MiTfAT

Vahid S. Bokharaie

Jul 08, 2020

CONTENTS:

CHAPTER

ONE

FIRST STEP

1.1 Installation

To install, in your command prompt or bash, simply type:

```
pip install mitfat
```

If you are using Anaconda on Windows, better to open an Anaconda command prompt and then type the above.

Or if you want to work with the latest beta-release, you can find it in:

https://gitlab.tuebingen.mpg.de/vbokharaie/mitfat

If you don't know anything about Python:

then you should not worry. In order to use this code, you do not need to know anything about python. You just need to install it and then follow the instructions in the Usage Section to be able to run the code. But you need to install Python first. Python is just a programming language and unlike Matlab is not owned by a company or organization, hence you do not need to bu a license to use it. There are various ways to install Python, but the easiest is to go here and install Miniconda Python 3.x (3.7 at the time of writing). This will install Python and a minimal set of commonly used libraries (Libraries in Python are equivalent to toolboxes in Matlab). A little detail to keep in mind for Windows users is that you need to open an Anaconda Prompt (which you can find in your Start menu) and then type `pip install mitfat` to install the MitfAT library. Typing it in a normal windows command prompt (which you can open by typing `cmd` in 'Search program or file' in Start menu) might not work properly.

When Python is installed, then follow the instructions below to use this code to analyse your fMRI data. I should add though, that I sincerely hope using this code can motivate you to learn a bit about Python. I learned how to use Matlab 20 years ago and still use it to this day. But as I learn more about Python and what is available in this ecosystem, I use Matlab less and Python more and more everyday. Python provides powerful tools for you that you did not know you are missing when you were writing programs in Matlab. If you want to learn the basics of Python, I can suggest this online book to start with.

1.1.1 Requirements

```
"pandas",
"numpy",
"scipy",
"matplotlib",
"nibabel",
"nilearn",
"pathlib",
"click",
"seaborn",
"openpyxl",
```

1.2 Compatibility

This code is tested under Python 3.7, and should work well for all current version of Python 3.

1.3 License

GNU General Public License (Version 3).

1.4 Citation

Please cite this code as follows:

Bokharaie VS (2019) "MiTfAT: A Python-based fMRI Analysis Tool", Zenodo. https://doi.org/10.5281/zenodo.3372365.

This code was originally developed for a collaboration which led to the following publications:

Savić T. , Gambino G., Bokharaie V. S., Noori H. R., Logothetis N.K., Angelovski G., "Early detection and monitoring of cerebral ischemia using calcium-responsive MRI probes", PNAS, 2019.

1.5 Authors

MiTfAT is maintained by Vahid Samadi Bokharaie.

CHAPTER

TWO

BASIC USAGE

If you know a bit of python, I recommend looking into the two the script coming with the code. It will get you started. The code loads all the relevant available data into an `object` and then does various analysis and visualization. The information on what data file, and mask file to load, and some other (optional) parameters such as where to save files, what to name this experiment, is save in a text file and given to the script as an input argument. i have used the very helpful `click` module to handle the arguments, but if you don't want to bother with them, you can change the scrip and manually enter the info file name. Detailed information on how to write a info file is available in the next section. To run the script, you can find the `script_test.py` in the folder in which mitfat is installed. To find it, in python, you can type: .. code-block:: python

    import mitfat print(mitfat.__file__)

Open a comand prompt (bash in Linux or Anaconda prompt in windows), and then type: .. code-block:: bash

    python script_test.py

This will open the info file `sample_info_file.txt` which comes with the code and has the required info for the data. If you want to use another info file, type: .. code-block:: bash

    python script_test.py my_info_file.txt

Looking into the script will tell you enough about what to do.

Also, if you have data with trials, look at `script_trials.py` for a template of what can be done. .. code-block:: bash

    python script_trials.py my_info_file.txt

CHAPTER

THREE

LOADING THE DATA

All you need to do to load you full dataset, including data file, mask file, time stamps, and optional descriptive parameters, you should fill in a text file in a certain format (as explained below), to let MiTfAT know where and what are your input data. When this file is setup properly, and saved, for example as 'my_first_config_file.txt', then all you need to do to load the data is run Python, and then enter the following lines:

```
from mitfat.file_io import read_data
list_of_datasets = read_data(info_file)
my_first_dataset = list_of_datasets[0]
```

If you have defined trials in your experiment and keep repeating the same stimulus and record the responses, then `list_of_datasets` would contain all of those datasets in a separate `fmri_dataset` format. If no trials are defined, the list would contain only one fmri_dataset object.

If you are new to Python, you should know that you can copy the above or any set of python commands, paste them to a file, save it with `.py` extension, and then in the command prompt, type

```
python my_first_python_file.py
```

These are just very basic issues that you know already if you have ever used python. But if you haven't, you just need to spend half an hour learning them to be able to use this and a wealth of other libraries written in Python.

Note: Sample dataset:

When you install MiTfAT, it comes with a sample dataset. You can use the data to familiarize yourself with the package. To save you some file, there is also a sample Python script which includes all the commands you can see in this page. The file is called `mitfat_test_script.py`. Just search for it in your computer after installing MiTfAT.

To load the sample dataset, you can run the following commands:

```
from mitfat.file_io import read_data
import pkg_resources
info_file = pkg_resources.resource_filename('mitfat', 'sample_info_file.
↪txt')
DATA_PATH = pkg_resources.resource_filename('mitfat', 'datasets/')
list_datasets = read_data(info_file)
dataset1 = list_datasets[0]
```

In doing so, you have loaded all the available data in the sample dataset into an object called `dataset1` which hold all the relevant info of an fMRI recording. The sample dataset includes a nifty data file of 852 voxels, each recorded in 59 time-steps. The data files also include a text file of time-steps (in minutes) which will be used in x-axis of all relevant plots. There have been two 'events' during the recording which are added to the dataset. These two events split our time steps into three time-segments, which are also reflected in the plots.

How to fill in the config file:

The config file has a specific format. Each line that starts with one of a number of keywords is assumed to be followed by the information. For example, the following line provides a description of the dataset to be loaded by MiTfAT:

```
DESC: Just a sample dataset.
```

In order to save a copy of a sample config file in the current folder, enter the following two commands in python:

```
from mitfat.file_io import print_info
print_info('sample_config_file.txt')
```

This sample file includes a full description of how the config file should be written and default values for optional parameters.

There is also test script accompanying the code that you can access by entering the following commands in python:

```
from mitfat.file_io import test_script
test_script()
```

It will save a file called `MiTfAT_test_script.py` into the current python working directory. If you want to change the output file name, you can give the file name as an argument to `test_script()` function.

CHAPTER

FOUR

BASIC PLOTS

There are already some basic operation done on the data. For example, linear regression of the data in each time segment. Now you can plot raw, normalised and linearly regressed time-series for each voxel as follows. Plots will be saved under `output` folder which is created in the folder from which you are running the python.

```
# Basic plots of normalized time-series
dataset1.plot_basics()

# plots of linearly regressed time-series. Linear regression is performed on␣
↪each segment separately.
dataset1.plot_basics('lin_reg')

# plots of raw time-series
dataset1.plot_basics('raw')
```

Here is an example of the output plots:

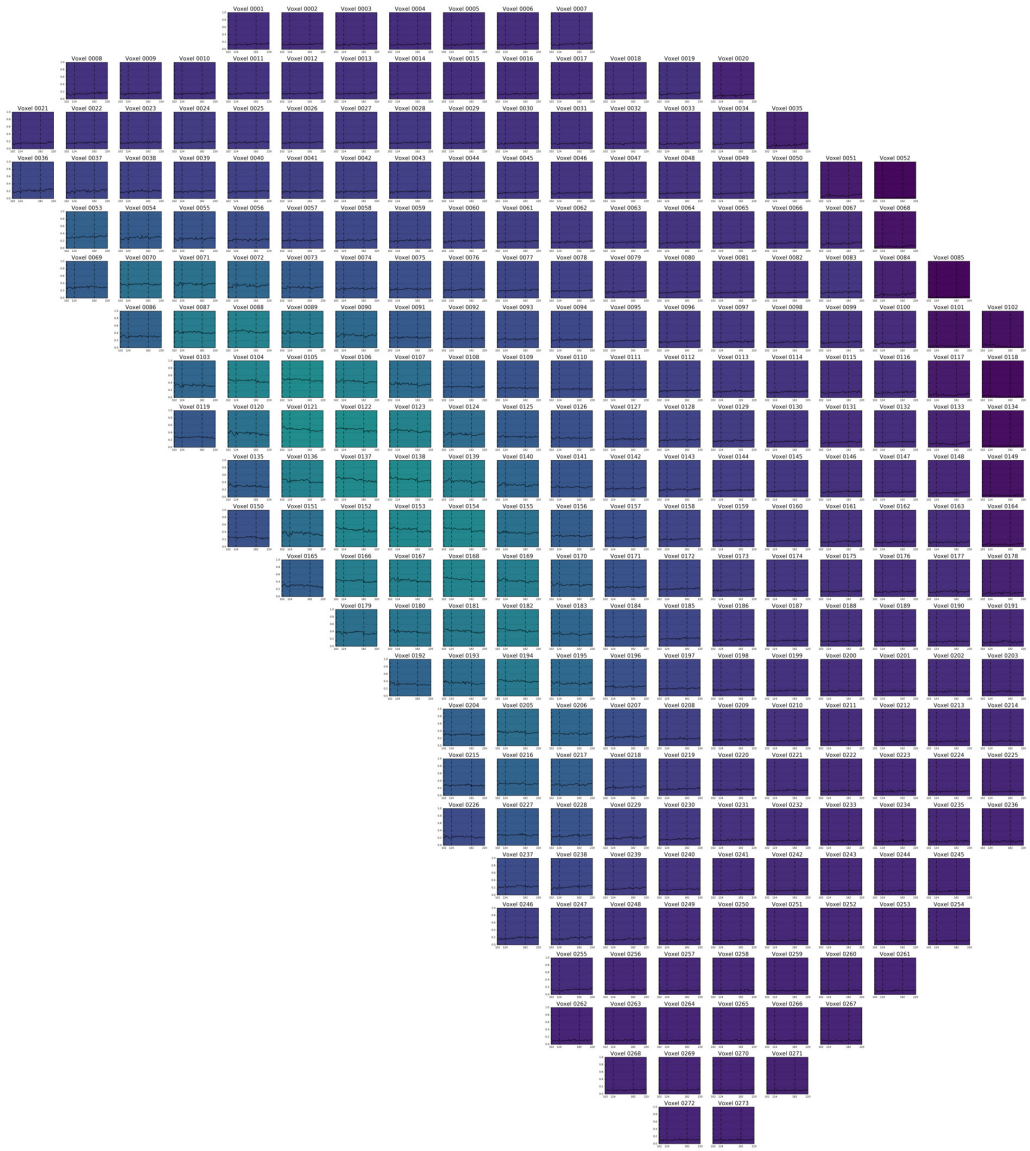

Fig. 1: Basic plots, layer 1 of the original mask.

Fig. 2: Basic plots, layer 2 of the original mask.

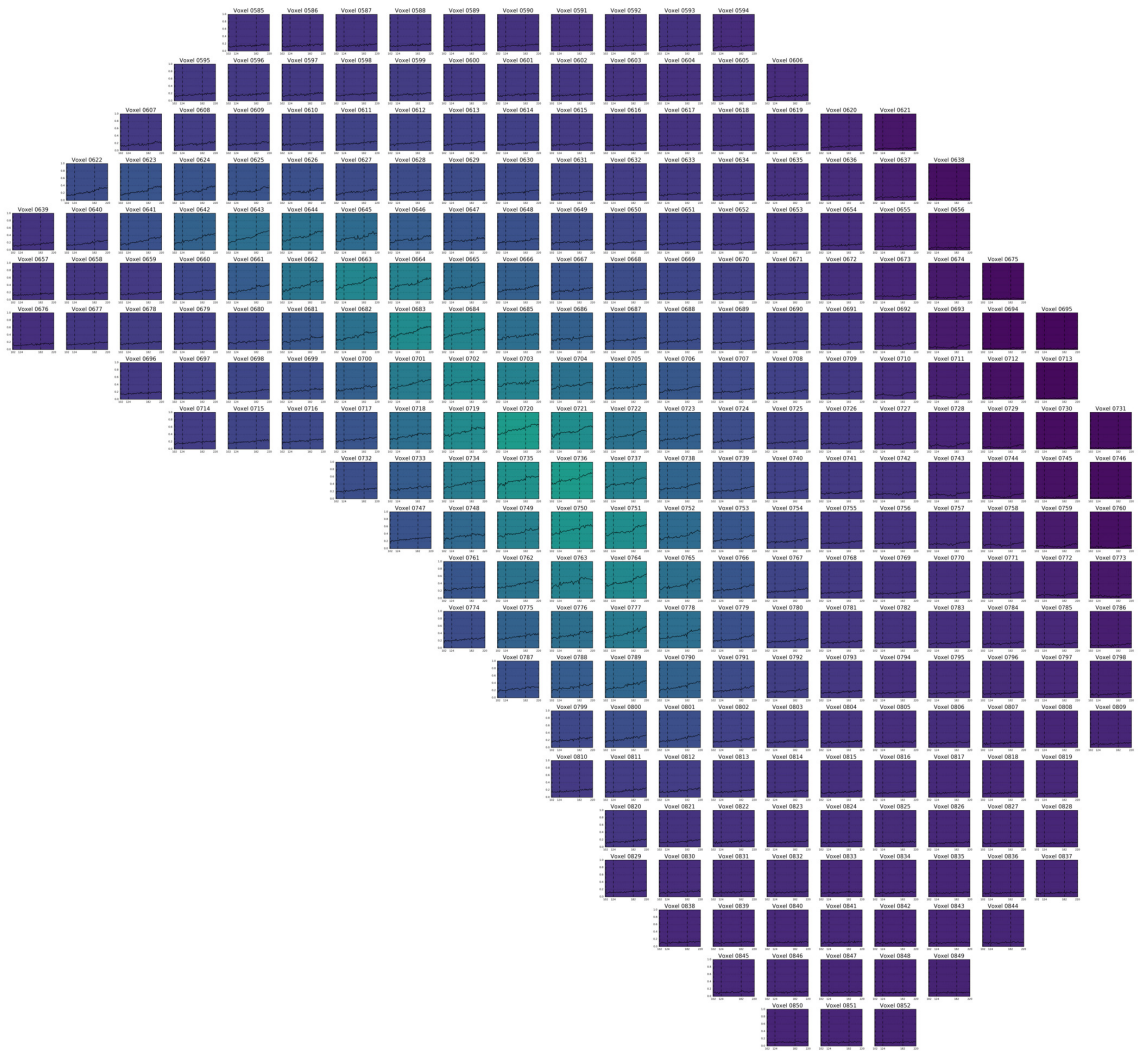

Fig. 3: Basic plots, layer 3 of the original mask.

CHAPTER

FIVE

CLUSTERING

You can also cluster the time-series (original or normalized versions), using Kmeans clustering from scikit-learn library:

```
X_train = dataset1.data
X_train_label = 'RAW_Normalised'  # used in plot titles only
num_clusters = 5
cluster_labels, cluster_centroid = dataset1.cluster(X_train, num_clusters)
dataset1.save_clusters(X_train, X_train_label, cluster_labels, cluster_
↪centroid)
dataset1.plot_clusters(X_train, X_train_label, cluster_labels, cluster_
↪centroid)
```

which would lead to plots such as these, plus an excel file including the centroids of the clusters:

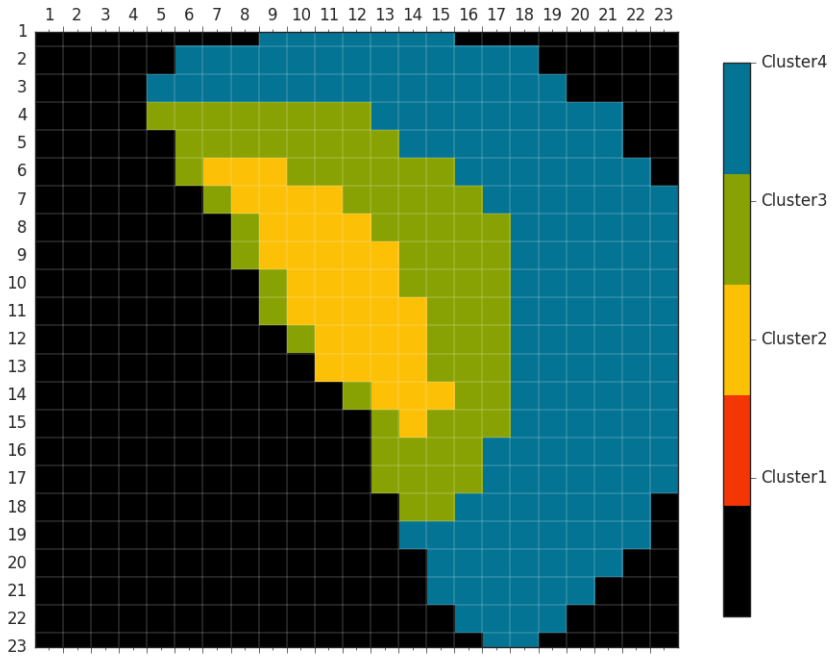

Fig. 1: Voxels color-coded based on their clusters, layer 3 of the mask.

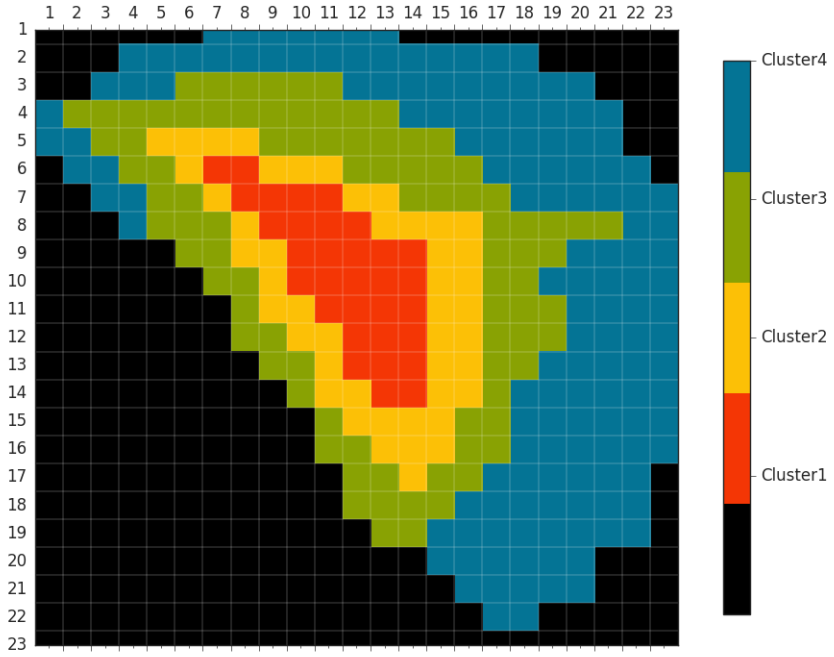

Fig. 2: Voxels color-coded based on their clusters, layer 3 of the mask.

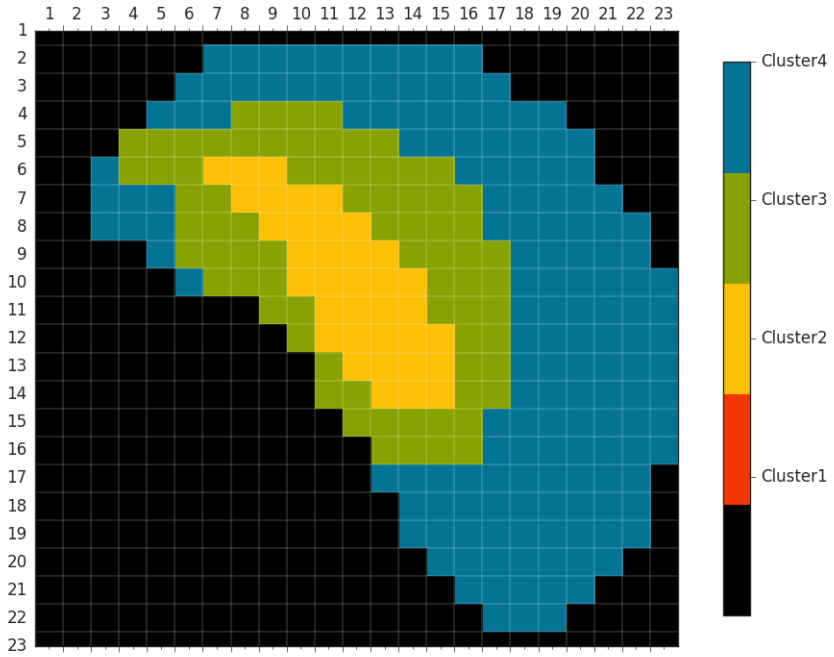

Fig. 3: Voxels color-coded based on their clusters, layer 3 of the mask.

And also the centroids:

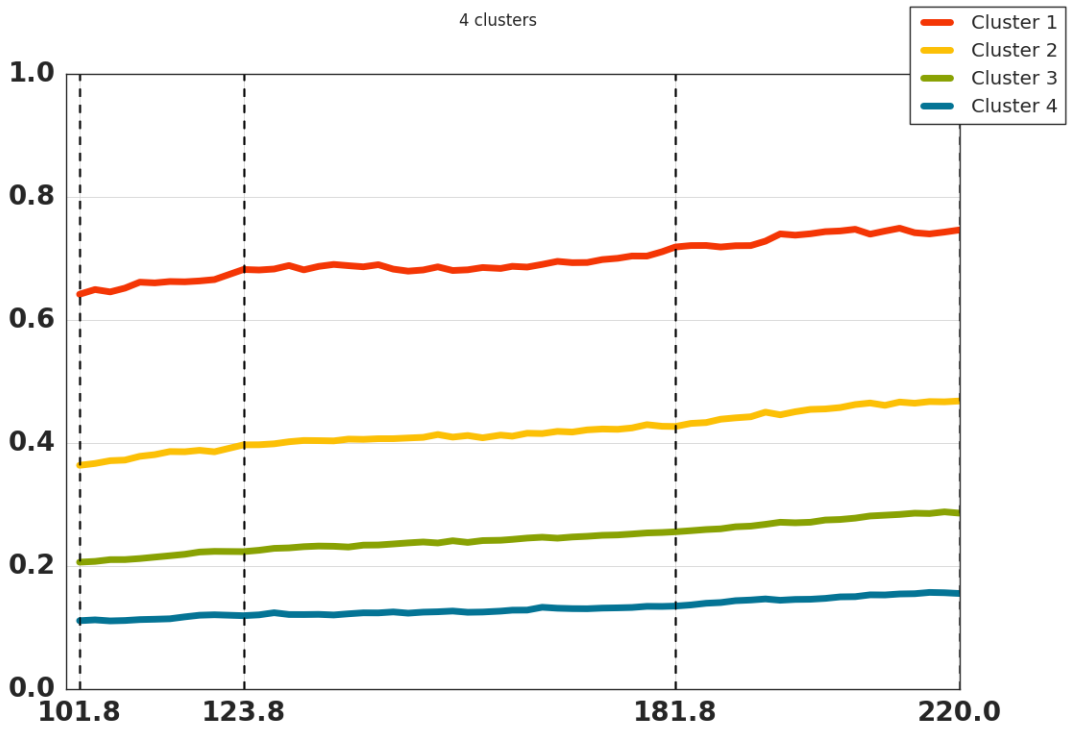

Fig. 4: Centroids.

or cluster voxels based on their mean value:

```
X_train = dataset1.data_mean
X_train_label = 'Mean_Normalised'
num_clusters = 4
cluster_labels, cluster_centroid = dataset1.cluster(X_train, num_clusters)
dataset1.save_clusters(X_train, X_train_label, cluster_labels, cluster_
↪centroid)
dataset1.plot_clusters(X_train, X_train_label, cluster_labels, cluster_
↪centroid)
```

or you can cluster your voxels based on the slope of your three segments.

```
X_train = dataset1.line_reg_slopes
X_train_label = 'Lin_regression_slopes_per_segments'
num_clusters = 4
cluster_labels, cluster_centroid = dataset1.cluster(X_train, num_clusters)
dataset1.save_clusters(X_train, X_train_label, cluster_labels, cluster_
↪centroid)
dataset1.plot_clusters(X_train, X_train_label, cluster_labels, cluster_
↪centroid, if_slopes=True)
```

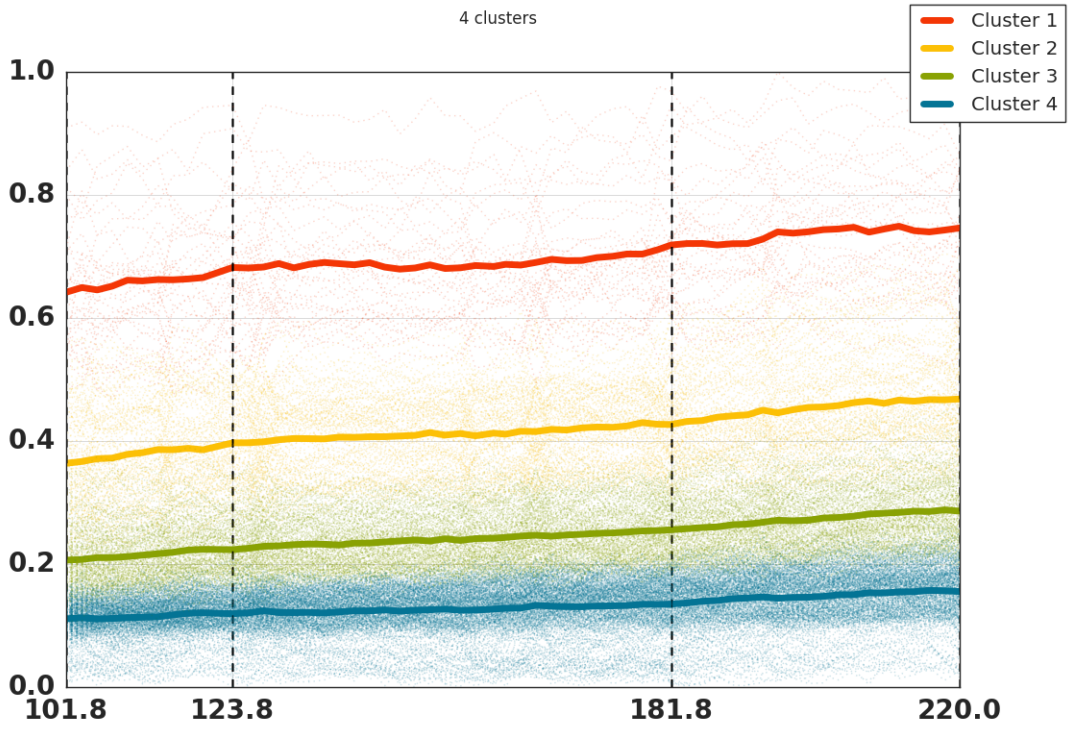

Fig. 5: Centroids with original data.

CHAPTER

SIX

HIERARCHICAL CLUSTERING

You can also do hierarchical clustering. This technique is quite useful when you are not sure about the size of your original mask. You might have included some voxels in the mask which are too noisy and including them in clustering might lead to redundant results. Hierarchical clustering works in two steps. In the first step, a Kmeans clustering with 2 clusters is performed. This will separate voxels based on their signal-to-noise ratio. The algorithm selects voxels corresponding with the centroid which has a higher absolute mean value. Then another Kmeans with two clusters is performed over these voxels, and the result is saved and plotted. You can do all that by simply typing:

```
signal = 'raw'  # can be 'raw', 'mean', 'slope', 'slope'Segments', 'mean_segments'
dataset1.cluster_hierarchial(signal, if_save_plot=True)
```

This leads to a decrease of the total number of considered voxels, which can be seen in the following figure.

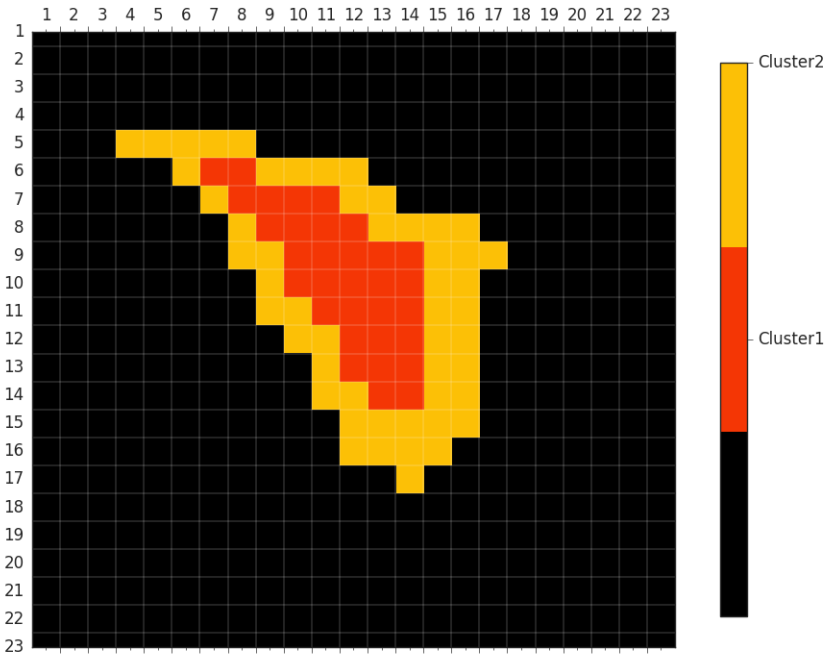

Fig. 1: Voxels color-coded based on their clusters, layer 2 of the mask.

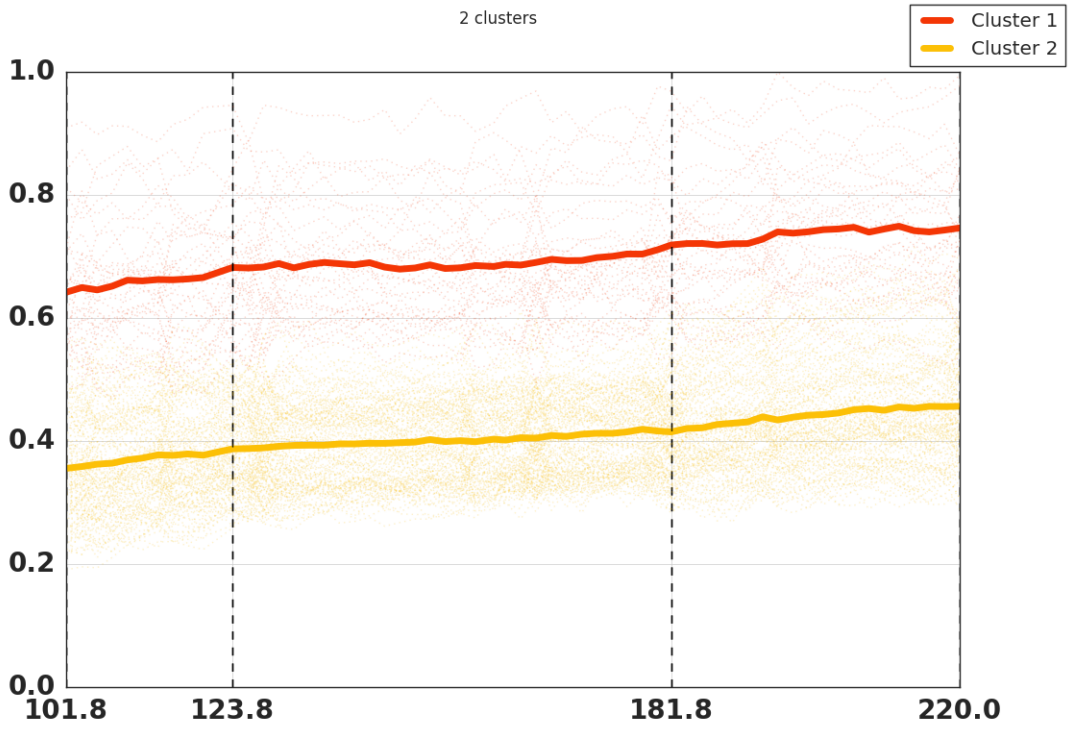

Fig. 2: Centroids.

CHAPTER

SEVEN

DETRENDING

You can detrend the data in MiTfAT. This is done in two different ways, based on the nature of the data. If there is no 'events' for the data, then detrending includes Spline smoothing of the data. If there are two events, which means time range is split into three time-segments, then a spline is fit to the first and third segments, and then this smoothed curve is used to interpolate the values of the second segment and then compared with the actual data. The underlying assumption has been that the recording conditions has changed in the second time-segment, and we want to extract the changes only due to that change from the information in the first and third segments. Detrending currently works only for data with zero or two 'events' (one or three time-segments), but will be extended to general case in future releases.

As an example, assume the average of a number of voxels of interests looks like the black line in the following image:

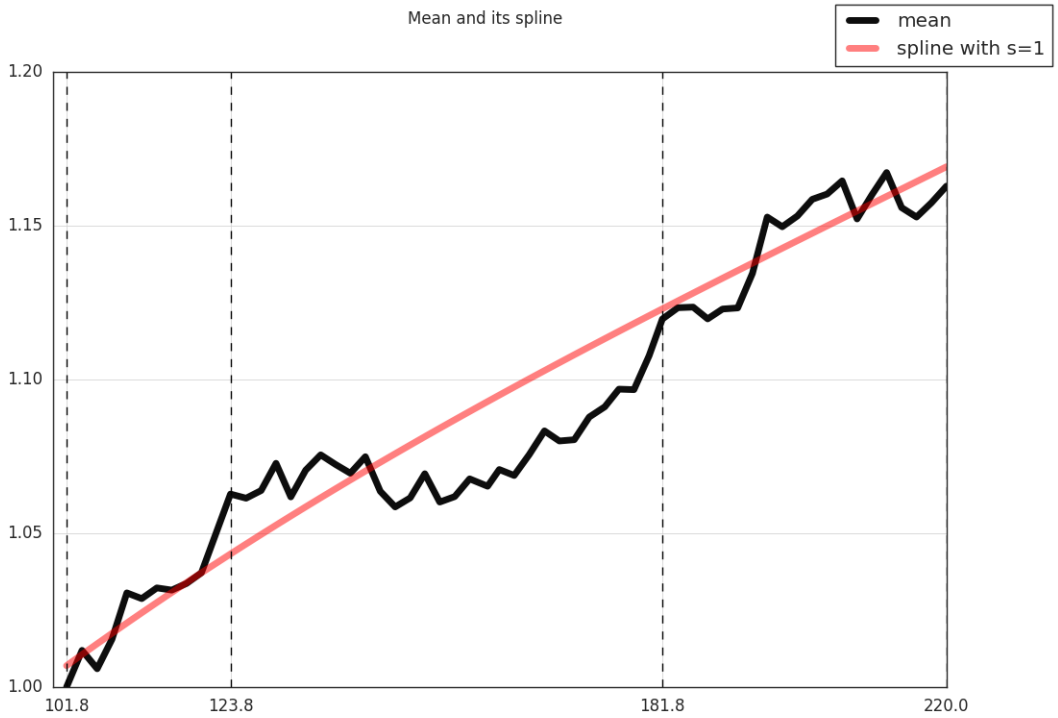

Fig. 1: The signal of interest (black), and the spline calculated only based on segments 1 and 3 (red).

The red signal is a spline which is fit only to first and last segment of the original signal. The part of the red curve in segment 2 represents the interpolation of segment 2 based on this spline curve. The difference between red and black lines in segment 2 represents the changes of the signal trend in segment 2.

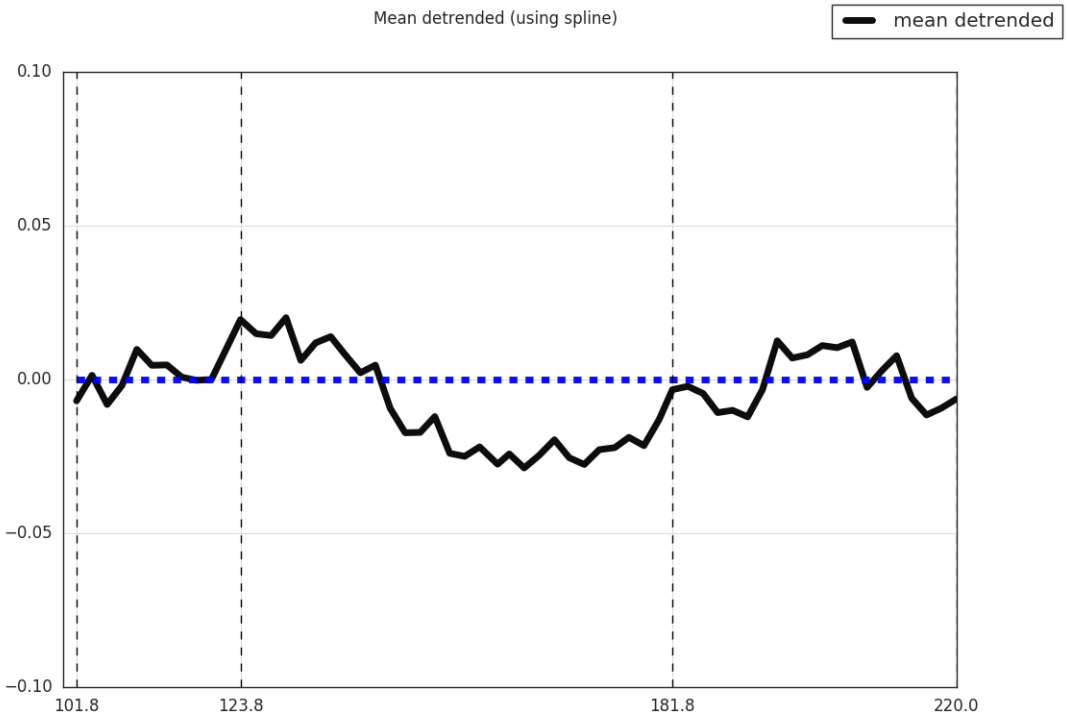

Fig. 2: The detrended signal.

CHAPTER

EIGHT

EXPERIMENTS INCLUDING TRIALS

If you run an experiment, which includes trials of a fixed length, you can use MiTfAT to analyze such data. Let's say each 10 time-steps in your fMRI recording represent one trial. To tell MiTfAT that it should divide the fMRI time-series into a number of trials, you should have the following lines in the info file:

```
FIRST_TRIAL_TIME: 0.0
TIME_STEP: 3
TRIAL_LENGTH: 30.0
EVENT_TIMES: 6.0, 12.0
```

`FIRST_TRIAL_TIME` specifies the time for the first step of the first trial. `TIME_STEP` specifies the time between each two samples. If you don't care about the actual time, just set this to 1. `TRIAL_LENGTH` specifies how long (in units of time) is each trial. `EVENT_TIMES` specifies possible event times of interest in each trial.

Looking at script_trails.py you can see what kinds of analysis you can perform on the data. For example, you can choose top N voxels (in terms of amplitude) in the overall time-series of the fMRI signal, and then average these N voxels, and cluster the average over all trials. You can then plot the centroids of these clusters and also a heat-map indicating to which cluster each trial belongs. This will be quite useful in checking possible patterns in the data.

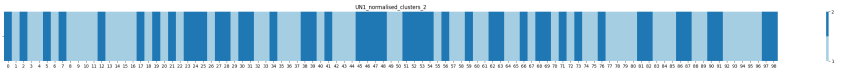

Fig. 1: Sequence of Cluster labels for each trial in an experiment with 99 trials.

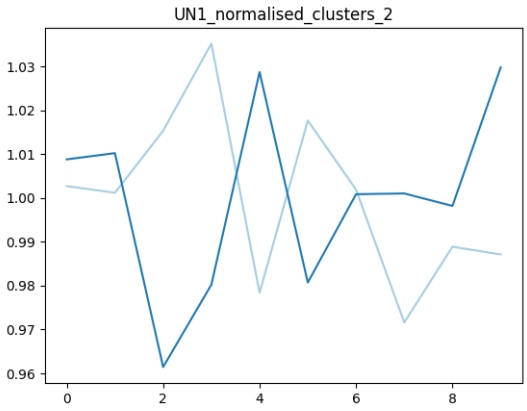

Fig. 2: Centroids of two clusters of the average signals for each trial

As a reminder, to make the above plots, we cacluated the mean for N voxels in each trials. And clustered these signals to see if and how the trials differ. It is also worth averaging all these signals to see the average response over all

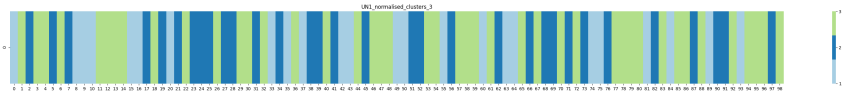

Fig. 3: Sequence of Cluster labels for each trial in an experiment with 99 trials.

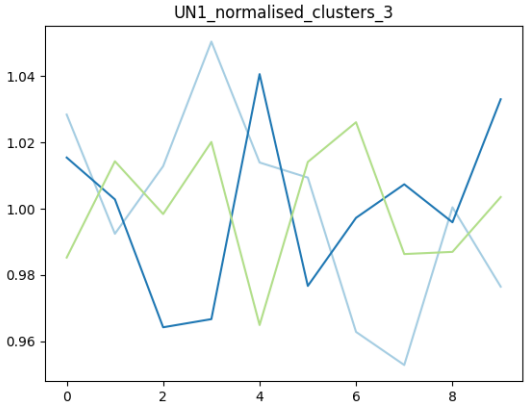

Fig. 4: Centroids of three clusters, time-series being the average signals of each trial

trials. This is possible in two way: either averaging the raw signals, or normalising the signal for each trial first, then averaging them.

We can also group trials and cluster these groups. This is useful when there might be a delay in the fMRI response with respect to the onset of the stimulus. Below, you can see cluster over signals obtained over each two consecutive trials. It means the first set if the signals obtained by concatenating trials 1 and 2, then 2 and 3, and so on.

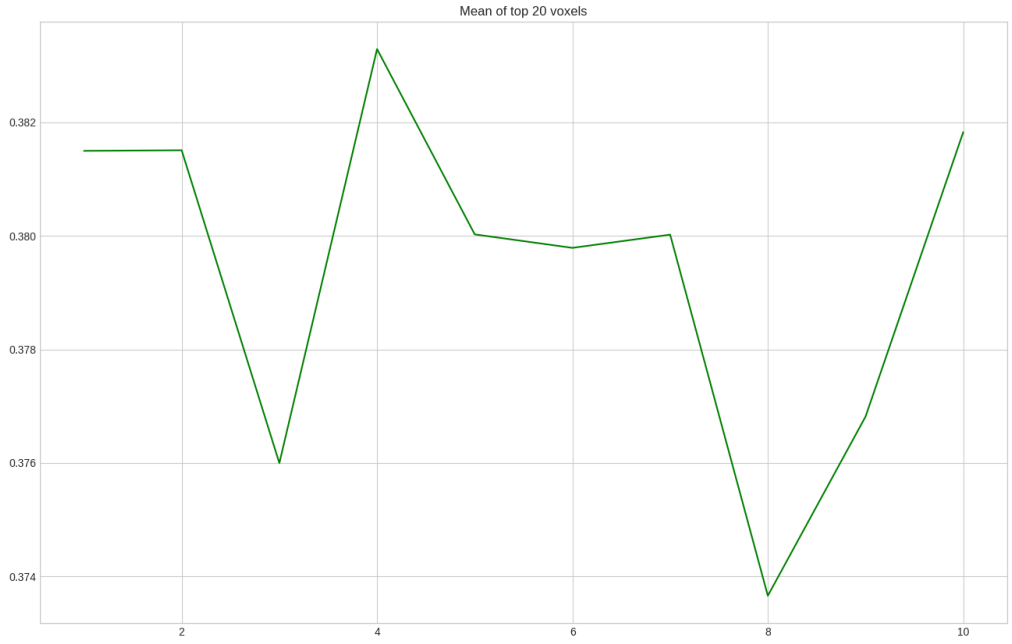

Fig. 5: Average of all 99 trials.

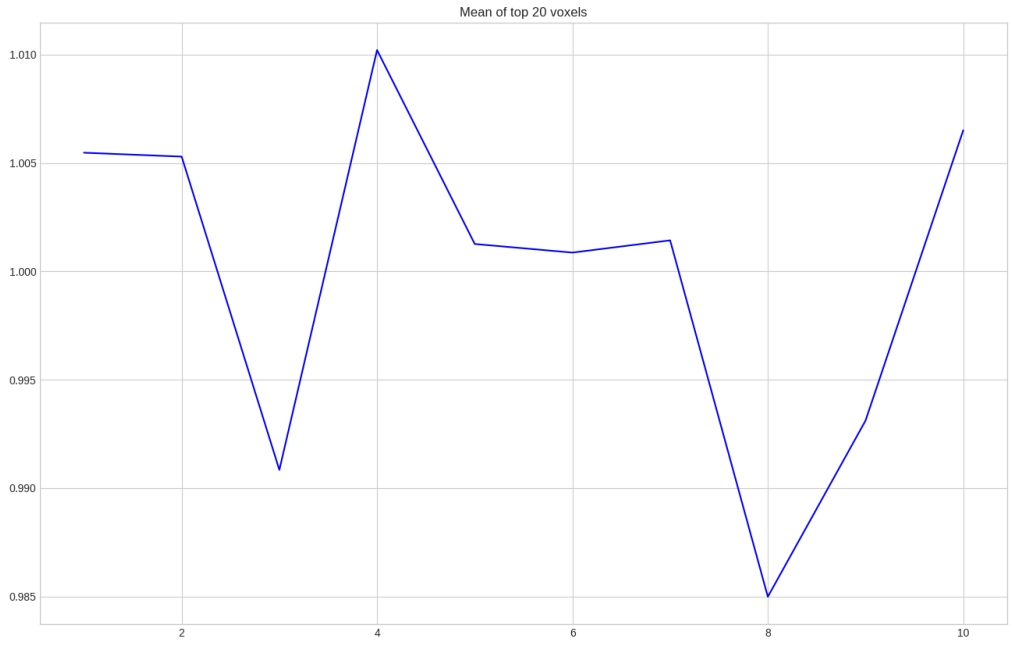

Fig. 6: Average of all 99 trials, each trial signal is normalised before averaging.

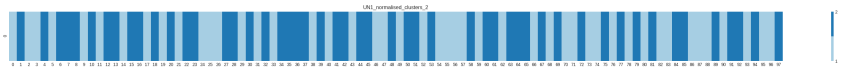

Fig. 7: Sequence of Cluster labels for each two consecutive trial in an experiment with 99 trials.

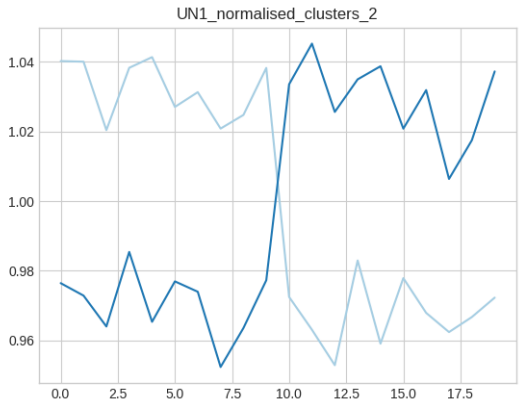

Fig. 8: Centroids of two clusters of the average signals for each two consecutive trial

CHAPTER

NINE

ALTERING THE DATA AFTER LOADING

If you want to change a property (attribute in Pythons-speak) of the dataset, different than what was written in the config file when you loaded the data, you can do so easily. For example, the example dataset is set to save the output plots and excel files into a subfolder called 'output' under the current python working directory. If you want to change it, you can simply do the following:

```
dataset1.dir_save = 'COPY_FOLDER_PATH_HERE'
```

You can change various other properties of the dataset. There are some of these attributes which are read directly from the input data-files:

```
data_raw  #raw data
data  # normalized data.
data_mean  # mean value of time-series for each voxel
mask  # the fMRI mask. Number of 1s in it should match data.shape[1]
num_voxels  # generated from data
num_time_steps  # generated from data
time_steps  # (OPTIONAL) a float array of time-steps.
            # Their length should match sdata.shape[0].
            # If there is no such data available, time-steps would be
            # consecutive integers to match data shape.
indices_cutoff  # (OPTIONAL) indices of 'events'.
                # If dataset has for example 59 time-steps,
                # it should be a list of integers in [0,58] range.
                # something like [11, 42].
dir_source  # (OPTIONAL) from which folder the data was loaded?
            # default is 'dataset' subfolder in the folder containing
            #  the config file (config file explained below)
dir_save  # (OPTIONAL) from which folder the data was loaded?
            # default is 'output' subfolder in the current folder in python
```

When the `dataset1` object is created, linear regression is automatically performed on the normalised data. If `indices_cutoff` is not empty, linear regression is performed on each segment separately. Results of linear regression are stored in the following attributes:

```
line_reg  # the same shape as data, but linearised version of the data
line_reg_slopes  # slopes. If there are for example 2 cutoff indices,
                 # it will of [3, N_voxels] shape. Otherwise [1, N_voxels]
line_reg_biases  # offsets for each linear regression.
                 # Same dimensions as linear_reg_slopes.
```

And then there are some which merely hold miscellaneous information and can be change at will:

```
signal_name  # a string, can be 'T1w', 'T2*', 'FISP'
             # or any other signal you have recorded
experiment_name  # a string
dataset_no  # an integer.
mol_name  # Molecule name. Useful for molecular fMRI studies.
description  # a short description
```

CHAPTER

TEN

MITFAT PACKAGE

10.1 Submodules

10.2 mitfat.bplots module

Created on Mon Feb 24 17:38:15 2020

@author: vbokharaie

mitfat.bplots.**plot_and_save_bbox_continuous**(*my_bbox*, *dir_save_bb*, *sup_title=[]*, *v_min=0.0*, *v_max=1.0*)

Plots continuous bbox when elemnts are continuous values (such as mean) default color pallete is default (virdis), 0 will be navy and 1 will be yellow.

my_bbox: 'numpy.ndarray' dir_save_bb: 'str'

pth to save folder

suptitle: 'str' optional used to title plots

mitfat.bplots.**plot_and_save_bbox_discrete**(*my_bbox*, *dir_save_bb*, *sup_title=[]*, *limits=[]*, *colours=[]*)

Plots continuous bbox when elemnts are continuous values (such as mean) default color pallete is default (virdis), 0 will be navy and 1 will be yellow.

my_bbox: 'numpy.ndarray' dir_save_bb: 'str'

path to save folder

suptitle: 'str' optional used to title plots

limits: 'list' ['float'] colours: 'list', matplotlib colors

mitfat.bplots.**plot_line**(*data_array*, *time_array*, *figsize=(16, 10)*, *title_str=''*, *label_str=''*, *vert_line_times=[]*, *color='black'*)

mitfat.bplots.**plot_trial_cluster_seq**(*centroid_to_plot*, *labels_to_plot*, *my_title*, *dir_save='./clusters_one_trial'*, *Cluster_num=2*)

mitfat.bplots.**save_fig**(*fig*, *dir_save*, *filename_in*, *figsize=(16, 12)*)

10.3 mitfat.clustering module

Created on Tue Jul 23 18:54:26 2019

@author: vbokharaie

This module includes methods of fmri_dataset class used for clustering.

mitfat.clustering.**cluster_raw**(*X_train*, *no_clusters*)
kmean clustering of time-series data. Sorts the cluster such that cluser 1 always corresponds with

the one with highest absolute mean.

X_train: 'numpy.ndarray', (N_time_steps, N_voxels) no_clusters: 'int'

cluster_labels_sorted: 'numpy.ndarray', (N_voxel, 1) cluster_centroids_sorted:'numpy.ndarray', (N_time_steps, N_voxels)

mitfat.clustering.**cluster_scalar**(*X_train*, *no_clusters*)
kmean clustering of scalara data. Sorts the cluster such that cluser 1 always corresponds with

the one with highest absolute mean.

X_train: 'numpy.ndarray', (N=1, N_voxels) no_clusters: 'int'

cluster_labels_sorted: 'numpy.ndarray', (N_voxel, 1) cluster_centroids_sorted:'numpy.ndarray',

mitfat.clustering.**kmean_it**(*self*, *cluster_no*)
a wrapper for sklearn.cluster.KMeans sorts the clusters, assigns cluster labels to bbox non-zero cells

cluster_no: 'int'

bbox_cat: 'numpy.ndarray', (e1, e2, e3) cluster_labels_sorted: 'numpy.ndarray' cluster_centroids_sorted: 'numpy.ndarray'

mitfat.clustering.**kmeans_missing**(*xx_*, *no_clusters*, *max_iter=30*)
Perform K-Means clustering on data with missing values. Found it in stackoverflow Argumentss:

xx_: An [n_samples, n_features] array of data to cluster. n_clusters: Number of clusters to form. max_iter: Maximum number of EM iterations to perform.

Returns: labels: An [n_samples] vector of integer labels. centroids: An [n_clusters, n_features] array of cluster centroids. x_hat: Copy of xx_ with the missing values filled in.

mitfat.clustering.**sort_cluster**(*cluster_labels*, *cluster_centroids*)
sorts the cluster numbers, such that cluster 1 always correspond with centroid with highest mean absolute value.

cluster_labels: cluster_centroids:

cluster_labels_out: clusters_centroids_out:

10.4 mitfat.collections module

Created on Fri Nov 22 12:58:00 2019

@author: vbokharaie

Thie mosule includes functions which will operate on a list of fmri_datasets, such as calcultating the average of a list of datasets.

mitfat.collections.**calc_mean_dataset**(*list_of_fmri_datasets*)
Reads a list of fmri_datasets, returns an fmri_dataset which contains the average of the data in all of them.

Parameters

- **list_of_fmri_datasets** – list of fmri_datastes to be avegred
- **first_trial_time** – the beginning time of the first trial, default 0.0

Raises Happiness level, and checks if input is a list and a list of fmri_dataset

Returns fmri_dataset, average of input list data

Return type 'fmri_dataset' type

10.5 mitfat.file_io module

Created on Thu Aug 13 14:55:19 2018

@author: vbokharaie

This module includes methods of fmri_dataset class used for I/O operations.

mitfat.file_io.**convert_real_time_to_index**(*time_steps*, *cutoff_times_to_convert*)
returns index values for first time steps bigger than input time-values. Parameters ————- times_files: 'str'

path to file including values of time-steps

cutoff_times_to_convert: 'list' of 'float'

output_list: 'list' if 'int' indices for time values

mitfat.file_io.**main_get_data**(*info_file_name*)
reads actual data based on info_file_name. uses read_info_file to read the config file and then loads the data.

A 'sample_info_file.txt' accompanying the library includes standrard format.

info_file_name: 'str' path to config file

data_nii_masked: 'numpy.ndarray', (N_time_steps, N_voxels) mask_roi_numpy: 'numpy.ndarray', (d1, d2, d3) time_steps: 'list' of 'float' signal_name: 'str' indices_cuttoff: 'list' if 'int' experiment_name: 'str' dataset_no: 'int' mol_name: 'str' dir_source: 'str' dir_save: 'str'mask_roi_numpy description: 'str'

mitfat.file_io.**make_list_of_trial_times**(*time_steps*, *first_trial_time=0.0*, *trial_length=None*)
Reads time-steps, first trial time and trial length, calculates ands returns the indices and time-steps of consecutive trials

Parameters

- **time_steps** (*list of floats*) – time-steps of the fMRI recording
- **first_trial_time** (*float, optional*) – the beginning time of the first trial, default 0.0
- **first_trial_time** – length of each trial, default None (wouold be set to max(time_steps))

Raises Happiness level

Returns list_of_trial_times, list_of_trial_times_indices

Return type list of floats, list of integers

mitfat.file_io.**print_info**(*filename_info=None*)

mitfat.file_io.**read_data**(*info_file_name*)
Reads the input dataset info from a file, returns a fmri_dataset object including all those files.

info_file_name: str an os.path filename object

fmri_dataset: obj

mitfat.file_io.**read_info_file**(*info_file_name*)

reads the config file. assigns default values to fmri_dataset object if not defined in config file. A 'sample_info_file.txt' accompanying the library includes standrard format. Parameters ———- info_file_name: 'str'

path to config file

data_file_name: 'str' mask_file_name: 'str' time_file_name: 'str' dir_source: 'str' dir_save: 'str' mol_name: 'str' exp_name: 'str' dataset_no: 'int' cutoff_times: 'list' of 'float' description: 'str' signal_name: 'str'

mitfat.file_io.**save_clusters**(*self*, *X_train*, *data_label*, *cluster_labels*, *cluster_centroids*, *subfolder_c=''*)

Save cluster to excel file.

X_train: 'numpy.ndarray', (N_clustered_data, N_voxels) N_clustered_data can be N_time_steps, 1, or N_segments

data_label: 'str' used in establishing save folders

cluster_labels: 'numpy.ndarray', (N_voxels, 1) cluster_centroids: 'numpy.ndarray', (N_clusters, N_clustered_data)

mitfat.file_io.**test_script**(*filename=None*)

10.6 mitfat.flags module

Created on Tue Jul 23 18:21:14 2019

@author: vbokharaie

10.7 mitfat.fmri_dataset module

Created on Thu Jul 26 10:55:19 2018

@author: vbokharaie

class mitfat.fmri_dataset.**fmri_dataset**(*data_masked*, *mask_numpy*, *time_steps*, *signal_name*, *indices_cutoff*, *experiment_name*, *dataset_no*, *mol_name*, *dir_source*, *dir_save*, *description*, *first_trial_time*, *trial_length*, *trial_no=0*)

Bases: object

A class as a container for all relvant data of an fMRI recording.

class method are distributed in different modules whose names start with '_'

calc_lin_reg()

Calculate linear regression for the time-series.

Uses numpy.linalg.lstsq.

line_reg : 'numpy.ndarray', data.shape line_reg_slopes: 'numpy.ndarray', (num_voxels, num_segments) line_reg_biases: 'numpy.ndarray', (num_voxels, num_segments)

calc_mean_segments()
Mean value of each time-segment in the time-series.

Time-segments are automatically defined based on specified values of cutoff indices ot times. If they are not defined, there would be only one time-segment including all time-steps.

mean_segments : 'numpy.ndarray', (len_segments-1, num_voxels)

cluster(*X_train*, *num_clusters*)
A wrapper for clustering mthoeds. calls cluster_scalar if input data is 1 dimentional, otherwise calls cluster_raw.

X_train: 'numpy.ndarray', (N=1, N_voxels) no_clusters: 'int'

cluster_labels_sorted: 'numpy.ndarray', (N_voxel, 1) cluster_centroids_sorted:'numpy.ndarray',

cluster_scalar, cluster_raw

cluster_TopN(*signal='raw'*, *n_top=20*, *if_save_plot=True*, *subfolder_c='Cluster_TopN'*)
Just plotting Top N voxels, i nterm of signal amplitude.

signal: {'raw', 'mean', 'slope', 'slope'Segments', 'mean_segments'} num_clusters: int

default=2

if_save_plot: 'bool' default=False

cluster_labels2: 'list' of 'int' cluster_centroid2: 'numpy.ndarray'

Incomplete for 'mean', 'slope', 'slope'Segments', 'mean_segments'

cluster_hierarchial(*signal='raw'*, *num_clusters=2*, *if_save_plot=False*, *subfolder_c='Cluster_Hierarchical'*)
Hiearchical clustering First 2 cluster over all data, choose all voxels corresponding to bigger centroid Then 2 clusters over these voxels.

signal: {'raw', 'mean', 'slope', 'slope'Segments', 'mean_segments'} num_clusters: int

default=2

if_save_plot: 'bool' default=False

cluster_labels2: 'list' of 'int' cluster_centroid2: 'numpy.ndarray'

Incomplete for 'mean', 'slope', 'slope'Segments', 'mean_segments'

data = None
normalised, unless flags.if_normalise=False

data_mean = None
mean of each voxels

data_raw = None
raw data, always

detrend()
Detrending the time-series. When cutoff times are set merely to begining and end times:

spline-smooth the time-series, and subtract it from the original signal.

For three segments: Fits a spline to first and third time-segments. Interpolates the second segment, and subtract it from the original time-series values. In doing so, chnages in the signal trend in segment 2 becomes evident.

Saves detrended plots, saved in png and optionaly eps format. Saves bar plots of detrended time segments, saved in png and optionaly eps format Saves an .xlsx file including values of detrended signals

fix_anomolies()
Fix overshoot and undershoots occasioanlly recorded by the fMRI scanners.

Replaces any value outside +-3 SD range of signal MEAN with np.nan.

my_data : 'numpy.ndarray', np.shape(self.data)

impute()
Interpolates nan values in the time series.

uses sklearn.impute.SimpleImputer

data: 'numpy.ndarray'

mask = None
3d mask

normalise()
Normalise the time-series. Divides all voxel time-series to their collective maximum value.

data_normalised : 'numpy.ndarray' data: 'numpy.ndarray' bbox_data_mean: 'numpy.ndarray' data_mean: 'numpy.ndarray' mean_segments : 'numpy.ndarray'

num_time_steps = None
int

num_voxels = None
inti

plot_basics(*data_type='normalised'*, *figsize=None*)
Plots the time-series corresponsing to each voxel. The plots are arranged in a layout which follows the mask. Each layer is saved in a separeta file. Parameters ———- data_type : {'normalised', 'raw', 'lin_reg'}

'normalised' (default): plot normalised time-series. 'raw': plot raw signal 'lin_reg': plot linear regression of time-series

figsize [set of float, (fig_w, fig_h)] fig dimensions in inches

NameError If data is not normalised and user tries to plot normalised data.

plot_clusters(*original_data*, *data_label*, *cluster_labels*, *cluster_centroids*, *if_slopes=False*, *if_hierarchical=False*, *subfolder_c=''*)
Plots the cluster, saves cntroid values. Including bbox plots, centroid plots. centroids are saved in .xlsx file in the same folder as plots.

The plots are arranged in a layout which follows the mask. Each layer is saved in a separeta file.

original_data: 'numpy.ndarray', (N_clustered_data, N_voxels) N_clustered_data can be N_time_steps, 1, or N_segments

data_label: 'str' used in establishing save folders

cluster_labels: 'numpy.ndarray', (N_voxels, 1) cluster_centroids: 'numpy.ndarray', (N_clusters, N_clustered_data) subfolder_c: str,

name of subfolder inside main dir_save to save the plots

NameError If data is not normalised and user tries to plot normalised data.

plot_voxel(*data_type='normalised'*, *voxel_num=1*, *list_voxel_num=None*, *figsize=(16, 10)*, *dir_save=None*, *filename_pre=''*)

Plots the time-series corresponsing to each voxel. The plots are arranged in a layout which follows the mask. Each layer is saved in a separeta file. Parameters ———- data_type : {'normalised', 'raw', 'lin_reg'}

'normalised' (default): plot normalised time-series. 'raw': plot raw signal 'lin_reg': plot linear regression of time-series

figsize [set of float, (fig_w, fig_h)] fig dimensions in inches

voxel_num: int voxels number to be plotter. shoud be in (1, self.voxel_no) range

NameError If data is not normalised and user tries to plot normalised data.

save_clusters(*X_train*, *data_label*, *cluster_labels*, *cluster_centroids*, *subfolder_c=''*)

Save cluster to excel file.

X_train: 'numpy.ndarray', (N_clustered_data, N_voxels) N_clustered_data can be N_time_steps, 1, or N_segments

data_label: 'str' used in establishing save folders

cluster_labels: 'numpy.ndarray', (N_voxels, 1) cluster_centroids: 'numpy.ndarray', (N_clusters, N_clustered_data)

time_steps = None

flow list

total_created_objects = 0

keeps track of total number of objects

10.8 mitfat.mitfat_test_script module

10.9 Module contents

MiTfAT - python-based fMRI Analysis Tool.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

T