DESIGN LAB

Documentation

To Visualize the EEG Microstates

Prepared by

Jamil Reza Chowdhury

Department: Electrical and Computer Engineering

2019-07-16

Table of Contents

Documentation for the Microstates

In this documentation we are using the microstates module to *"Segment a continuous signal into microstates"* and then we have implemented and visualized the microstates in topographical maps. Also, the code plots *"auto scaled data"* just to visualize the raw data that was selected by the user. The preferred file format is ".mul" and the preferred channel location file can have the following formats:

'.elc','.txt','.loc','.locs','csd','elp','hpts',' sfp','eloc' or '.bvref'.

Here we have used the concept of ***Global field power*** [2]. First the user is prompted to give the desired data file and channel location file of the EEG data. Then using the microstates module, the segment method can be called to find the Peaks in the global field power (GFP). These are used to find microstates, using a modified K-means algorithm. Several runs of the modified K-means algorithm are performed, using different random initializations. The run that resulted in the best segmentation, as measured by global explained variance (GEV), is used. The following parameters are given to call the method(function) from the microstate module.

Method (function) Parameters:

data: ndarray, shape (n_channels, n_samples)

The data for finding the EEG microstates where the data has 63 channels(by default) and 30000 samples

n_states : int

The number of unique microstates to find. Defaults to 4.

n_inits : int

The number of random initializations to use for the k-means algorithm. The best fitting segmentation across all initializations is used. Defaults to 10 but can be also 30/40/50.

max_iter : int

The maximum number of iterations to perform in the k-means algorithm. Defaults to 1000.

thresh : float

The threshold of convergence for the k-means algorithm, based on relative change in noise variance. Defaults to 1e-6.

normalize : bool

Whether to normalize the data across time before running the k-means algorithm. Defaults to ``False``.

min_peak_dist : int

Minimum distance (in samples) between peaks in the GFP. Defaults to 2.

max_n_peaks : int

Maximum number of GFP peaks to use in the k-means algorithm. Chosen randomly. Defaults to 10000.

random_state : int | numpy.random.RandomState | None

The seed or ``RandomState`` for the random number generator. Defaults to ``None``, in which case a different seed is chosen each time this function is called.

verbose: input type: int(integer) | bool | None => This controls the verbosity.

The method(function) returns the followings:

Returns:

maps : ndarray, shape (n_channels, n_states)

The topographic maps of the found unique microstates.

segmentation : ndarray, shape (n_samples,)

For each sample, the index of the microstate to which the sample has

been assigned.

Next we want to plot the topographical map of the microstates. For this we called the ***plot_maps*** method(function) from the microstate module.

"""Plotting prototypical microstate maps.

Parameters:

maps : ndarray, shape (n_channels, n_maps)

The prototypical microstate maps.

info : instance of mne.io.Info. The info structure of the dataset, containing the location of the

sensors.

"""

Lastly the module can be used to plot the microstate segmentation for the data.

"""Plot a microstate segmentation.

Parameters:

segmentation : list of int (integer)

For each sample in time, the index of the state to which the sample has been assigned.

times : list of float => The time-stamp for each sample.

References:

[1] Pascual-Marqui, R. D., Michel, C. M., & Lehmann, D. (1995). Segmentation of brain electrical activity into microstates: model estimation and validation. IEEE Transactions on Biomedical Engineering.

[2] Global Field Power is a related, parametric assessment of map strength, defined as the sum of the absolute microvolt values measured at all electrodes divided by the number of electrodes; the assessment must be done after the values in each map have been expressed as deviations from the mean of all momentary values (spatial DC offset removal, 'average reference') computed as standard deviation of the momentary potential values (Lehmann and Skrandies, 1980).

http://www.scholarpedia.org/article/EEG_microstates#Brain_electric_fields

Visualizing EEG Microstates:

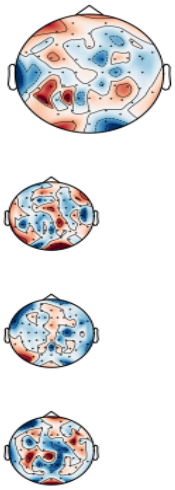

Fig. 1 Four EEG microstates

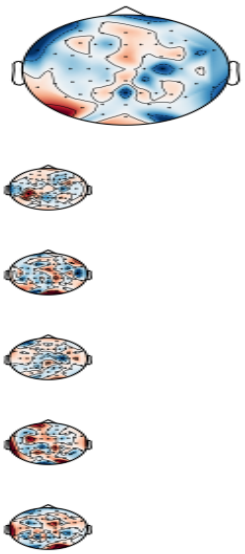

Fig. 2 Six EEG microstates

Topographic maps of EEG microstates in a continuous EEG signal. This depends on the number of microstates as given by the user.

Code:

The coding is done with Python 3.7. We have used the *ANACONDA* package for the python version 3.7. However, one can use the following packages, libraries and modules to implement the code. The setup file is given as *"setup.py"* for the easy installation of *certain* libraries

Required packages, Libraries and modules:

We adopted the following modules, packages and libraries:

1. Numpy
2. MNE python
3. Matplotlib
4. Pathlib library and Path module
5. Microstates
6. SciPy for scientific computing and technical computing
7. Tkinter package for python 3.7
8. Operation Sytem (Os) module

Python 3.7 CODE:

MAIN:

File name: *"MAIN.py"*

```
import tkinter as tk
from tkinter import filedialog
import numpy as np
import mne
import microstates
import os
from pathlib import Path
#Function to read the MUL data file
def read_data(path):
    with open (path,'r') as f:
        lines = f.readlines()
        matrix = []
        for line in lines:
            res=[]
            temp = line.split(' ')
            for num in temp:
                if num:
res.append(float(num))
            matrix.append(res)
```

```
        return
np.asarray(matrix)


#Tkinter package for user
selection of files: DATA and
Channel Location file

root = tk.Tk()

root.withdraw()


print("Please select the data
file")

data_file_path                =
filedialog.askopenfilename()

print("Please    select    the
channel location file")

channel_file_path             =
filedialog.askopenfilename()


# "kind" and "path" variables
for  mne.channels_read_montage
function

p = Path(channel_file_path)

k = p.parts[-1]

d = k.find('.')

kind = k[0:d]


f = p.parts

f=f[0:len(f)-1]

path = os.path.join(*f)


#Loading    of    the    data:
Prefereable format .MUL

data1                         =
read_data(data_file_path)

data                          =
np.resize(data1,(63,30000))


#The name of the channels. It
can be modified as desired. By
default 63

ch_names  =  ['FP1','Fz','F3',
'F7',  'FT9',  'FC5',  'FC1',
'C3','T7','TP9',


'CP5','CP1','Pz','P3','P7','O1
','Oz','O2','P4','P8','TP10','
CP6',


'CP2','C4','T8','FT10','FC6','
FC2','F4','F8','FP2','AF7','AF
3',


'AFz','F1','F5','FT7','FC3','F
Cz','C1','C5','TP7','CP3','P1'
,'P5',


'PO7','PO3','POz','PO4','PO8',
'P6','P2','CPz','CP4','TP8','C
6',


'C2','FC4','FT8','F6','F2','AF
4','AF8']


#In the unit parameter "cm"/"m"
can be given
```

```
montage                          =
mne.channels.read_montage(kind
='Cap63',ch_names = ch_names,


path    =    path,    unit='cm',
transform=False)

#creating  the  channel  info
instance

info                             =
mne.create_info(ch_names     =
ch_names,   sfreq=250,ch_types
='eeg',

                         montage
= montage, verbose = None)

#Creating the raw instance of
the data

raw                              =
mne.io.RawArray(data,info,firs
t_samp= 0, verbose = None)



#""""  OPTIONAL  PARTS  Raw  data
visualization """

#Auto scaling option

scalings ='auto'

raw.plot(n_channels    =    63,
scalings=scalings,title='Auto-
scaled Data from arrays',

         show=True,block=True)
```

```
#EEG Microstates

# Segment the data in number of
microstates

n_states  =  int(input("Please
provide    the    number    of
Microstates: "))

if n_states <2 :

    print("The    number    of
microstates  must  be  equal
greater than or equal to 2" )


n_inits  =  int(input("Please
give the number of random
initializations to use for the
k-means algorithm: "))


maps,      segmentation      =
microstates.segment(raw.get_da
ta(),    n_states=    n_states,
n_inits = n_inits)


# Plot the topographic maps of
the   microstates   and   the
segmentation

print("    Visualizing    the
topographical maps of the EEG
Micrsotates ")


microstates.plot_maps(maps,
raw.info)
```

```
#Plotting the segementation for first 600 time samples

microstates.plot_segmentation(segmentation[:600],
raw.get_data()[:, :600],raw.times[:600])
```

Microstates Module Code:

The required code for Microstates module: "microstates.py"

```
"""

Functions to segment EEG into
microstates. Based on the
Microsegment toolbox

for EEGlab, written by Andreas
Trier Poulsen [1]_.


Reference:

Author: Marijn van Vliet
<w.m.vanvliet@gmail.com>


References

----------

.. [1]  Poulsen, A. T., Pedroni,
A., Langer, N., &  Hansen, L. K.
(2018).

        Microstate EEGlab
toolbox: An introductionary
guide. bioRxiv.

"""

import warnings

import numpy as np

from scipy.stats import zscore

from scipy.signal import
find_peaks

from scipy.linalg import eigh

import matplotlib as mpl

from matplotlib import pyplot as
plt

import mne

from mne.utils import logger,
verbose


@verbose

def segment(data, n_states=4,
n_inits=10, max_iter=1000,
thresh=1e-6,

            normalize=False,
min_peak_dist=2,
max_n_peaks=10000,

            random_state=None,
verbose=None):

    """Segment a continuous
signal into microstates.


    Peaks in the global field
power (GFP) are used to find
microstates, using a

    modified K-means algorithm.
Several runs of the modified K-
means algorithm

    are performed, using
different random
initializations. The run that

    resulted in the best
segmentation, as measured by
global explained variance

    (GEV), is used.


    Parameters

    ----------

    data : ndarray, shape
(n_channels, n_samples)
```

The data to find the microstates in

n_states : int

The number of unique microstates to find. Defaults to 4.

n_inits : int

The number of random initializations to use for the k-means algorithm.

The best fitting segmentation across all initializations is used.

Defaults to 10.

max_iter : int

The maximum number of iterations to perform in the k-means algorithm.

Defaults to 1000.

thresh : float

The threshold of convergence for the k-means algorithm, based on

relative change in noise variance. Defaults to 1e-6.

normalize : bool

Whether to normalize (z-score) the data across time before running the

k-means algorithm. Defaults to ``False``.

min_peak_dist : int

Minimum distance (in samples) between peaks in the GFP. Defaults to 2.

max_n_peaks : int

Maximum number of GFP peaks to use in the k-means algorithm. Chosen

randomly. Defaults to 10000.

random_state : int | numpy.random.RandomState | None

The seed or ``RandomState`` for the random number generator. Defaults

to ``None``, in which case a different seed is chosen each time this

function is called.

verbose : int | bool | None

Controls the verbosity.

Returns

----

maps : ndarray, shape (n_channels, n_states)

The topographic maps of the found unique microstates.

segmentation : ndarray, shape (n_samples,)

For each sample, the index of the microstate to which the sample has

been assigned.

References

.. [1] Pascual-Marqui, R. D., Michel, C. M., & Lehmann, D. (1995).

```
            Segmentation of brain
electrical activity into
microstates: model

            estimation and
validation. IEEE Transactions on
Biomedical

            Engineering.
    """

    logger.info('Finding %d
microstates, using %d random
intitializations for the k-means
algorithm' %

                  (n_states,
n_inits))



    # Convert min_peak_dist to
samples

    # min_peak_dist = 1 +
int(round(min_peak_dist *
raw.info['sfreq']))



    # Find peaks in the global
field power (GFP)

    gfp = data.std(axis=0)

    peaks, _ = find_peaks(gfp,
distance=min_peak_dist)

    n_peaks = len(peaks)



    # Limit the number of peaks
by randomly selecting them

    if max_n_peaks is not None:

        max_n_peaks =
min(n_peaks, max_n_peaks)

        if not
isinstance(random_state,
np.random.RandomState):

            random_state =
np.random.RandomState(random_sta
te)

        chosen_peaks =
random_state.choice(n_peaks,
size=max_n_peaks,


replace=False)

        peaks =
peaks[chosen_peaks]



    # Run microstates analysis
on selected data

    if normalize:

        data = zscore(data,
axis=1)



    # Cache this value for later

    gfp_sum_sq = np.sum(gfp **
2)



    # Do several runs of the k-
means algorithm, keep track of
the best

    # segmentation.

    best_gev = 0

    best_maps = None

    best_segmentation = None

    for _ in range(n_inits):

        maps, segmentation =
_mod_kmeans(data, n_states,
n_inits, max_iter,


thresh, random_state, verbose)
```

```
        map_corr =
_corr_vectors(data,
maps[segmentation].T)


        # Compare across
iterations using global
explained variance (GEV) of

        # the found microstates.

        gev = sum((gfp *
map_corr) ** 2) / gfp_sum_sq

        logger.info('GEV of
found microstates: %f' % gev)

        if gev > best_gev:

            best_gev, best_maps,
best_segmentation = gev, maps,
segmentation


    return best_maps,
best_segmentation




@verbose

def _mod_kmeans(data,
n_states=4, n_inits=10,
max_iter=1000, thresh=1e-6,


random_state=None,
verbose=None):

    """The modified K-means
clustering algorithm.


    See :func:`segment` for the
meaning of the parameters and
return

    values.

    """

    if not
isinstance(random_state,
np.random.RandomState):

        random_state =
np.random.RandomState(random_sta
te)

    n_channels, n_samples =
data.shape


    # Cache this value for later

    data_sum_sq = np.sum(data **
2)


    # Select random timepoints
for our initial topographic maps

    init_times =
random_state.choice(n_samples,
size=n_states, replace=False)

    maps = data[:, init_times].T

    maps /= np.linalg.norm(maps,
axis=1, keepdims=True)  #
Normalize the maps


    prev_residual = np.inf

    for iteration in
range(max_iter):

        # Assign each sample to
the best matching microstate

        activation =
maps.dot(data)

        segmentation =
np.argmax(np.abs(activation),
axis=0)

        # assigned_activations =
np.choose(segmentations,
all_activations)
```

```
        # Recompute the
topographic maps of the
microstates, based on the

        # samples that were
assigned to each state.

        for state in
range(n_states):

            idx = (segmentation
== state)

            if np.sum(idx) == 0:


warnings.warn('Some microstates
are never activated')

                maps[state] = 0

                continue


            # Find largest
eigenvector

            #cov = data[:,
idx].dot(data[:, idx].T)

            #_, vec = eigh(cov,
eigvals=(n_channels - 1,
n_channels - 1))

            #maps[state] =
vec.ravel()

            maps[state] =
data[:,
idx].dot(activation[state, idx])

            maps[state] /=
np.linalg.norm(maps[state])


        # Estimate residual
noise

        act_sum_sq =
np.sum(np.sum(maps[segmentation]
.T * data, axis=0) ** 2)

        residual =
abs(data_sum_sq - act_sum_sq)

        residual /=
float(n_samples * (n_channels -
1))


        # Have we converged?

        if (prev_residual -
residual) < (thresh * residual):


logger.info('Converged at %d
iterations.' % iteration)

            break


        prev_residual = residual

    else:

        warnings.warn('Modified
K-means algorithm failed to
converge.')


    # Compute final microstate
segmentations

    activation = maps.dot(data)

    segmentation =
np.argmax(activation ** 2,
axis=0)


    return maps, segmentation



def _corr_vectors(A, B, axis=0):

    """Compute pairwise
correlation of multiple pairs of
vectors.
```

```
    Fast way to compute
correlation of multiple pairs of
vectors without

    computing all pairs as would
with corr(A,B). Borrowed from
Oli at Stack

    overflow. Note the resulting
coefficients vary slightly from
the ones

    obtained from corr due
differences in the order of the
calculations.

    (Differences are of a
magnitude of 1e-9 to 1e-17
depending of the tested

    data).


    Parameters

    ----------

    A : ndarray, shape (n, m)

        The first collection of
vectors

    B : ndarray, shape (n, m)

        The second collection of
vectors

    axis : int

        The axis that contains
the elements of each vector.
Defaults to 0.


    Returns

    -------

    corr : ndarray, shape (m,)

        For each pair of
vectors, the correlation between
them.
    """

    An = A - np.mean(A,
axis=axis)

    Bn = B - np.mean(B,
axis=axis)

    An /= np.linalg.norm(An,
axis=axis)

    Bn /= np.linalg.norm(Bn,
axis=axis)

    return np.sum(An * Bn,
axis=axis)



def
plot_segmentation(segmentation,
data, times):

    """Plot a microstate
segmentation.


    Parameters

    ----------

    segmentation : list of int

        For each sample in time,
the index of the state to which
the sample has been assigned.


times : list of float

        The time-stamp for each
sample.
    """

    gfp = data.std(axis=0)
```

```
    n_states =
len(np.unique(segmentation))

    plt.figure(figsize=(6 *
np.ptp(times), 2))

    cmap =
plt.cm.get_cmap('plasma',
n_states)

    plt.plot(times, gfp,
color='black', linewidth=1)

    for state, color in
zip(range(n_states),
cmap.colors):

        plt.fill_between(times,
gfp, color=color,


where=(segmentation == state))

    norm =
mpl.colors.Normalize(vmin=0,
vmax=n_states)

    sm =
plt.cm.ScalarMappable(cmap=cmap,
norm=norm)

    sm.set_array([])

    plt.colorbar(sm)

    plt.yticks([])

    plt.xlabel('Time (s)')

    plt.title('Segmentation into
%d microstates' % n_states)

    plt.autoscale(tight=True)

    plt.tight_layout()



def plot_maps(maps, info):

    """Plot prototypical
microstate maps.

    Parameters

    ----------

    maps : ndarray, shape
(n_channels, n_maps)

        The prototypical
microstate maps.

    info : instance of
mne.io.Info

        The info structure of
the dataset, containing the
location of the

        sensors.
    """


    plt.figure(figsize=(5*
len(maps), 2))

    layout =
mne.channels.find_layout(info)

    for i, map in
enumerate(maps):

plt.subplot(1, len(maps), i+1)

mne.viz.plot_topomap(map,
layout.pos[:, :2])

# plt.title('%d' % i)
```