obob$_{m}ne$*Documentation*

Release 0.0.15

Thomas Hartmann

Jul 25, 2019

Contents

Hello World

CHAPTER 1

Introduction

The intro goes here

1.1 Reference

1.1.1 Raw Files

LoadFromSinuhe(subject_id[, block_nr])	mne.io.Raw mixin to facilitate loading data from sinuhe.
AdvancedEvents(*args, **kwargs)	Integrate event loading and handling into mne.io.Raw.
AutomaticBinaryEvents(*args, **kwargs)	Mixin for binary events.

obob_mne.mixins.raw.LoadFromSinuhe

class obob_mne.mixins.raw.**LoadFromSinuhe**(*subject_id*, *block_nr=None*, ***kwargs*)

mne.io.Raw mixin to facilitate loading data from sinuhe.

By including this mixin in your study specific Raw class, you can facilitate loading the raw data files.

Supposed your fif files follow the usual pattern: '19800908igdb_run01.fif', you can define a Raw class like this one:

```
class Raw(mne.io.fiff.Raw, LoadFromSinuhe):
    study_acronym = 'test_study'
```

Loading run 2 of subject '19800908igdb' can then be done like this:

```
raw_data = Raw(subject_id='19800908igdb', block_nr=2, preload=True)
```

classmethod get_all_subjects()

Return a list of all subjects in the study.

Returns **all_subjects** – A list of strings with all subjects codes found.

Return type list

classmethod get_fif_filename(*subject_id*, *run_nr*)
Find the fif file for the subject and run.

Parameters

- **subject_id** (*str*) – The subject id
- **run_nr** (*int*) – The run number

Returns **fname** – The filename of the respective fif file

Return type str

classmethod get_number_of_runs(*subject_id*)
Return the number of runs for the given subject.

Parameters **subject_id** (*str*) – The subject id

Returns **n_runs** – The number of runs for the subject.

Return type int

obob_mne.mixins.raw.AdvancedEvents

class obob_mne.mixins.raw.**AdvancedEvents**(**args*, ***kwargs*)
Integrate event loading and handling into `mne.io.Raw`.

Including this mixin in your study specific `Raw` class provides event handling features directly in that class.

More specifically, it provides three extra properties:

1. events
2. event_id
3. evt_metadata

Which are automatically filled and kept up-to-date. They correspond to the respective meaning in `mne.Epochs`.

You can also create a subclass of this class and use `_process_events()` to process the events (fill the event_id, modify the event codes. . . .)

event_id
The event_ids

Type dict

events
The event matrix.

Type `numpy.ndarray`

evt_metadata
The metadata

Type `pandas.DataFrame`

get_filtered_event_id(*condition_filter*)
Return a filtered version of the event_id field.

Refer to `obob_mne.events.filter_event_id()` for further details.

has_filtered_events(*condition_filter*)
Check whether the event_ids are present.

Parameters **condition_filter** (*str*) – The event_ids to check

Returns **has_events** – True if the filtered events are present.

Return type bool

resample(**args*, ***kwargs*)
Resample the data and reloads the events.

For the rest, refer to mne.io.Raw.resample().

obob_mne.mixins.raw.AutomaticBinaryEvents

class obob_mne.mixins.raw.**AutomaticBinaryEvents**(**args*, ***kwargs*)
Mixin for binary events.

If your triggers code events with binary triggers, this mixin can help you a lot.

Let's suppose, you have an experiment with two types of blocks. At the beginning of each block, the type of the block is signalled by a a trigger code:

1. Attend Auditory: Trigger 1
2. Attend Visual: Trigger 2

Then you present either:

1. A tone: Trigger 4
2. An image: Trigger 8

And sometimes, one of them is an oddball which is marked by adding 1 to the trigger codes.

In this case, you can use this mixin and write something like this:

```
class Raw(mne.io.fiff.Raw, LoadFromSinuhe, AutomaticBinaryEvents):
    study_acronym = 'test_study'

    condition_triggers = {
        'attention': {
            'auditory': 1,
            'visual': 2
        }
    }

    stimulus_triggers = {
        'modality': {
            'audio': 4,
            'visual': 8
        },
        'oddball': 1
    }
```

This will automatically result in mne-python aware event_ids like:

```
'attention:visual/modality:audio/oddball:True'
```

1.1.2 Decoding

Temporal Decoding

Temporal(epochs, pipeline[, epochs_test, ...])	Apply a decoding pipeline to every sample.

obob_mne.decoding.Temporal

class obob_mne.decoding.**Temporal**(*epochs*, *pipeline*, *epochs_test=None*, *cv=2*, *scoring='accuracy'*, *n_jobs=-1*, *metadata_querylist=None*, *metadata_querylist_test=None*)

Apply a decoding pipeline to every sample.

Use this class to perform Temporal decoding. This means that a classifier is trained on every sample and tested on that very sample.

The minimum requirements are the data as `mne.Epochs` and the pipeline as `sklearn.pipeline.Pipeline`, which is most commonly generated using `sklearn.pipeline.make_pipeline()`.

This class treats every individual `event_id` as an individual target. If you want to combine multiple `events_id` into one target, you can use `epochs.collapse_conditions()`.

If you want the training and testing data to be different, you can supply the training data to `epochs` and the testing data to `epochs_test`.

Cross validation will be run only if training and testing data are equal (i.e., when `epochs_test=None`).

Parameters

- **epochs** (`mne.Epochs`) – The epochs to apply the classifier pipeline to.
- **pipeline** (`sklearn.pipeline.Pipeline`) – The classifier pipeline to use. Most likely created with `sklearn.pipeline.make_pipeline()`
- **epochs_test** (`mne.Epochs` or None, optional) – If set, the classifier gets tested on this data. The event_ids must be equal. No crossvalidation will be run in this case.
- **cv** (*int, optional*) – The amount of folds for cross validation
- **scoring** (*str or callable, optional*) – The scoring function to use.
- **n_jobs** (*int, optional*) – Number of CPU cores to use

decoder
The decoder.

Generalized Temporal Decoding

GeneralizedTemporal(epochs, pipeline[, ...])	Apply a decoding pipeline using Temporal Generalization.
GeneralizedTemporalAverage(data_list)	Create average instance of Temporal Generalization results.
GeneralizedTemporalStatistics(data_list[, ...])	Run statistics on Temporal Generalization results.

obob_mne.decoding.GeneralizedTemporal

class obob_mne.decoding.**GeneralizedTemporal**(*epochs*, *pipeline*, *epochs_test=None*, *cv=2*, *scoring='accuracy'*, *n_jobs=-1*, *metadata_querylist=None*, *metadata_querylist_test=None*)

Apply a decoding pipeline using Temporal Generalization.

Use this class to perform Temporal Generalization decoding. This means that a classifier is trained on every sample and then tested on all samples. So, if you supply epochs with 100 samples, a classifier gets trained on the data data of the first sample. This classifier is then tested on the data of the first sample, then the second sample and so on. The process is then repeated by training on the second sample and testing on all samples and so forth.

The minimum requirements are the data as `mne.Epochs` and the pipeline as `sklearn.pipeline.Pipeline`, which is most commonly generated using `sklearn.pipeline.make_pipeline()`.

This class treats every individual `event_id` as an individual target. If you want to combine multiple `events_id` into one target, you can use `epochs.collapse_conditions()`.

If you want the training and testing data to be different, you can supply the training data to `epochs` and the testing data to `epochs_test`.

Cross validation will be run only if training and testing data are equal (i.e., when `epochs_test=None`).

Parameters

- **epochs** (`mne.Epochs`) – The epochs to apply the classifier pipeline to.
- **pipeline** (`sklearn.pipeline.Pipeline`) – The classifier pipeline to use. Most likely created with `sklearn.pipeline.make_pipeline()`
- **epochs_test** (`mne.Epochs` or None, optional) – If set, the classifier gets tested on this data. The event_ids must be equal. No crossvalidation will be run in this case.
- **cv** (*int, optional*) – The amount of folds for cross validation
- **scoring** (*str or callable, optional*) – The scoring function to use.
- **n_jobs** (*int, optional*) – Number of CPU cores to use

obob_mne.decoding.GeneralizedTemporalAverage

class obob_mne.decoding.**GeneralizedTemporalAverage**(*data_list*)

Create average instance of Temporal Generalization results.

Given a list of Temporal Generalization Results, this class computes the average of their weights and returns an instance of *GeneralizedTemporalArray* so you can plot the average scores and weights.

Parameters **data_list** (iterable of *GeneralizedTemporalArray*) – A list (or any other iterable) of *GeneralizedTemporalArray*

obob_mne.decoding.GeneralizedTemporalStatistics

class obob_mne.decoding.**GeneralizedTemporalStatistics**(*data_list*, *stat_function=<function ttest_1samp>*, *pop_mean=None*, ***kwargs*)

Run statistics on Temporal Generalization results.

Given a list of Temporal Generalization Results, this class calculates a statistic on the weights and returns an instance of *GeneralizedTemporalArray*.

get_temporal_from_training_interval (*tmin*, *tmax*)
Average the scores of a training interval.

Parameters

- **tmin** (*int*) – Start time in seconds of the training interval to average.
- **tmax** (*int*) – End time in seconds of the training interval to average.

Returns **data** – The *TemporalArray* with the averaged scores.

Return type instance of *TemporalArray*

plot_scores (*axes=None*, *show=True*, *cmap='Reds'*, *colorbar=True*, *mask_below_chance=False*, *interpolation='bessel'*, *mask_p=None*)
Plot the scores as a Matrix.

Parameters

- **axes** (matplotlib.axes.Axes or None, optional) – The axes where to draw the plot. If None, a new figure is created.
- **show** (*bool, optional*) – True to actually show the plot.
- **cmap** (str or matplotlib.colors.Colormap, optional) – The colormap.
- **colorbar** (*bool, optional*) – Whether to draw the colorbar.
- **mask_below_chance** (*bool, optional*) – If True, values below chance level get masked.
- **interpolation** (*str, optional*) – The interpolation method used.
- **mask_p** (*float or None, optional*) – If set, the plot is masked for the given p-value.

1.2 Reference of low level classes and functions

1.2.1 Decoding

Temporal Decoding

TemporalArray(raw_scores, weights, . . . [, . . .])	Base class for temporal decoding.

obob_mne.decoding.TemporalArray

class obob_mne.decoding.**TemporalArray** (*raw_scores*, *weights*, *n_classes*, *info*, *tmin*, *scoring_name*, *c_factors_training*, *nave=1*, *nave_testing=None*, *c_factors_testing=None*)
Base class for temporal decoding.

Parameters

- **raw_scores** (numpy.ndarray shape (n_times) or (n_folds, n_times)) – The scores of the classification
- **weights** (numpy.ndarray shape (n_channels, n_times)) – Classifier weights

- **n_classes** (*int*) – The number of classes
- **info** (*dict*) – Info dict
- **tmin** (*float*) – Time of the first sample in seconds
- **scoring_name** (*str*) – Name of the scoring function (i.e. Accuracy...)
- **c_factors_training** (*str*) – Name of the factors over which was collapsed in the training set
- **nave** (*int, optional*) – Number of epochs in the training set.
- **nave_testing** (*int or None*) – Number of epochs in the testing set. If `None`, it is copied from nave.
- **c_factors_testing** (*None or str*) – Name of the factors over which was collapsed in the testing set. If `None`, it is copied from c_factors_training.

__init__(*raw_scores*, *weights*, *n_classes*, *info*, *tmin*, *scoring_name*, *c_factors_training*, *nave=1*, *nave_testing=None*, *c_factors_testing=None*)

Methods

`__init__`(raw_scores, weights, n_classes, ...)	
`add_channels`(add_list[, force_update_info])	Append new channels to the instance.
`add_proj`(projs[, remove_existing, verbose])	Add SSP projection vectors.
`animate_topomap`([ch_type, times, ...])	Make animation of evoked data as topomap time-series.
`anonymize`()	Anonymize measurement information in place.
`apply_baseline`([baseline, verbose])	Baseline correct evoked data.
`apply_hilbert`([picks, envelope, n_jobs, ...])	Compute analytic signal or envelope for a subset of channels.
`apply_proj`()	Apply the signal space projection (SSP) operators to the data.
`as_type`([ch_type, mode])	Compute virtual evoked using interpolated fields.
`copy`()	Copy the instance of evoked.
`crop`([tmin, tmax])	Crop data to a given time interval.
`decimate`(decim[, offset])	Decimate the evoked data.
`del_proj`([idx])	Remove SSP projection vector.
`detrend`([order, picks])	Detrend data.
`drop_channels`(ch_names)	Drop channel(s).
`filter`(l_freq, h_freq[, picks, ...])	Filter a subset of channels.
`get_peak`([ch_type, tmin, tmax, mode, ...])	Get location and latency of peak amplitude.
`interpolate_bads`([reset_bads, mode, origin, ...])	Interpolate bad MEG and EEG channels.
`pick`(picks[, exclude])	Pick a subset of channels.
`pick_channels`(ch_names)	Pick some channels.
`pick_types`([meg, eeg, stim, eog, ecg, emg, ...])	Pick some channels by type and names.
`plot`([picks, exclude, unit, show, ylim, ...])	Plot evoked data using butterfly plots.
`plot_field`(surf_maps[, time, time_label, n_jobs])	Plot MEG/EEG fields on head surface and helmet in 3D.
`plot_image`([picks, exclude, unit, show, ...])	Plot evoked data as images.
`plot_joint`([times, title, picks, exclude, ...])	Plot evoked data as butterfly plot and add topomaps for time points.

Continued on next page

Table 5 – continued from previous page

plot_projs_topomap([ch_type, layout, axes])	Plot SSP vector.
plot_scores([axes, show])	Plot the scores as a line plot.
plot_sensors([kind, ch_type, title, ...])	Plot sensor positions.
plot_topo([layout, layout_scale, color, ...])	Plot 2D topography of evoked responses.
plot_topomap([times, ch_type, layout, vmin, ...])	Plot topographic maps of specific time points of evoked data.
plot_white(noise_cov[, show, rank, ...])	Plot whitened evoked response.
rename_channels(mapping)	Rename channels.
reorder_channels(ch_names)	Reorder channels.
resample(sfreq[, npad, window, n_jobs, pad, ...])	Resample data.
save(fname)	Save dataset to file.
savgol_filter(h_freq[, verbose])	Filter the data using Savitzky-Golay polynomial method.
set_channel_types(mapping)	Define the sensor type of channels.
set_eeg_reference([ref_channels, ...])	Specify which reference to use for EEG data.
set_montage(montage[, set_dig, verbose])	Set EEG sensor configuration and head digitization.
shift_time(tshift[, relative])	Shift time scale in evoked data.
time_as_index(times[, use_rounding])	Convert time to indices.
to_data_frame([picks, index, scaling_time, ...])	Export data in tabular structure as a pandas DataFrame.

Attributes

ch_names	Channel names.
chance_level	The chance level of the classifier.
compensation_grade	The current gradient compensation grade.
data	The data matrix.
nclasses	
proj	Whether or not projections are active.
scores	The scores of the classification

Generalized Temporal Decoding

GeneralizedTemporalArray(scores_raw, ...[, ...])	Base class for Temporal Generalization Decoding.
GeneralizedTemporalFromCollection(data_list, ...)	Base class for Temporal Generalization data from multiple subjects.

obob_mne.decoding.GeneralizedTemporalArray

class obob_mne.decoding.**GeneralizedTemporalArray**(*scores_raw*, *scoring_name*, *n_classes*, *c_factors_training*, *weights*, *info*, *times_training*, *tmin*, *times_testing=None*, *c_factors_testing=None*, *nave=1*, *nave_testing=None*)

Base class for Temporal Generalization Decoding.

Parameters

- **scores_raw** (numpy.ndarray shape (n_times, n_times) or (n_folds, n_times, n_times)) – Classifier accuracies.
- **scoring_name** (*str*) – Name of the scoring function (i.e. Accuracy...)
- **n_classes** (*int*) – Number of classes of the classification
- **c_factors_training** (*str*) – Name of the factors over which was collapsed in the training set
- **weights** (numpy.ndarray shape (n_channels, n_times)) – Classifier weights
- **info** (*dict*) – Info dict
- **times_training** (numpy.ndarray shape (n_times)) – Array of times in seconds of the training data
- **tmin** (*int*) – ???
- **times_testing** (None or numpy.ndarray shape (n_times)) – Array of times in seconds of the testing data. If None, it is copied from times_training.
- **c_factors_testing** (*None or str*) – Name of the factors over which was collapsed in the testing set. If None, it is copied from c_factors_training.
- **nave** (*int*) – Number of epochs in the training set.
- **nave_testing** (*int or None*) – Number of epochs in the testing set. If None, it is copied from nave.

__init__(*scores_raw*, *scoring_name*, *n_classes*, *c_factors_training*, *weights*, *info*, *times_training*, *tmin*, *times_testing=None*, *c_factors_testing=None*, *nave=1*, *nave_testing=None*)

Methods

__init__(scores_raw, scoring_name, ...[, ...])	
diagonal_as_temporal()	Return the non-generalized results.
drop_channels(chs)	Drop the channels from the weights.
get_temporal_from_training_interval(tmin, tmax)	Average the scores of a training interval.
plot_scores([axes, show, cmap, colorbar, ...])	Plot the scores as a Matrix.

Attributes

chance_level	The chance level of the classifier.
info	Measurement info
nave	Number of epochs in the training set.
nave_testing	Number of epochs in the testing set.
scores	The scores of the classification
tmin	tmin
weights	The classifier weights

obob_mne.decoding.GeneralizedTemporalFromCollection

class obob_mne.decoding.**GeneralizedTemporalFromCollection**(*data_list*, *raw_scores*, *weights*)

Base class for Temporal Generalization data from multiple subjects.

The individual elements must match in number of classes, times etc.

Parameters

- **data_list** (iterable of *GeneralizedTemporalArray*) – A list (or any other iterable) of *GeneralizedTemporalArray*
- **raw_scores** (numpy.ndarray shape (n_channels, n_times) or (n_folds, n_channels, n_times)) – The already processed (i.e., averaged, statistically tested) scores.
- **weights** (numpy.ndarray shape (n_channels, n_times)) – The already processed (i.e., averaged, statistically tested) weights.

__init__(*data_list*, *raw_scores*, *weights*)

Methods

__init__(data_list, raw_scores, weights)	
diagonal_as_temporal()	Return the non-generalized results.
drop_channels(chs)	Drop the channels from the weights.
get_temporal_from_training_interval(tmin, tmax)	Average the scores of a training interval.
plot_scores([axes, show, cmap, colorbar, ...])	Plot the scores as a Matrix.

Attributes

chance_level	The chance level of the classifier.
info	Measurement info
must_be_equal	
nave	Number of epochs in the training set.
nave_testing	Number of epochs in the testing set.
scores	The scores of the classification
tmin	tmin
weights	The classifier weights

Index

Symbols

A

D

E

G

H

L

P

R

T