD matrices in tangent space by Euclidean logarithmic map.

The projection of a SPD matrix X in the tangent space by Euclidean logarithmic map according to a reference matrix $\mathbf{C}_{\text{ref}}$ is:

$$\mathbf{X}_{\text{new}} = \mathbf{X} - \mathbf{C}_{\text{ref}}$$

Parameters

X
[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

Returns

X_new
[ndarray, shape (..., n_channels, n_channels)] SPD matrices projected in tangent space.

Notes

New in version 0.3.1.

pyriemann.utils.tangentspace.log_map_logeuclid

pyriemann.utils.tangentspace.**log_map_logeuclid**(*X*, *Cref*)

Project SPD matrices in tangent space by Log-Euclidean logarithmic map.

The projection of a SPD matrix X in the tangent space by Log-Euclidean logarithmic map according to a reference matrix $\mathbf{C}_{\text{ref}}$ is:

$$\mathbf{X}_{\text{new}} = \log(\mathbf{X}) - \log(\mathbf{C}_{\text{ref}})$$

Parameters

X
[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

Returns

X_new
[ndarray, shape (..., n_channels, n_channels)] SPD matrices projected in tangent space.

Notes

New in version 0.3.1.

pyriemann.utils.tangentspace.log_map_riemann

pyriemann.utils.tangentspace.**log_map_riemann**(*X*, *Cref*, *C12=False*)

Project SPD matrices in tangent space by Riemannian logarithmic map.

The projection of a SPD matrix X in the tangent space by Riemannian logarithmic map according to a reference matrix $\mathbf{C}_{\text{ref}}$ is:

$$\mathbf{X}_{\text{new}} = \log(\mathbf{C}_{\text{ref}}^{-1/2}\mathbf{X}\mathbf{C}_{\text{ref}}^{-1/2})$$

When C12=True, it returns the full Riemannian logarithmic map:

$$\mathbf{X}_{\text{new}} = \mathbf{C}_{\text{ref}}^{1/2}\log(\mathbf{C}_{\text{ref}}^{-1/2}\mathbf{X}\mathbf{C}_{\text{ref}}^{-1/2})\mathbf{C}_{\text{ref}}^{1/2}$$

Parameters

X
[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

C12
[bool, default=False] If True, it returns the full Riemannian logarithmic map.

Returns

X_new
[ndarray, shape (..., n_channels, n_channels)] SPD matrices projected in tangent space.

Notes

New in version 0.3.1.

pyriemann.utils.tangentspace.upper

pyriemann.utils.tangentspace.**upper**(*X*)

Return the weighted upper triangular part of symmetric matrices.

This function computes the minimal representation of a matrix in tangent space [1]: it keeps the upper triangular part of the symmetric matrix and vectorizes it by applying unity weight for diagonal elements and $\sqrt{2}$ weight for out-of-diagonal elements.

Parameters

X

[ndarray, shape (..., n_channels, n_channels)] Symmetric matrices.

Returns

T

[ndarray, shape (..., n_channels * (n_channels + 1) / 2)] Weighted upper triangular parts of symmetric matrices.

Notes

New in version 0.3.1.

References

[1]

pyriemann.utils.tangentspace.unupper

pyriemann.utils.tangentspace.**unupper**(*T*)

Inverse upper function.

This function is the inverse of upper function: it computes symmetric matrices from their weighted upper triangular parts.

Parameters

T

[ndarray, shape (..., n_channels * (n_channels + 1) / 2)] Weighted upper triangular parts of symmetric matrices.

Returns

X

[ndarray, shape (..., n_channels, n_channels)] Symmetric matrices.

See also:

upper

Notes

New in version 0.3.1.

pyriemann.utils.tangentspace.tangent_space

pyriemann.utils.tangentspace.**tangent_space**(*X*, *Cref*, ***, *metric='riemann'*)

Transform SPD matrices into tangent vectors.

Transform SPD matrices into tangent vectors, according to a reference matrix Cref and to a specific logarithmic map.

Parameters

X
[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

metric
[string, default='riemann'] The metric used for logarithmic map, can be: 'euclid', 'logeuclid', 'riemann'.

Returns

T
[ndarray, shape (..., n_channels * (n_channels + 1) / 2)] Tangent vectors.

See also:

log_map_euclid
log_map_logeuclid
log_map_riemann
upper

pyriemann.utils.tangentspace.untangent_space

pyriemann.utils.tangentspace.**untangent_space**(*T*, *Cref*, ***, *metric='riemann'*)

Transform tangent vectors back to SPD matrices.

Transform tangent vectors back to SPD matrices, according to a reference matrix Cref and to a specific exponential map.

Parameters

T
[ndarray, shape (..., n_channels * (n_channels + 1) / 2)] Tangent vectors.

Cref
[ndarray, shape (n_channels, n_channels)] The reference SPD matrix.

metric
[string, default='riemann'] The metric used for exponential map, can be: 'euclid', 'logeuclid', 'riemann'.

Returns

X

[ndarray, shape (..., n_channels, n_channels)] SPD matrices.

See also:

unupper
exp_map_euclid
exp_map_logeuclid
exp_map_riemann

5.13.8 Base

sqrtm(C)	Square root of SPD matrices.
invsqrtm(C)	Inverse square root of SPD matrices.
expm(C)	Exponential of SPD matrices.
logm(C)	Logarithm of SPD matrices.
powm(C, alpha)	Power of SPD matrices.
nearest_sym_pos_def(X[, reg])	Find the nearest SPD matrices.

pyriemann.utils.base.sqrtm

pyriemann.utils.base.**sqrtm**(*C*)

Square root of SPD matrices.

The matrix square root of a SPD matrix C is defined by:

$$\mathbf{D} = \mathbf{V} \left(\mathbf{\Lambda} \right)^{1/2} \mathbf{V}^\top$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{C}$.

Parameters

C

[ndarray, shape (..., n, n)] SPD matrices, at least 2D ndarray.

Returns

D

[ndarray, shape (..., n, n)] Matrix square root of C.

pyriemann.utils.base.invsqrtm

pyriemann.utils.base.**invsqrtm**(*C*)

Inverse square root of SPD matrices.

The matrix inverse square root of a SPD matrix C is defined by:

$$\mathbf{D} = \mathbf{V} \left(\mathbf{\Lambda} \right)^{-1/2} \mathbf{V}^\top$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{C}$.

Parameters

C

[ndarray, shape (..., n, n)] SPD matrices, at least 2D ndarray.

Returns

D
[ndarray, shape (..., n, n)] Matrix inverse square root of C.

pyriemann.utils.base.expm

pyriemann.utils.base.**expm**(*C*)

Exponential of SPD matrices.

The matrix exponential of a SPD matrix C is defined by:

$$\mathbf{D} = \mathbf{V} \exp{(\mathbf{\Lambda})} \mathbf{V}^\top$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{C}$.

Parameters

C
[ndarray, shape (..., n, n)] SPD matrices, at least 2D ndarray.

Returns

D
[ndarray, shape (..., n, n)] Matrix exponential of C.

pyriemann.utils.base.logm

pyriemann.utils.base.**logm**(*C*)

Logarithm of SPD matrices.

The matrix logarithm of a SPD matrix C is defined by:

$$\mathbf{D} = \mathbf{V} \log{(\mathbf{\Lambda})} \mathbf{V}^\top$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{C}$.

Parameters

C
[ndarray, shape (..., n, n)] SPD matrices, at least 2D ndarray.

Returns

D
[ndarray, shape (..., n, n)] Matrix logarithm of C.

pyriemann.utils.base.powm

pyriemann.utils.base.**powm**(*C*, *alpha*)

Power of SPD matrices.

The matrix power α of a SPD matrix C is defined by:

$$\mathbf{D} = \mathbf{V} \left(\mathbf{\Lambda}\right)^\alpha \mathbf{V}^\top$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues and $\mathbf{V}$ the eigenvectors of $\mathbf{C}$.

Parameters

C
[ndarray, shape (…, n, n)] SPD matrices, at least 2D ndarray.

alpha
[float] The power to apply.

Returns

D
[ndarray, shape (…, n, n)] Matrix power of C.

pyriemann.utils.base.nearest_sym_pos_def

pyriemann.utils.base.**nearest_sym_pos_def**(*X*, *reg=1e-06*)

Find the nearest SPD matrices.

A NumPy port of John D'Errico's *nearestSPD* MATLAB code [1], which credits [2].

Parameters

X
[ndarray, shape (…, n, n)] Square matrices, at least 2D ndarray.

reg
[float] Regularization parameter.

Returns

P
[ndarray, shape (…, n, n)] Nearest SPD matrices.

Notes

New in version 0.3.1.

References

[1], [2]

5.13.9 Aproximate Joint Diagonalization

ajd_pham(X, *[, init, eps, n_iter_max, ...])	Approximate joint diagonalization based on Pham's algorithm.
rjd(X, *[, init, eps, n_iter_max])	Approximate joint diagonalization based on Jacobi angles.
uwedge(X, *[, init, eps, n_iter_max])	Approximate joint diagonalization based on UWEDGE.

pyriemann.utils.ajd.ajd_pham

pyriemann.utils.ajd.**ajd_pham**(*X*, ***, *init=None*, *eps=1e-06*, *n_iter_max=15*, *sample_weight=None*)

Approximate joint diagonalization based on Pham's algorithm.

This is a direct implementation of the Pham's AJD algorithm [1].

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices to diagonalize.

init
[None | ndarray, shape (n_channels, n_channels), default=None] Initialization for the diagonalizer.

eps
[float, default=1e-6] Tolerance for stoping criterion.

n_iter_max
[int, default=15] The maximum number of iterations to reach convergence.

sample_weight
[None | ndarray, shape (n_matrices,), default=None] Weights for each matrix, strictly positive. If None, it uses equal weights.

Returns

V
[ndarray, shape (n_channels, n_channels)] The diagonalizer, an invertible matrix.

D
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of quasi diagonal matrices.

See also:

rjd
uwedge

Notes

New in version 0.2.4.

References

[1]

pyriemann.utils.ajd.rjd

pyriemann.utils.ajd.**rjd**(*X*, ***, *init=None*, *eps=1e-08*, *n_iter_max=1000*)

Approximate joint diagonalization based on Jacobi angles.

This is a direct implementation of the AJD algorithm by Cardoso and Souloumiac [1] used in JADE. The code is a translation of the Matlab code provided in the author website.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of symmetric matrices to diagonalize.

init
[None | ndarray, shape (n_channels, n_channels), default=None] Initialization for the diagonalizer.

eps
[float, default=1e-8] Tolerance for stopping criterion.

n_iter_max
[int, default=1000] The maximum number of iterations to reach convergence.

Returns

V
[ndarray, shape (n_channels, n_channels)] The diagonalizer, an orthogonal matrix.

D
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of quasi diagonal matrices.

See also:

ajd_pham
uwedge

Notes

New in version 0.2.4.

References

[1]

pyriemann.utils.ajd.uwedge

pyriemann.utils.ajd.**uwedge**(*X*, ***, *init=None*, *eps=1e-07*, *n_iter_max=100*)

Approximate joint diagonalization based on UWEDGE.

Implementation of the AJD algorithm by Tichavsky and Yeredor [1] [2]: uniformly weighted exhaustive diagonalization using Gauss iterations (U-WEDGE). This is a translation from the matlab code provided by the authors.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of symmetric matrices to diagonalize.

init
[None | ndarray, shape (n_channels, n_channels), default=None] Initialization for the diagonalizer.

eps
[float, default=1e-7] Tolerance for stoping criterion.

n_iter_max
[int, default=100] The maximum number of iterations to reach convergence.

Returns

V
[ndarray, shape (n_channels, n_channels)] The diagonalizer.

D
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of quasi diagonal matrices.

See also:

ajd_pham
rjd

Notes

New in version 0.2.4.

References

[1], [2]

5.13.10 Matrix Tests

is_square(X)	Check if matrices are square.
is_sym(X)	Check if all matrices are symmetric.
is_skew_sym(X)	Check if all matrices are skew-symmetric.
is_real(X)	Check if all complex matrices are strictly real.
is_hermitian(X)	Check if all matrices are Hermitian.
is_pos_def(X[, fast_mode])	Check if all matrices are positive definite.
is_pos_semi_def(X)	Check if all matrices are positive semi-definite.
is_sym_pos_def(X)	Check if all matrices are symmetric positive-definite.
is_sym_pos_semi_def(X)	Check if all matrices are symmetric positive semi-definite.

pyriemann.utils.test.is_square

pyriemann.utils.test.**is_square**(*X*)

Check if matrices are square.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

ret
[boolean] True if matrices are square.

pyriemann.utils.test.is_sym

pyriemann.utils.test.**is_sym**(*X*)

Check if all matrices are symmetric.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

ret
[boolean] True if all matrices are symmetric.

pyriemann.utils.test.is_skew_sym

pyriemann.utils.test.**is_skew_sym**(*X*)

Check if all matrices are skew-symmetric.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

ret
[boolean] True if all matrices are skew-symmetric.

pyriemann.utils.test.is_real

pyriemann.utils.test.**is_real**(*X*)

Check if all complex matrices are strictly real.

Better management of numerical imprecisions than np.all(np.isreal()).

Parameters

X
[ndarray] The set of matrices.

Returns

ret
[boolean] True if all complex matrices are strictly real.

pyriemann.utils.test.is_hermitian

pyriemann.utils.test.**is_hermitian**(*X*)

Check if all matrices are Hermitian.

Check if all matrices are Hermitian, ie with a symmetric real part and a skew-symmetric imaginary part.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

ret
[boolean] True if all matrices are Hermitian.

pyriemann.utils.test.is_pos_def

pyriemann.utils.test.**is_pos_def**(*X*, *fast_mode=False*)

Check if all matrices are positive definite.

Check if all matrices are positive definite, fast verification is done with Cholesky decomposition, while full check compute all eigenvalues to verify that they are positive.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

fast_mode
[boolean, default=False] Use Cholesky decomposition to avoid computing all eigenvalues.

Returns

ret
[boolean] True if all matrices are positive definite.

pyriemann.utils.test.is_pos_semi_def

pyriemann.utils.test.**is_pos_semi_def**(*X*)

Check if all matrices are positive semi-definite.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

ret
[boolean] True if all matrices are positive semi-definite.

pyriemann.utils.test.is_sym_pos_def

pyriemann.utils.test.**is_sym_pos_def**(*X*)

Check if all matrices are symmetric positive-definite.

Parameters

X
[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

ret
[boolean] True if all matrices are symmetric positive-definite.

pyriemann.utils.test.is_sym_pos_semi_def

pyriemann.utils.test.**is_sym_pos_semi_def**(*X*)

Check if all matrices are symmetric positive semi-definite.

Parameters

X

[ndarray, shape (..., n, n)] The set of square matrices, at least 2D ndarray.

Returns

ret

[boolean] True if all matrices are symmetric positive semi-definite.

5.13.11 Visualization

plot_confusion_matrix(*args, **kwargs)

Warning: DEPRECATED: plot_confusion_matrix is deprecated and will be removed in 0.4.0; please use sklearn confusion_matrix and ConfusionMatrixDisplay; see examples/ERP/plot_classify_EEG_tangentspace.py

plot_embedding(X[, y, metric, title, ...])	Plot 2D embedding of SPD matrices.
plot_cospectra(cosp, freqs, *[, ylabels, title])	Plot cospectral matrices.
plot_waveforms(X, display, *[, times, ...])	Display repetitions of a multichannel waveform.

pyriemann.utils.viz.plot_confusion_matrix

pyriemann.utils.viz.**plot_confusion_matrix**(**args*, ***kwargs*)

> **Warning:** DEPRECATED: plot_confusion_matrix is deprecated and will be remove in 0.4.0; please use sklearn confusion_matrix and ConfusionMatrixDisplay; see examples/ERP/plot_classify_EEG_tangentspace.py

Plot Confusion Matrix.

pyriemann.utils.viz.plot_embedding

pyriemann.utils.viz.**plot_embedding**(*X*, *y=None*, ***, *metric='riemann'*, *title='Embedding of covariances'*, *embd_type='Spectral'*, *normalize=True*)

Plot 2D embedding of SPD matrices.

Parameters

X
[ndarray, shape (n_matrices, n_channels, n_channels)] Set of SPD matrices.

y
[None | ndarray, shape (n_matrices,), default=None] Labels for each matrix.

metric
[string, default='riemann'] Metric used in the embedding. Can be {'riemann' ,'logeuclid' , 'euclid'} for Locally Linear Embedding and {'riemann' ,'logeuclid' , 'euclid' , 'logdet', 'kullback', 'kullback_right', 'kullback_sym'} for Spectral Embedding.

title
[str, default="Embedding of covariances"] Title string for plot.

embd_type
[{'Spectral' ,'LocallyLinear'}, default='Spectral'] Embedding type.

normalize
[bool, default=True] If True, the plot is normalized from -1 to +1.

Returns

fig
[matplotlib figure] Figure of embedding.

pyriemann.utils.viz.plot_cospectra

pyriemann.utils.viz.**plot_cospectra**(*cosp*, *freqs*, ***, *ylabels=None*, *title='Cospectra'*)

Plot cospectral matrices.

Parameters

cosp
[ndarray, shape (n_freqs, n_channels, n_channels)] Cospectral matrices.

freqs
[ndarray, shape (n_freqs,)] The frequencies associated to cospectra.

Returns

fig
[matplotlib figure] Figure of cospectra.

Notes

New in version 0.2.7.

pyriemann.utils.viz.plot_waveforms

pyriemann.utils.viz.**plot_waveforms**(*X*, *display*, ***, *times=None*, *color='gray'*, *alpha=0.5*, *linewidth=1.5*, *color_mean='k'*, *color_std='gray'*, *n_bins=50*, *cmap=None*)

Display repetitions of a multichannel waveform.

Parameters

X
[ndarray, shape (n_reps, n_channels, n_times)] Repetitions of the multichannel waveform.

display
[{'all', 'mean', 'mean+/-std', 'hist'}] Type of display:

- 'all' for all the repetitions;
- 'mean' for the mean of the repetitions;
- 'mean+/-std' for the mean +/- standard deviation of the repetitions;
- 'hist' for the 2D histogram of the repetitions.

time
[None | ndarray, shape (n_times,), default=None] Values to display on x-axis.

color
[matplotlib color, optional] Color of the lines, when `display=all`.

alpha
[float, optional] Alpha value used to cumulate repetitions, when `display=all`.

linewidth
[float, optional] Line width in points, when `display=mean`.

color_mean
[matplotlib color, optional] Color of the mean line, when `display=mean`.

color_std
[matplotlib color, optional] Color of the standard deviation area, when `display=mean+/-std`.

n_bins
[int, optional] Number of vertical bins for the 2D histogram, when `display=hist`.

cmap
[Colormap or str, optional] Color map for the histogram, when `display=hist`.

Returns

fig
[matplotlib figure] Figure of waveform (one subplot by channel).

Notes

New in version 0.3.

BIBLIOGRAPHY

[1] A Plug and Play P300 BCI Using Information Geometry A. Barachant, M. Congedo. Research report, 2014.

[2] A New generation of Brain-Computer Interface Based on Riemannian Geometry M. Congedo, A. Barachant, A. Andreev. Research report, 2013.

[3] Classification de potentiels evoques P300 par geometrie riemannienne pour les interfaces cerveau-machine EEG A. Barachant, M. Congedo, G. van Veen, and C. Jutten, 24eme colloque GRETSI, 2013.

[1] MEG decoding using Riemannian Geometry and Unsupervised classification A. Barachant. Technical report with the solution of the DecMeg 2014 challenge.

[1] Instantaneous and lagged measurements of linear and nonlinear dependence between groups of multivariate time series: frequency decomposition R. Pascual-Marqui. Technical report, 2007.

[2] Identifying true brain interaction from EEG data using the imaginary part of coherency G. Nolte, O. Bai, L. Wheaton, Z. Mari, S. Vorbach, M. Hallett. Clinical Neurophysioly, Volume 115, Issue 10, October 2004, Pages 2292-2307

[3] Non-Parametric Synchronization Measures used in EEG and MEG M. Congedo. Technical Report, 2018.

[1] https://en.wikipedia.org/wiki/Hankel_matrix

[2] Spatio-spectral filters for improving the classification of single trial EEG S. Lemm, B. Blankertz, B. Curio, K-R. Muller. IEEE Transactions on Biomedical Engineering 52(9), 1541-1548, 2005.

[1] Beyond Covariance: Feature Representation with Nonlinear Kernel Matrices L. Wang, J. Zhang, L. Zhou, C. Tang, W Li. ICCV, 2015.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_kernels.html

[1] https://scikit-learn.org/stable/modules/generated/sklearn.covariance.shrunk_covariance.html

[1] Clustering and dimensionality reduction on Riemannian manifolds A. Goh and R. Vidal, in 2008 IEEE Conference on Computer Vision and Pattern Recognition

[1] Laplacian Eigenmaps for dimensionality reduction and data representation M. Belkin and P. Niyogi, in Neural Computation, vol. 15, no. 6, p. 1373-1396 , 2003

[1] Nonlinear Dimensionality Reduction by Locally Linear Embedding S. Roweis and L. K. Saul, in Science, Vol 290, Issue 5500, pp. 2323-2326, 2000.

[2] Clustering and dimensionality reduction on Riemannian manifolds A. Goh and R. Vidal, in 2008 IEEE Conference on Computer Vision and Pattern Recognition

[1] Multiclass Brain-Computer Interface Classification by Riemannian Geometry A. Barachant, S. Bonnet, M. Congedo, and C. Jutten. IEEE Transactions on Biomedical Engineering, vol. 59, no. 4, p. 920-928, 2012.

[2] Riemannian geometry applied to BCI classification A. Barachant, S. Bonnet, M. Congedo and C. Jutten. 9th International Conference Latent Variable Analysis and Signal Separation (LVA/ICA 2010), LNCS vol. 6365, 2010, p. 629-636.

[1] Riemannian geometry applied to BCI classification A. Barachant, S. Bonnet, M. Congedo and C. Jutten. 9th International Conference Latent Variable Analysis and Signal Separation (LVA/ICA 2010), LNCS vol. 6365, 2010, p. 629-636.

[2] Classification of covariance matrices using a Riemannian-based kernel for BCI applications A. Barachant, S. Bonnet, M. Congedo and C. Jutten. Neurocomputing, Elsevier, 2013, 112, pp.172-178.

[1] Classification of covariance matrices using a Riemannian-based kernel for BCI applications A. Barachant, S. Bonnet, M. Congedo and C. Jutten. Neurocomputing, Elsevier, 2013, 112, pp.172-178.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[1] The Riemannian Minimum Distance to Means Field Classifier M Congedo, PLC Rodrigues, C Jutten. BCI 2019 - 8th International Brain-Computer Interface Conference, Sep 2019, Graz, Austria.

[1] Defining and quantifying users' mental imagery-based BCI skills: a first step F. Lotte, and C. Jeunet. Journal of neural engineering, 15(4), 046030, 2018.

[1] Classification of covariance matrices using a Riemannian-based kernel for BCI applications A. Barachant, S. Bonnet, M. Congedo and C. Jutten. Neurocomputing, Elsevier, 2013, 112, pp.172-178.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

[1] The Riemannian Potato: an automatic and adaptive artifact detection method for online experiments using Riemannian geometry A. Barachant, A Andreev, and M. Congedo. TOBI Workshop lV, Jan 2013, Sion, Switzerland. pp.19-20.

[2] The Riemannian Potato Field: A Tool for Online Signal Quality Index of EEG Q. Barthélemy, L. Mayaud, D. Ojeda, and M. Congedo. IEEE Transactions on Neural Systems and Rehabilitation Engineering, IEEE Institute of Electrical and Electronics Engineers, 2019, 27 (2), pp.244-255

[1] The Riemannian Potato Field: A Tool for Online Signal Quality Index of EEG Q. Barthélemy, L. Mayaud, D. Ojeda, and M. Congedo. IEEE Transactions on Neural Systems and Rehabilitation Engineering, IEEE Institute of Electrical and Electronics Engineers, 2019, 27 (2), pp.244-255

[1] Multiclass Brain-Computer Interface Classification by Riemannian Geometry A. Barachant, S. Bonnet, M. Congedo, and C. Jutten. IEEE Transactions on Biomedical Engineering, vol. 59, no. 4, p. 920-928, 2012.

[2] Classification of covariance matrices using a Riemannian-based kernel for BCI applications A. Barachant, S. Bonnet, M. Congedo and C. Jutten. Neurocomputing, Elsevier, 2013, 112, pp.172-178.

[1] Riemannian geometry applied to BCI classification A. Barachant, S. Bonnet, M. Congedo and C. Jutten. 9th International Conference Latent Variable Analysis and Signal Separation (LVA/ICA 2010), LNCS vol. 6365, 2010, p. 629-636.

[2] Classification of covariance matrices using a Riemannian-based kernel for BCI applications A. Barachant, S. Bonnet, M. Congedo and C. Jutten. Neurocomputing, Elsevier, 2013, 112, pp.172-178.

[1] xDAWN algorithm to enhance evoked potentials: application to brain-computer interface B. Rivet, A. Souloumiac, V. Attina, and G. Gibert. IEEE Transactions on Biomedical Engineering, 2009, 56 (8), pp.2035-43.

[2] Theoretical analysis of xDAWN algorithm: application to an efficient sensor selection in a P300 BCI B. Rivet, H. Cecotti, A. Souloumiac, E. Maby, J. Mattout. EUSIPCO 2011 19th European Signal Processing Conference, Aug 2011, Barcelone, Spain. pp.1382-1386.

[1] Spatial Patterns Underlying Population Differences in the Background EEG Z. Koles, M. Lazar, and S. Zhou. Brain Topography 2(4), 275-284, 1990.

[2] Optimizing Spatial Filters for Robust EEG Single-Trial Analysis B. Blankertz, R. Tomioka, S. Lemm, M. Kawanabe, K-R. Muller. IEEE Signal Processing Magazine 25(1), 41-56, 2008.

[3] Common Spatial Pattern revisited by Riemannian geometry A. Barachant, S. Bonnet, M. Congedo and C. Jutten. IEEE International Workshop on Multimedia Signal Processing (MMSP), p. 472-476, 2010.

[4] Multiclass common spatial patterns and information theoretic feature extraction IEEE Transactions on Biomedical Engineering, Volume 55, Issue 8, August 2008. pp. 1991 - 2000

[1] SPoC: a novel framework for relating the amplitude of neuronal oscillations to behaviorally relevant parameters S. Dahne, F. C. Meinecke, S. Haufe, J. Hohne, M. Tangermann, K-R. Muller, and V. V. Nikulin. NeuroImage, 86, 111-122, 2014.

[1] On the blind source separation of human electroencephalogram by approximate joint diagonalization of second order statistics M. Congedo, C. Gouy-Pailler, C. Jutten. Clinical Neurophysiology, Elsevier, 2008, 119 (12), pp.2677-2686.

[2] Group indepedent component analysis of resting state EEG in large normative samples M. Congedo, R. John, D. de Ridder, L. Prichep. International Journal of Psychophysiology, Elsevier, 2010, 78, pp.89-99.

[3] Joint approximate diagonalization of positive definite Hermitian matrices D.-T. Pham. SIAM Journal on Matrix Analysis and Applications, Volume 22 Issue 4, 2000

[1] Channel selection procedure using riemannian distance for BCI applications A. Barachant and S. Bonnet. The 5th International IEEE EMBS Conference on Neural Engineering, Apr 2011, Cancun, Mexico.

[1] https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation-iterators

[1] Transfer Learning: A Riemannian Geometry Framework With Applications to Brain–Computer Interfaces P Zanini et al, IEEE Transactions on Biomedical Engineering, vol. 65, no. 5, pp. 1107-1116, August, 2017

[1] Riemannian Procrustes analysis: transfer learning for brain-computer interfaces PLC Rodrigues et al, IEEE Transactions on Biomedical Engineering, vol. 66, no. 8, pp. 2390-2401, December, 2018

[1] Riemannian Procrustes analysis: transfer learning for brain-computer interfaces PLC Rodrigues et al, IEEE Transactions on Biomedical Engineering, vol. 66, no. 8, pp. 2390-2401, December, 2018

[2] An introduction to optimization on smooth manifolds N. Boumal. To appear with Cambridge University Press. June, 2022

[1] Transfer learning for SSVEP-based BCI using Riemannian similarities between users E. Kalunga, S. Chevallier and Q. Barthelemy, in 26th European Signal Processing Conference (EUSIPCO), pp. 1685-1689. IEEE, 2018.

[2] Minimizing Subject-dependent Calibration for BCI with Riemannian Transfer Learning S. Khazem, S. Chevallier, Q. Barthelemy, K. Haroun and C. Nous, 10th International IEEE/EMBS Conference on Neural Engineering (NER), pp. 523-526. IEEE, 2021.

[1] A new method for non-parametric multivariate analysis of variance M. Anderson. Austral ecology, Volume 26, Issue 1, February 2001.

[1] Riemannian Gaussian distributions on the space of symmetric positive definite matrices S. Said, L. Bombrun, Y. Berthoumieu, and J. Manton. IEEE Trans Inf Theory, vol. 63, pp. 2153–2170, 2017.

[est] https://scikit-learn.org/stable/modules/covariance.html

[corr] https://numpy.org/doc/stable/reference/generated/numpy.corrcoef.html

[cov] https://numpy.org/doc/stable/reference/generated/numpy.cov.html

[lwf] https://scikit-learn.org/stable/modules/generated/sklearn.covariance.ledoit_wolf.html

[mcd] https://scikit-learn.org/stable/modules/generated/sklearn.covariance.MinCovDet.html

[mest] *pyriemann.utils.covariance.covariance_mest()*

[oas] https://scikit-learn.org/stable/modules/generated/sklearn.covariance.OAS.html

[sch] *pyriemann.utils.covariance.covariance_sch()*

[scm] https://scikit-learn.org/stable/modules/generated/sklearn.covariance.empirical_covariance.html

[1] Complex Elliptically Symmetric Distributions: Survey, New Results and Applications E. Ollila, D.E. Tyler, V. Koivunen, H.V. Poor. IEEE Transactions on Signal Processing, 2012.

[2] Robust antenna array processing using M-estimators of pseudo-covariance E. Ollila, V. Koivunen. PIMRC, 2003.

[3] Influence functions for array covariance matrix estimators E. Ollila, V. Koivunen. IEEE SSP, 2003.

[4] A distribution-free M-estimator of multivariate scatter D.E. Tyler. The Annals of Statistics, 1987.

[1] A shrinkage approach to large-scale covariance estimation and implications for functional genomics J. Schafer, and K. Strimmer. Statistical Applications in Genetics and Molecular Biology, Volume 4, Issue 1, 2005.

[1] A special form of SPD covariance matrix for interpretation and visualization of data manipulated with Riemannian geometry M. Congedo and A. Barachant, MaxEnt - 34th International Workshop on Bayesian Inference and Maximun Entropy Methods in Science and Engineering (MaxEnt'14), Sep 2014, Amboise, France. pp.495

[1] https://en.wikipedia.org/wiki/Cross-spectrum

[1] On the blind source separation of human electroencephalogram by approximate joint diagonalization of second order statistics M. Congedo, C. Gouy-Pailler, C. Jutten. Clinical Neurophysiology, Elsevier, 2008, 119 (12), pp.2677-2686.

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.mahalanobis.html

[1] Approximate Joint Diagonalization and Geometric Mean of Symmetric Positive Definite Matrices M. Congedo, B. Afsari, A. Barachant, M. Moakher. PLOS ONE, 2015

[1] Geometric Means T. Ando, C.-K. Li, and R. Mathias. Linear Algebra and its Applications. Volume 385, July 2004, Pages 305-334.

[1] Symmetric positive-definite matrices: From geometry to applications and visualization M. Moakher and P. Batchelor. Visualization and Processing of Tensor Fields, pp. 285-298, 2006

[1] Geometric means in a novel vector space structure on symmetric positive-definite matrices V. Arsigny, P. Fillard, X. Pennec, and N. Ayache. SIAM Journal on Matrix Analysis and Applications. Volume 29, Issue 1 (2007).

[1] Matrix Power means and the Karcher mean Y. Lim and M. Palfia. Journal of Functional Analysis, Volume 262, Issue 4, 15 February 2012, Pages 1498-1514.

[2] Fixed Point Algorithms for Estimating Power Means of Positive Definite Matrices M. Congedo, A. Barachant, and R. Bhatia. IEEE Transactions on Signal Processing, Volume 65, Issue 9, pp.2211-2220, May 2017

[1] A differential geometric approach to the geometric mean of symmetric positive-definite matrices M. Moakher, SIAM Journal on Matrix Analysis and Applications. Volume 26, Issue 3, 2005

[1] Geometric Radar Processing based on Frechet distance: Information geometry versus Optimal Transport Theory F. Barbaresco. 12th International Radar Symposium (IRS), October 2011

[1] Geodesically-convex optimization for averaging partially observed covariance matrices F. Yger, S. Chevallier, Q. Barthélemy, and S. Sra. Asian Conference on Machine Learning (ACML), Nov 2020, Bangkok, Thailand. pp.417 - 432.

[1] Geodesically-convex optimization for averaging partially observed covariance matrices F. Yger, S. Chevallier, Q. Barthélemy, and S. Sra. Asian Conference on Machine Learning (ACML), Nov 2020, Bangkok, Thailand. pp.417 - 432.

[1] Sur le point pour lequel la somme des distances de n points donnés est minimum E Weiszfeld. Tohoku Mathematical Journal, 1937, 43, pp. 355-386.

[2] The multivariate L1-median and associated data depth Y Vardi and C-H Zhan. Proceedings of the National Academy of Sciences, 2000, vol. 97, no 4, p. 1423-1426

[3] https://numpy.org/doc/stable/reference/generated/numpy.median.html

[1] The geometric median on Riemannian manifolds with application to robust atlas estimation PT. Fletcher, S. Venkatasubramanian S and S. Joshi. NeuroImage, 2009, 45(1), S143-S152

[2] Riemannian median, geometry of covariance matrices and radar target detection L Yang, M Arnaudon and F Barbaresco. 7th European Radar Conference, 2010, pp. 415-418

[1] Classification of covariance matrices using a Riemannian-based kernel for BCI applications A. Barachant, S. Bonnet, M. Congedo and C. Jutten. Neurocomputing, Elsevier, 2013, 112, pp.172-178.

[1] Classification of covariance matrices using a Riemannian-based kernel for BCI applications A. Barachant, S. Bonnet, M. Congedo and C. Jutten. Neurocomputing, Elsevier, 2013, 112, pp.172-178.

[1] Pedestrian detection via classification on Riemannian manifolds O. Tuzel, F. Porikli, and P. Meer. IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 30, Issue 10, October 2008.

[1] nearestSPD J. D'Errico, MATLAB Central File Exchange

[2] Computing a nearest symmetric positive semidefinite matrix N.J. Higham, Linear Algebra and its Applications, vol 103, 1988

[1] Joint approximate diagonalization of positive definite Hermitian matrices D.-T. Pham. SIAM Journal on Matrix Analysis and Applications, Volume 22 Issue 4, 2000

[1] Jacobi angles for simultaneous diagonalization J.-F. Cardoso and A. Souloumiac, SIAM Journal on Matrix Analysis and Applications, Volume 17, Issue 1, Jan. 1996.

[1] A Fast Approximate Joint Diagonalization Algorithm Using a Criterion with a Block Diagonal Weight Matrix P. Tichavsky, A. Yeredor and J. Nielsen. 2008 IEEE International Conference on Acoustics, Speech and Signal ProcessingICASSP.

[2] Fast Approximate Joint Diagonalization Incorporating Weight Matrices P. Tichavsky and A. Yeredor. IEEE Transactions on Signal Processing, Volume 57, Issue 3, March 2009.

INDEX

Symbols

A

G

H

I

K

L

M

N

P

R

S

T

U

W

X