easyupMVPA – Manual

Author: Maurice Hollmann

Version: 1.1 beta

Date: 09/2012

Contents

Introduction 3
Installation 3
Initialization 3
Basic Concepts 4
Data Management 4
2D case: 4
4D case: 5
Saving datasets to hard disk 6
Preprocessing 7
Classification 8
Feature Selection 8
Full Example: The complete workflow on single-subject test-data 8
The example-dataset 8
The Script 8
Method Reference 12

Introduction

The easyupMVPA toolbox is aimed to enable fast and simple access to the multivariate analysis of data typical for neuroscience using pattern classification and regression. The focus is functional MRI data (time series of 3D native space) but with the toolbox any data (i.e. EEG, MEG, …) may be analyzed. The idea is to allow the easy implementation of the complete workflow of a pattern analysis using a very low number of commands by providing high-level methods to access the complete preprocessing, model-estimation, and prediction functionality.

The toolbox uses parallel processing if the Matlab "Distributed Computing Toolbox" is installed and there are more than one central processing unit / cores available.

All functions of the toolbox have a detailed HTML-description in the API-Documentation:

{dir_toolbox}/doc/ API_doc/easyupMVPA/index.html

Installation

To install the toolbox copy the toolbox files to any desired folder. Then set the matlab path variable to the toolbox path using Matalb-GUI or the command `addpath('directory of the toolbox')`.
Needed Matlab toolboxes are the "Statistics Toolbox" and if parallel computing should be used the "Distributed Computing Toolbox".

Initialization

At the beginning of every matlab-script using the easyupMVPA toolbox, the toolbox has to be initialized. This is done by the function (given arguments here restrict the number of cores to use to 2 and suppress toolbox-messages during processing):

```
easyupMVPA_init('nmbCores', 2,'quietMode', true);
```

Called without any arguments the toolbox will use the maximum number of cores available for parallel processing and toolbox messages will be allowed.

Basic Concepts

Data Management

This section describes how the information necessary for classification is stored in the toolbox. There are two objects that can be used for data storage: *dataset4D* and *dataset2D*. The 4D variant is especially defined for functional fMRI data and allows for easy access an manipulation if the time-series should be processed. Another possibility is to save all data in the "allround" dataset2D where every kind of data, like EEG, can be saved.

Of course a dataset2D can contain fMRI data too, but then a 3D volume has already to be transformed into a 1D vector, which means that the user has to take care of the transformation between the 1D and 3D space representation.

The information stored in datasets cover the following data:

- The raw data or already processed data
    - For functional MRI usually 4D (time series of 3D volumes)
    - For non-3D data the 2D representation may be used
- If applicable, a mask
    - For functional MRI defined in the 3D-space of the functional data
    - For 2D datasets this is a 1D vector
- If applicable, a feature selection map
    - For functional MRI defined in the 3D-space of the functional data
    - For 2D datasets this is a 1D vector
- Information about the structure of your data (classes, runs etc.)
- Further information like file-lists or header information of functional data

The main data structure in the toolbox is called **dataset**. An empty dataset may be created using the function: *getEmptyDataset4D()* or *getEmptyDataset2D().*

A *dataset* is a simple matlab-struct that has the following fields:

2D case:

dataset.type

- Always set to "dataset2D"

dataset.is4D

- Always set to false

dataset.is2D

- Always set to true

dataset.data

- A series of 1D volumes with the dimensions: **X** × **N** (*N* is the number of samples). This data can contain any raw or preprocessed data, or statistical results
  (for EEG data this may a concatenated time course of the signal of some electrodes (1D) for *N* events)

dataset.mask

- A 1D vector that is optional and can describe parts of the feature space that should be excluded (for EEG this may be time points that should not be analyzed)

dataset.featureSelectionMap

- A 1D vector that is usually set during the analysis by "feature selection" methods. It describes parts of the feature space that should be included in classification steps. Elements that are not part the *mask1D* will not be included in a *featureSelectionMap1D*.

dataset.classIDs

- A vector containing your class labels (starting from 0) , for every sample in the data1D field a classID must be given, that means the vector contains always *N* elements (e.g. [0 0 0 1 1 1 0 0 0 1 1 1])
- If the dataset should be used for regression, the labels can also be scalars (e.g. [1.2 0.345 0.56 0.01 1.32 1 2.1 0])

dataset.chunks

- A vector containing further information about your data structure e.g. the runs in your experiment – For example the vector [0 1 1 0 2 2 0 3 3 0 4 4] codes that the samples 2 and 3 are part of the first run, volume 5 and 6 from the second and so on. The length of the chunks vector must also be *N*.

4D case:

dataset.type

- Always set to "dataset4D"

dataset.is4D

- Always set to true

dataset.is2D

- Always set to false

dataset.data

- A time series of 3D volumes with the dimensions: **X** × **Y** × **Z** × **N** (*N* is the number of scans or samples). This data can contain raw fMRI data, preprocessed fMRI data, statistical results like beta-maps from a general linear model, or any other 3D data (e.g. structural images).

dataset.mask

- A 3D volume that is optional and can describe parts of the volume that should be included in the analysis (e.g. a mask that defines brain tissue used to exclude the background)

dataset.featureSelectionMap

- A 3D volume that is usually set during the analysis by "feature selection" methods. It describes parts of the volume that should be included in classification steps. voxels that are not part the *mask3D* will not be included in a *featureSelectionMap3D*.

dataset.classIDs

- A vector containing your class labels (starting from 0) , for every 3D volume in the data4D field a classID must be given, that means the vector contains always *N* elements (e.g. [0 0 0 1 1 1 0 0 0 1 1 1])
- If the dataset should be used for regression the labels can also be scalars (e.g. [1.2 0.345 0.56 0.01 1.32 1 2.1 0])

dataset.chunks

- A vector containing further information about your data structure e.g. the runs in your experiment – For example the vector [0 1 1 0 2 2 0 3 3 0 4 4] codes that the volumes 2 and 3 are part of the first run, volume 5 and 6 from the second and so on. It also sets a zero for the first volume of every run, which may be a transition scan that should be marked for later processing steps. The length of the chunks vector must also be *N*.

dataset.dataFilelist

- If the 4D data was created using functional images, a list of these is stored here. , This field is usually set automatically during data preparation.

dataset.data4D_3DNiftiHdr

- If the 4D data was created using functional images, the header information of a single 3D volume is stored here. This field is usually set automatically during data preparation.

Important: The information in a *dataset* can be assessed directly, but it is strongly recommended to set the content solely by the functions provided by the toolbox (see example section).

Saving datasets to hard disk

When working with large datasets it is advantageous not to do all data creation or preprocessing steps every time again for the same dataset. Therefore it is a clever idea to save the dataset for example after preprocessing and to load it if one wants to work with it. Let us assume we have a dataset object called "myDataset". Using the following code Matlab saves it to the hard disk (using very efficient encoding):

```
%choose a filename and the file location
fName = ['path_to_save_to', filesep(), 'nameOfDataset.mat']];
save(fName, 'myDataset');
```

The function *filesep()* just returns the file separator for the actual operating system (e.g. "\" for Windows).
Now we can load the above saved dataset using:

```
load(fName);
```

This call will create the variable *myDataset* in the current workspace. Maybe you already recognized that the load-function creates a variable that does not need to be known before. Thus it might be a clever idea to

name the saved .mat file in a manner that you know which variable will be available after loading it (i.e. “myDataset_sav110301.mat” for the object “myDataset”).

Preprocessing

The preprocessing offered in this multivariate pattern analysis toolbox consists of three basic methods:

- Elementwise (i.e. Voxelwise) Linear Detrending
- Elementwise (i.e. Voxelwise) High-Pass Filtering
- Elementwise (i.e. Voxelwise) Z-Scoring

Other preprocessing steps like slice-time correction, motion correction, normalization, (moderate) smoothing in space domain etc. can (and should) be done before processing the data with *easyupMVPA*. For this purposes there is already very sophisticated software available like SPM or FSL. The Linear Detrending and the High-Pass Filtering both preserve the mean value (in time-course) for every voxel - the Z-scoring does not.

Important: All preprocessing methods get a dataset as input and return a modified dataset. If you don’t want to loose the unprocessed data you will have to store it in a different dataset-object (i.e. calling the methods with a different return variable name than the given argument) or save it before processing!

The **Linear Detrending** algorithm used here calculates the fit of a line on the time-course of every single voxel. It is possible to call the method with the optional argument *breakpoints*, that allow to define an expected break in the time-course, i.e. if the input consists of several time-independent sessions (see also example section).

Example Call:

```
%first of two sessions ends at scan 100 (overwrites the data in myDataset)
myDataset = doLinearDetrending(myDataset, [100 101]);
```

Highpass-Filtering is done using a phase-preserving algorithm that takes the sampling frequency and the cutoff-frequency as input. All frequencies below the input frequency will be removed. Please choose the cutoff with respect to your experimental design (should be at least 2 times lower than your stimulation frequency). The standard value used if no cutoff-frequency is given is 0.0078 Hz (128s wave length).

Example Call:

```
%TR = 2s, cutoff-freq = 0.0033 Hz (300s wavelength)
myDataset = doHighpassFiltering(myDataset, 0.5, 0.0033);
```

The **Z-Scoring** consists of the voxelwise substraction of the mean and scaling the data to have a standard-deviation of one. This is necessary if the results of several subjects should be compared.

Example Call:

```
myDataset = doZScoring(myDataset);
```

Classification

Coming soon…

Feature Selection

Coming soon…

Full Example: Classification - The complete workflow on single-subject test-data

The example-dataset

The data provided in the file: **easyupMVPA_Examle.zip**. In the subfolders you will find all necessary data and the example script. For additional information to the experiment see [Hollmann, M. et al. (2011) Neural correlates of the volitional regulation of the desire for food. Int. J. Obes. DOI: 10.1038/ ijo.2011.125].

MRI parameters of the experiment:

TR: 2000 ms, TE: 29 ms, Slice Thickness: 4.5 mm, Matrix: 64x64, Slices: 28, FOV:

The experimental design was a 2 x 2 factorial. The volunteers saw luminance and contrast matched images of unhealthy food that was rated by them beforehand for tastiness (chosen images: **tasty**, **non-tasty**). The 2 goal conditions were: **crave** (allow the desire for the food) and **non-crave** (suppressing the desire for the food). The whole experiment was split in 2 sessions a 507 scans with a break of 3 minutes. A session was divided in runs that consisted of 3 images each. Every run was introduced with the written sentence: "Zulassen" (Allow) or "Unterdrücken" (supress), that coded the task for the volunteers for the following 3 images. The ISI between images was randomized btw. 8 and 10 seconds and after every run the volunteers rated their performance by pressing one of four buttons with the fingers of the right hand.

The test-data consist of the two sessions of one volunteer and the raw-fMRI dataset was preprocessed using SPM 5. The preprocessing included: slice-time-correction, motion-correction, normalization to 3 x 3 x 3 m^3 MNI, and smoothing with a gauss-kernel of 8 x 8 x 8 m^3.

The Script

The complete source code of the example can be found in the "example/ example_4D_fMRI_classification/"-folder in the file: **easyupMVPA_exampleSingleSubject.m**.

At first the scan directory is saved as a string, the path to the toolbox is set (please modify `dir_toolbox` for your setup) and the toolbox is initialized:

```
%set the folder pointing to the data files
dir_scans = [pwd, filesep(), 'exampleData', filesep()];
```

```
%initialize the toolbox (string dir_toolbox points to the place where easyupMVPA is saved)
addpath(dir_toolbox);
easyupMVPA_init();
```

An empty dataset is created:

```
%create an empty dataset
myDataset = getEmpty4DDataset();
```

Now an array of strings is defined, that holds all header files of the functional data used here (If SPM is installed also the spm_get method may be used to select files in a folder: `fileList = spm_get('Files', dir_scans, '*.hdr')`). This array is used to set the *data* field of the dataset:

```
%fileList will be a character array in this case
fileList = [[dir_scans, 'swra_session1_epi_2D_standard_fkt.hdr']; ...
            [dir_scans, 'swra_session2_epi_2D_standard_fkt.hdr']];

%set the field data4D of the dataset
myDataset = setDataset_data_ByFilelist(myDataset, fileList);
```

Now the filename for the attributes-file is set and this file is used to set both: the *chunks* and the *classIDs* of the dataset:

```
%get the attributes from a txt file
attribFile = [dir_scans,'attributes_CTvsNCT.txt'];

%set the chunks
myDataset= setDataset_chunks_ByAttribFile(myDataset, attribFile);

%set the classIDs
myDataset = setDataset_classIDs_ByAttribFile(myDataset, attribFile);
```

It is always a good idea to exclude voxels that potentially have no information for classification. Because of this a brain-mask is set, that excludes all non-brain voxels from further processing. The original data is preserved, the *mask3D* field, if set, is just used to select a subset voxels. Of course this mask can be any other selection too. All non-zero voxels in an image file are used as inclusion:

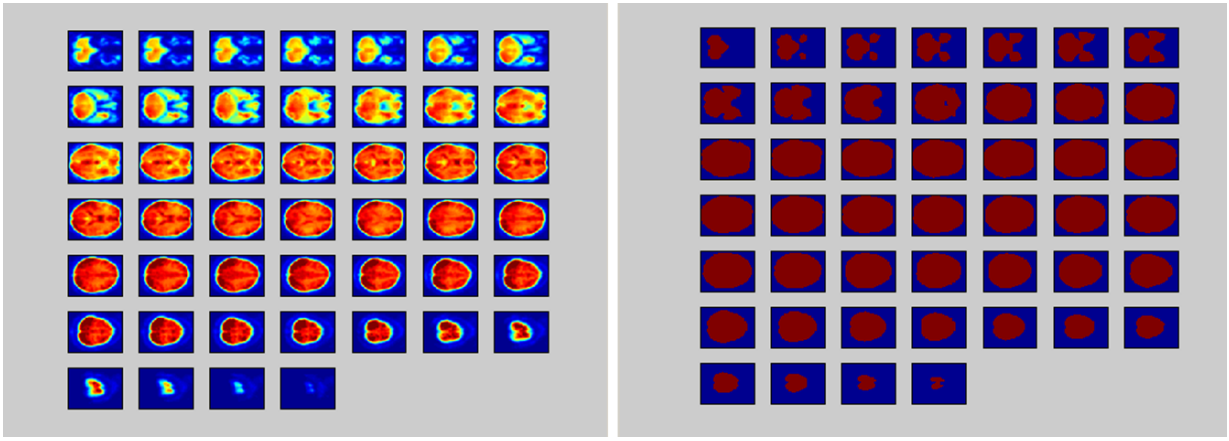
```
%set a brain mask
myDataset = setDataset_mask_ByImageFile(myDataset, ['templates',…
                                        filesep(),'brainMask_53_63_46.hdr']);
```

The dataset is filled with content. Now we have a look at the information by the **printDatasetInfo-**method. Furthermore It is possible to show a 3D-matrix as a color-coded MOSAIC-view (**showDataAsImage**). If the input is 4D, a third argument after the title of the resulting figure gives the index of the element in 4th dimension that should be shown.

```
%print the content of dataset
printDatasetInfo(myDataset);

%show the first image of data4D (last argument sets the figure title)
showDataAsImage(myDataset.data, 'First element in data', 1);

%show the mask3D (last argument sets the figure title)
showDataAsImage(myDataset.mask, 'Dataset mask');
```

This is the output at the Matlab Command Window (Please notice – the length of *classIDs*- and *chunks*-vectors is exactly the size of the 4th dimension of *data* and is equal to 2 x 507 = 1014, which is the number of scans in the whole experiment):

```
*** Info Dataset ***

Dataset Type: dataset4D

field data (4D):
              size : 53    63    46  1014
              class: int16
field dataFilelist:
              size : 2  1
              class: cell
field data_3DNiftiHdr:
              size : 0  0
              class: double
field mask (3D):
              size                  : 53  63  46
              nmb non-zero elements: 75533
              class                 : int16
field featureSelectionMap (3D):
               No featureSelectionMap is defined for dataset.
field chunks:
              length: 1014
              class : uint8
0   0   0   0   0   1   1   1   1   1   1   0   2   2   2   2   2   2   0   3   3   3 …
field classIDs:
              length: 1014
              class : double
0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  …
```



The two figures are showing the subset of *data* and the *mask* in MOSAIC-view.

Now we add the preprocessing. Here we deal with functional data and thus expect fluctuations and drifts in single voxel time courses. As it is standard in fMRI data processing we remove low frequencies (including linear drifts):

```
%highpass-filtering (Take a careful look at your design - it is possible that useful
%frequencies are removed!)
%In this case just a very low freq of more than 300 seconds is the cutoff,
%because of trial design (look at classIDs)
%The TR in our experiment is 2 s, thus the second argument (sampling frequency) is 0.5!
myDataset = doHighpassFiltering(myDataset, 0.5, 1/300);
```

Usually the preprocessing takes a while and one would not like to repeat it several times. Thus it might be a good idea to save the whole dataset after preprocessing (e.g. `save(fileName, 'myDataset');`) and load it each time the data should be re-processed (e.g. `load(fileName);`).

The following code further prepares the given data. We decided to remove transition scans (the first two scans after a stimulus onset) and thus apply a new attribute file where transition scans are marked with chunk = 0. Afterwards we select just the samples that are useful for us (chunks > 0).

```
%set new chunks with excluded transition scans
attribFile = fullfile(dir_scans,'attributes_noTransitions_CTvsNCT.txt');
myDataset = setDataset_chunks_ByAttribFile(myDataset, attribFile);

%select all samples that are not transitions
myDataset = selectSamples(myDataset,'chunks > 0');
```

For classification we now could use each single scan as input, but we reduce noise by averaging scans that belong to the same trial (they are assigned to the same chunks) by averaging over the chunks. This just means that scans having the same chunk id are averaged. This results in 60 single samples in the averaged dataset, because the whole data is compromised of 60 runs:

```
%average over samples (in example using chunks)
%the number of resulting samples is the number of unique chunks ids
myDataset = averageOverChunks(myDataset);
```

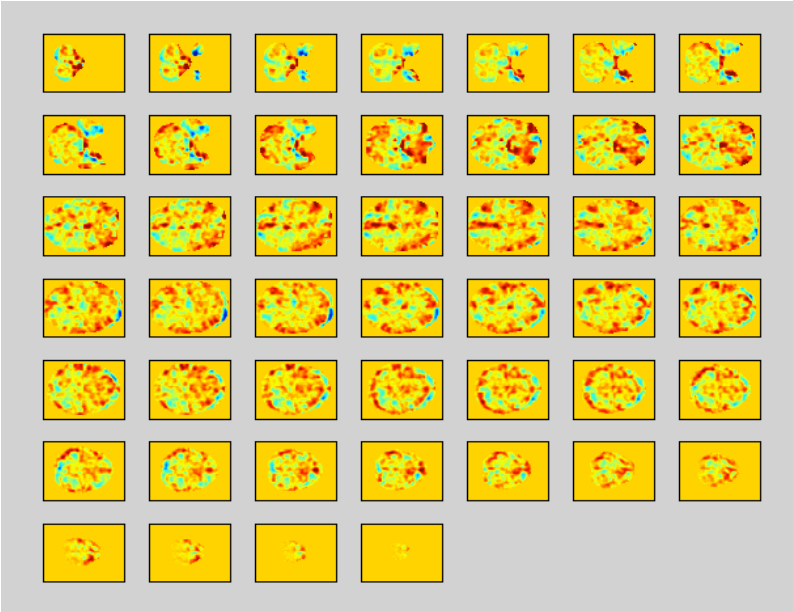
Eventually it gets exciting! We define a data splitter that creates a classification scheme for the following leave one out cross validation (LOOCV). This scheme is of type “oneSampleOut” which means, that in each iteration a different sample is selected as test set while all other samples are the training set. This results in as many iterations as samples are available.

```
%Leave one out cross validation with a oneSampleOut scheme
splitterOSO = getDataSplitter(myDataset, 'oneSampleOut');

%the LOOCV is called with the arguments: dataset, splitter, svmType, svmKernelType, slack variable
%for SVM tolerance
[myDataset, resultStruct, avgWeights3D] = doLeaveOneOutCrossValidation_SVM(myDataset,
splitterOSO,'classification', 'linear', 0.5);

printResultStruct(resultStruct);

showDataAsImage(avgWeights3D, 'Average weights3D');
```

If we print the resultStruct it should give the following output, which means that in the LOOCV around 71.7% of all tests were correct:

```
*** RESULT STRUCT:  ***
Number of Tests: 60
Accuracy:        71.6667 %
True Positives:  21
True Negatives:  22
False Positives: 8
False Negatives: 9
Sensitivity:     0.7  (TP/TP+FN = Proportion of true positives to all positives).
Specificity:     0.73333  (TN/TN+FP = Proportion of true negatives to all negatives).

Predicted class IDs / regression values:
                 size : 1  60
                 class: double
************************
```

The average weights the classifier assigned to the different feature space dimensions (i.e. voxels) are shown below. The absolute value of a weight depicts the importance of the voxel for classification. **Important:** Interpreting these weights directly in the way claimed before is just possible for **linear** SVMs!



Average weights assigned to the single voxels (red=positive, blue=negative)

Method Reference

For a HTML-reference of all methods of the toolbox see:

{dir_toolbox}/doc/ API_doc/easyupMVPA/index.html

Appendix

Toolbox versions

V 1.0

First stable version

V 1.1

- SVM Regression included