Lecture 9: BCILAB Scripting and Plugins

Introduction to Modern Brain-Computer Interface Design

Christian A. Kothe
SCCN, UCSD

Outline

1. Prerequisites
2. Defining an Approach
3. All Other Steps

9.1 Prerequisites

Finding the Right Functions

- There is a scriptable function for every GUI command
- For documentation on script functions see Help menu or type `doc function_name` or `help function_name`
- Most functions have a brief summary, documentation for all input arguments, and code examples
- Some functions have paper references, some have cross-references

Calling Syntax

- Most functions take their arguments in the order in which they are listed in the documentation, and some can *alternatively* called with all parameters passed in as name-value pairs (using the same names as in the help text, in CamelCase)
- If in doubt, pass them in by name – less chance of getting the order wrong, etc.
- It is usually a bad idea to try to mix positional and name-value arguments in one call – don’t do it unless that’s the default way to call the function
- **Example:**

  ```
  bci_train(mydata,myapproach)
  bci_train(‘Data’,mydata,’Approach’,myapproach)
  ```

Loading Data

- A data set (no matter what file format) is loaded using the function io_loadset()
- It is almost always enough pass in just the file name, as in the example:

  ```
  data = io_loadset(‘/somepath/somefile.xyz’)
  ```

9.2 Defining an Approach

Defining a new Approach

- Defining an approach is the most complex area in scripting because a data structure must be constructed
- Since an approach is a particular instance of a BCI paradigm (used with custom parameters), an approach definition consists of:
  - The name of the paradigm (e.g., CSP, WindowMeans)
  - Optionally a list of arguments for the paradigm’s calibrate() function
- The default way to specify an approach is as a cell array whose first element is the name of the paradigm and whose remaining elements are arguments to its calibrate() function
- **Example:**

```
appr = {‘CSP’,‘SignalProcessing’,...,‘FeatureExtraction’,...};
```

Approach Parameters

- The parameters are a list of name-value pairs
- **Important:** The argument of an approach are not passed in a long ‘flat’ list, but they are organized in a hierarchy, i.e. some parameters have *named sub-parameters*
- **Example:**

```
app = {‘CSP’,’Prediction’,{‘MachineLearning’, …}};
```

MachineLearning is a sub-parameter of Prediction

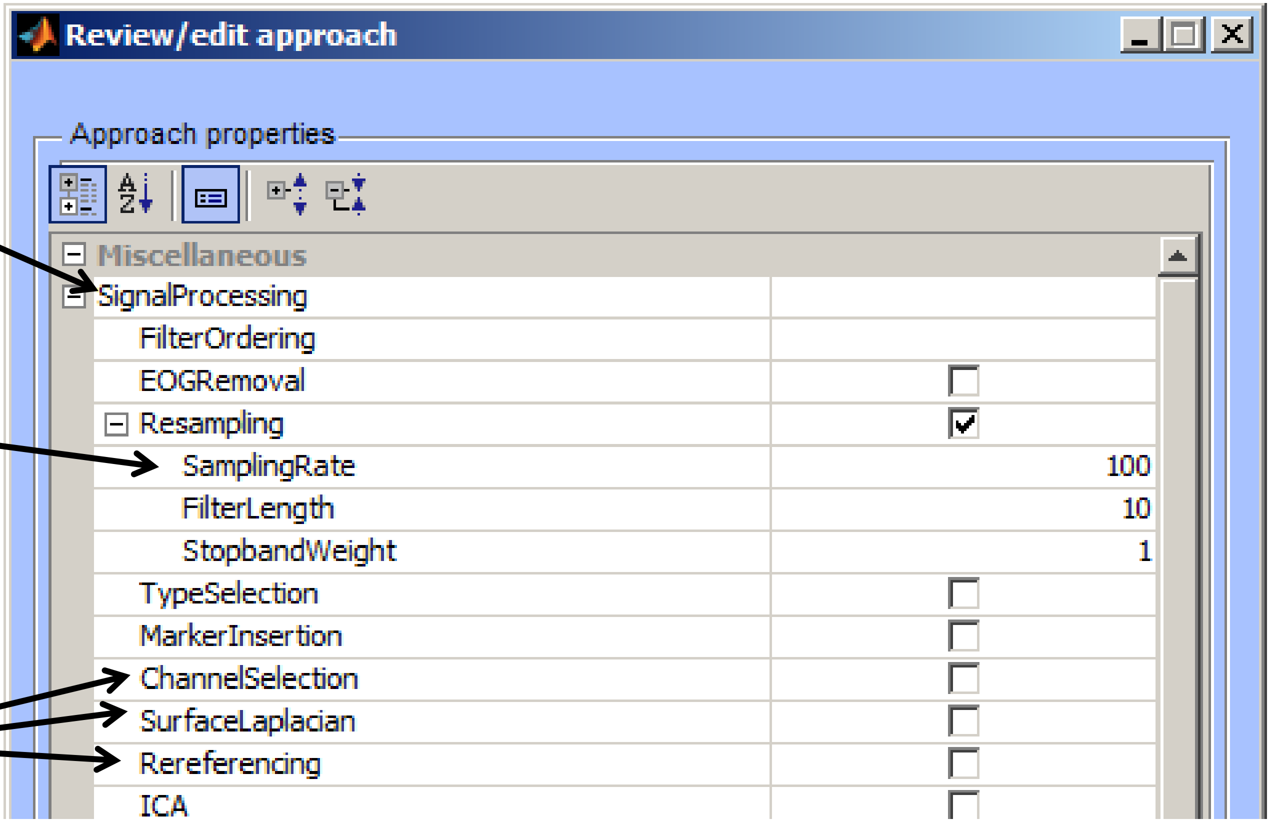
Prediction is a “top-level” parameter

Approach Parameters

- Which parameter names a BCI paradigm exposes is the business of the BCI paradigm
- However, practically all of them adhere to a uniform scheme of 2 top-level parameter names:
  - **SignalProcessing** is a top-level parameter that determines the signal processing stages that shall be used
  - **Prediction** is a top-level parameter that governs how the prediction function is being calibrated or applied

Correspondence With The GUI

- There is a 1:1 correspondence between the hierarchy of parameters that are specified in scripts and the layout of the parameter tree in the approach definition GUI



⊟ Miscellaneous

⊟ SignalProcessing	
FilterOrdering	
EOGRemoval	☐
⊟ Resampling	☑
SamplingRate	100
FilterLength	10
StopbandWeight	1
TypeSelection	☐
MarkerInsertion	☐
ChannelSelection	☐
SurfaceLaplacian	☐
Rereferencing	☐
ICA	☐

The SignalProcessing parameter

Sub-Parameter of Resampling (itself a sub-parameter of SignalProcessing)

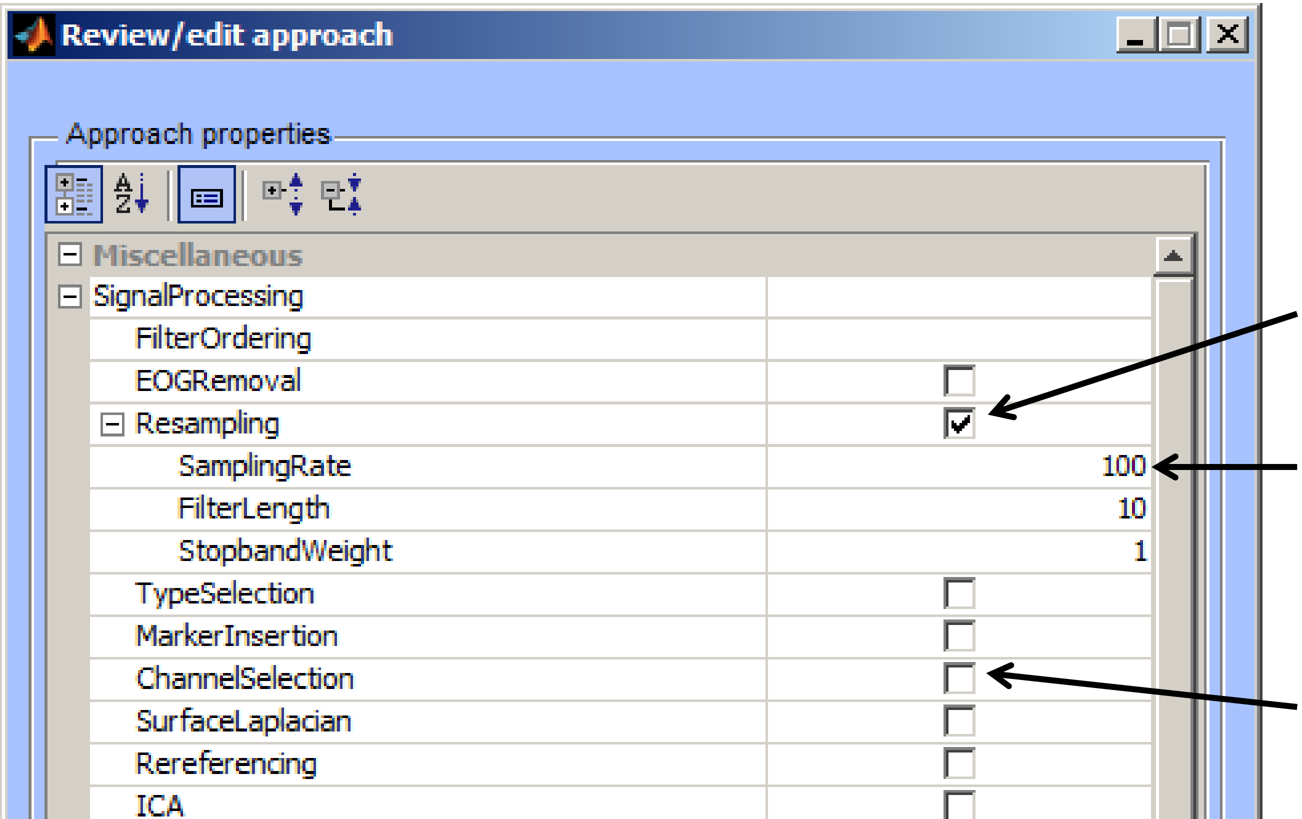
Sub-parameters of SignalProcessing

Correspondence With The GUI

- **Therefore:** If in doubt about parameter names, look them up in the GUI
- It is also okay to look up the parameter names in the function documentation or code, but they can be nested in a hierarchy of functions calling each other

Default Values

- Each parameter has a default value (unless it makes *absolutely no sense*), which can also be looked up in the GUI



Miscellaneous	
SignalProcessing	
FilterOrdering	
EOGRemoval	☐
Resampling	☑
SamplingRate	100
FilterLength	10
StopbandWeight	1
TypeSelection	☐
MarkerInsertion	☐
ChannelSelection	☐
SurfaceLaplacian	☐
Rereferencing	☐
ICA	☐

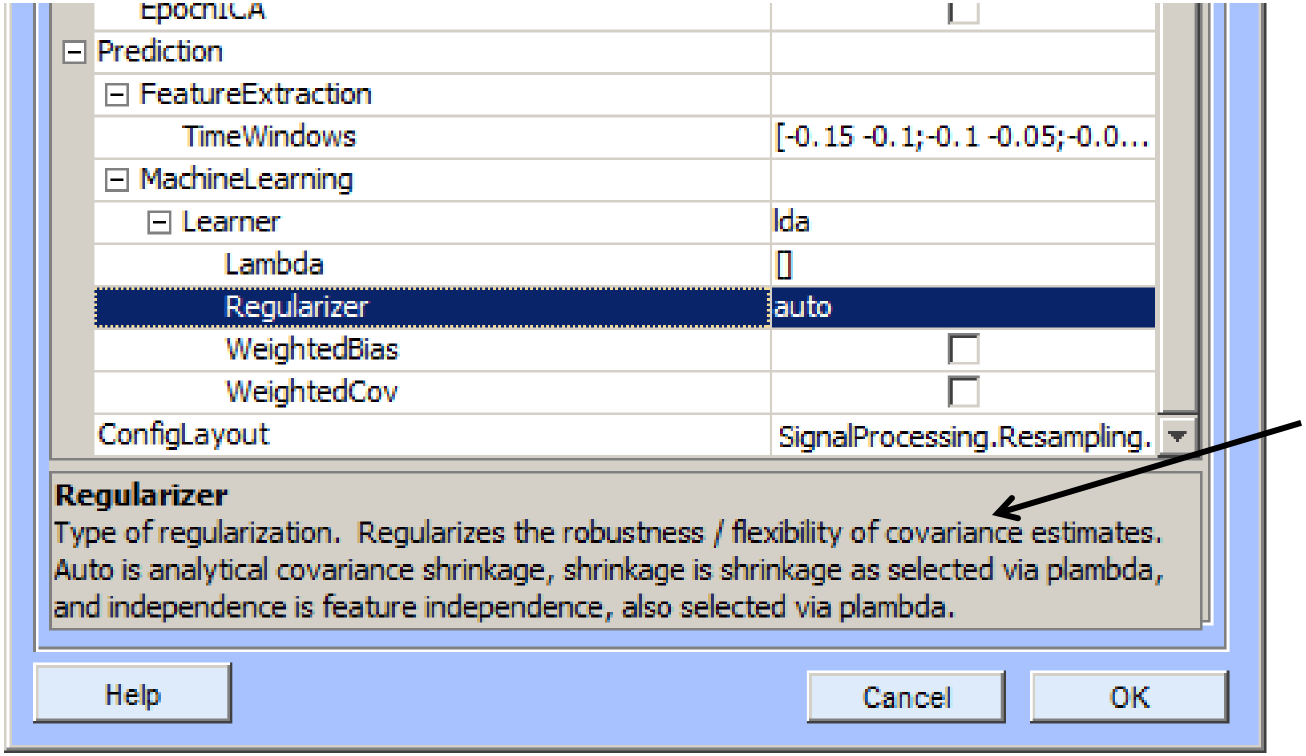
By default enabled

Default = 100

By default disabled

Parameter Help

- Each parameter has a help text, which is also visible in the GUI panel (at the bottom)



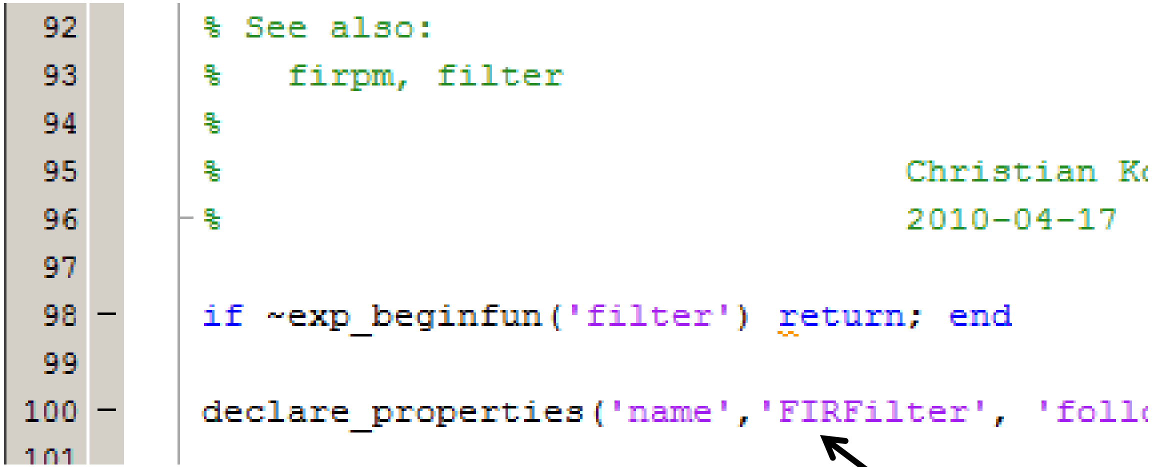
EpochICA	☐
⊟ Prediction	
⊟ FeatureExtraction	
TimeWindows	[-0.15 -0.1;-0.1 -0.05;-0.0...
⊟ MachineLearning	
⊟ Learner	lda
Lambda	[]
Regularizer	auto
WeightedBias	☐
WeightedCov	☐
ConfigLayout	SignalProcessing.Resampling.

Regularizer
Type of regularization. Regularizes the robustness / flexibility of covariance estimates. Auto is analytical covariance shrinkage, shrinkage is shrinkage as selected via plambda, and independence is feature independence, also selected via plambda.

Help text for the selected parameter

The SignalProcessing Parameter

- Has one named sub-parameter for every signal processing plugin that can be used (these are found automatically)
- The name under which a given signal processing plugin appears is up to the plugin – they declare this property at the beginning of their code (you can look it up there)



```
% See also:
%   firpm, filter
%
%                                  Christian K
%                                  2010-04-17

if ~exp_beginfun('filter') return; end

declare_properties('name','FIRFilter', 'foll
```

Name of the sub-parameter as which this plugin shows up in the approach definition (below SignalProcessing)

The SignalProcessing Parameter

- The plugins that are listed under SignalProcessing are those in the directories:
  - code/filters (file names beginning with flt_)
  - code/dataset_editing (file names beginning with set_)
- The value assigned to a sub-parameter (e.g., FIRFilter) that is presented by a function (e.g., flt_fir.m) is by default a cell array of arguments to that function
- The arguments can be passed in any format accepted by the function, but preferably they should again be passed as name-value pairs to avoid confusion

Configuring Signal Processing Stages

- **Example:**

(MATLAB line break)

```
app={‘CSP’,’SignalProcessing’, ...
  {‘FIRFilter’,{‘Frequencies’,[7 8 14 15]}}};
```

- This example defines a CSP-based approach that uses a particular Frequencies value in its FIR filter
- The FIR filter is now also “enabled” if it was not before

Disabling Signal Processing Stages

- It is sometimes useful to disable a parameter that is enabled by default: This can be written (by convention) as follows:

```
app={‘CSP’,’SignalProcessing’,{‘Resampling’,[]};
```

- Note that these are [] brackets – using {} accidentally would still enable the filter, but passes an empty argument list to it!

Shortcuts for the Impatient

- BCILAB has the unhealthy habit of allowing *short forms for most things* – I recommend to avoid them whenever possible, but it helps recognizing them
- The most salient short-cut form is when a parameter that has sub-parameters is not assigned a cell array of arguments (like it should), but instead directly the value of the first sub-argument
- **Example:**

```
app={‘CSP’,’SignalProcessing’,{‘Resampling’,200}};
```

This number is assigned to the first sub-argument of the resampling filter (=the target sampling rate)

Shortcuts for the Impatient

- BCILAB has the unhealthy habit of allowing *short forms for most things* – I recommend to avoid them whenever possible, but it helps recognizing them
- The most salient short-cut form is when a parameter that has sub-parameters is not assigned a cell array of arguments (like it should), but instead directly the value of the first sub-argument
- **Example:**

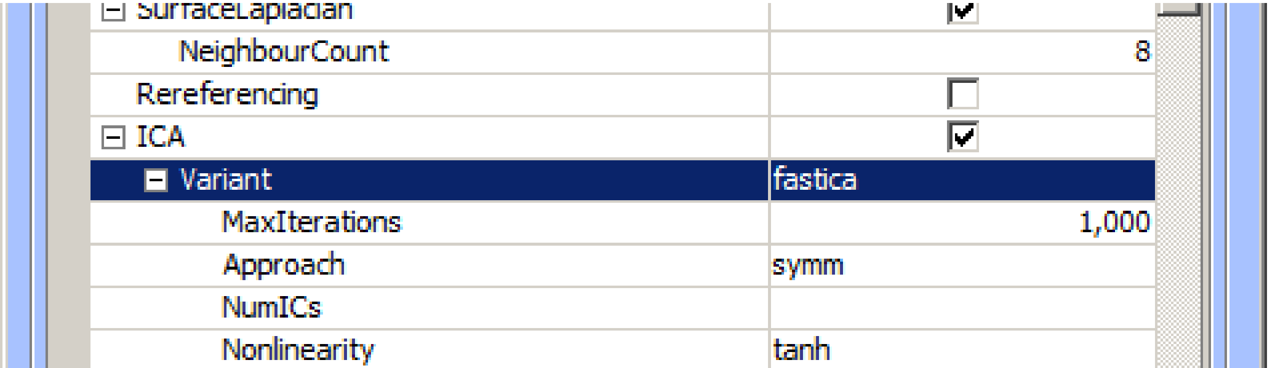
```
app={‘CSP’,’SignalProcessing’,{‘Resampling’,200}};
```

- **… is equivalent to:**

```
app={‘CSP’,’SignalProcessing’,...
  {‘Resampling’,{‘SamplingRate’,200}}};
```

Multi-Option Parameters

- The last kind of parameter that deserves mention are multi-option parameters, which consists of a *selection* argument (a string) and for each possible value a different list of sub-arguments
- An example are the different alternative variants supported by the ICA filter: amica, infomax, etc., all of which have algorithm-specific sub-arguments
- Below, the parameter named Variant is set to ‘fastica’, and the MaxIterations sub-parameter of Variant for the *fastica case* is set to 1000



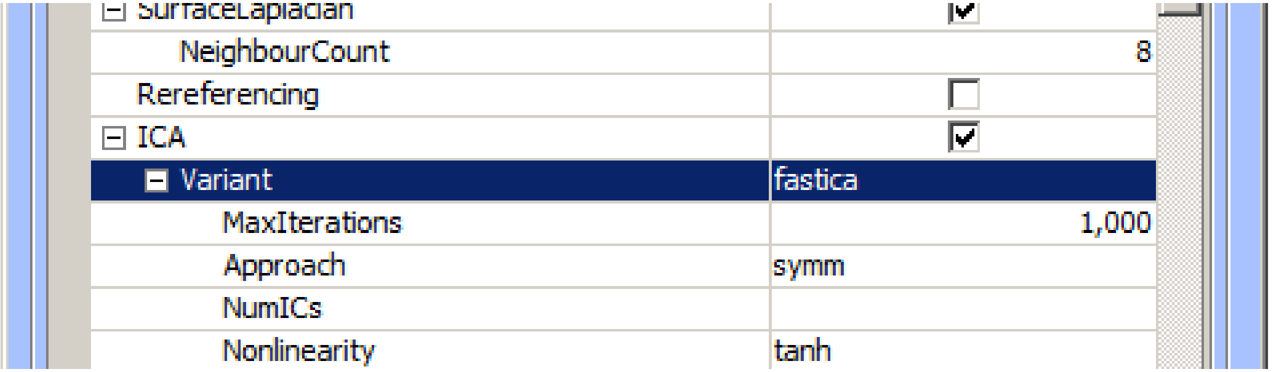
⊟ SurfaceLaplacian	☑
NeighbourCount	8
Rereferencing	☐
⊟ ICA	☑
⊟ Variant	fastica
MaxIterations	1,000
Approach	symm
NumICs	
Nonlinearity	tanh

Multi-Option Parameters

- In scripts, multi-option parameters are written just like the overall approach definition: as a cell array whose first element is the name of the selection followed by name-value pairs for this case
- **Example:**

```
…,‘Variant’,{‘fastica’,‘MaxIterations’,1000,‘Approach’,’symm’}
```

- **… is equivalent to setting what is shown here in the GUI:**



SurfaceLaplacian	☑
NeighbourCount	8
Rereferencing	☐
ICA	☑
Variant	fastica
MaxIterations	1,000
Approach	symm
NumICs	
Nonlinearity	tanh

Other Paradigm Parameters

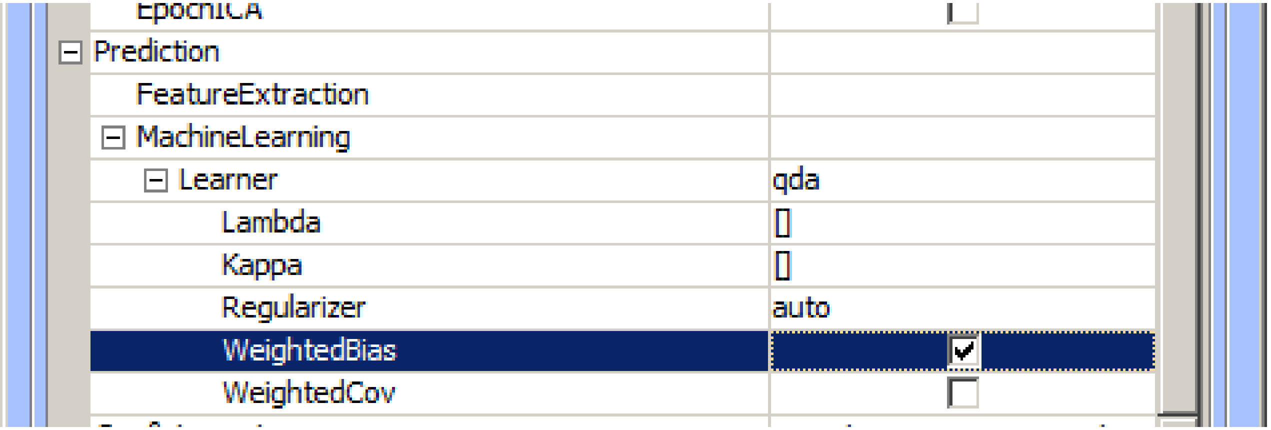
- The other parameters behave in exactly the same ways
- **Example:**
  - MachineLearning is a sub-parameter of Prediction, it has a Learner sub-parameter
  - Learner is a multi-option parameter with one case for each machine learning plugin (e.g., ‘lda’, ’qda’, ’logreg’, …)
  - The sub-parameters of the respective case are those that are exposed by the respective plugin function (e.g., ml_trainqda.m)

Configuring the Machine Learning Stage

- Thus, the following is a valid way to configure the machine learning function of a paradigm:

```
app={‘CSP’, ’Prediction’,{‘MachineLearning’, …
     {‘Learner’,{‘qda’ ‘WeightedBias’,true}}}};
```

- It corresponds to the following GUI setting:



EpochICA	☐
⊟ Prediction	
FeatureExtraction	
⊟ MachineLearning	
⊟ Learner	qda
Lambda	[]
Kappa	[]
Regularizer	auto
WeightedBias	☑
WeightedCov	☐

Shortcut for Multi-Options

- Here is one last shortcut for today:

```
app={‘CSP’, ’Prediction’,{‘MachineLearning’, …
     {‘Learner’,‘qda’}}};
```

Instead of at least {‘qda’}

9.3 All Other Steps

Calibrating (“Training”) a Model

- A new BCI model is created using a previously loaded data set (the training set) and a previously defined approach
- This is done using the function bci_train (the equivalent of the “Train new model…” dialog)
- **Example:**

```
raw = io_loadset(‘imag.set’)
app = {‘SpecCSP’, ... };
[loss,model,stats] = bci_train(‘Data’,raw,’Approach’,app, ...
    ‘TargetMarkers’,{‘S  1’,’S  2’});
```

Calibrating a Model

- The bci_train function usually takes 3 inputs:
  - The data (Data parameter)
  - The approach (Approach parameter)
  - The description of how event types map onto class labels (TargetMarkers, same as in the GUI)
- The function returns three outputs:
  - The overall loss estimate (e.g. error rate)
  - The learned model
  - Statistics about the model and training process, including results of a cross-validation

Calibrating a Model

- The bci_train function therefore not only returns a model but also produces estimates about the likely future performance
- If this is too slow, it can be disabled (in an extra parameter to bci_train)

Visualizing a Model

- Models are visualized using the function bci_visualize
- **Example:**
  bci_visualize(mymodel)

- This function can take extra arguments that are passed on to the responsible drawing function (but few drawing functions have arguments)

Applying a Model to Test Data

- For *offline application* to test data, the function bci_predict can be used – it applies the BCI model to each trial in the data and calculates loss statistics
- **Example:**

```
[outputs,loss,stats] = ...
    bci_predict(‘Data’,mydata,’Model’,mymodel);
```

- **Note:** the first output are the model’s predictions for each trial in the data

Annotating Data with Continuous BCI Outputs

- The BCI output can be attached as an extra channel (or multiple channels, each representing the probability of class k) to a data set, using the function bci_annotate
- **Example:**

```
newset = bci_annotate(‘Data’,mydata,’Model’,mymodel)
```

Reading Real-Time Data

- Real-time data can be acquired from a device and written into a named workspace variable using the online reader plugins (run_read* functions)
- **Examples:**

```
run_readbiosemi(); # read from a BioSemi device

run_readdataset(‘MatlabStream’,’mystream’,’Dataset’,myset);
```

Sending Real-Time Outputs

- The outputs of a BCI model as applied to some stream(s) can be calculated in the background online and passed on to some destination – this is done using the online writer plugins (run_write*)
- These functions take usually the name of the model to use and the name(s) of the stream(s) to use
- **Example:**

```
run_writevisualization(‘Model’,’mymodel’, ...
    ’SourceStream’,’mystream’)
```

Performing Batch Analyses

- Using bci_batchtrain, a single approach can be efficiently applied to a list of data sets or file names
- Also multiple approaches can be applied to one or more data sets in an automated manner
- Can not just train models but also make predictions and evaluate losses on test data sets

Parameter Search

- It is possible to replace (practically) any value in an approach definition by a so-called “search range”, i.e. a list of possible values to try automatically in a systematic manner
- A search range is specified by writing the expression search(value1, value2, …, valueN)
- Multiple search parameters in one approach lead to combinatorial grid search (slow!)
- **Example:**

```
app={‘CSP’,’Prediction’,{‘FeatureExtraction’,{ ...
    ‘PatternPairs’,search(1,2,3)}}};
```

L9 Questions?