ReadMe of Matlab based data annotation baseline software

Essentially this is a brief review of the work I, Christian Ward, carried out over the summer of 2015. First there will be a review of what I set out to achieve and then how I went about tackling the problem. Following this there will be a review of the key files that enable all this software to work and how to use them so others understand what was done. If I do this correctly others should be able to port much of the work to a more viable software language; Matlab is really about prototyping anyway.

The task at hand was to develop a generic machine learning platform that could take in any data type and allow a user to highlight events of interest to train the algorithm. This is a supervised learning approach that assumes the system could not (and perhaps should not) be programmed to run open loop on the data. The target data set was EEG data from the DBS devices at Medtronic. However, this data proved to be very complicated to parse without the aid of a clinician. The evolution of the project showed that by finding less complex data sources (accelerometers, events from a patient diary, other bio signals) non-experts could train the algorithm on these types. This would then link back to reducing the amount of complex data (EEG) needed to be reviewed by a clinician in the future. Ideally, the algorithm that operates on the less complex data will also operate on the complex data. The beauty is that with minimal training (less than one hour) by the user 10,000 hours of data could be easily reduced down 1,000 of feature dense complex data. These 1,000 hours of data would then only require an hour of clinician/expert time to train a new algorithm that could then operate on the entire 10,000 hours of data. With persistent stream devices on the horizon, 10,000 hours is 416 days of patient data.

I took real sample data from explore cloud to build an initial model system that had user input from a Matlab prompt. Given that Matlab isn't the best I/O setup, it worked well enough. It attempts to match the time series signal the user selects to potential signals in the next sample window. The user can specify the input file and the parameters of the window based program. This is all triggered from the **dataAnnotation()** file.

dataAnnotation( ‘*filename’*, *sample\_rate*, *window\_size* )

* Sample rate is in hertz
* window size is in seconds
* filename needs to be in single quotes due to how Matlab handles strings

This generates a window for the user to start selecting features, once three samples of the same feature are provided, the algorithm starts to match. There are hooks in place to enable capturing features of different types, but they aren’t finished or tested. The main function is **annotationWindow()** that produces the window and processes all the data via various sub-functions.

annotationWindow(*raw\_data*,*window\_count*,*sample\_rate*,*data\_length*)

* raw\_data anticipates an *N* by *M* vector, where *N* are the data channels
* window\_count is the same value from before in determining how much data to display
* sample\_rate is from dataAnnotation
* data\_length is probably just lazy coding given it is the full length of raw\_data

After this initial version it quickly showed various flaws of window methodology and difficulties when testing having to continually require user input. This lead to the second iteration which enabled automated inputs and automated data analysis for empirical study. To achieve this I collected real data from four interns with a Samsung Galaxy SIII Mini. I then build up templates of the data so I would know where my features were located so I could generate automated test/training data. This resulted in the creation of **generateFeatureVector()**.

generateFeatureVector({*feature\_file\_list},feature\_count,spacing\_max,test\_name,warp\_factor*)

* feature\_file\_list is a cell of file names corresponding to the templates to be used
* feature\_count lists the number of times each feature is to be included in the test vector, this should be an array with one entry for each entry in feature\_file\_list
* spacing\_max provides an upper bound on the number of samples between features, it stiches them together with a linear interpolation and a combine cosine+sine wave
* test\_name provides a name for the generated files to be tracked back to the same test
* warp\_factor is a binary toggle of enabling the features to be randomly resampled from their original templates, making them larger or smaller in the time domain

Once a test is generated calling **featureTesting()** or **featureTestingPlot()** enables validation of the algorithm against the randomly generated test. featureTesting() runs once with the ability to pass in a noise coefficient for setting the SNR ratio. featureTestingPlot enables a vector of noise coefficients to be passed in (or left to the default) and the resultant data and plots saved to a new folder tagged with the test name and date of the test. There are a few ‘hanging chads’ of **featureTestingPlot2()** and **featureTestingSingle()** were developed to repeated testing of the same vectors to ensure robust results. Updates to featureTestingPlot() never made their way into these two files however.

These are the main files, along with the data files, that one needs to understand to reproduce the results of my work. I will not overview the methodology of the work in an effort to make clear the process by which these files follow to achieve their results.

The present work relies on having flat files to process and no effort was made to address the eventual streaming nature of the system. Ideally a known feature (or set of features) in the raw signal are identified and catalogued by the algorithm. In my case, versions 0.2 and 0.3 are fed in these features as templates. V0.1 worked to build them at the discretion of the user. Once given features both systems using a sliding window of sample size equal to the same rate, with 75% overlap, to find potential matches.

Given that the user defined features and the sliding window are going to be of different sizes the use of dynamic time warping is essential to match the figures. The resultant distance between the samples could be used as the distance metric, but I have preferred to instead use the two newly built warped signals. These two signals can be feed to a mean squared error calculation which can be the feature component of the KMeans clustering algorithm.

A point of interest is that the signals are not normalized before applying the dynamic time warping. In a majority of papers it is encouraged to normalize them prior to warping so that the warping can be balanced between weighting time and amplitude. As the implemented algorithm worked fine without this feature I never enabled it. I did however use the reciprocal of the mean squared error as the true feature component. The KMeans cluster worked by breaking the set into two groups, feature or not feature, which requires it to be called for each unique feature. This allows for features to overlap, which could be useful for diagnosis, and also does not require complicated grouping of the features when defining them for testing. Later layers of learning should be used to the group the features as the initial machine learning needs to be as general as possible so as not to miss any data.

All of this is carried out in the function **featureFinderAuto()** which operates within the featureTesting() function. The function **featureCompare()** is used to find the mean squared error between two features and their given feature components (it could also be done for their FFT or whatever is desired as long as it becomes a component of the **FeatureCrate** object). featureCrate was designed to be a place to store all necessary functions and data associated with the signal of interest. When a feature was found the raw signal could be stored alongside the time warped version of it, the Fourier Transform of it, along with the select energy band transform values or whatever would be necessary given its type. The type of the feature is also afford a spot given that some features may be of different events or groupings depending on the need of the user. Sadly, most of this stayed as ideas with only storage of data being implemented.

The dynamic time warping used is a mash up of two sets of open source code, both of which come from the mathworks file exchange. One implements the distance calculation of the warping matrix in mex(C) while the other would return the two resultant warped signals and a nice plot of how the system worked. As the full Matlab version with plot took far too long to run, I used the Mex version to generate the initial *M* by *N* distance calculation matrix. In this sense **cdtw2()** is the front end that uses **dtw\_c2.c** to generate the distance matrix. The results are quite reasonable, but there is no native package for this in Matlab so the user is left to their own devices.