13.5/15 points

2019ws-BCIIL-Sheet03-Karastoyanov,Mohan,Al-Asadi

November 12, 2019

0.0.1 General rules:

- For all figures that you generate, remember to add meaningful labels to the axes (including units), and provide a legend and colorbar, if applicable.
- Do not hard code constants, like number of samples, number of channels, etc in your program. These values should always be determined from the given data. This way, you can easily use the code to analyse other data sets.
- Do not use high-level functions from toolboxes like scikit-learn.
- Before submitting, check your code by executing: Kernel -> Restart & run all.
- Replace *Template* by your *FirstnameLastname* in the filename, or by *Lastname1Lastname2* if you work in pairs.

1 BCI-IL - Exercise Sheet #03

Name:

```
In [1]: % matplotlib inline
        import numpy as np
        import scipy as sp
        from matplotlib import pyplot as plt

        import bci_minitoolbox as bci
```

1.1 Exercise 1: Nearest Centroid Classifier (NCC) (2 point)

2/2

Implement the calculation of the nearest centroid classifier (NCC) as a Python function `train_NCC`. The function should take two arguments, the first being the data matrix **X** where each column is a data point ($\mathbf{x_k}$), and the second being class labels of the data points. Two output arguments should return the weight vector `w` and bias `b`.

```
In [2]: def train_NCC(X, y):
            '''
            Synopsis:
                w, b= train_NCC(X, y)
            Arguments:
                X: data matrix (features X samples)
                y: labels with values 0 and 1 (1 x samples)
            Output:
```

```
    w: NCC weight vector
    b: bias term
'''
# compute a centroid for every class given the column data xk
mu0 = np.mean(X[:, y == 0],axis = 1)
mu1 = np.mean(X[:, y == 1],axis = 1)

w = (mu1 - mu0)
b = ((w.T.dot(mu0) + w.T.dot(mu1))/2)
return w, b
```

1.2 Exercise 2: Linear Discriminant Analysis (LDA) (4 points)

4/4

Implement the calculation of the LDA classifier as a Python function `train_LDA`. The function should take two arguments, the first being the data matrix **X** where each column is a data point ($\mathbf{x_k}$), and the second being class labels of the data points. Two output arguments should return the weight vector `w` and bias `b`.

```
In [3]: def train_LDA(X, y):
            '''
            Synopsis:
                w, b= train_LDA(X, y)
            Arguments:
                X: data matrix (features X samples)
                y: labels with values 0 and 1 (1 x samples)
            Output:
                w: LDA weight vector
                b: bias term
            '''
            mu0 = np.mean(X[:, y == 0],axis = 1)
            mu1 = np.mean(X[:, y == 1],axis = 1)

            cov_0 = np.cov(X[:, y==0])
            cov_1 = np.cov(X[:, y==1])
            cov = (cov_0 + cov_1) /2

            w = np.linalg.inv(cov).dot(mu1 - mu0)
            b = ((w.T.dot(mu0) + w.T.dot(mu1))/2)
            return w, b
```

1.3 Exercises 3: Cross-validation with weighted loss (2 points)

1/2

Complete the implementation of `crossvalidation` by writing a loss function `loss_weighted_error` which calculates the weighted loss as explained in the lecture.

```
In [4]: def crossvalidation(classifier_fcn, X, y, nFolds=10, verbose=False):
            '''
            Synopsis:
```

```
        loss_te, loss_tr= crossvalidation(classifier_fcn, X, y, nFolds=10, verbose=Fal
    Arguments:
        classifier_fcn: handle to function that trains classifier as output w, b
        X:              data matrix (features X samples)
        y:              labels with values 0 and 1 (1 x samples)
        nFolds:         number of folds
        verbose:        print validation results or not
    Output:
        loss_te: value of loss function averaged across test data
        loss_tr: value of loss function averaged across training data
    '''
    nDim, nSamples = X.shape
    inter = np.round(np.linspace(0, nSamples, num=nFolds + 1)).astype(int)
    perm = np.random.permutation(nSamples)
    errTr = np.zeros([nFolds, 1])
    errTe = np.zeros([nFolds, 1])

    for ff in range(nFolds):
        idxTe = perm[inter[ff]:inter[ff + 1] + 1]
        idxTr = np.setdiff1d(range(nSamples), idxTe)
        w, b = classifier_fcn(X[:, idxTr], y[idxTr])
        out = w.T.dot(X) - b
        errTe[ff] = loss_weighted_error(out[idxTe], y[idxTe])
        errTr[ff] = loss_weighted_error(out[idxTr], y[idxTr])

    if verbose:
        print('{:5.1f} +/-{:4.1f}  (training:{:5.1f} +/-{:4.1f})  [using {}]'.format(e
                                                                                      e
                                                                                      c

    return np.mean(errTe), np.mean(errTr)
```

```
In [5]: def loss_weighted_error(out, y):
    '''
    Synopsis:
        loss= loss_weighted_error( out, y )
    Arguments:
        out:  output of the classifier
        y:    true class labels
    Output:
        loss: weighted error
    '''
    class_0_error = 0;
    class_1_error = 0;

    for i in range(y.shape[0]):
        if out[i] < 0 and y[i] == 1:
            class_0_error = class_0_error+1
        if out[i] >= 0 and y[i] == 0:
```

```
                class_1_error = class_1_error+1

        return ((class_0_error/y.shape[0]) + (class_1_error/y.shape[0])) /2
```

1.4 Preparation: Load Data

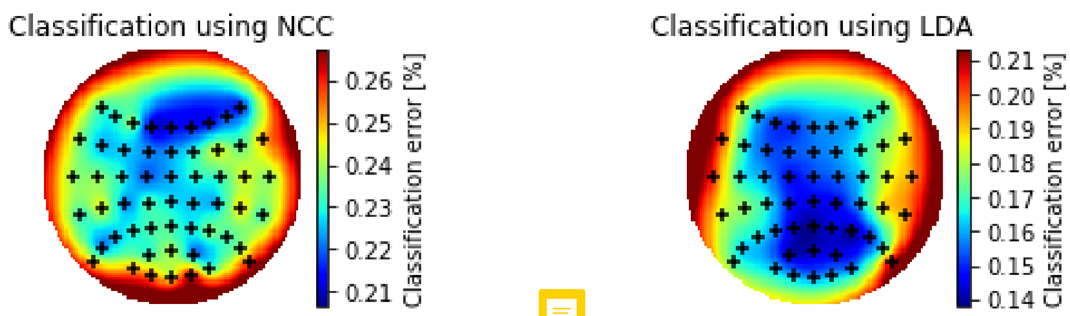
```
In [6]: fname = 'erp_hexVPsag.npz'
        cnt, fs, clab, mnt, mrk_pos, mrk_class, mrk_className = bci.load_data(fname)
```

1.5 Exercise 4: Classification of Temporal Features (3 points)

3/3

Extract as temporal features from single channels the epochs of the time interval 0 to 1000 ms. Determine the error of classification with LDA and with NCC on those features using 10-fold cross-validation for each single channel. Display the resulting (test) error rates for all channel as scalp topographies (one for LDA and one for NCC).

```
In [7]: ival= [0, 1000]
        epo, epo_t = bci.makeepochs(cnt, fs, mrk_pos, ival)

        loss_test_ncc_list =[]
        loss_test_lda_list =[]

        for i in range(len(clab)):
            loss_test_ncc_list.append(crossvalidation(train_NCC, epo[:,i,:], mrk_class, nFolds=
            loss_test_lda_list.append(crossvalidation(train_LDA, epo[:,i,:], mrk_class, nFolds=

        a = np.array([[loss_test_ncc_list],[loss_test_lda_list]])

        classifiers = ['NCC', 'LDA']
        for i in range(len(classifiers)):
            plt.subplot(1,2,i+1)
            bci.scalpmap(mnt, a[i], clim='minmax', cb_label="Classification error [%]")

            plt.title('Classification using ' + classifiers[i])
        plt.subplots_adjust(left  = 0.001, right = 1, bottom = 0.1, top = 0.9, wspace = 1, hspa
```



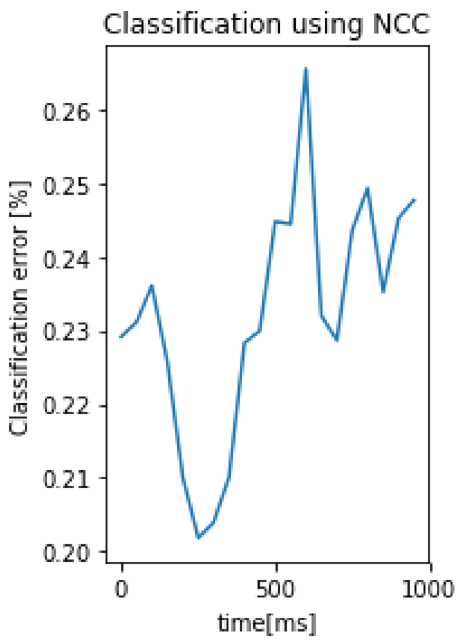
1.6 Exercise 5: Classification of Spatial Features (4 points)

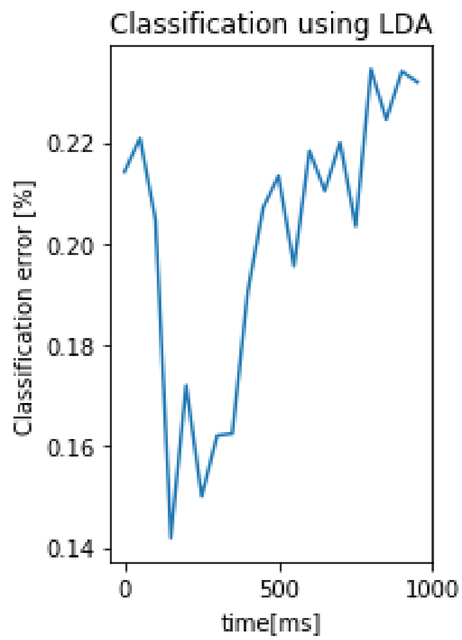
3.5/4

Perform classification (*target* vs. *nontarget*) on spatial features (average across time within a 50 ms interval) in a time window that is shifted from 0 to 1000 ms in steps of 10 ms, again with both, LDA and NCC. Visualize the time courses of the classification error. Again, use 10-fold cross-validation. Here, use a baseline correction w.r.t. the prestimulus interval -100 to 0 ms.

```
In [8]: ival= [0, 1000]
        ref_ival= [-100, 0]

        # Segment continuous data into epochs:
        epo, epo_t = bci.makeepochs(cnt, fs, mrk_pos, ival)

        # Baseline correction:
        epo = bci.baseline(epo, epo_t, ref_ival)

        intervals = np.arange(0,101,5)

        loss_test_ncc_list_2 =[]
        loss_test_lda_list_2 =[]

        for i in range(len(intervals)-1):
            X = epo[intervals[i]:intervals[i+1],:,:].mean(axis=0)
            loss_test_ncc_list_2.append(crossvalidation(train_NCC, X, mrk_class, nFolds=10, ve
            loss_test_lda_list_2.append(crossvalidation(train_LDA, X, mrk_class, nFolds=10, ve

        b = np.array([[loss_test_ncc_list_2],[loss_test_lda_list_2]])

        classifiers = ['NCC', 'LDA']
        for i in range(len(classifiers)):
            plt.subplot(1,2,i+1)
            plt.plot(intervals[:-1]*10, b[i].T)

            plt.title('Classification using ' + classifiers[i])
            plt.xlabel("time[ms]")
            plt.ylabel("Classification error [%]")

        plt.subplots_adjust(left  = 0.001, right = 1, bottom = 0.1, top = 0.9, wspace = 1, hspa
```

-0.5





In [9]:
```
#print(sfksdmf)
#print(sfksdmf)
```

In [10]:
```
#print(sfksdmf)
#print(sfksdmf)
#print(sfksdmf)
```