General Information

Python requirements

You need a fully setup Python installation (e.g., Anaconda**)** including the **numpy**, **scipy** and **matplotlib** modules.

Additional modules: Please also install the mne toolbox (a Python toolbox dedicated specifically to EEG data analysis). In a terminal, simply type

```
pip3 install mne
```

Course material

- `data` (1) For part I and II: EEG data from a P300 Speller BCI calibration session which you will use to implement your own calibration script. (2) For part III: eCoG data from a Motor Imagery task.

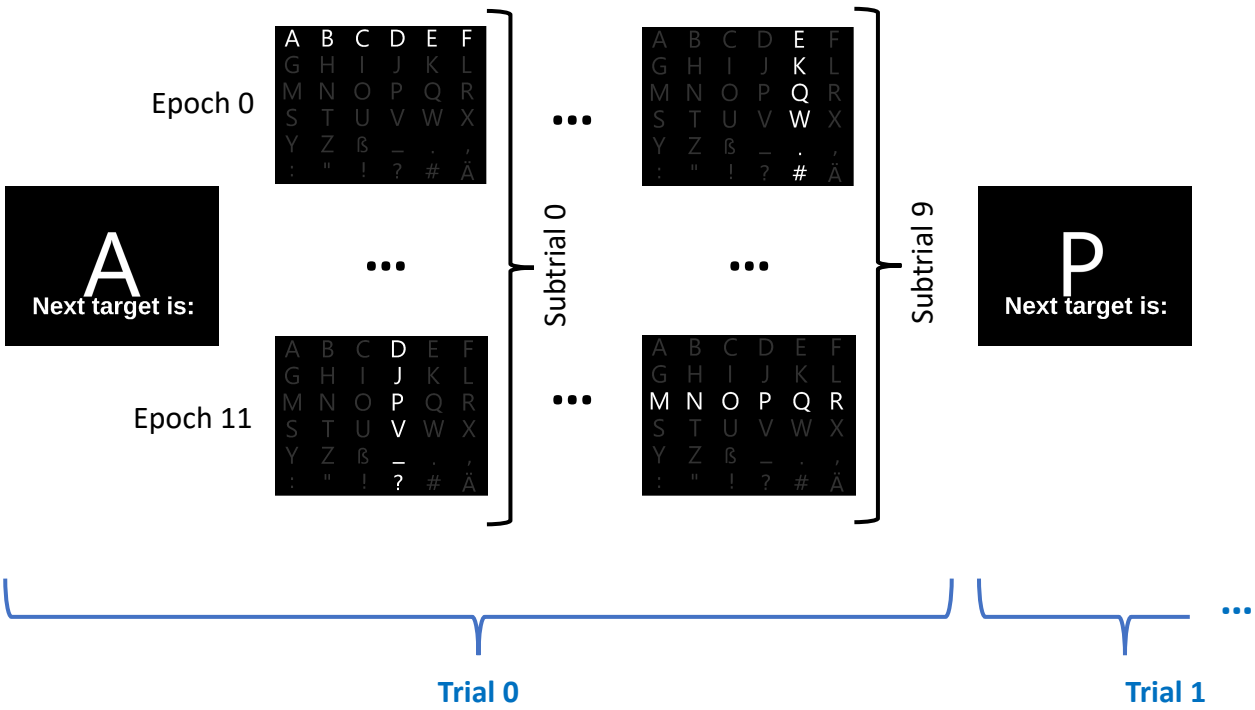
- `toolbox` Some helper functions in Python for the P300 calibration task, the wavelet transform, and the eCoG MI classification task.

NumPy reference: https://docs.scipy.org/doc/numpy/reference/
SciPy reference: https://docs.scipy.org/doc/scipy-1.0.0/reference/
Matplotlib.pyplot: https://matplotlib.org/api/pyplot_summary.html
MNE toolbox reference: https://martinos.org/mne/stable/python_reference.html

Python console: Is opened by typing `ipython3` in a terminal. You can run scripts inside the console by calling `%run somescript.py`. All variables (name, type and size) currently in the workspace (those created in the script but not inside functions called from that script) can be shown by typing `whos`. Copied code snippets can be directly pasted into the console with the `%paste` command.

Part I: The P300 Speller calibration task

Structure of experiment (P300 BCI Training)



Variables contained in dataset *p3bci_data.npz*

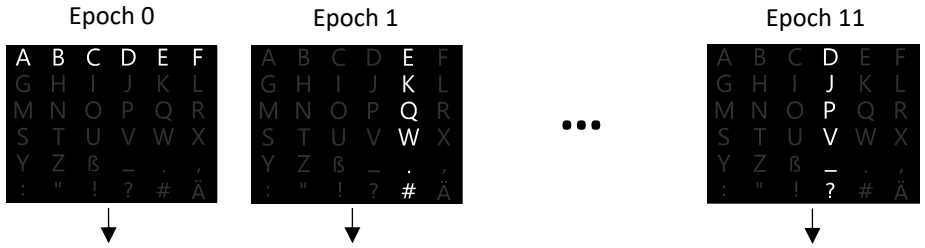
Name	Dimension	Description
`data`	`Matrix [137889x10]`	`Continuously (over the whole experiment) sampled 10–channel EEG data`
`flashseq`	`3D-Array 30x10x12`	`Index of row/ column flashing in epoch`
`onsets`	`3D-Array 30x10x12`	`Timestamps of flash onsets in stimulus presentation`
`targets`	`Vector [30]`	`Index of target cell for each trial`
`timestamps`	`Vector[137889]`	`Timestamp for each EEG data sample`

Task 1: Segmentation

Load the data using the script p300_load_data.py. Create EEG epochs starting at the onset of a flash and lasting for 800 ms (205 samples @256 Hz).
You need the variables `data`, `timestamps` and `onsets`.
This (ideally) results in a 3D-Array with dimension [3600 x 205 x 10] → [epochs x samples x channels].
Epoch number is 30 trials * 10 subtrials * 10 flashes = 3600.
Hint: For each onset, you need to find **min(abs(timestamps-onset([i,j,k]))**

data [137889x10] → Amplitude in muV

Samples	Channels										timestamps [137889x1]
	Fz	Cz	Pz	Oz	C4	P4	PO8	C3	P3	PO7	
	0	1	2	3	4	5	6	7	8	9	
	-34.2116	-14.6184	-26.3894	-18.2760	-16.3372	-16.7786	-12.0880	-12.7202	-19.7992	-11.2867	t0
	-37.1727	-14.3919	-24.5748	-12.9586	-14.4914	-12.9029	-7.5815	-12.3270	-19.3088	-9.2431	t1
	-41.7361	-14.2078	-24.0219	-12.8760	-17.0238	-14.0816	-4.9931	-14.8641	-20.2108	-8.5117	t2
	-41.5582	-17.5688	-27.8683	-17.2784	-17.5524	-18.7101	-13.9011	-18.0195	-21.5927	-9.7515	t3
	-56.2664	-32.4222	-38.9705	-23.9682	-31.2643	-27.9027	-21.3607	-29.2959	-32.5913	-16.2275	t4
	-57.8894	-30.8098	-37.3009	-22.0572	-27.0669	-27.0702	-18.3261	-28.9332	-31.7366	-15.7235	

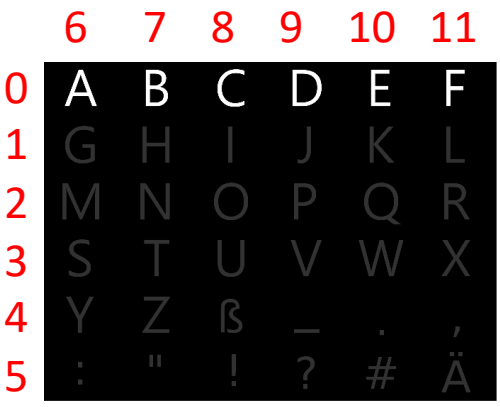


onsets [30x10x12]

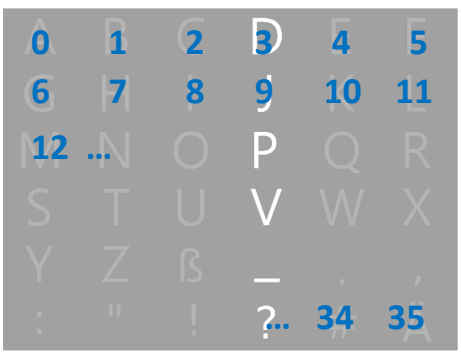
Timestamp, when flash of row or column started. Same format as variable `timestamps`.

Task 2: Grouping

Divide the epochs in the newly created array into the two relevant groups (or conditions, or classes): **target** and **non-target**, depending on whether the row/column flash contained the target symbol or not. Please generate a `label` vector (needed for classifier training) with dimension [3600 x 1]. This vector should contain a 1 if the respective epoch is a target epoch, and 0 otherwise.
You need the variables `flashseq` and `targets`.



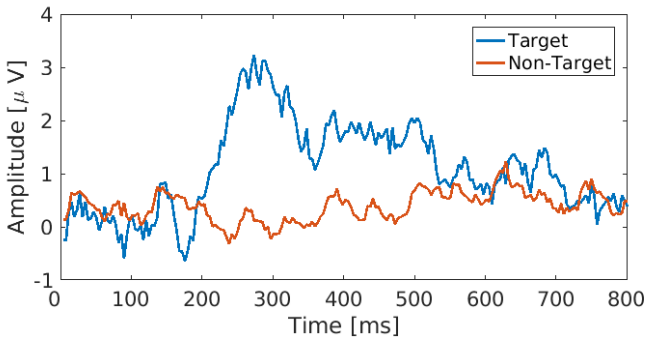
flashseq [30x10x12]



targets [1x30]

Task 3: Event-Related Potential (ERP) plots

To verify segmentation and grouping, please generate channel-wise ERP plots. To do so, you need to average over all epochs *per group*. Your plot should look like this (channel 2). You can call `plt.ion()` before plotting to get non-blocking interactive plots. Use `plt.plot(x,y)` (assuming plt is matplotlib.pyplot).



Task 4: ERPs and topoplots

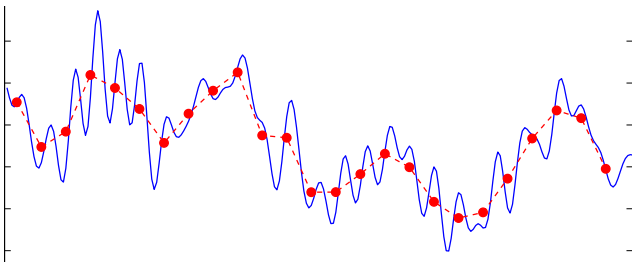
You will now plot the grouped ERP data onto a 2D head model. First, you need to generate an info object that holds the names and positions of the channels in the data set. Use the function `eeginfo = chan4plot()` from utils.py to do that. Inside this function, you need to adjust the path to the p300speller.txt file containing the channel labels.
The function `sba = viz.plot_topomap(erp, eeginfo, show=True)`
From the toolbox mne.viz creates nice such head plots (i.e., the interpolated activation distribution for one point in time, erp is a [1 x 10] row vector). Also plot the $\mathbf{r^2}$ values for different time points. $\mathbf{r^2}$ is the *squared difference between the target and the non-target average*.
Observe how the activation, especially the P300 component, evolves over time.

Task 5: Preparing for cross-validation

You will now prepare the data for assessing the performance of a classifier in a cross-validation scheme. For a start, the feature extraction (i.e., dimension reduction) will take the simplest yet effective form: Down sampling the signal by averaging over a specified window (10 samples).

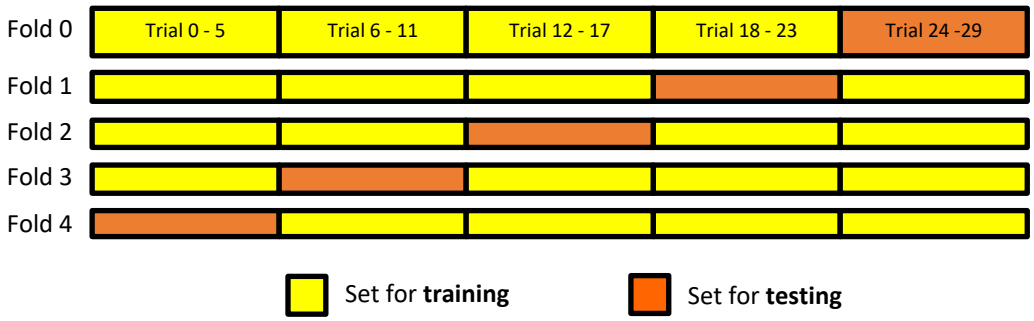
Please use the function (expects an array [epochs x samples x channels]) from utils.py

```
data = downsample(data,10)
```



Your data should now have the dimension [3600 x 20 x 10].
Now, create feature vectors by concatenating the reduced *samples* and the *channels*, resulting in a matrix with dimension [3600 x 200]. This, plus the label, is needed for the initial cross-validation.

Divide your data matrix in 5 equally sized sets (or create indices for accessing respective chunks), each containing an appropriate ratio of target to non-target epochs (1:5). For the sake of later steps, it is beneficial to retain the original trial/ subtrial structure (although here not strictly necessary).
The labels need to be divided accordingly! You should have 720 data epochs and labels per set.



Task 6: Cross-validation and FDA classifier performance with ROC and AUC

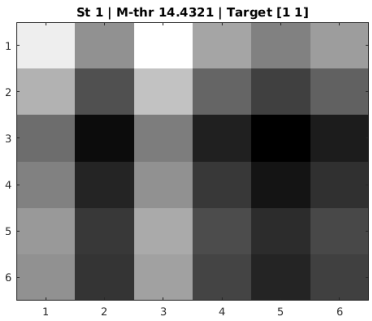
Create an FDA training function (hint: example code lecture slides) that expects a training data set and the corresponding label. The function should return the weight vector (fda.w) and the bias (fda.b). Please use the function `cov_shrinkage` from utils.py to compute the covariance matrices (gives a better estimate than the Numpy built-in function `numpy.cov`

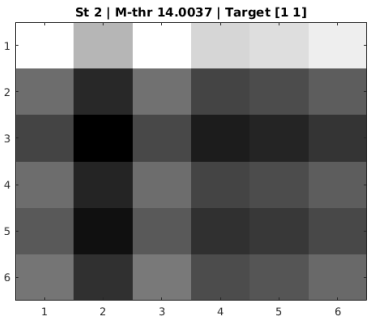
Iterate over your folds choosing the appropriate training and testing sets. Classify the test set using the obtained FDA parameters: `ys = numpy.mutmul(testdata,fda_w)`

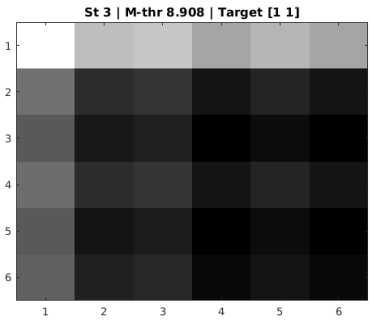
No compute the Receiver Operator Characteristics (ROC) curve to assess the performance. You need the above ys and the true labels of the test set. Scale the ys to an interval [0,1]. The ROC is generated for a number (typically 100) of different values for fda_b in the range [0,1]. You can create them with
`bias = numpy.linspace(0,1,steps)`
Iterate over these 100 bias values. Set all labels for the scaled ys<bias to 0, and all other labels to 1. Calculate the confusion matrix at each iteration using the function
`TP,FP,FN,TN = calc_confusion(prelabel,truelabel,1,0)`
Store the number of TPs and FPs at each iteration. They will ultimately constitute the ROC, which is obtained by plotting the 100 TP values against the FPs. Please take a moment to understand how the ROC is generated by systematically shifting the bias parameter. Plot the ROC. The function
auc = calc_AUC(roc) will return the AUC, that is, the area under your ROC curve.
Finally, train the full data set, generating a final `fda_w` and `fda_b` . If the BCI was to be used later for an online run, these two variables needed to be stored (unfortunately, there is no online BCI this year).

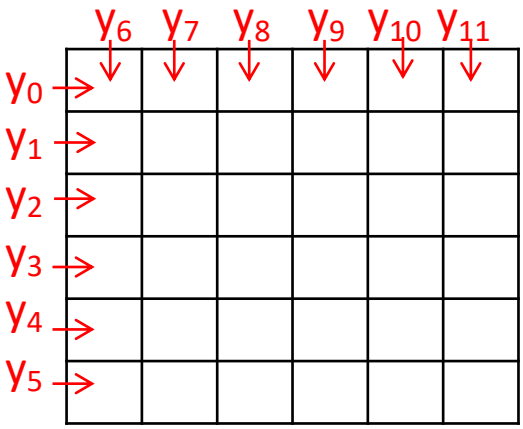
Task 7: Calculating threshold for dynamic subtrial limitation

When running the online system, the number of subtrials per trial should be determined automatically by the system, based on the confidence of the classification result. This requires (1) a measure for this confidence and (2) a threshold derived from the training data for the measure.
(1) We refer to this measure as “matrix brightness”. This is computed by taking the real-valued outputs of the FDA, `ys`, and sum them up subtrial-wise according to the row/column flash they belong to. The entries in the resulting Matrix M are scaled onto the interval [0 1]. The next figure shows images of M for three subtrials. The ”brightness” `M_thr` is computed by summing up all elements of M. With increasing subtrials, this overall "brightness” will decrease while the value of the individual target cell will increase.
(2) The threshold is determined by selecting the value of `M_thr` at the subtrial, when the winning cell corresponds to the true target. The final M_thr for the online system is the mean over values obtained during the cross-validation.









In your code, this step is added inside the cross-validation loop, right after the `ys = numpy.matmul(testset,fda.w)` call. Use the function `TPscore,M_tp,sbtrix = calcTPscore(ys,flashseq,target,isSpeller,doImg)`

Check the help for this function to see, which dimensions the input and output arguments have and adapt your variables accordingly.
The function, if the `doImg` flag is set to 1, will also plot the "brightness" images like the ones above. Note that you should not do that for the full cross-validation, but only for selected trials (you will end up will many plots!).

The function is called once per test trial. Collect all M_thr values in a matrix or vector and average them after the cross-validation. Add the result to the fda struct by setting `fda.M_thr = M_thr` before saving `fda`. You are now all set to train the classifier on your own data and then use it to run the online P300 Speller BCI.

Task 8: PCA for feature extraction

The final task is to replace the simplified feature extraction by downsampling with Principal Component Analysis (PCA). Remove the `downsample` line from your code an replace it with the filtering routine you have just created. PCA is computed inside the fold loop only on the train set and subsequently applied to both train and test set. Compute the covariance of the data first by using `numpy.cov(data)`. Note that the data has to have shape [samples x epochs] and then call `pcamat,eigval,_ = scipy.linalg.svd(Xcov)`
`pcamat` contains the eigenvectors of the data covariance matrix in the columns (the principal components) and the corresponding eigenvalues in `eigval`. Compute how many PCs are needed to capture 99,9% of the variance using the eigenvalues. These n PCs are used for the projection, which reduces the dimensionality. Do this for the training and testing set at each iteration of the fold loop. The rest of the code (FDA, M_thr calculation, AUC) remains the same.
Check the AUC by running the modified code – if it is fine, great! You are done.

Part II: Frequency Transformations

Task 1: Wavelets and spectrogram

You will now compute the spectrogram of the ERP data (the *averaged* data from previous task). This results in a time-frequency-bandpower representation. Use the complex Morlet wavelet (known from the lecture) for this task. The wavelet is defined by

$$\Psi(f,x) = \sqrt{\pi\sigma} \cdot \exp(2\pi i f x) \cdot \exp\left(-\frac{x^2}{\sigma}\right)$$

The mother wavelet has 2 parameters: frequency *f* und band width *σ*. Create one such wavelet by applying the above formula for the interval [-5 5]. Plot it. Investigate, how different values for *f* and *σ* change the wavelet. The Parameter *σ* determines the width of the Gaussian window and thus the number of cycles (i.e., periods) of the wavelet. This, in turn, defines the temporal resolution of the analysis. The more cycles, the lower the temporal resolution. The latter relationship also means that with more cycles, you need a longer signal segment.

Keep in mind that the CWT multi-resolution analysis. Remember the relation between temporal and frequency resolution. The function

```
coeffs, Ws = wavelet_coeffs(data,freqs,srate,nco)
```

from wavelet.py accepts a parameter nco, which stands for the number of cycles. Ws holds a list of the created wavelets. Inspect them for different values of nco. When you are satisfied with the parameters, transform the ERPs (target and non-target averages) with the above function. The function `power = wavelet_power(coeffs)` return the real—valued, squared coefficients suited for plotting. Plot the spectrograms using the function

```
fig,ax = plt.subplots()
cax = ax.imshow(power,cmap='jet',aspect='auto',\
            extent=[0,power.shape[1],power.shape[0],0],origin='upper')
```

and compare them.

Task 2: Bandpass filtering with FFT

You will now bandpass filter your data. This requires the 3D-Arrays holding the epochs (no more averaging). FFT is used to create such a filter.

Numpy provides the functions `fft` and `ifft` for this purpose. The cut-off frequencies must be calculated depending on the sampling rate (256 Hz) and the length of the input segment (205 samples = 800 ms). Do so. Data should be filtered using a 1-10 Hz bandpass. First, transform the data into the frequency space using `coeffs=numpy.fft.fft(data)`. Then, remove all coefficients that correspond to frequency outside the desired band. Do so by setting these coefficients to zero. Finally, transform your data back into the time-amplitude plane by using the inverse FFT and retaining only the real part `filtdata=real(numpy.fft.ifft(coeffs))`.

Part III: Motor Imagery Data Recorded with eCoG Grid

All resources for this task are located in `data/ecog_mi`.
The file *Dataset_description.png* contains a brief description of the dataset and how it was acquired. *BCI-Comp-III-eCoG_NIPS_2004.pdf* is the research paper associated with the dataset and contains in depth information.

The script `ecog_load_data.py` contains code how to load the datasets.

Dataset Competition_train.mat

Name	Dimension	Description
data	3D Array[278x64x3000]	Epoched data with 278 trials, 64 eCoG channels and 3000 samples (3 sec duration).
label	Vector[278]	The class labels, {-1,1}

Task 1: Filtering

```
import mne.filter as filter
filtdata = filter.filter_data(data,1000,hi,lo,method='iir',n_jobs=6)
```

With `hi` and `lo` being the respective cut-off frequencies and 1000 is the sampling rate in Hz.

Task 2: Common Spatial Pattern (CSP)

First, the data needs to be grouped with respect to the label. Then, implement the CSP algorithm following these steps.

1. Input data fo the form: $\vec{x} = \begin{pmatrix} x_{1,1} \dots x_{1,d} \\ \ddots \\ x_{m,1} \dots x_{m,d} \end{pmatrix}$

   d Number of channels
   m Number of samples

   The gouped data **is averaged** over all epochs in the respective group.

2. Computation of the group-wise covariance matrices:
   1. Averaging over all $\{\vec{x_i}\}_{C_1}$ and $\{\vec{x_i}\}_{C_2}$
   2. Compute the between-class covariance matrix: $C = C_1 + C_2$
3. Factorize the eigenvalues and eigenvectors like in PCA: $C = V\lambda V^{\mathsf{T}}$
4. Where $\boldsymbol{V}$ Matrix is the matrix of eigenvectors and λ the diagonal matrix of eigenvalues.
5. Normalization of the eigenvectors $\boldsymbol{V}$ to 1 using the Whitening transformation $P = \sqrt{\lambda^{-1}}V^{\mathsf{T}}$
6. Factorization of $\boldsymbol{C_1}$ und $\boldsymbol{C_2}$ in $S_1 = PC_1P^{\mathsf{T}}$ and $S_2 = PC_2P^{\mathsf{T}}$
7. Them, $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$ **have common** eigenvectors: $S_1 = B\lambda_1 B^{\mathsf{T}}$ and $S_2 = B\lambda_2 B^{\mathsf{T}}$ with $\lambda_1 + \lambda_2 = \mathbb{I}$
8. Based on the latter condition the largest EV of $\boldsymbol{S_1}$ the smallest EV of $\boldsymbol{S_2}$. Sort the EVs and their eigenvectors in descending order!
9. Projektion (or CSP) matrix $W = (B^{\mathsf{T}}P)^{\mathsf{T}}$

In Python:

```
lambda,V = scipy.linalg.eig(C)        → Standard Eigenvalue problem
lambda,B = scipy.linalg.eig(S1,S2)    → Generalized Eigenvalue problem
```

The columns of W are the spatial filters, the rows of inv(W) are the spatial patterns. You will need the spatial filters (with reduced dimensionality) to project the data.

Task 3: Cross-validation of training data including CSP computation and application

Please implement (or reuse) a cross-validation scheme for the eCoG data.

The CSP matrix is computed inside the cross-validation only on the training data and then applied to both, training and testing data.
The dimensionality of the CSP matrix is reduced by retaining only a number of the first and the last spatial filters (i.e., columns of W). After removing the middle vectors, W has shape [64 x dim*2], where dim is the number of vectors removed at each side. Each epoch is shaped [3000 x 64] for projection (i.e., times x channels), resulting in reduced epochs of shape [3000 x dim*2]. Finally, we take

```
featurevec = np.log(np.var(epoch,axis=0))
```

resulting in feature vectures with shape [1 x dim*2]. Taking the log variance means to consider only the band power (w.r.t. the cut-off frequencies used for filtering, e.g., 8 to 30Hz) in this frequency band and at the spatially filtered channel.

Training, testing and AUC calculation are just like for the P300 Speller. Finally train a classifier on the full set and save it for application to the test set. Save fda_w, fda_b, and the dim-reduced CSP matrix.

Task 4: Classifying the test set

Dataset Competition_test.mat (Use the code snippet from load_data.py)

Name	**Dimension**	**Description**
`data`	`3D Array[100x64x3000]`	`Epoched data with 100 trials, 64 eCoG channels and 3000 samples (3 sec duration).`
`truelabel`	`Vector[100]`	`The class labels, {-1,1}`

Load the stored classifier file. Filter the data using the same cut-off frequency as during training. Apply the CSP (don't forget the log-var step). Classify the test set with the loaded fda_w and compute the ROC and AUC. If the ROC looks like *classresults_roc.png* – perfect!

You're done ☺