Supplement of J. Sens. Sens. Syst., 7, 489–506, 2018
https://doi.org/10.5194/jsss-7-489-2018-supplement

JSSS Journal of Sensors and Sensor Systems Open Access

Supplement of

DAV^{3}E – a MATLAB toolbox for multivariate sensor data evaluation

Manuel Bastuck et al.

Correspondence to: Manuel Bastuck (m.bastuck@lmt.uni-saarland.de)

Here we demonstrate the command line interface and use the steps involved to arrive from raw data to Fig. 16b. The presented syntax is based on a DAV³E version currently in development with improved command line interface.

The first step is to create a project as a container for everything else:

```
prj = Project();
```

The clusters are created directly from plain text files (tab-separated). In this case, the sampling rate must be supplied as it is not available from this file type.

```
cluster100Hz = Cluster.fromFile(...
    {'PS1','PS2','PS3','PS4','PS5','PS6','EPS1'},'tsv',...
    'samplingPeriod',0.01);
cluster10Hz = Cluster.fromFile(...
    {'FS1','FS2'},'tsv','samplingPeriod',0.1);
cluster1Hz = Cluster.fromFile(...
    {'TS1','TS2','TS3','TS4','VS1','CE','CP','SE'},'tsv',...
    'samplingPeriod',1);
```

We then create a feature definition set (fds) which holds one or more feature definitions (fd). A feature definition holds, in turn, a range, defining what part of the cycle to operate on, and the function to compute on this part. Note that the range is created from the cluster. This allows to give its begin and end in actual sample points, which are then internally converted to a timestamp, which is important to make the ranges independent of the sampling rate. The range that is created here will span the whole cycle independent of its sampling rate. Having only one range and one feature definition here is a very naïve approach and can, and should, be replaced by a more sophisticated selection for a real evaluation. The feature definition set is finally set as default in the project, so that it is automatically applied to every newly added sensor.

```
fds = FeatureDefinitionSet();
fd = FeatureDefinition(@FeatureExtraction.meanFeature,...
    cluster1Hz.makeIndexRange([1,60]));
fds.addFeatureDefinition(fd);
prj.poolFeatureDefinitionSets = fds;
```

The project setup is then finished by adding the clusters and defining a range to select which cycles shall be included in the evaluation. Similar to above, we create only one range spanning the whole measurement. This is because the target values are already available for each cycle. If this were not the case, and the data had to be annotated manually, several ranges like shown in Fig. 13 would significantly facilitate this process because groups of ranges can be annotated at once.

```
prj.addCluster([cluster100Hz,cluster10Hz,cluster1Hz]);
prj.ranges = cluster1Hz.makeCycleRange([1,2205]);
```

In the following, the 17 defined features (the mean value from each sensor) are computed with one simple command and can be fused in parallel employing a data processing block. Data processing blocks are the common interface for almost all functions in DAV³E, so that the user can extend many different areas of functionality after learning this basic, simple interface. They can have parameters and plot functions attached and, thus, provide improved functionality over a simple function while requiring less special knowledge from the user compared to writing class definitions. The output is a data object, which contains the concatenated features, but also meta data like target values, feature captions, etc. Features and cycles can be (de-)selected in the dataset and it ensures an always consistent state. Here, we load the target values from an external file.

```
features = prj.computeFeatures();
parallelFusion = DataProcessingBlock(@FeatureFusion.parallel);
data = parallelFusion.apply(features);
targets = dlmread('profile.txt','\t');
data.setTarget(targets(:,2),'numeric');
```

The data object will also contain predictions after a model has been applied to it, so that it can always traced back which data lead to which prediction. Hence, it makes sense to also define splits of the dataset here. DAV³E supports

nested cross-validation, which is automatically applied when both validation and testing are set to cross-validation. In this case, we only validate the model with cross-validation, but test it with a hold-out set. Defining this within in the data object again ensures traceability and prevents data leakage from training into testing sets.

```
data.setValidation('kFold','folds',10);
data.setTesting('holdout','percent',10);
```

Now we can set up the model. In this case, a simple PLS regression is enough, but additional blocks, e.g. feature preprocessing, could be added as the data processing blocks are elements of a double-linked list. The head of this list is given to the model which then takes care of training, validating, and testing the whole processing chain with the supplied data. Note that the options struct enables to give parameters for each element of the processing chain, and also supports lists of hyperparameters. If a list is given for more than one hyperparameter, the model automatically does a grid search and computes training, validation, and testing error for all parameter combinations.

```
plsr = DataProcessingBlock(Regression.plsr);
mdl = Model(plsr);
options = struct('plsr',struct('nComp',1:17));
mdl.train(data,options);
```

The next line then creates the graph in Fig. 16b. Eventually, the project is saved in its current state, so that it can be loaded again with all data and results in a new session.

```
mdl.plotErrors('plsr.nComp');
save('savedProject.mat','prj');
```

As an example for the data processing block interface, we show the definition of the PLS regression used in the code above:

```
function info = plsr()
    info.type = DataProcessingBlockTypes.Regression;
    info.caption = 'PLS regression';
    info.shortCaption = mfilename;
    info.description = '';
    info.parameters = [...
        Parameter('shortCaption','trained', 'value',false,...
            'hidden',true)...
        Parameter('shortCaption','beta0', 'hidden',true)...
        Parameter('shortCaption','offset', 'hidden',true)...
        Parameter('shortCaption','nComp', 'value',1)...
        ];
    info.apply = @apply;
    info.train = @train;
end

function [data,params] = apply(data,params)
    if ~params.trained
        error('Regressor must first be trained.');
    end
    b = params.beta0(:,params.nComp);
    o = params.offset(params.nComp);
    pred = data.getSelectedData() * b + o;
    data.setSelectedPrediction(pred);
end

function params = train(data,params)
    params.trained = true;
    [b,o] = quickPLSR(data.data,data.target);  % external function
    params.beta0 = b;
    params.offset = o;
end
```