**USER GUIDE**

**DETECT: DETection of Events in Continuous Time Toolbox**

DETECT is a MATLAB toolbox for detecting and labeling events in epoched and continuous time series. To use DETECT, you must have examples of labeled data to create a classifier or model. Once you have created a model, you can apply it to label either epoched or continuous data. DETECT provides some utility functions specifically for manually labeling EEG data in an efficient way. This process is useful for producing labeled data to train a model for artifacts and other features.

DETECT uses autoregressive features by default, but you are free to provide your own feature functions. The toolbox includes several functions for building models of events as well as sample datasets for detecting artifact segments in EEG data.

**Requirements:**

DETECT requires the following

* MATLAB™ version R2011A or higher. Other versions of MATLAB™ may work; version R2011A and later are officially supported
* EEGLAB version 10 or higher, if you wish to use the DETECT plotting functions.

**Installation:**

1. Download the toolbox and extract the .zip file.
2. Add the extracted folder to the MATLAB Path (File → Set Path). Use the “Add with Subfolders” option.
3. Add EEGLAB and its subfolders to the MATLAB Path if you wish to use the DETECT plotting functions.

**Installation notes:**

1. There are two versions of the toolbox available for download, depending on your installation. Currently we have versions for 64-bit Windows Vista/7 and 64-bit Linux platforms. You will need to compile LibSVM for any other installation. To recompile the toolbox, navigate to LIBSVM\_DETECT/matlab and run the make.m file.
2. If you plan to use DETECT frequently, you may want to put commands to automatically add the DETECT folder and its subdirectories in your startup.m file located in your MATLAB startup folder.

**Contents:**

|  |  |
| --- | --- |
| **GENERAL FUNCTIONS (can be used to label any type of time series data)** | |
| **Name** | **Description** |
| getARfeatures | Estimate autoregressive model coefficients of specified order for a 3D array of input data (*channels* × *windowSize* × *windows*) and return a (*windows* × *featureSize*) array of features to be used for classification |
| getModel | Create a model or classifier based on data (*channels* × *windowSize* × *windows*) array of training data and a *(windows)* length vector of class labels |
| labelData | Label data (*channels* × *frames*) as a function of time based on a classification model and, report certainty of each label. |
| labelWindows | Label windows (*channels* × *windowSize* × *windows*) based on a classification model and also return classification accuracy if ground truth labels are passed in for comparison |
| compareLabels | Compares two sets of labeled data, either from an automated labeling (by DETECT) or manual labeling (from using markEvents) or both (one set from a manual labeling and the other from an automated labeling). |
| thresholdPolicy | Applies the certainty post-processing policy used in this paper. If the certainty is below a given threshold and one of the top two possible classes is the baseline, the prediction type will be set to the baseline. No change is made if neither of the top two possible classes is the baseline. |
| unknownPolicy | A post-processing policy that incorporates a new decision class of “Unknown”. If the certainty is below a given threshold and one of the top two possible classes is the baseline, the prediction type is set to the baseline. Otherwise, the prediction type is set to “Unknown.” |
| **EEG RELATED FUNCTIONS (depend on EEGLAB)** | |
| getLabels | Convert a continuous dataset into an epoched or windowed dataset, epoching by user-highlighted regions. |
| plotLabeledData | Display results of continuous labeled data using a modified EEGLAB plot window |
| plotMarkedData | Plot a manually labeled dataset using a modified EEGLAB plot window. |
| plotWindowData | Display results of labeling windowed dataset using a modified EEGLAB plot window. |
| markEvents | Manually label data based on given categories. Can update a previously labeled dataset to add/remove events and categories. Uses a modified EEGLAB plot window. |

**EXAMPLE 1: Labeling trials (epoched data)**

Suppose you have epoched data consisting of a three-dimensional array (*channels* × *windowSize* × *windows*) and each trial or epoch is a *channels* × *windowSize* array. To label the data, you must have a labeled set of windows called the *training* data. The windows you wish to label are called the *testing* data. In general, the training data and testing data should not overlap.

The following MATLAB code labels the windows in the testing data based on the labeled data in the training data. Set the MATLAB Current Directory to be the directory containing the DETECT Toolbox functions and run:

load data/training-epochs.mat; % load training windows + labels

model = getModel(training, training\_labels); % create classifier

load data/testing-epochs.mat; % load testing windows to be labeled

[results, accuracy] = labelWindows(testing, model, testing\_labels);

The results structure looks like this:

results =

1x56 struct array with fields:

label

actualLabel

certainty

likelihoods

prob\_estimates

We can call the results from the first window in this array with:

results(1)

ans =

label: 'None'

actualLabel: 'None'

certainty: 0.7793

likelihoods: {7x1 cell}

prob\_estimates: [0.6566 0.0133 0.0141 0.0393 0.1118 0.1449 0.0201]

labelOrder: {7x1 cell}

The entries in this structure are:

results.label A string label indicating predicted category for the window

results.actualLabel A string label indicating the actual category of the window. This

is left blank if the true labels were not passed into the

labelWindows function.

results.certainty The certainty of the prediction (see below)

results.likelihoods A cell array of strings denoting the categories, from most likely

to least likely. The first entry in this array is the same as

results.label.

results.prob\_estimates Estimated probability distribution of the classes in

results.labelOrder.

results.labelOrder Cell array of strings to identify the categories for

results.prob\_estimates. The first entry of prob\_estimates

denotes the probability of the first entry in labelOrder.

We calculate a certainty measure as a means to assess the confidence in the predictions. The certainty is defined as

where and are the first and second largest prediction probabilities for the data sample. This is a relative probability measure that quantifies the strength of the prediction: if most of the probability is concentrated in one class, this measure will be close to 1, while if the probabilities are more distributed across the classes, the measure will be close to 0.

Here, we can call results(1).labelOrder:

results(1).labelOrder

ans =

'None'

'Jaw Clench'

'Jaw Movement'

'Eye Blink'

'Eye Left Movement'

'Eye Up Movement'

'Eyebrow Movement'

The first entry of prob\_estimates, .6566, indicates that the probability that the data is in the class ‘None’ is .6566. The second entry, .0133 indicates the probability that the data is in the ‘Jaw Clench’ category, and so forth.

The second output, accuracy, is the classification accuracy. Here, the accuracy is:

accuracy =

98.2143

By default getModel uses all of the channels, 4-fold cross validation, and the getARfeatures function with parameter 2. This feature function fits an autoregressive (AR) model to each channel given. You can explicitly specify the channels to be used, the number of cross validations to perform, and your own feature function with an arbitrary number of arguments as illustrated by the following example:

model = getModel(training, training\_labels, 1:64, 4, @getARfeatures, 2);

This example uses only the first 64 channels in the training data to build the classifier and four cross-validations to validate the model. We use the function @getARfeatures to extract autoregressive coefficients for each channel individually and concatenate all the features to form one long feature vector. The additional argument of 2 is the model order to fit. Any feature extraction function can be used here, as long as the output of the function is a matrix of size (windows x featureSize)with the number of features fixed. Other types of features can be used, such as spectral-based features or other features such as connectivity-based measures such as granger causality or directed coherence.

Both the training and testing data used in this example were EEG data sampled at 256Hz using a Biosemi 64-channel Active Two System. The experiments also recorded four EOG (electrooculography) channels (channels 65-68), which are not used to build the model.

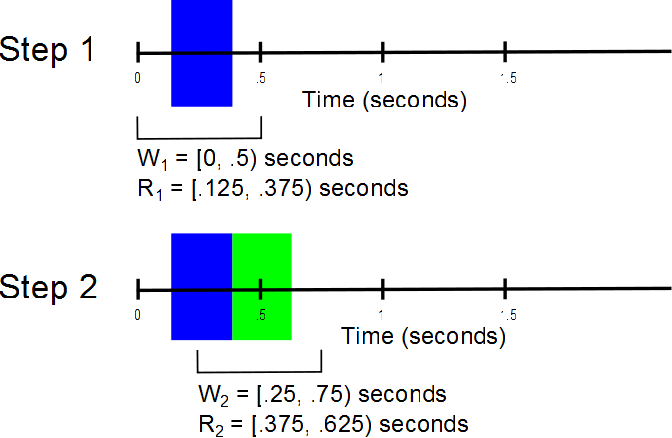
**EXAMPLE 2: Labeling continuous time series**

The primary purpose of DETECT is to label continuous time series. That is, DETECT produces a time series of labels corresponding to a two-dimensional array (*channels* × *frames*) of data by sliding a window across the time series and predicting a label for each slide.

For example, suppose you want to label artifacts in your data. Pick out fixed size intervals in the time series where you recognize the artifact and label these intervals. Also choose intervals that don’t contain these artifacts and label them as such (say with the label ‘none’). Although DETECT does not require a balanced training set (i.e., the same number of trials for each type of feature), you should take care to provide enough training data for each type of feature that you want to classify (the more the better). Once you have labeled the training data and created a model with getModel, you can then apply labelData to produce a continuously labeled data set.

Typically, the amount of data to be labeled is very large compared with the training set, so the fact that the training set is exacted from the much larger testing test is not a cause for concern.

The size of the slide (which is given in frames or samples) is usually greater than 1. The window size must be the same as the epoch or trial size used to train the data. The following figure illustrates the process.



**Figure 1:** Illustration of the association of label with time in labeling of continuous data. The example data is sampled at 256Hz with a window size of 0.5 s and a slide width of 0.25 s.

Figure 1 is based on the training data provided in the DETECT toolbox, using training epochs of 0.5 s (128 frames). The first window starts at time 0, and the length of the window is 0.5 s. We label data in time using the following formula:

*Ri* = [*Mi* - 0.5\**S*, *Mi* + 0.5\**S*)

where *Ri* is the *ith* region of the data, *Mi* is the midpoint of the *ith* window and *S* is the slide width, all in seconds. This procedure is performed until the end of the dataset. DETECT ignores data if the slide cannot be performed (i.e., if the slide window is 0.5 s but only 0.2 s of data remain at the end).

The following MATLAB code labels continuous testing based on the labeled data in the training data using a slide width of .125s and a sampling rate of 256Hz.

load data/training; % load the training data

load data/labels; % load labels for training data

model = getModel(training, labels); % create classifier using defaults

load data/testing; % load continuous testing data

results = labelData(testing, model, 256, .125); % label testing data

The third (sampling rate) and fourth (slide width) arguments of labelData are 256Hz and .01s by default. The results structure contains the following fields:

results.label cell array of predicted labels for continuous data

results.time 2-dimensional vector with [startTime, endTime] in seconds

results.certainty vector of probability-like quality indicators

results.likelihoods array of probability estimates for each possible label

The code above generates the following output:

results =

1x3837 struct array with fields:

label

time

certainty

likelihoods

We can call an entry in this array using results(1):

results(1)

ans =

label: 'None'

time: [0.1836 0.3047]

certainty: 0.8704

likelihoods: {7x1 cell}

There were 3837 number of slides, each slide at a width of 0.125 s long. A certainty thresholding policy can be applied to this output to reduce false positives in the data. For example:

results1 = thresholdPolicy(results, 'None', .5);

filters the relabels the data as ('None') if the label certainty is less than 0.5 and one of the top two possible classes is the baseline class ('None'). The input arguments are:

results the output from labelData

baseline\_class the class which is considered the baseline class used in

building the model (here, ‘None’)

certainty\_threshold the threshold value (here, .5).

**EXAMPLE 3: Manually labeling data**

DETECT provides a GUI for manually labeling continuous time series based on EEGLAB’s eegplot function. This function, getLabels, allows you to view your data in a continuous scrolling window and to easily mark and categorize intervals in a continuous time series. This function is useful for creating training sets for your own data. This function can handle both EEGLAB EEG datasets as well as MATLAB matrix inputs where the dimensions of the matrix are (*channels* × *frames*)

A second function, plotLabeledData, displays labeled data based on the results of the classification. This function is useful for accurately labeling features such as artifacts and can be used as a preliminary step to manual removal of artifacts.

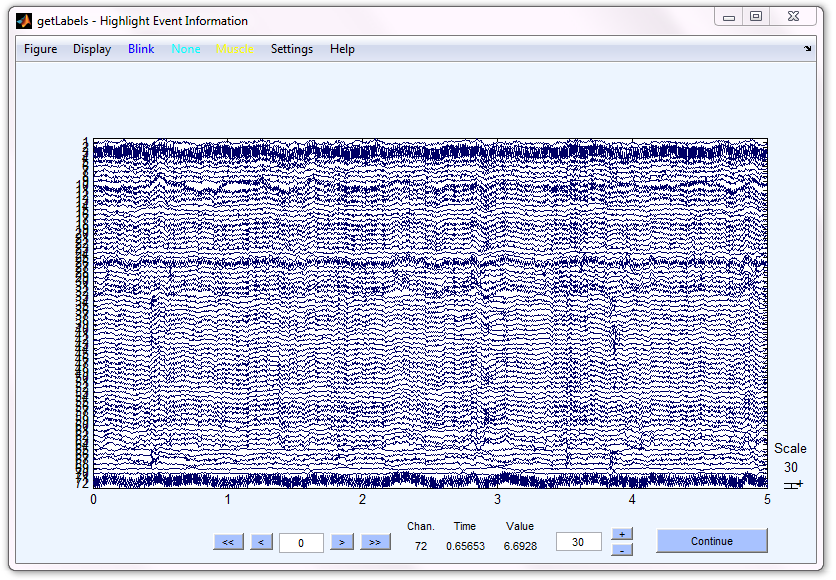
First off, load a continuous dataset:

load data/testing; % load data to be manually labeled

You can use this command to highlight the data containing blinks and muscle artifacts, with a desired event interval length of .5 seconds:

[dataWindows, labels] = getLabels(testing, {'Blink', 'None', 'Muscle'}, .5)

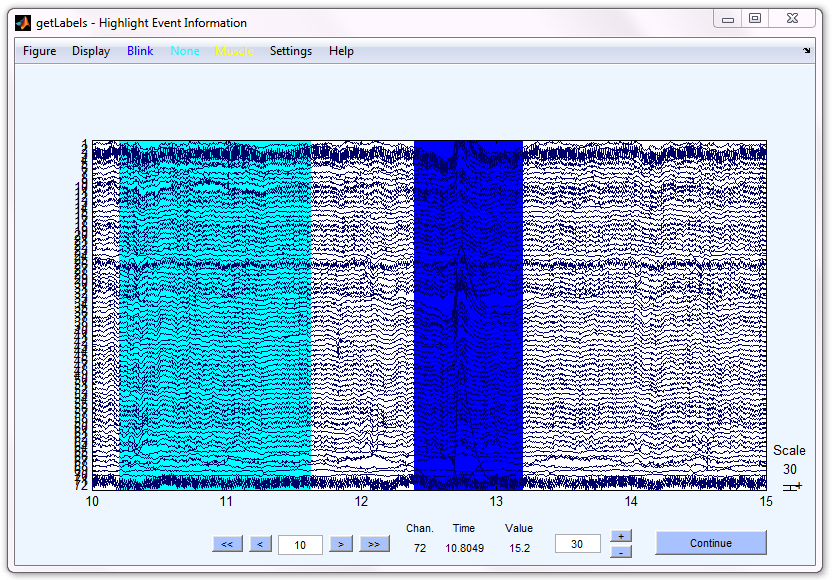
This command brings up the following GUI:



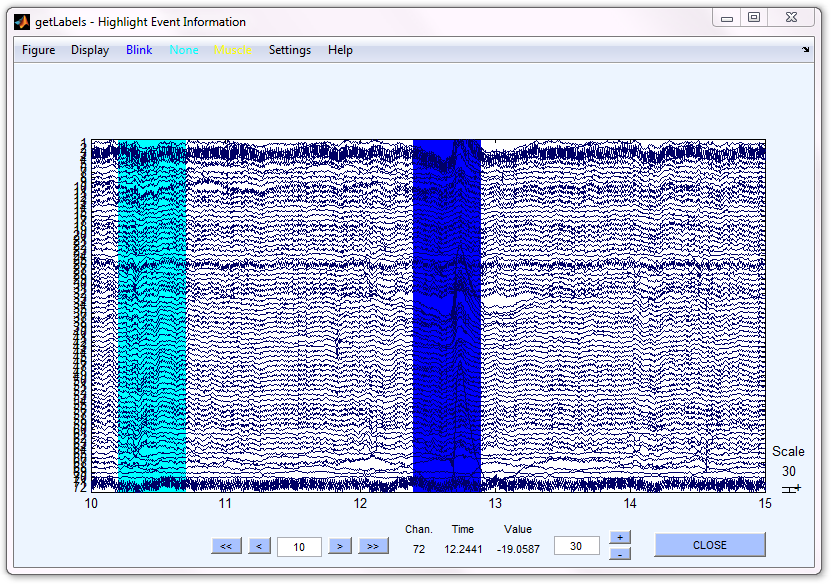
The three colored buttons on the top toolbar (“Blink”, “None” and “Muscle”) correspond to the labels passed as the second argument to getLabels. These buttons determine the colors and labels of the highlighted regions. Note that if the input data was an EEGLAB EEG structure, the channel locations (vertical axis) as well as the event information will be plotted in addition to the data. Use this GUI to highlight regions of data to be labeled as follows:

1. Press the button corresponding to the label you choose.
2. Click the cursor on the data at the starting point and drag the mouse while holding the mouse button down.
3. Release the button when you have reached the end of the segment you wish to label.

It is generally a good rule of thumb to have an equal number of trials for each event. In this example, one ‘None’ region for every ‘Blink’ and ‘Muscle’ region selected helps to ensure the accurate estimation of classification models. An example of some highlighted regions is shown below.



Once you are finished highlighting your data, hit the “Continue” button at the bottom right. Another GUI will pop up, this time with all the events aligned to be exactly the length specified in the function call. The GUI that is shows for our particular example is shown below:



In the MATLAB command window, another prompt will appear:

Adjusting event timings for the desired event length of 0.500 seconds

Do you want to:

1. save this labeling(s),

2. continue labeling(c), or

3. quit without saving(q)? [s/c/q]:

Here, three options are available. The first option, “s”, extracts the highlighted regions and provides a label for each region. The second option “c” is used whenever you need to adjust the highlighted regions (for example, it may not cover the desired area, or you may need to add additional events). The final option “q” will quit the function without saving the results.

Hit “Close” to close the figure after you have inspected the highlighted regions for accuracy. Typing “s” will calculate two variables, dataWindows and labels. The dataWindows output is an array that contains the data in windowed form, as a 3D array of size *channels* × *windowSize* × *windows*, where *windowSize* is the number of samples in a trial, and *windows* is the total number of labeled trials in the data. The labels output is a cell array of length *windows* that contains a label for each labeled trial. In the above example, we only highlighted two regions, so there are only two entries in the labels variable: {'None'; 'Blink'}. If the input to getLabels was an EEGLAB EEG structure, the output will also be an EEGLAB EEG structure whose data is a 3D array, while if the input was a 2D data matrix, the output will be a 3D data matrix.

We can use the output of getLabels to train a model:

model = getModel(dataWindows, labels); % create classifier using defaults

(See Example 1 for details on the defaults and additional parameters for getModel.)

**EXAMPLE 4: Plotting labeled data**

We can use DETECT to plot detected events in time series as colored segments in a scrolling plot. As an example, suppose we are interesting in labeling sections of EEG data containing artifacts. To build the artifact classification model using EEGLAB .set files:

training = pop\_loadset('data/training.set'); % load the training data

load('data/labels.mat'); % load labels for training data

model = getModel(training, labels); % create classifier using defaults

testing = pop\_loadset('data/testing.set'); % load continuous testing data

results = labelData(testing, model, 256, .125); % label testing data

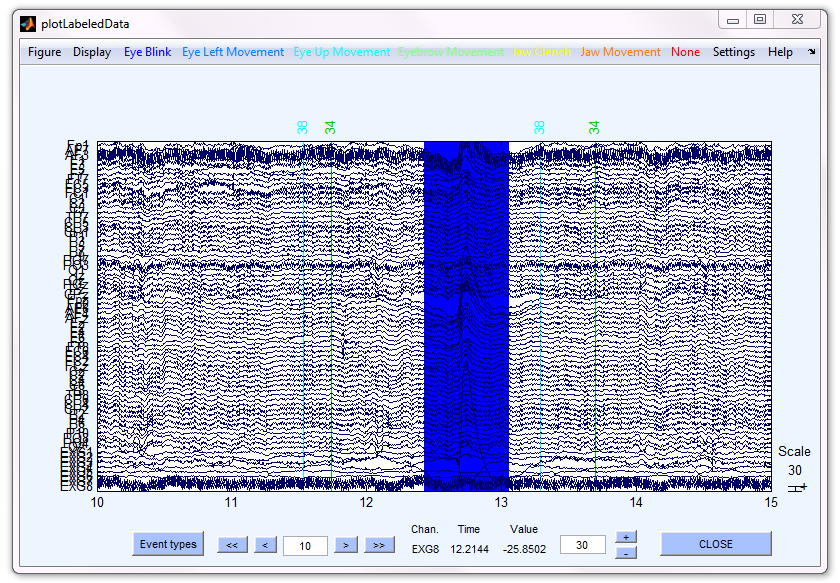
This code builds a continuous detection model using a slide width of .125s for EEG data sampled at 256 Hz. The artifact types included in the labels are: “Jaw Clench”, “Jaw Movement”, “Eye Blink”, “Eye Left Movement”, “Eye Up Movement”, and “Eyebrow Movement”.

Once the calculation finishes, we run the following code to see the effect of labeling. In the example, we are only interested in eye blinks (the fifth input argument in the function):

% plot the data

labelSet = plotLabeledData(testing, model, results, 'srate', ...

256, 'includeClasses', {'Eye Blink'})



All the artifact types appear as uniquely colored buttons at the top toolbar. The artifact types are: Jaw Clench, Jaw Movement, Eye Blink, Eye Left Movement, Eye Up Movement, Eyebrow Movement and None. Only Eye Blinks are displayed, however.

When you push the CLOSE button, the GUI closes and returns the labelSet in a structure similar to this:

labelSet =

'Eye Blink' [ 12.4336] [ 13.0547]

'Eye Blink' [ 19.8086] [ 20.3047]

'Eye Blink' [ 25.8086] [ 26.4297]

'Eye Blink' [ 40.9336] [ 41.5547]

'Eye Blink' [ 93.8086] [ 94.1797]

'Eye Blink' [119.0586] [119.6797]

'Eye Blink' [124.1836] [124.4297]

'Eye Blink' [173.4336] [174.0547]

'Eye Blink' [175.1836] [175.9297]

'Eye Blink' [179.0586] [179.5547]

'Eye Blink' [180.4336] [180.5547]

'Eye Blink' [184.5586] [185.1797]

'Eye Blink' [274.1836] [274.6797]

'Eye Blink' [285.1836] [285.6797]

'Eye Blink' [296.9336] [297.4297]

'Eye Blink' [350.0586] [350.6797]

'Eye Blink' [355.6836] [356.3047]

'Eye Blink' [357.0586] [357.1797]

'Eye Blink' [357.3086] [357.4297]

'Eye Blink' [361.8086] [362.6797]

'Eye Blink' [362.8086] [363.3047]

'Eye Blink' [446.6836] [447.5547]

The first column is the detected event, while the second and third columns denote the start and end times, respectively, of the event. In this case, we only wanted to display one event, “Eye Blink”, and so only the times where eye blinks are present are shown. If more than one type of event is chosen for display, the output of this function will show the start and end times of each event type chronologically.

**EXAMPLE 5: Updating a previously labeled dataset**

DETECT has functionality to update a previously labeled dataset. This previous labeling can either be from a manual labeling (using the function markEvents) or from an automated labeling (use plotLabeledData on labeling generated from LabeledData or LabeledWindows ).

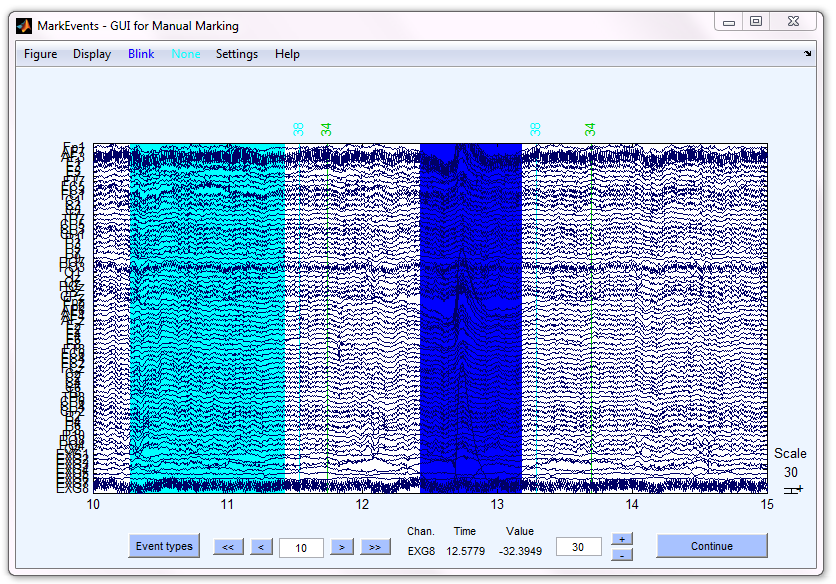
Example: Load the dataset:

EEG = pop\_loadset('data/testing.set')

Then run the following command and manually highlight events with Blinks and None:

labelSet1 = markEvents(EEG, {'Blink', 'None'}, 'srate', 256)

This function call generates a GUI that is similar to getLabels. Use the same procedure for highlighting data as previously shown in the examples for getLabels. An example labeling is shown below:



Hitting Continue will output the following event structure:

labelSet1 =

'None' [10.2821] [11.4245] []

'Blink' [12.4334] [13.1827] []

The first column is the event type, the second and third columns are the start and end time, in seconds and the fourth column is an index of bad channels (here it is empty).

If you want to update the markings, use

labelSet2 = markEvents(EEG, {'Blink', 'Muscle'}, 'srate', 256, ...

'regions', labelSet1)

This will take the existing events field and display it on the data scroll plot. Any modifications made to the data will be automatically saved to labelSet2 when you hit “Continue”.

**EXAMPLE 6: Comparing two labeled datasets**

DETECT also has functionality to automatically compare two labeled datasets. To compare two labeled datasets, first load up an EEG dataset:

EEG = pop\_loadset('data/testing.set')

load('data/labeled-data.mat'); % load the labeled data

We can plot the data using plotMarkedData.

plotMarkedData(EEG, labelSet1) % plot the data

plotMarkedData(EEG, labelSet2) % plot the data

The first label set has two events while the second label set has only one event.

labelSet1 =

'Blink' [0.9795] [1.3351] []

'Muscle' [1.3382] [1.5794] []

labelSet2 =

'Blink' [0.9720] [1.5061] []

We are interested in measuring the agreement between the two label sets. We allow for a timing error as an additional input in the comparison. For example, in the first label set, the ‘Blink’ starts a little bit later than in the second label set. The function call for this is :

[results errorInfo timeInfo] = compareLabels(EEG, labelSet1, labelSet2, ...

0, 256)

The outputs look like this:

-----------------------------------------------------

Total Time in Agreement = 479.734 seconds

Total Time in TypeError = 0.168 seconds

Total Time in FalsePositive = 0.004 seconds

Total Time in FalseNegative = 0.070 seconds

Total Time of Data = 479.996 seconds

-----------------------------------------------------

results =

'NullAgreement' [ 0] [ 0.9609]

'FalsePositive' [0.9648] [ 0.9688]

'Agreement' [0.9727] [ 1.3281]

'TypeError' [1.3320] [ 1.5000]

'FalseNegative' [1.5039] [ 1.5742]

'NullAgreement' [1.5781] [479.9961]

errorInfo =

'Null' 'Blink' [0.9648] [0.9688]

'Muscle' 'Blink' [1.3320] [1.5000]

'Muscle' 'Null' [1.5039] [1.5742]

timeInfo =

agreement: 479.7344

typeError: 0.1680

falsePositive: 0.0039

falseNegative: 0.0703

totalTime: 479.9961

‘NullAgreement’ is the agreement between the two regions when no events are highlighted. Here, the null agreement is from time 0 to the beginning of the first event in time among the two datasets. A ‘FalsePositive’ error is generated because the ‘Blink’ in the second label set starts before the blink in the first label set. A period of ‘Agreement’ follows because both regions are labeled with the same type at the same time. Starting at time 1.3320, the first label set called the type as “Muscle” while the second label set still called the region “Blink.”. Therefore, there is a type error, labeled ‘TypeError’ in the output. The ‘Muscle’ area in the first label set extends to time 1.5974, which is further than the ‘Blink’ area in label set 2, so the area generates a ‘FalseNegative’ error from 1.5039 to 1.5742. After this, there is ‘NullAgreement’ until the end of the dataset. Note that small numerical differences are expected since we convert the labeled regions into data points in frames to compare the regions.

The output errorInfo describes the type of errors generated from ‘FalsePositive’, ‘FalseNegative’ and ‘TypeError’ conditions. For example, the first entry in errorInfo describes the ‘FalsePositive’ error where Region 2 did not have a label where Region 1 did (Blink). The second region in errorInfo describes the TypeError where Muscle was in Region 1 while Blink was in Region 2. The third and fourth columns of this matrix denote the start and end time, in seconds, of the disagreement.

The output timeInfo gives a summary of the total time in each of the possible states in seconds. Here, Agreement is the total time where the two regions agree (both agree on the absence or presence of an event). The totalTime is the length of the data in seconds.

If the allowable timing error is 0.1 s, the output is now:

[results errorInfo timeInfo] = compareLabels(EEG, labelSet1, labelSet2, ...

.1, 256)

-----------------------------------------------------

Total Time in Agreement = 479.840 seconds

Total Time in TypeError = 0.070 seconds

Total Time in FalsePositive = 0.000 seconds

Total Time in FalseNegative = 0.070 seconds

Total Time of Data = 479.996 seconds

-----------------------------------------------------

results =

'NullAgreement' [ 0] [ 0.8711]

'Agreement' [0.8750] [ 1.4258]

'TypeError' [1.4297] [ 1.5000]

'FalseNegative' [1.5039] [ 1.5742]

'NullAgreement' [1.5781] [479.9961]

errorInfo =

'Muscle' 'Blink' [1.4297] [1.5000]

'Muscle' 'Null' [1.5039] [1.5742]

timeInfo =

agreement: 479.8398

typeError: 0.0703

falsePositive: 0

falseNegative: 0.0703

totalTime: 479.9961

The only difference here is the ‘FalsePositive’ is now an ‘Agreement’ because, while the ‘Blink’ in the first label set starts later than in the second label set, the start times are within 0.1 seconds of each other, and so the labelings are said to be equal. The agreement starts and ends 0.1 s earlier and later, respectively, to account for the allowable timing error. This also pushes the start of the ‘TypeError’ by 0.1 s later since the regions are concurrent. The allowable timing error is used only when the label sets have a region of the same type; no timing error adjustment is used when the label ets have different type, such as the ‘FalseNegative’ entry that stays the same in both scenarios.

Note that we can plot the results of compareLabels using the plotMarkedData function:

plotMarkedData(EEG, results, 'srate', 256)

