Machine Learning Algorithms on P300&EEG Data

March 18, 2021

1 Machine Learning Algorithms on P300/EEG Data

1.0.1 Taylor Han, Tae Kwang (Jason) Chung, Ahmad Sadeed

1.1 Introduction & Motivation

BCI is the process of transforming brain activity into transcribable signal data that can be manipulated and find trends. BCI is currently being used in multiple aspects in the field, not limited to medical research to help those who may have lost their motor ability. Thus, through the use of supervised machine learning, models can be trained so that it is able to detect and classify signals accordingly (in this case, good or bad feedback).

For our project, we were interested in solving the Kaggle BCI Challenge @ NER 2015. This Jupyter Notebook file showcases our efforts in cleaning the noisy data and training multiple models that classify whether a feedback was good or bad. The data consists of 56 P300 electrode signals across the brain, EEG evoked responses recorded every 5 milliseconds. The brain wave data consists of test subjects with the goal of spelling a word by only paying attention to visual stimuli. The goal of the competition is to detect errors during the spelling task, given the subject's brain waves.

The "P300-Speller" is a well-known brain-computer interface (BCI) paradigm which uses Electroencephalography (EEG) and the so-called P300 response evoked by rare and attended stimuli in order to select items displayed on a computer screen. In the experiment, each subject was presented with letters and numbers (36 possible items displayed on a matrix) to spell words. Each item of a word is selected one at a time, by flashing screen items in groups and in random order.

1.2 Related Work

In 2017, a paper "Comparison of signal decomposition methods in classification of EEG signals for motor-imagery BCI system" was published by Kevric et. al. This paper investigated ML performance on BCI data. Although it considered motor imagery tasks, the work from that paper is still related to P300 response exploration done in this challenge. Our group also took notes of how the findings were presented and attempted to stage our own findings in an organized manner.

1.3 Data Introduction

Our data is collected at 200Hz across 26 subjects, 16 for training and 10 for testing. Each subject participated in 5 different sessions. Each session included 60 target stimulus, with the last session

of each subject containing 100 target stimulus. In total, there were 340 target stimulus for each subject.

The feedback label consists of either 0 or 1, bad or good, respectively. Bad feedback is when the selected item is different from the expected item. Good feedback is when the selected item is similar to the expected item.

1.4 Methods

We first downloaded all the files from the Kaggle BCI Challenge and unzipped them in a subdirectory called "data/". From there, the train and test data were already splitted, and the train data had its labels on a separate file. Keeping this in mind, the first decision we made was to discard all unnecessary columns from the original data files. These included not only the obvious columns such as Time, but we also figured that the most important electrodes would be the ones in the middle of the brain (i.e. Fz, FCz, Cz, CPz, Pz and POz) and thus we discarded all those columns as well. Thus, our data files were considerably reduced from 59 columns to only 6 columns.

Next, for filtering the data we decided to use the FIR filter design using the window method. We extract the train data first by running the FIR filter for each train data file. Then, we epoching the data by 20 indices before and 100 indices after each feedback event, leading to 120 indices per feedback event. Finally, for each feedback event we generate a 2-dimensional array of shape (6, 120) so that each row represents data for an electrode at that particular feedback event. Because there were in total 5440 feedback events occurred during the training data, the final result of the extraction was a 3-dimensional array of shape (5440, 6, 120). We store this array in a separate file so that this 3D array could be leveraged without extracting the same training data multiple times.

1.5 Preprocessing Raw EEGNet Data

[18]:
```
import numpy as np
import pandas as pd
import glob
import time
from scipy import signal
from tqdm.notebook import tqdm
import copy

from sklearn import ensemble
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_curve

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import auc

import matplotlib.pyplot as plt
import matplotlib.patches as patches

import random
```

[2]:
```
train_files = glob.glob('data/Train/Data*.csv')
test_files = glob.glob('data/Test/Data*.csv')
print(train_files[:5])
```

```
['data/Train\\Data_S02_Sess01.csv', 'data/Train\\Data_S02_Sess02.csv',
'data/Train\\Data_S02_Sess03.csv', 'data/Train\\Data_S02_Sess04.csv',
'data/Train\\Data_S02_Sess05.csv']
```

[3]:
```
# CONSTANTS
training_subjects = 16 #num of training subjects
num_of_fb = 340 #num of feedbacks / subject
freq = 200 #sampling rate
epoch_time = 0.5 #proposed epoching time in seconds
epoch = int(freq * epoch_time) #epoch in indices
pre_fb_time = 0.1
pre_fb = int(freq * pre_fb_time) #sampling time before the feedback
num_of_cols = 59
eeg_cols = 6
b_s = int(-0.4*freq) #index where baseline starts relative to feedback (-400ms)
b_e = int(-0.3*freq) #index where baseline ends relative to feedback (-300ms)
order = 2 #butterworth order
low_pass = 0.1 #low frequency pass for butterworth filter
high_pass = 20. #high frequency pass for butterworth filter

all_channels = ['Fp1', 'Fp2', 'AF7', 'AF3', 'AF4', 'AF8', 'F7', 'F5', 'F3',␣
↪'F1',
                'Fz', 'F2', 'F4', 'F6', 'F8', 'FT7', 'FC5', 'FC3', 'FC1', 'FCz',
                'FC2', 'FC4', 'FC6', 'FT8', 'T7', 'C5', 'C3', 'C1', 'Cz', 'C2',
                'C4', 'C6', 'T8', 'TP7', 'CP5', 'CP3', 'CP1', 'CPz', 'CP2',␣
↪'CP4',
                'CP6', 'TP8', 'P7', 'P5', 'P3', 'P1', 'Pz', 'P2', 'P4', 'P6',␣
↪'P8',
                'PO7', 'POz', 'PO8', 'O1', 'O2']
channels = ['Fz', 'FCz', 'Cz', 'CPz', 'Pz', 'POz']
# [10, 19, 28, 37, 46, 52]
channels_indices = []
for channel in channels:
    channels_indices.append(all_channels.index(channel))
```

[4]:
```
def butter_filter(order, low_pass, high_pass, fs,sig):
    nyq = 0.5 * fs
#     lp = low_pass / nyq
#     hp = high_pass / nyq
    lp = low_pass
    hp = high_pass
    coeffs = [lp, hp]
    btype = 'bandpass'
#     sos = signal.butter(order, [lp, hp], btype=btype, output = 'sos', fs =␣
 ↪200)
#     return signal.sosfiltfilt(sos, sig, axis=0)
    b = signal.firwin(numtaps=151, cutoff=coeffs, pass_zero=btype, fs=200) #␣
 ↪order = odd number (31 - 201)
    a = 1
    filt = signal.filtfilt(b, a, sig, axis=0)
    return filt



def extract_data(files, e_s = None, baseline = True, bandpass = True):
    start = time.time()

    temp = np.empty((1,epoch+pre_fb, len(channels)), float)
    enum_files = list(enumerate(files))
    for i, f in tqdm(enum_files):
        df = pd.read_csv(f) #read each file
        index_fb = df[df['FeedBackEvent'] == 1].index.values
        df_array = np.array(df)

        #uncomment below for butterworth filter or firwin filter
        if bandpass == True:
            eeg = df_array[:, channels_indices] #only eeg values to apply␣
 ↪butterworth filter
            for i, channel in enumerate(channels):
                raw_eeg = df[channel].values
                eeg_filtered = butter_filter(order, low_pass, high_pass, freq,␣
 ↪raw_eeg) #butterworth filter applied
                eeg[:,i] = eeg_filtered
            df = np.array(df)
            df[:, channels_indices] = eeg #replacing old eeg values with new␣
 ↪ones
        else:
            df = np.array(df)
```

```
        for j, indx in enumerate(index_fb): #epoching 100 indexes (0.5 seconds)␣
↪after each stimulus
            epoch_array = df[indx-pre_fb:indx+epoch, channels_indices]
            epoch_array = epoch_array.reshape((1, epoch_array.shape[0],␣
↪epoch_array.shape[1]))

            #uncomment below for baseline correction
            if baseline == True:
                #baseline correction of 100ms (20 indexes), 400ms to 300ms␣
↪before fb
                baseline_array = df[indx+b_s:indx+b_e, channels_indices]
                baseline_array = baseline_array.
↪reshape((1,20,int(baseline_array.shape[1])))
                baseline_mean = np.mean(baseline_array, axis = 1)
                #noise subtracted from epoched data
                epoch_array[:,:,:] = epoch_array[:,:,:] - baseline_mean

            temp = np.vstack((temp,epoch_array))

    now = time.time()
    print('Elapsed Time: ' + str(int(now-start)) + ' seconds')
    return temp
```

[5]:
```
train = extract_data(train_files) # 80 iterations
print(train.shape)
test = extract_data(test_files) # 50 iterations
print(test.shape)
np.save('tr1.npy',train[1:,:,:])
np.save('te1.npy',test[1:,:,:])

train = np.load('tr1.npy')
test = np.load('te1.npy')
```

```
  0%|          | 0/80 [00:00<?, ?it/s]


Elapsed Time: 129 seconds
(5441, 120, 6)
  0%|          | 0/50 [00:00<?, ?it/s]


Elapsed Time: 68 seconds
(3401, 120, 6)
```

[6]:
```
train_shape = train.shape[0]
test_shape = test.shape[0]
```

```
train = np.reshape(train, (train_shape, eeg_cols, epoch + pre_fb))
test = np.reshape(test, (test_shape, eeg_cols, epoch + pre_fb))

EEG_train = train[:,:,:].reshape(5440*(epoch + pre_fb), eeg_cols)
EEG_test = test[:,:,:].reshape(3400*(epoch + pre_fb), eeg_cols)

train_filtered = EEG_train.reshape(5440, int(eeg_cols), epoch + pre_fb)
test_filtered = EEG_test.reshape(3400, int(eeg_cols), epoch + pre_fb)

print(train_filtered.shape)
print(test_filtered.shape)
```

```
(5440, 6, 120)
(3400, 6, 120)
```

[7]:
```
train_filtered[0].shape
```

[7]:
```
(6, 120)
```

[8]:
```
np.save('data/X_train_bwbs.npy',train_filtered)
np.save('data/X_test_bwbs.npy',test_filtered)
```

1.6 Generate Graphs

For graphs, we have chosen to draw multiple graphs from different versions of extracted data: data without filtering and baseline corrections, data with filtering but without baseline corrections, and data with both filtering and baseline corrections. We also split the extracted data by correct and incorrect data per electrode to visualize the signal differences in amplitude so that we are able to make an appropriate hypothesis of which subset of electrodes would be ideal for the classifier.

[9]:
```
import matplotlib.pyplot as plt
```

[10]:
```
train_files = glob.glob('data/train/Data*.csv')
test_files = glob.glob('data/test/Data*.csv')
print(train_files[:5])
```

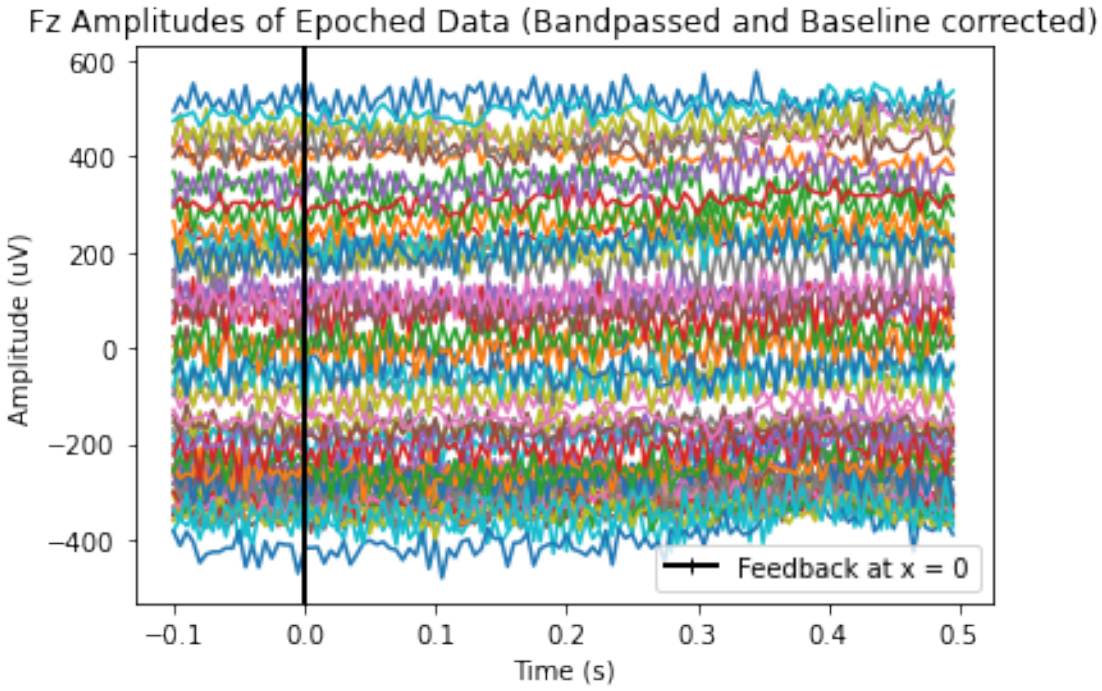
```
['data/train\\Data_S02_Sess01.csv', 'data/train\\Data_S02_Sess02.csv',
'data/train\\Data_S02_Sess03.csv', 'data/train\\Data_S02_Sess04.csv',
'data/train\\Data_S02_Sess05.csv']
```

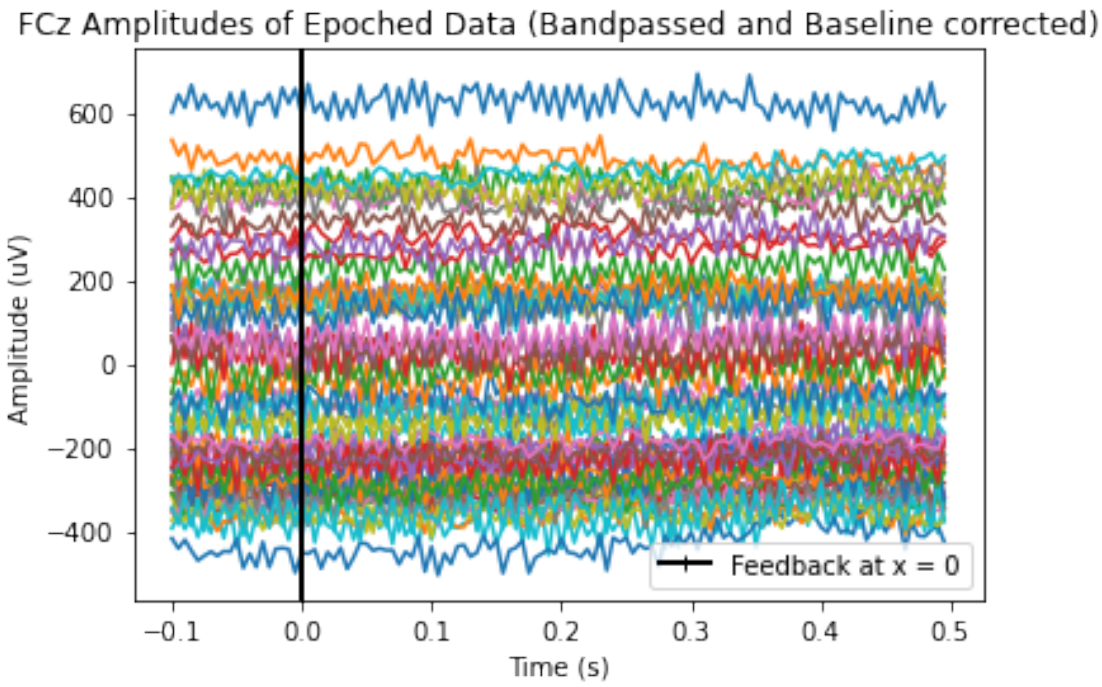
[11]:
```
#Not baseline corrected no filter
x = np.arange(-0.1,0.5,0.005) # length = 120
train = extract_data(train_files[:1], -0.4, False, False)
train = train[1:,:,:]

for i, c in enumerate(channels):
```

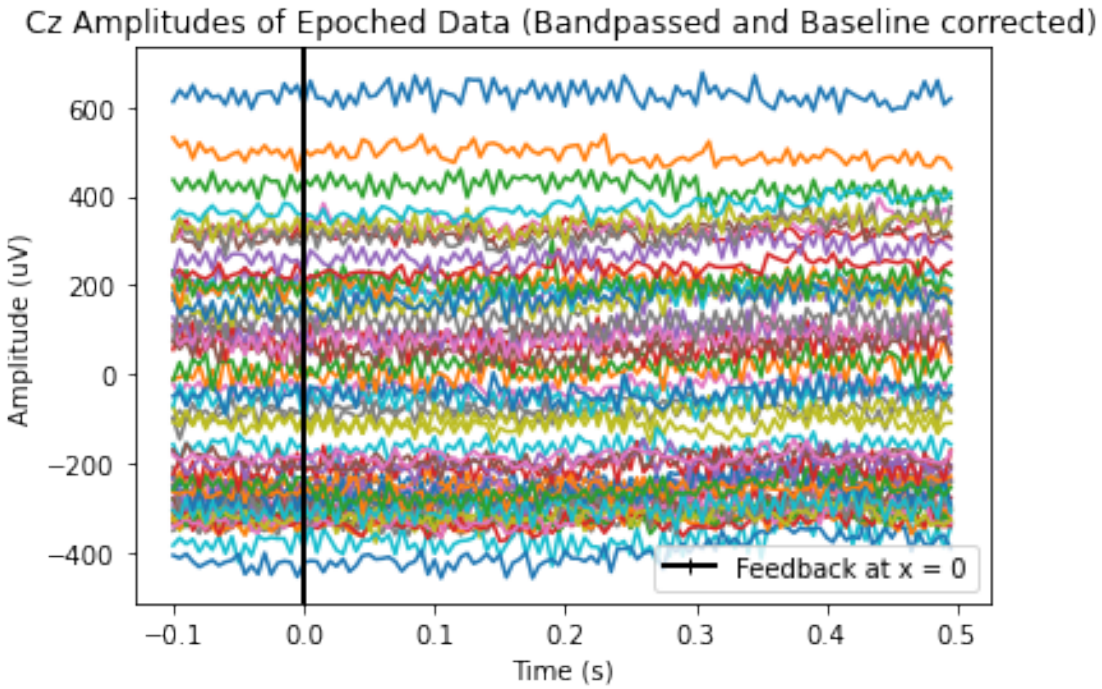
```
    first_electrode = train[:,:120,i]

    for j in np.arange(60):
        y = first_electrode[j,:]
        plt.plot(x, y)

    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude (uV)')
    plt.title('{} Amplitudes of Epoched Data (Bandpassed and Baseline␣
↪corrected)'.format(c))
    plt.axvline(x=0, marker = '|', linewidth = 2, label = 'Feedback at x = 0',␣
↪color = 'black')
    plt.legend()
    plt.show()
```

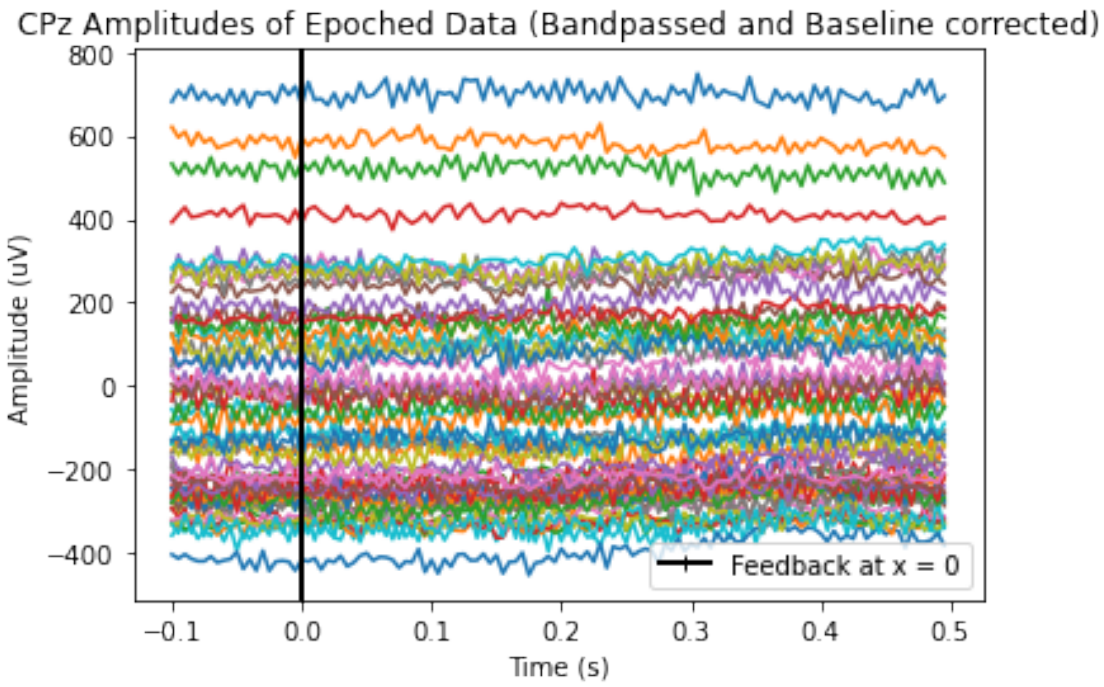
```
  0%|          | 0/1 [00:00<?, ?it/s]
```

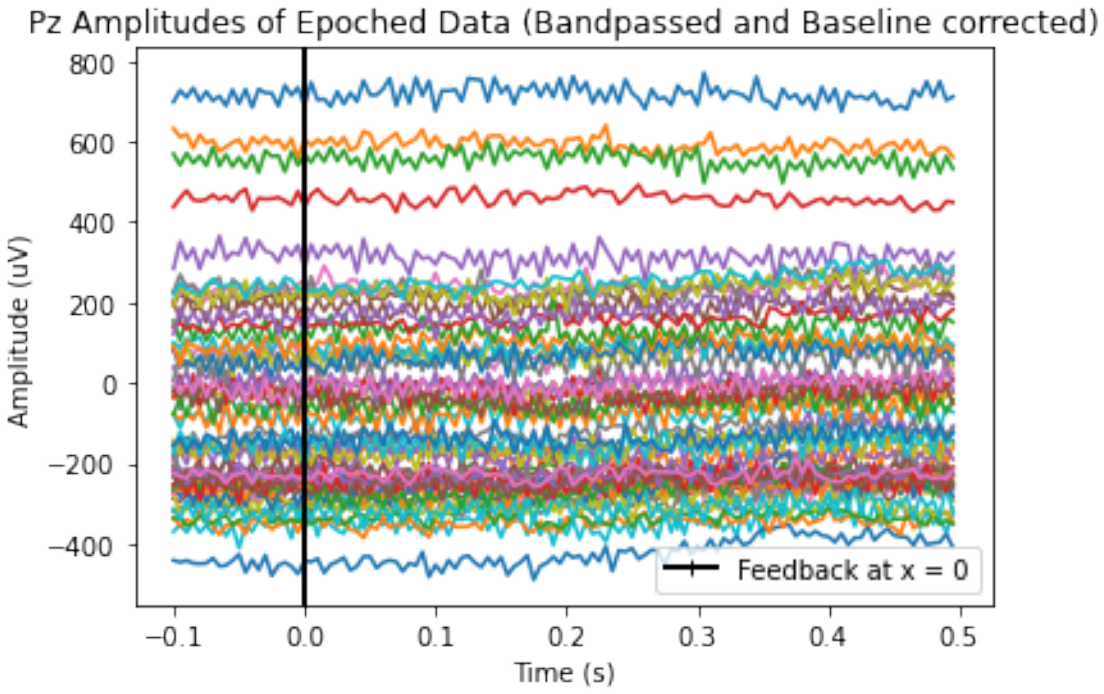
```
Elapsed Time: 0 seconds
```

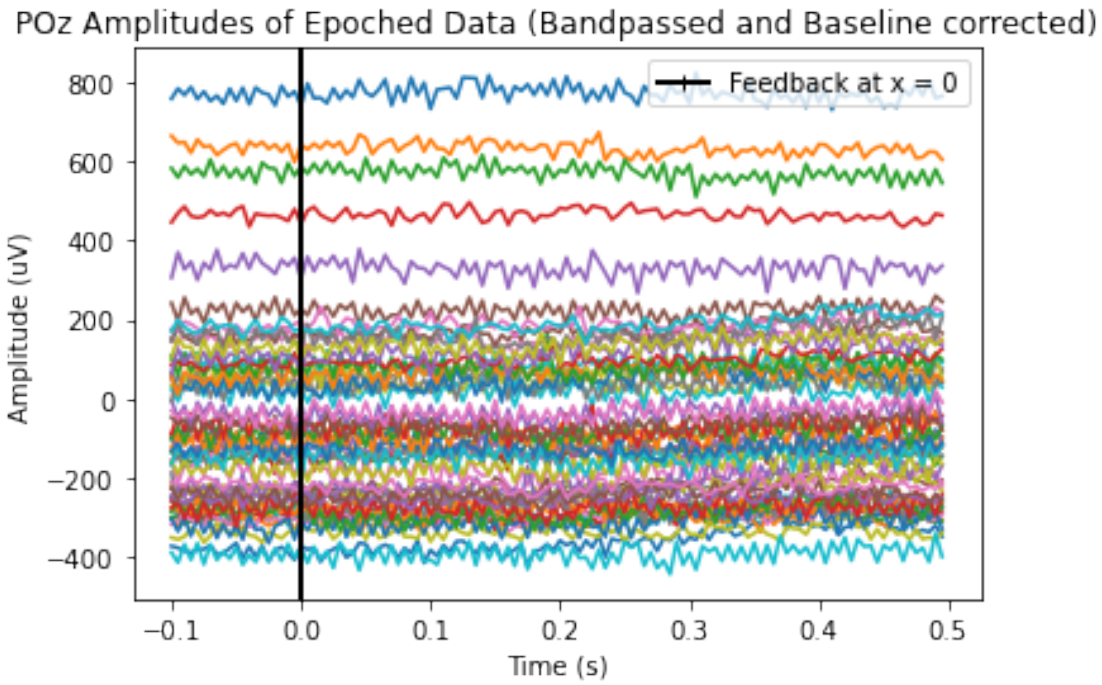



FCz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
600
400
200
0
−200
−400
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)
Feedback at x = 0


Cz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
600
400
200
0
−200
−400
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)
Feedback at x = 0

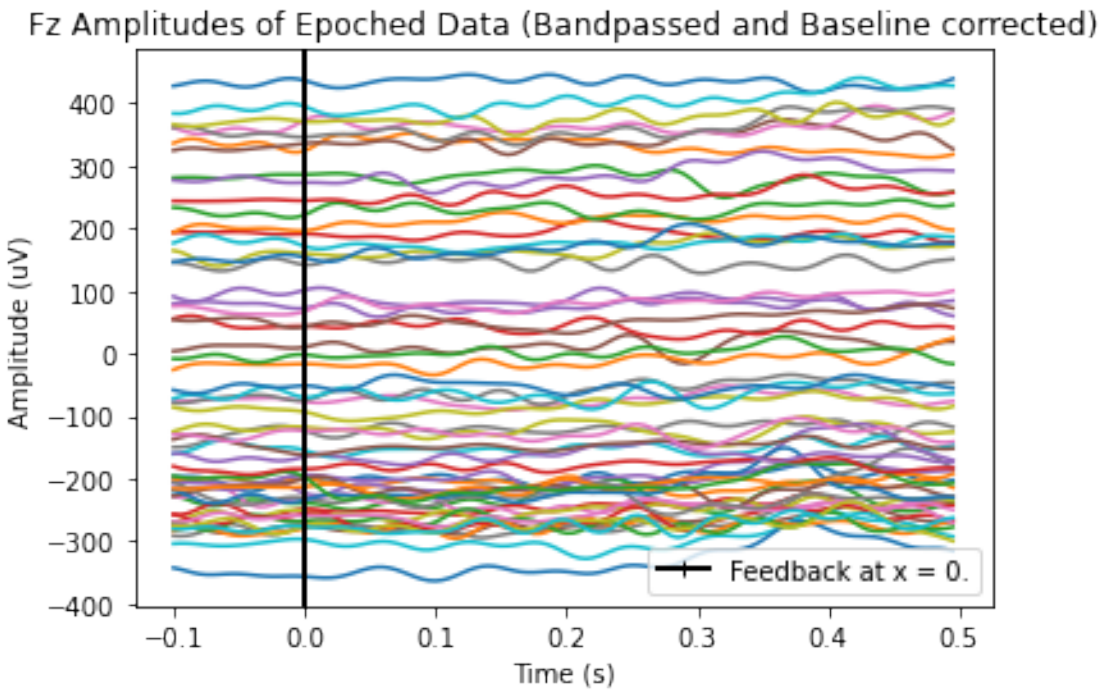

CPz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
800
600
400
200
0
−200
−400
Amplitude (uV)
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)
Feedback at x = 0

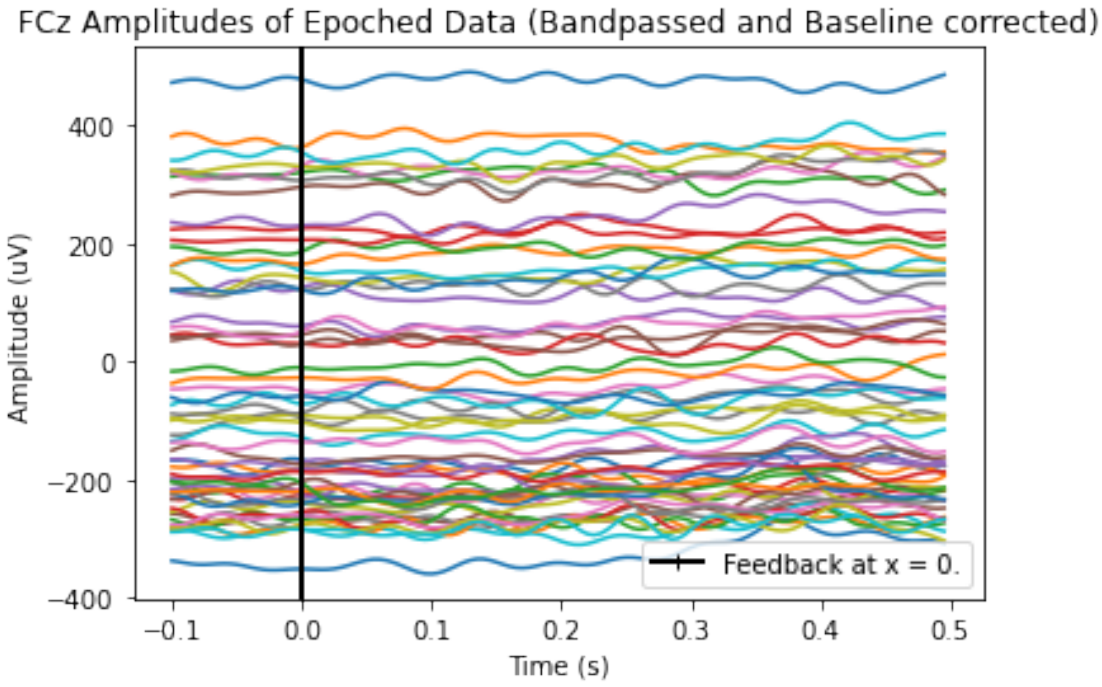

Pz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
800
600
400
200
0
−200
−400
Amplitude (uV)
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)
Feedback at x = 0

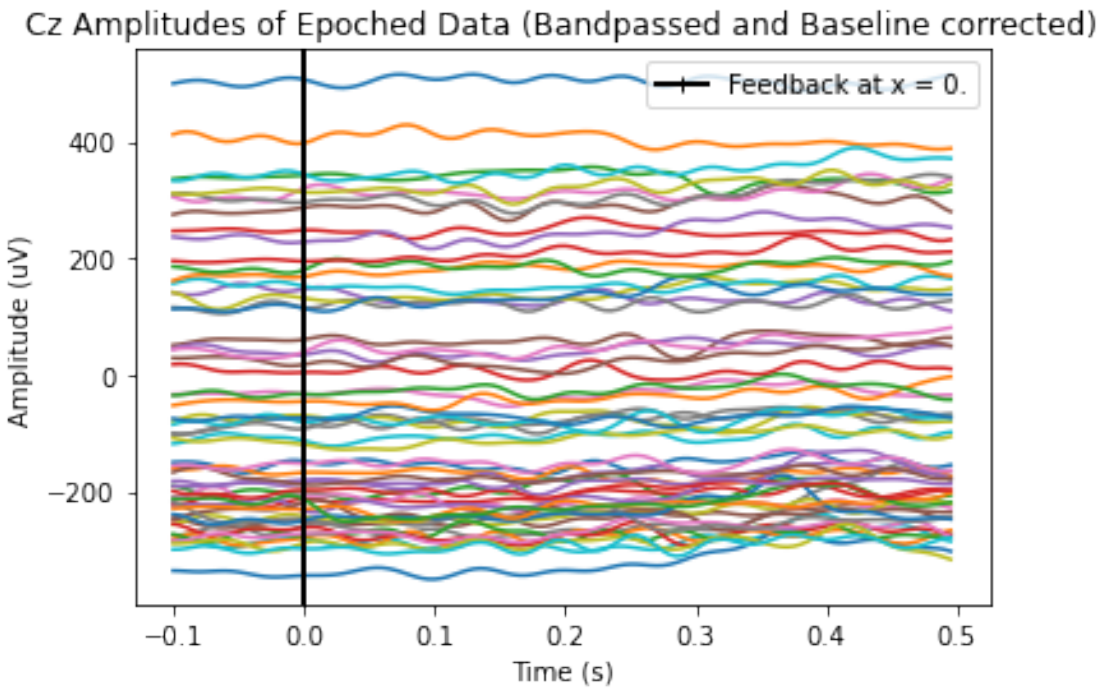


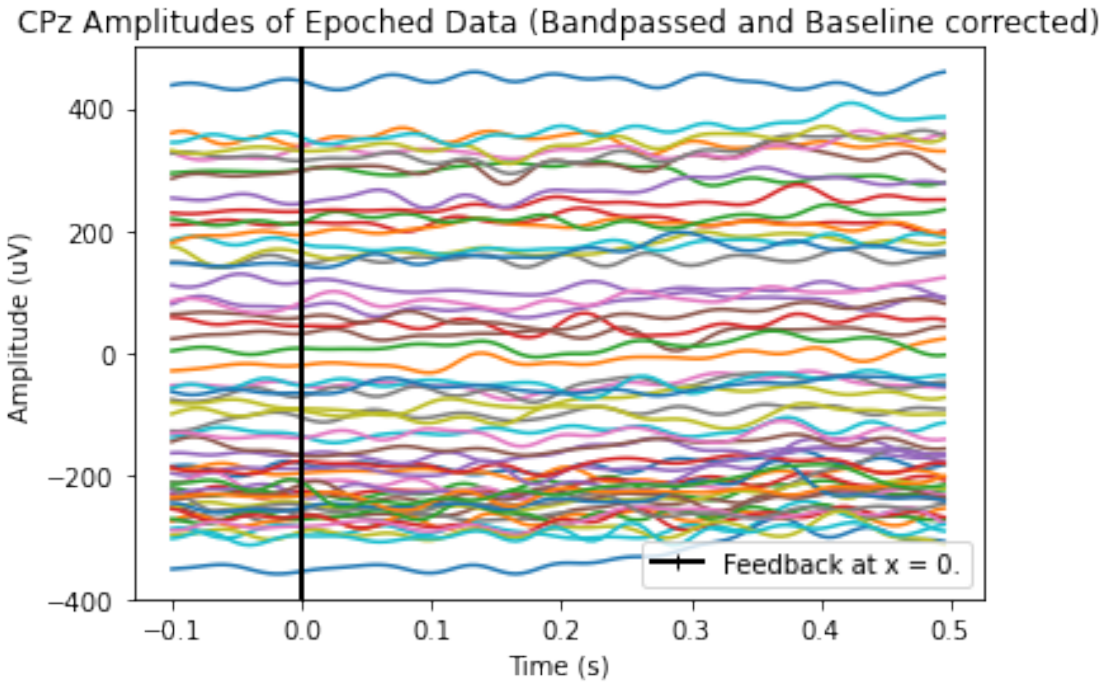
[12]:
```
#Bandpassed, baseline not corrected
x = np.arange(-0.1,0.5,0.005)
train = extract_data(train_files[:1], -0.4, baseline = False, bandpass = True)
train = train[1:,:,:]

for i, c in enumerate(channels):
    first_electrode = train[:,:120,i]

    for j in np.arange(60):
        y = first_electrode[j,:]
        plt.plot(x, y)

    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude (uV)')
    plt.title('{} Amplitudes of Epoched Data (Bandpassed and Baseline␣
↪corrected)'.format(c))
    plt.axvline(x=0, marker = '|', linewidth = 2, label = 'Feedback at x = 0.',␣
↪color = 'black')
    plt.legend()
    plt.show()
```

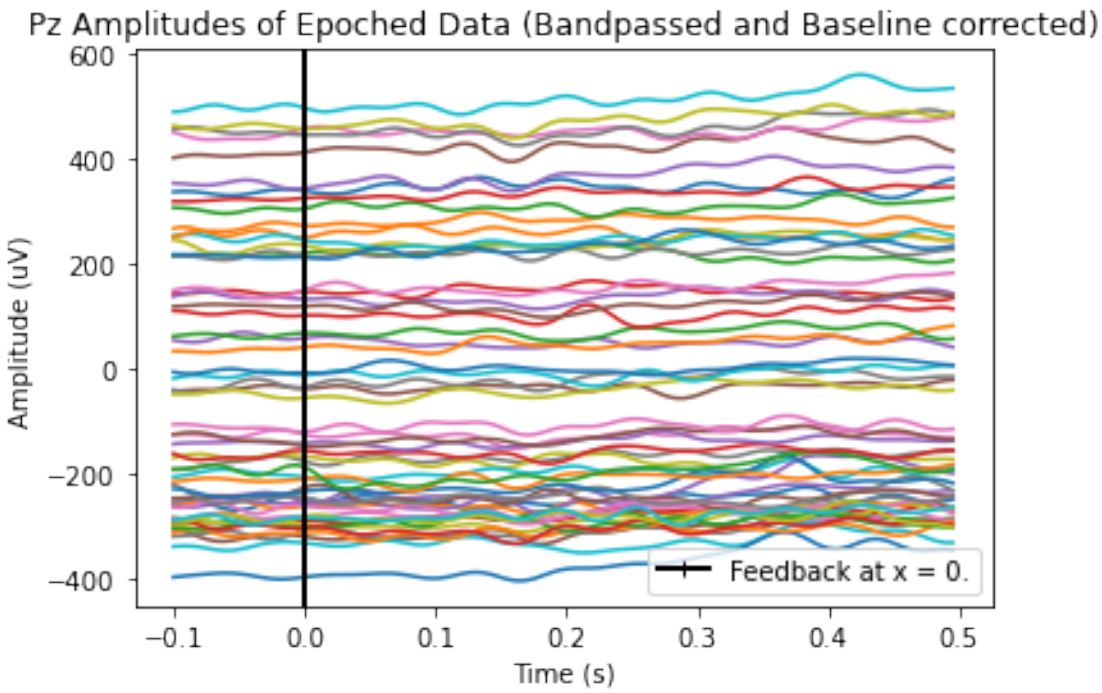
```
  0%|          | 0/1 [00:00<?, ?it/s]

Elapsed Time: 1 seconds
```

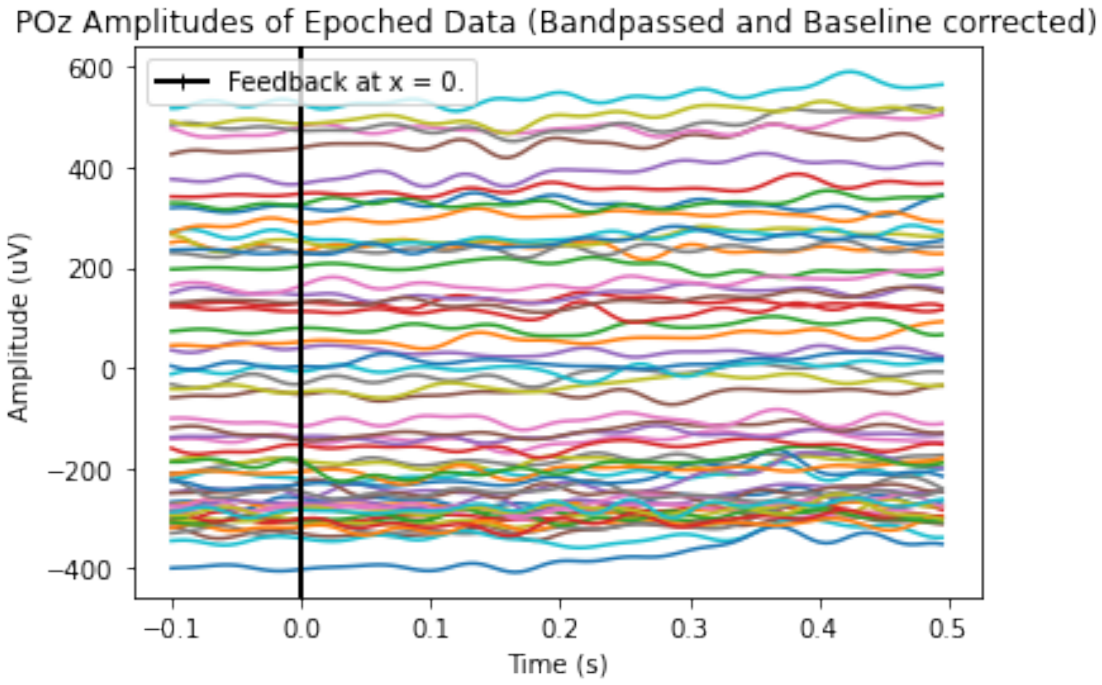

Fz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
Time (s)
Feedback at x = 0.


FCz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
Time (s)
Feedback at x = 0.


Cz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Feedback at x = 0.
400
200
0
−200
Amplitude (uV)
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)

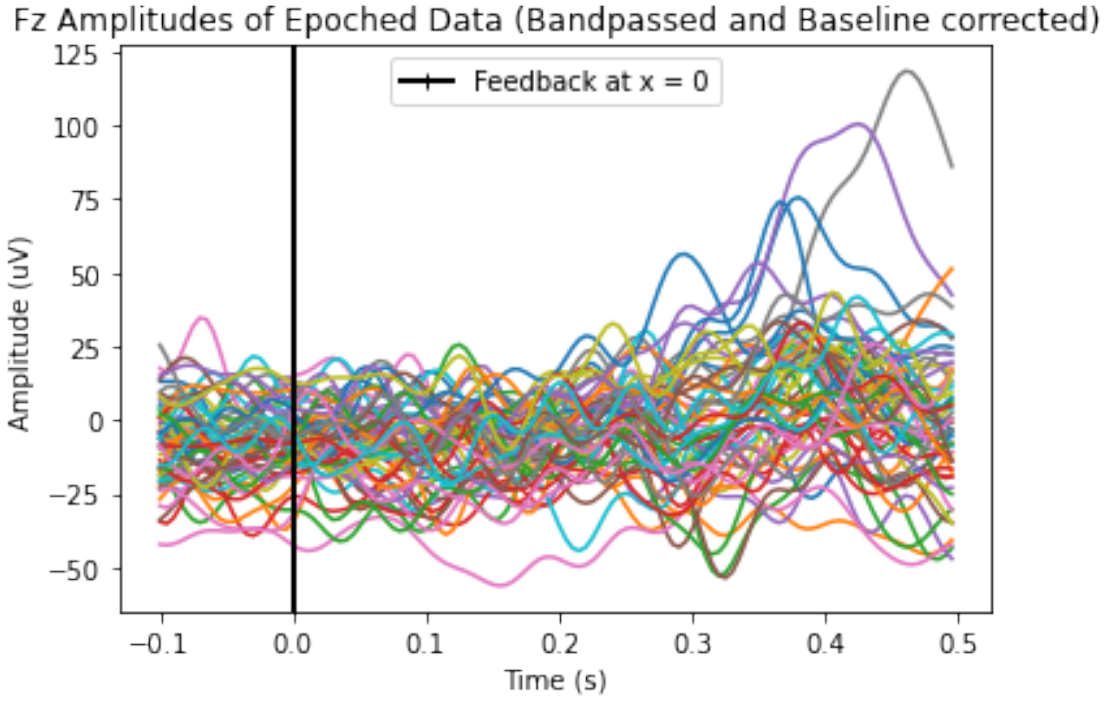

CPz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
400
200
0
−200
−400
Amplitude (uV)
Feedback at x = 0.
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)

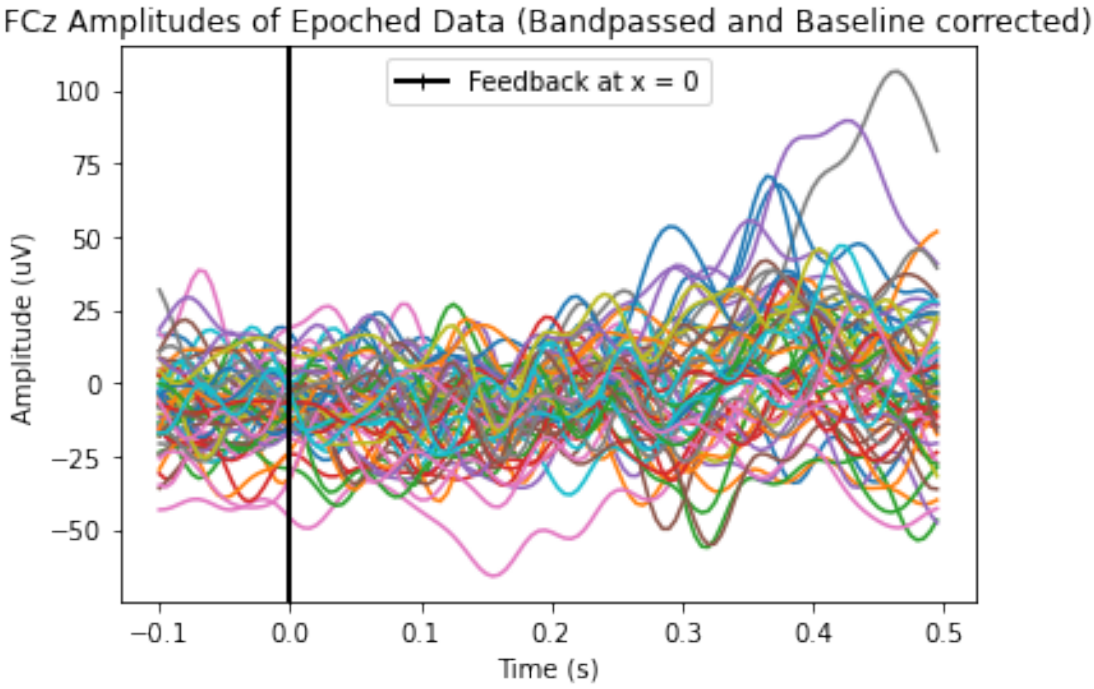

Pz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
600
400
200
0
−200
−400
Amplitude (uV)
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)
Feedback at x = 0.

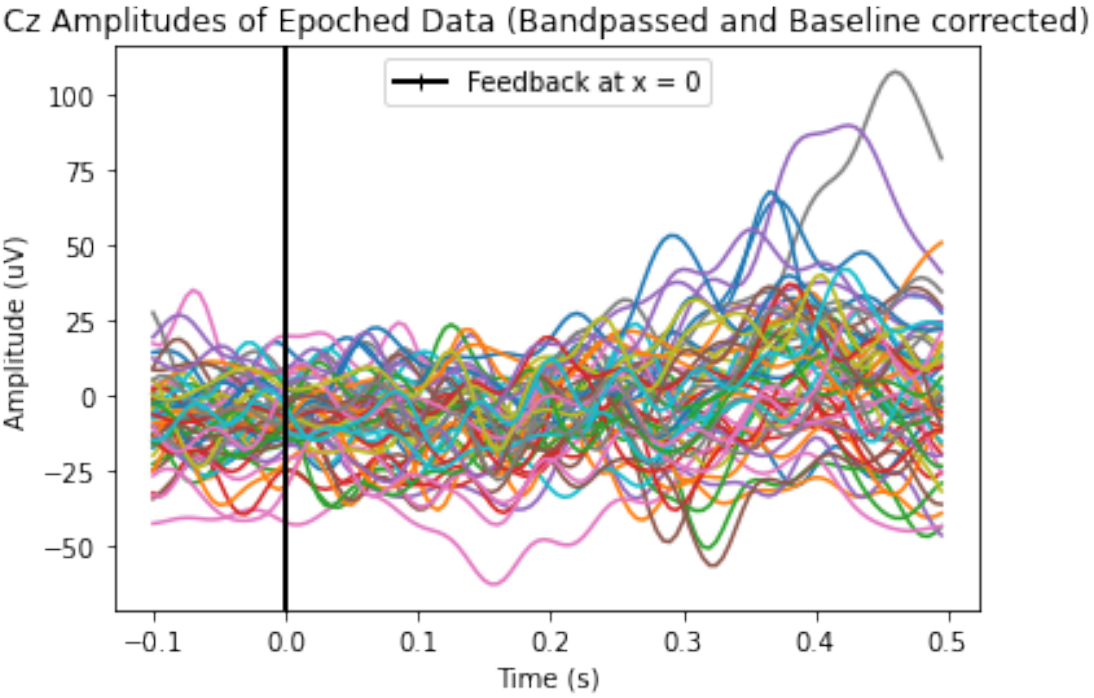

POz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
600
400
200
0
−200
−400
Amplitude (uV)
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)
Feedback at x = 0.

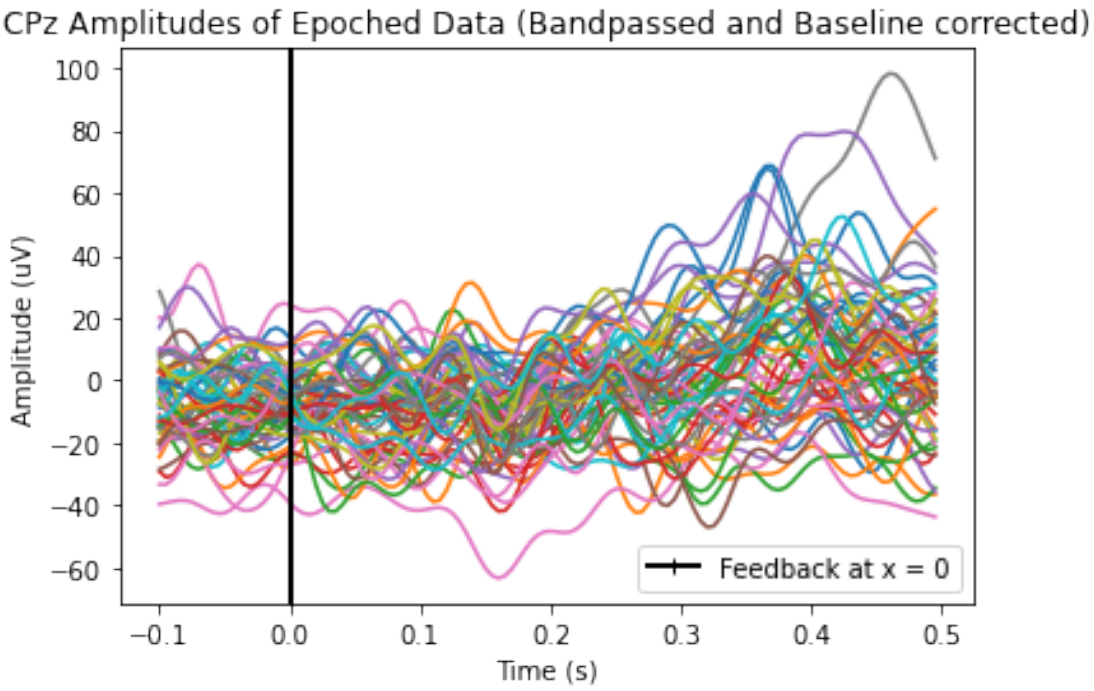
[13]:
```
#Bandpassed, and baseline corrected
x = np.arange(-0.1,0.5,0.005)
train = extract_data(train_files[:1], -0.4, baseline = True, bandpass = True)
train = train[1:,:,:]
print(train.shape)

for i, c in enumerate(channels):
    first_electrode = train[:,:120,i]

    for j in np.arange(60):
        y = first_electrode[j,:]
        plt.plot(x, y)

    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude (uV)')
    plt.title('{} Amplitudes of Epoched Data (Bandpassed and Baseline␣
↪corrected)'.format(c))
    plt.axvline(x=0, marker = '|', linewidth = 2, label = 'Feedback at x = 0',␣
↪color = 'black')
    plt.legend()
    plt.show()
```

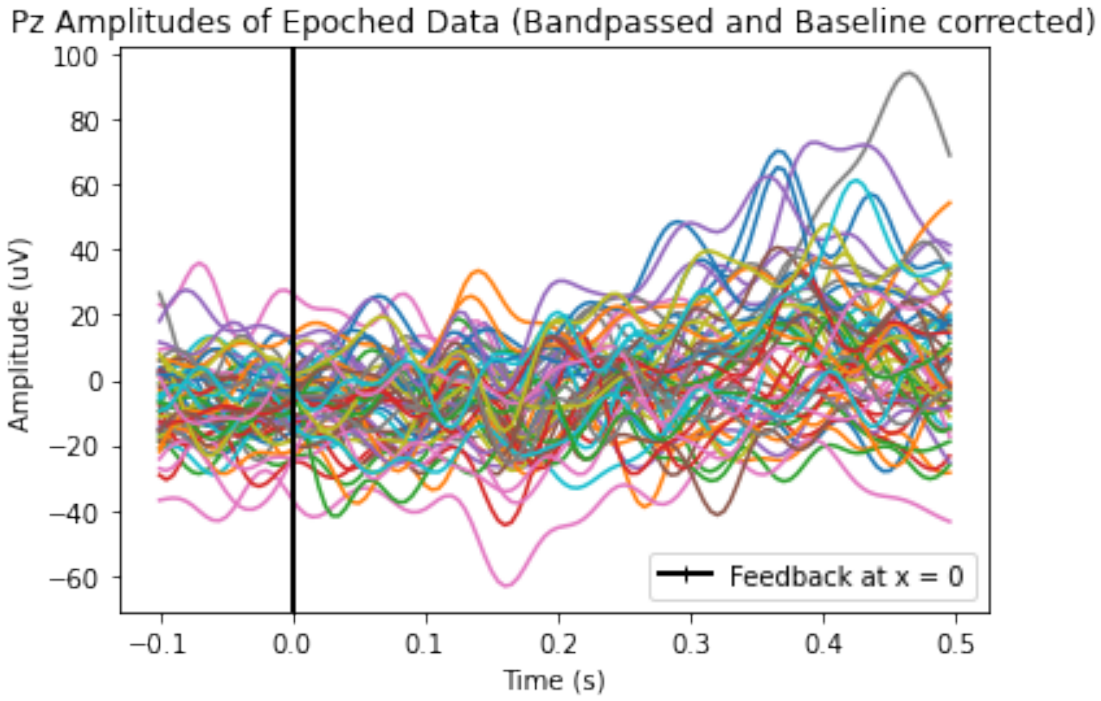
```
  0%|          | 0/1 [00:00<?, ?it/s]

Elapsed Time: 0 seconds
(60, 120, 6)
```

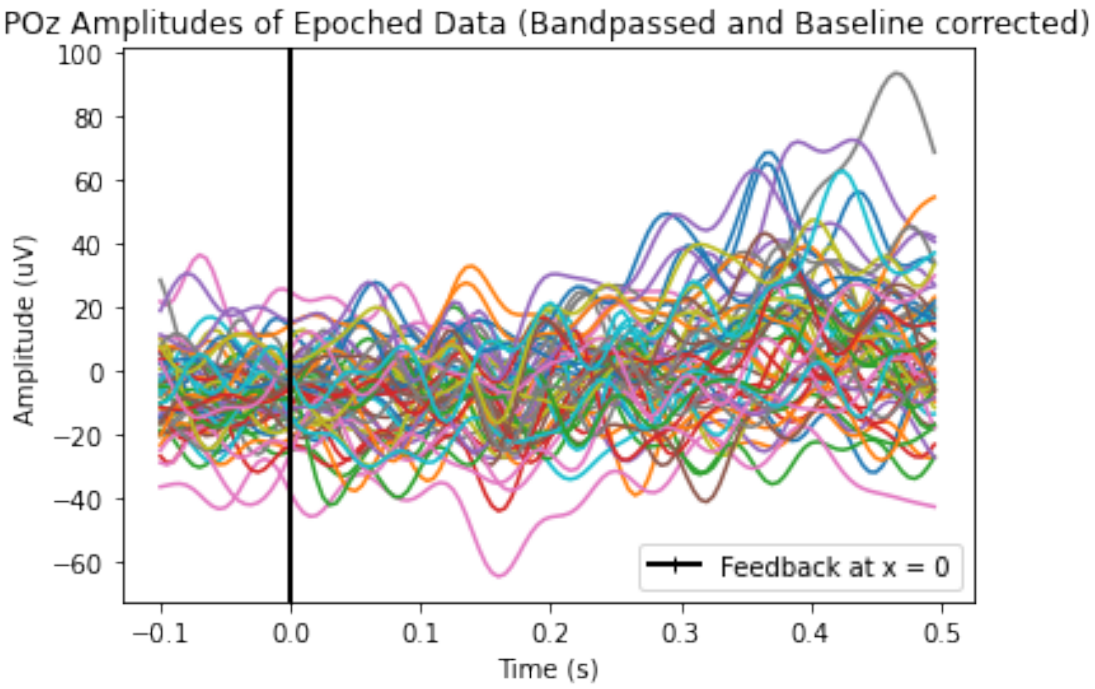



FCz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Feedback at x = 0
100
75
50
25
0
−25
−50
Amplitude (uV)
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)


Cz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Feedback at x = 0
100
75
50
25
0
−25
−50
Amplitude (uV)
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)

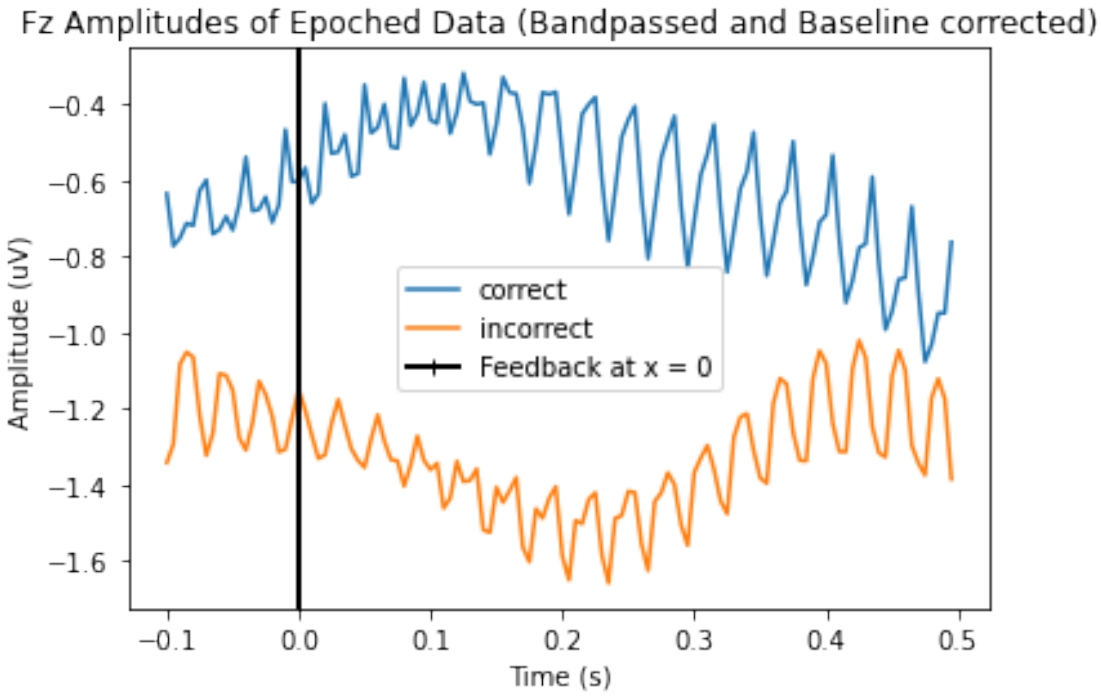

CPz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
Time (s)
Feedback at x = 0

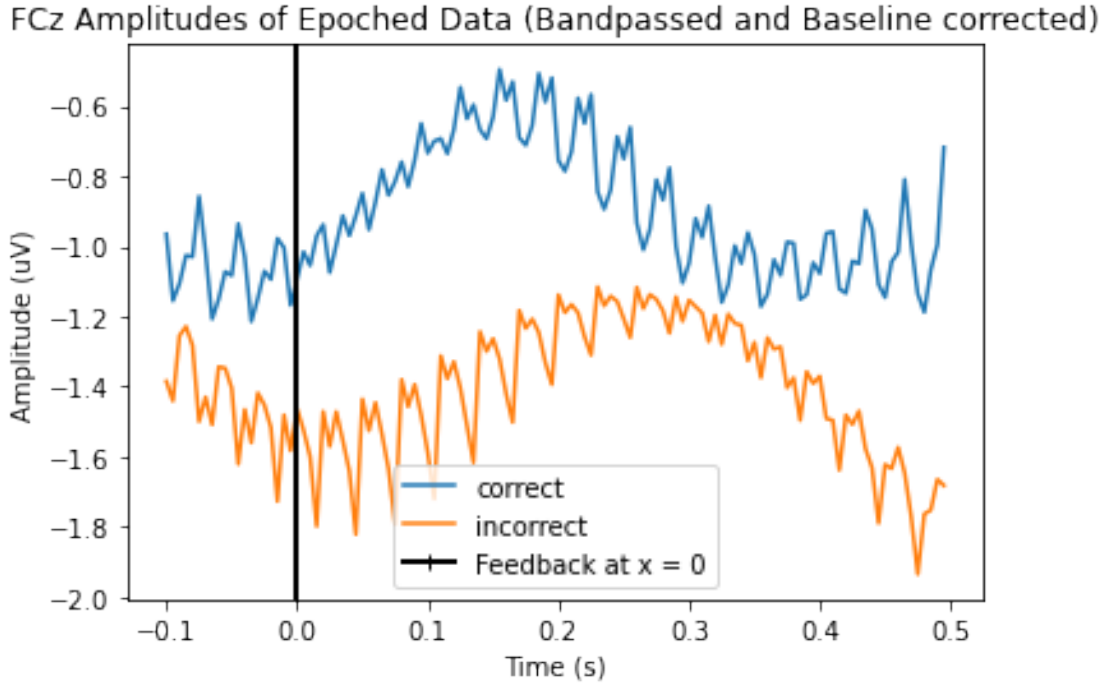

Pz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
Time (s)
Feedback at x = 0

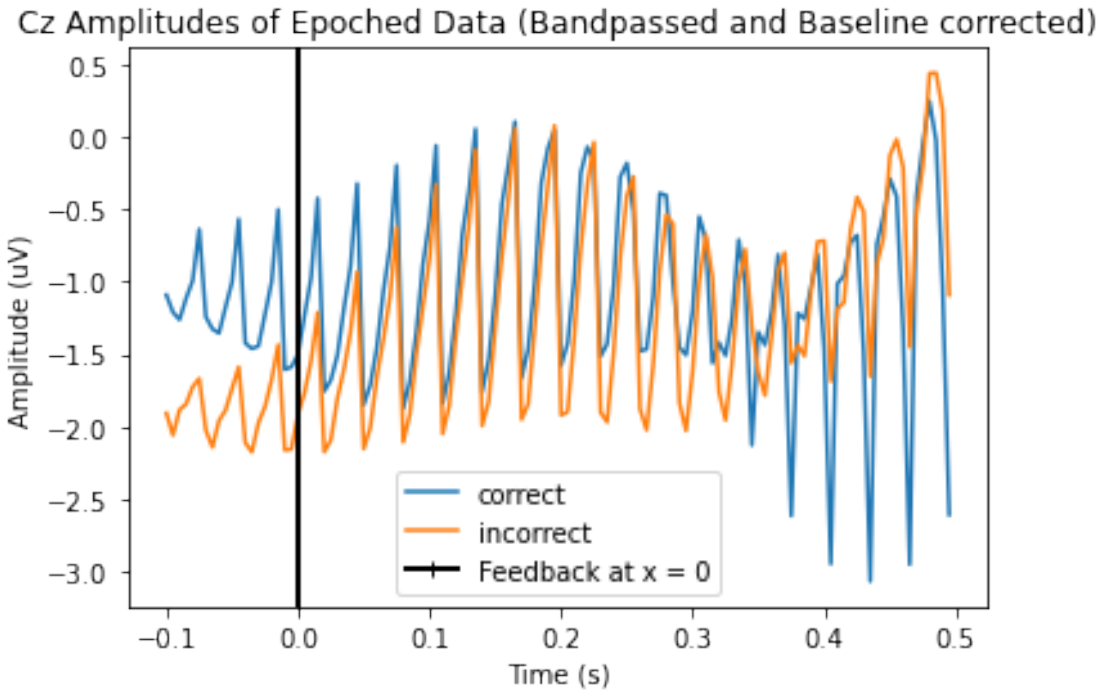


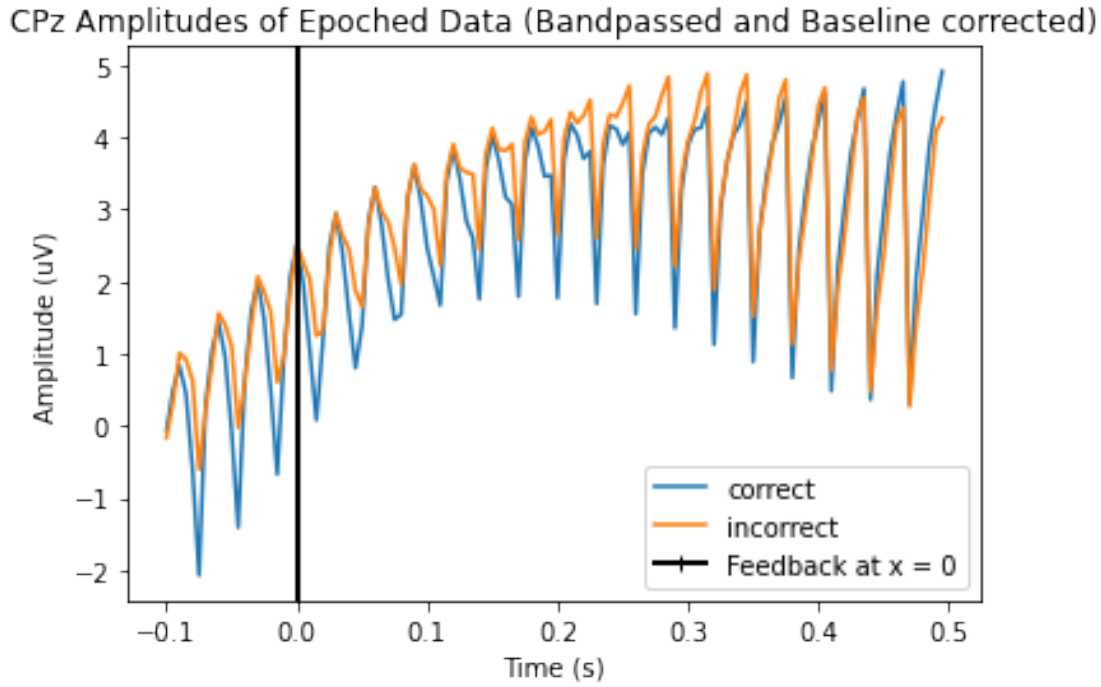
[14]:
```
# 1. Split data by correct / incorrect data

f = 'data/TrainLabels.csv'
train_labels = pd.read_csv(f)
train_labels = train_labels.Prediction.values
# print(train_labels.shape)

train_filtered = np.load('data/X_train_bwbs.npy')
# print(train_filtered.shape)

X_train_correct, X_train_incorrect = [], []
for i in range(len(train_labels)):
    label = train_labels[i]
    if label:
        X_train_correct.append(train_filtered[i])
    else:
        X_train_incorrect.append(train_filtered[i])

X_train_correct = np.array(X_train_correct)
X_train_incorrect = np.array(X_train_incorrect)

np.save('data/X_train_correct.npy', X_train_correct)
np.save('data/X_train_incorrect.npy', X_train_incorrect)
```

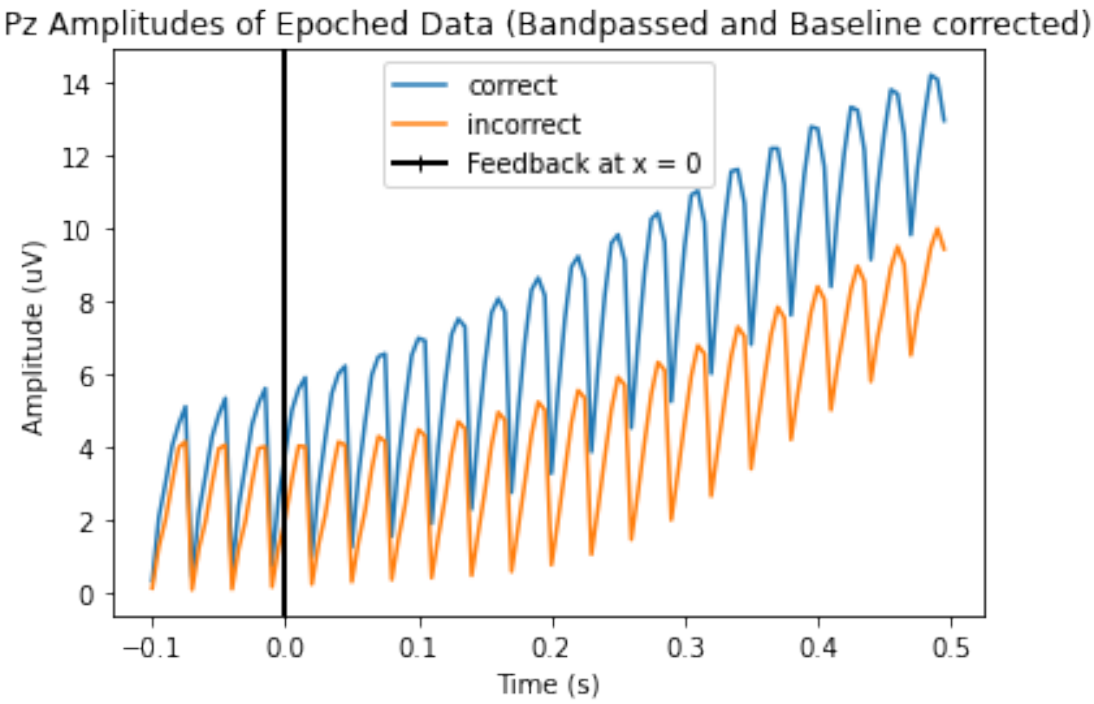
[15]:
```
# 2. Take average for graph instead of displaying all lines

#Not baseline corrected no filter
x = np.arange(-0.1,0.5,0.005) # length = 120
X_train_correct = np.load('data/X_train_correct.npy')
X_train_incorrect = np.load('data/X_train_incorrect.npy')
# print(X_train_correct.shape)

for i, c in enumerate(channels):
    correct, correct_mean = X_train_correct[:,i,:], []
    incorrect, incorrect_mean = X_train_incorrect[:,i,:], []

    for j in range(len(x)):
        correct_mean.append(sum(correct[:, j]) / len(correct))
        incorrect_mean.append(sum(incorrect[:, j]) / len(incorrect))

    plt.plot(x, correct_mean, label='correct')
    plt.plot(x, incorrect_mean, label='incorrect')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude (uV)')
    plt.title('{} Amplitudes of Epoched Data (Bandpassed and Baseline␣
↪corrected)'.format(c))
    plt.axvline(x=0, marker = '|', linewidth = 2, label = 'Feedback at x = 0',␣
↪color = 'black')
    plt.legend()
    plt.show()
```

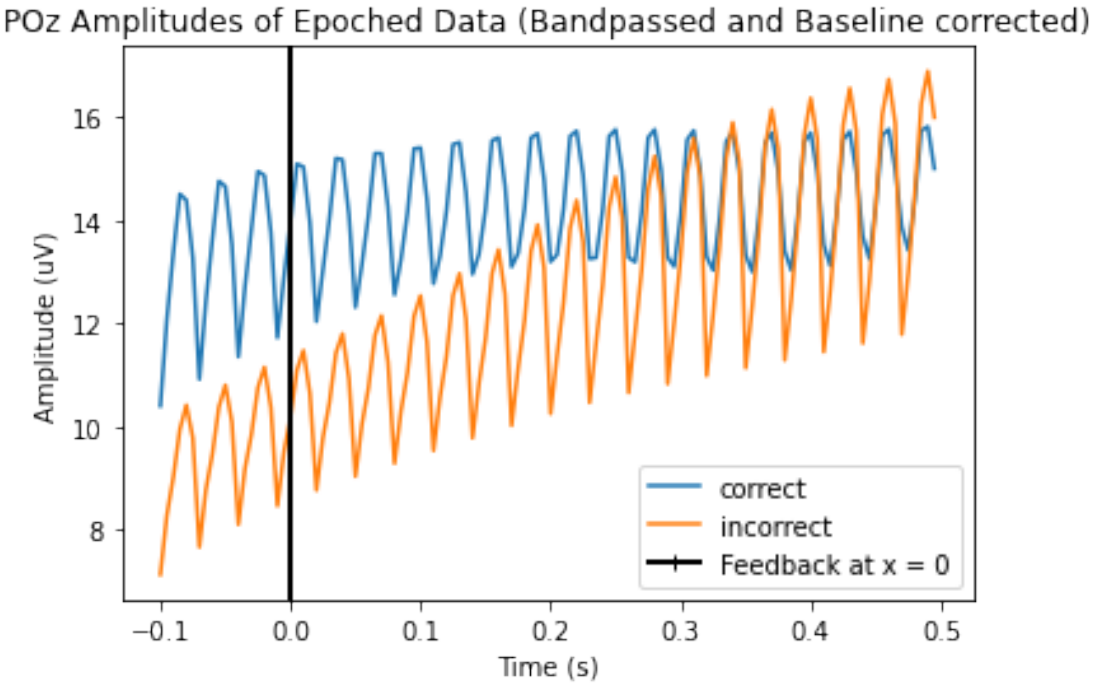

Fz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
−0.4
−0.6
−0.8
−1.0
−1.2
−1.4
−1.6
correct
incorrect
Feedback at x = 0
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)


FCz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
−0.6
−0.8
−1.0
−1.2
−1.4
−1.6
−1.8
−2.0
correct
incorrect
Feedback at x = 0
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Time (s)


Cz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
Time (s)
correct
incorrect
Feedback at x = 0


CPz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
Amplitude (uV)
Time (s)
correct
incorrect
Feedback at x = 0


Pz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
correct
incorrect
Feedback at x = 0
Amplitude (uV)
Time (s)
14
12
10
8
6
4
2
0
−0.1
0.0
0.1
0.2
0.3
0.4
0.5


POz Amplitudes of Epoched Data (Bandpassed and Baseline corrected)
correct
incorrect
Feedback at x = 0
Amplitude (uV)
Time (s)
16
14
12
10
8
−0.1
0.0
0.1
0.2
0.3
0.4
0.5

1.7 Models

1.8 SVC

For the models, we have initially chosen the Support-Vector Classifier (SVC) simply because it is widely known as one of the most basic and popular classifiers. We first randomly splitted the training data into train and test pools in a 4:1 ratio. Then, for each electrode, we have generated multiple pipelines supported using the SVC model with various C values to determine which C constant would be the most optimal for each electrode. We determine the "best C value" for each electrode by computing the Area Under the Receiver Operating Characteristic Curve (roc_auc_score) and figuring out which constant outputs the largest roc_auc_score. Finally, once we got the best C values for all 6 electrodes, we tested the SVC models on the testing pool for each electrode, and the accuracy results of predicting the labels were in a consensus of being slightly above 0.500. Because this meant that our SVC model was trumping a random guesser by an extremely small margin, we decided to search for a better model so that the accuracy was approximately 0.7.

[19]:
```
channels = ['Fz', 'FCz', 'Cz', 'CPz', 'Pz', 'POz']

train_filtered = np.load('data/X_train_bwbs.npy')

f = 'data/TrainLabels.csv'
train_labels = pd.read_csv(f)
train_labels = train_labels.Prediction.values

# Silence warning messages
import warnings
warnings.filterwarnings('ignore')
# warnings.filterwarnings(action='once')
```

[20]:
```
#Using SVM to find the optimal C in the dataset with baseline
# LDA SVM
C_list = [0.001,0.01,0.1,10]

opt_C = None
scaler = StandardScaler()
best_Cs = []

start = time.time()

for i in range(len(train_filtered[0])):
    x_train, x_test, y_train, y_test = train_test_split(train_filtered[:,i,:],␣
 ↪train_labels, test_size=0.2)
    auc_best=0
    for c in C_list:
#         estimator = SVC(kernel = 'linear', max_iter=10000, probability=True,␣
 ↪C = c)
```

```
#         parameter = {'kernel':('linear',), 'C':[c]}
#         grid_search = GridSearchCV(estimator, parameter, cv=5)
#         grid_search.fit(scaler.fit_transform(x_train), y_train)
#         pred_y = grid_search.predict(scaler.transform(x_test))
#         auc_score = roc_auc_score(y_test, pred_y)
        clf = make_pipeline(StandardScaler(), SVC(kernel = 'linear',␣
 ↪max_iter=10000, probability=True, C = c))
        clf.fit(x_train, y_train)
        pred_y = clf.predict(x_test)
        auc_score = roc_auc_score(y_test, pred_y)
#         print('AUC:'+"{0:.3f}".format(auc_score))
        if auc_score > auc_best:
            auc_best = auc_score
            opt_C = c
    best_Cs.append(opt_C)

print('Time taken: {}'.format(time.time() - start))
for i in range(len(channels)):
    print('Best parameter C* for {} = {}'.format(channels[i], best_Cs[i]))
```

```
Time taken: 118.48688769340515
Best parameter C* for Fz = 10
Best parameter C* for FCz = 10
Best parameter C* for Cz = 10
Best parameter C* for CPz = 10
Best parameter C* for Pz = 0.001
Best parameter C* for POz = 10
```

[21]:
```
c_values = best_Cs # from prev cell
start = time.time()

for i in range(len(c_values)):
    x_train, x_test, y_train, y_test = train_test_split(train_filtered[:,i,:],␣
 ↪train_labels, test_size=0.2)
    estimator = SVC(kernel = 'linear', max_iter=10000, probability=True)
    parameter = {'kernel':('linear',), 'C':[c_values[i]]}
    grid_search = GridSearchCV(estimator, parameter, cv=5)
    grid_search.fit(scaler.fit_transform(x_train), y_train)
    pred_y = grid_search.predict(scaler.transform(x_test))
    auc_score = roc_auc_score(y_test, pred_y)
    print('{} electrode score: {}'.format(channels[i], auc_score))

print()
for i in range(len(c_values)):
    x_train, x_test, y_train, y_test = train_test_split(train_filtered[:,i,:],␣
 ↪train_labels, test_size=0.2)
    pred_y = [random.randint(0, 1) for i in range(len(y_test))]
```

```
    auc_score = roc_auc_score(y_test, pred_y)
    print('{} electrode score: {}'.format(channels[i], auc_score))
```

```
Fz electrode score: 0.48846405423083794
FCz electrode score: 0.4958808189079858
Cz electrode score: 0.49725396497064
CPz electrode score: 0.5162611197803382
Pz electrode score: 0.5
POz electrode score: 0.513825285376175

Fz electrode score: 0.5141091192837635
FCz electrode score: 0.49842441330126874
Cz electrode score: 0.5324666098323388
CPz electrode score: 0.4857942349878501
Pz electrode score: 0.501171875
POz electrode score: 0.5272782694440012
```

1.9 LDA

The final model we chose was the Linear Discriminant Analysis model (LDA). We have chosen this model because we wanted to reduce the number of features before classification so that it could deliver better performance in terms of accurately predicting the labels. We have tried multiple versions of the LDA model by alternating the least square solver and the single-value decomposition (SVD) solver as well as leveraging the automated shrinkage mode. By going through the same processes from the SVC model, all three versions of the LDA model produced an accuracy of 0.7044.

[31]:
```
# Train our classifier
clfs = []
lda_types = ['lsqr (auto)', 'lsqr', 'svd']
ldas = {'lsqr (auto)': LinearDiscriminantAnalysis(solver = 'lsqr',  shrinkage =␣
↪'auto'),
        'lsqr': LinearDiscriminantAnalysis(solver = 'lsqr'),
        'svd': LinearDiscriminantAnalysis(solver = 'svd')}

for lda_type in lda_types:
    for i in range(len(c_values)):
        x_train, x_test, y_train, y_test = train_test_split(train_filtered[:,i,:
↪], train_labels, test_size=0.2)
        clf_lsqrs = ldas[lda_type].fit(x_train, y_train)
        clfs.append(clf_lsqrs)
        # Let's do 5-fold cross validation
        score_lsqrs = cross_val_score(clf_lsqrs.fit(x_train, y_train), x_train,␣
↪y_train, cv = 5)

        # We will print out the mean score
```

```
        print("solver = {}, channel: {}, accuracy: {}".format(lda_type,␣
↪channels[i], np.mean(score_lsqrs)))
    print()
```

```
solver = lsqr (auto), channel: Fz, accuracy: 0.7056526386634466
solver = lsqr (auto), channel: FCz, accuracy: 0.710937619594336
solver = lsqr (auto), channel: Cz, accuracy: 0.7042733283186191
solver = lsqr (auto), channel: CPz, accuracy: 0.7070314211436187
solver = lsqr (auto), channel: Pz, accuracy: 0.702664396848648
solver = lsqr (auto), channel: POz, accuracy: 0.699907623685287

solver = lsqr, channel: Fz, accuracy: 0.6985293690697706
solver = lsqr, channel: FCz, accuracy: 0.7031241669635906
solver = lsqr, channel: Cz, accuracy: 0.692095754648508
solver = lsqr, channel: CPz, accuracy: 0.7090985391345659
solver = lsqr, channel: Pz, accuracy: 0.7102527152038218
solver = lsqr, channel: POz, accuracy: 0.7127782836480726

solver = svd, channel: Fz, accuracy: 0.7040439711258034
solver = svd, channel: FCz, accuracy: 0.7033543159533896
solver = svd, channel: Cz, accuracy: 0.7022059463953443
solver = svd, channel: CPz, accuracy: 0.7038146139329875
solver = svd, channel: Pz, accuracy: 0.7051910210222099
solver = svd, channel: POz, accuracy: 0.7077218681130159
```

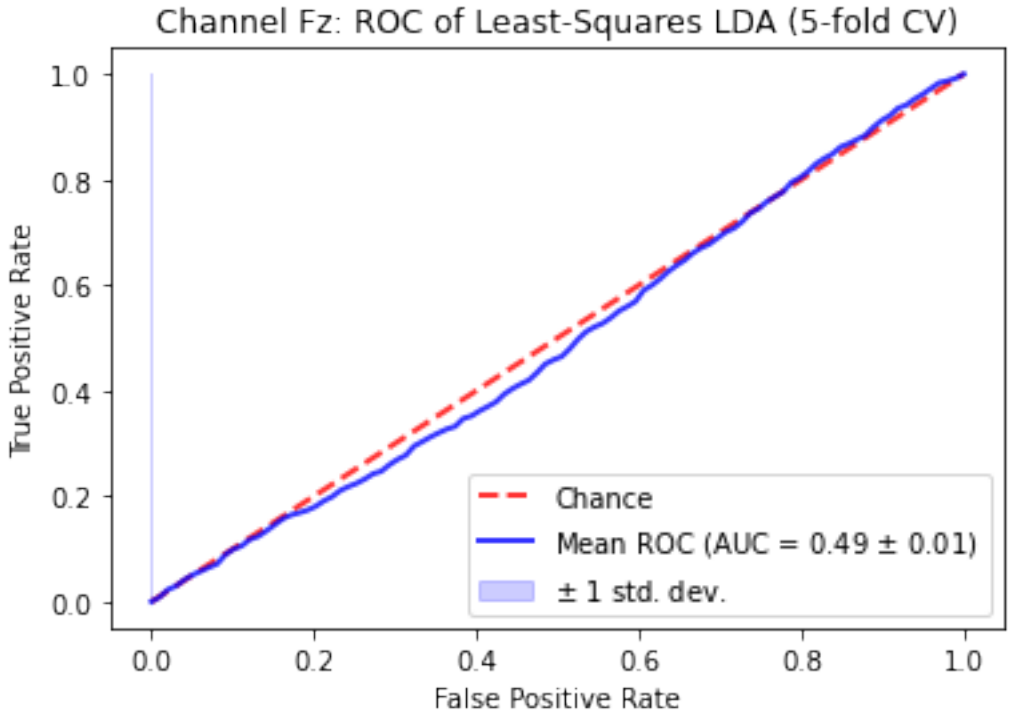
[23]:
```
# Test using classifier
lda_types = ['lsqr (auto)', 'lsqr', 'svd']
ldas = {'lsqr (auto)': LinearDiscriminantAnalysis(solver = 'lsqr',  shrinkage =␣
↪'auto'),
        'lsqr': LinearDiscriminantAnalysis(solver = 'lsqr'),
        'svd': LinearDiscriminantAnalysis(solver = 'svd')}

for lda_type in lda_types:
    average_score = 0
    for i in range(len(c_values)):
        x_train, x_test, y_train, y_test = train_test_split(train_filtered[:,i,:
↪], train_labels, test_size=0.2)
        clf_lsqrs = ldas[lda_type].fit(x_train, y_train)
        # Let's do 5-fold cross validation
        score_lsqrs = cross_val_score(clf_lsqrs.fit(x_train, y_train), x_test,␣
↪y_test, cv = 5)
        average_score += score_lsqrs
    average_score /= len(c_values)

    # We will print out the mean score
    print("solver = {} accuracy: {}".format(lda_type, np.mean(score_lsqrs)))
```

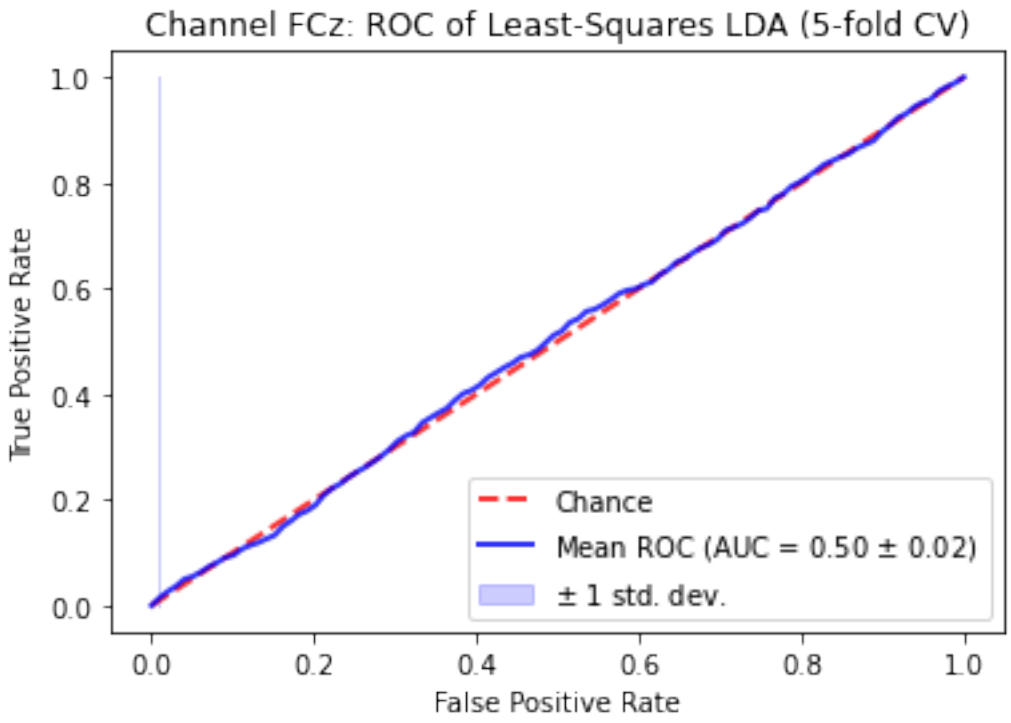
```
solver = lsqr (auto) accuracy: 0.7132541326681604
solver = lsqr accuracy: 0.708705026846489
solver = svd accuracy: 0.6994419312560775
```

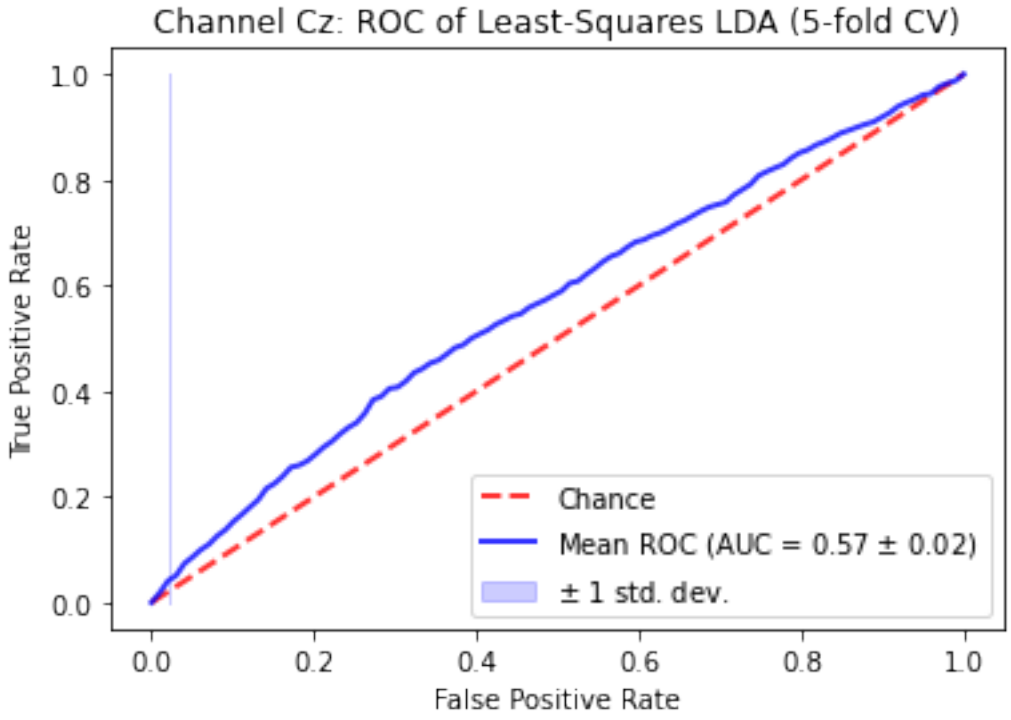
[24]:
```
nfolds = 5;
cv = StratifiedKFold(n_splits = nfolds)
tprs_lsqrs = []
aucs_lsqrs = []
mean_fpr_lsqrs = np.linspace(0, 1, 100)

for i in range(len(c_values)):
    x_train, x_test, y_train, y_test = train_test_split(train_filtered[:,i,:],␣
 ↪train_labels, test_size=0.2)
    X = x_train
    y = y_train
    temp_tprs_lsqrs, temp_aucs_lsqrs = [], []
    for train, test in cv.split(X, y):
        probas_lsqrs = clf_lsqrs.fit(X[train], y[train]).predict_proba(X[test])
        fpr_lsqrs, tpr_lsqrs, th_lsqrs = roc_curve(y[test], probas_lsqrs[:, 1])
        temp_tprs_lsqrs.append(np.interp(mean_fpr_lsqrs, fpr_lsqrs, tpr_lsqrs))
        temp_tprs_lsqrs[-1][0] = 0.0
        roc_auc_lsqrs = auc(fpr_lsqrs, tpr_lsqrs)
        temp_aucs_lsqrs.append(roc_auc_lsqrs)
    tprs_lsqrs.append(temp_tprs_lsqrs)
    aucs_lsqrs.append(temp_aucs_lsqrs)
```

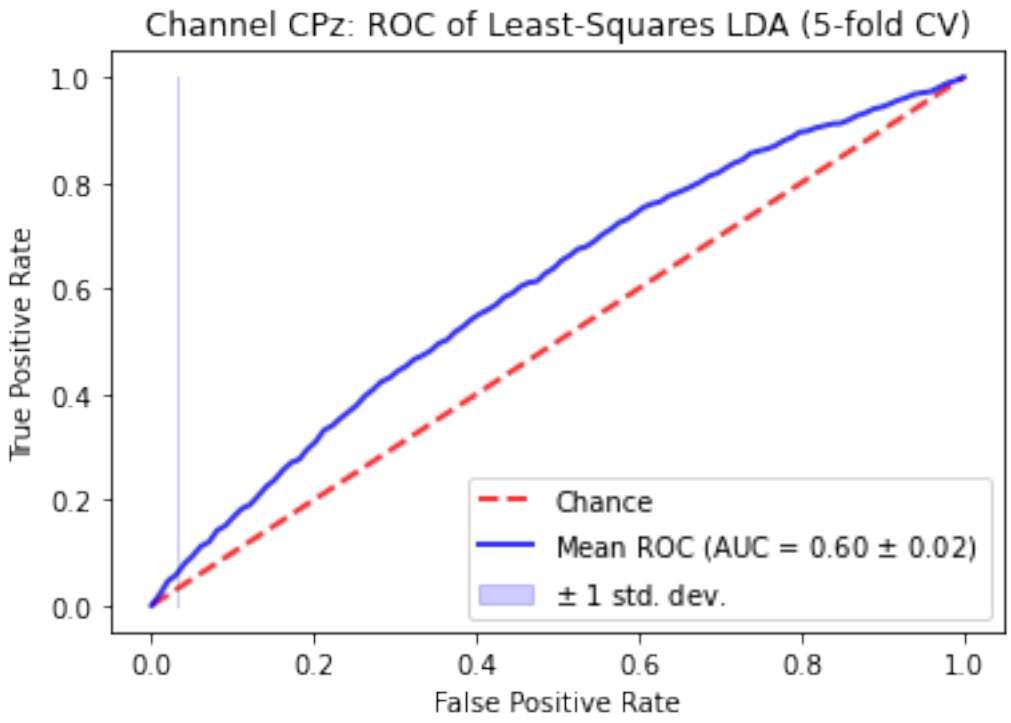
[25]:
```
# plot the data
mean_tpr_lsqrs = [np.mean(x, axis=0) for x in tprs_lsqrs]
# mean_tpr_lsqrs[-1] = 1.0
mean_auc_lsqrs = [auc(mean_fpr_lsqrs, x) for x in mean_tpr_lsqrs]
std_auc_lsqrs = [np.std(x) for x in aucs_lsqrs]

for i in range(len(c_values)):
    plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label='Chance',␣
 ↪alpha=.8)
    plt.plot(mean_fpr_lsqrs, mean_tpr_lsqrs[i], color='b',
             label=r'Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc_lsqrs[i],␣
 ↪std_auc_lsqrs[i]),
             lw=2, alpha=.8)

    std_tpr_lsqrs = np.std(tprs_lsqrs[i], axis=0)
    tprs_upper_lsqrs = np.minimum(mean_tpr_lsqrs[i] + std_tpr_lsqrs[i], 1)
    tprs_lower_lsqrs = np.maximum(mean_tpr_lsqrs[i] - std_tpr_lsqrs[i], 0)
    plt.fill_between(mean_fpr_lsqrs[i], tprs_lower_lsqrs, tprs_upper_lsqrs,␣
 ↪color='b', alpha=.2, label=r'$\pm$ 1 std. dev.')

    plt.xlim([-0.05, 1.05])
```

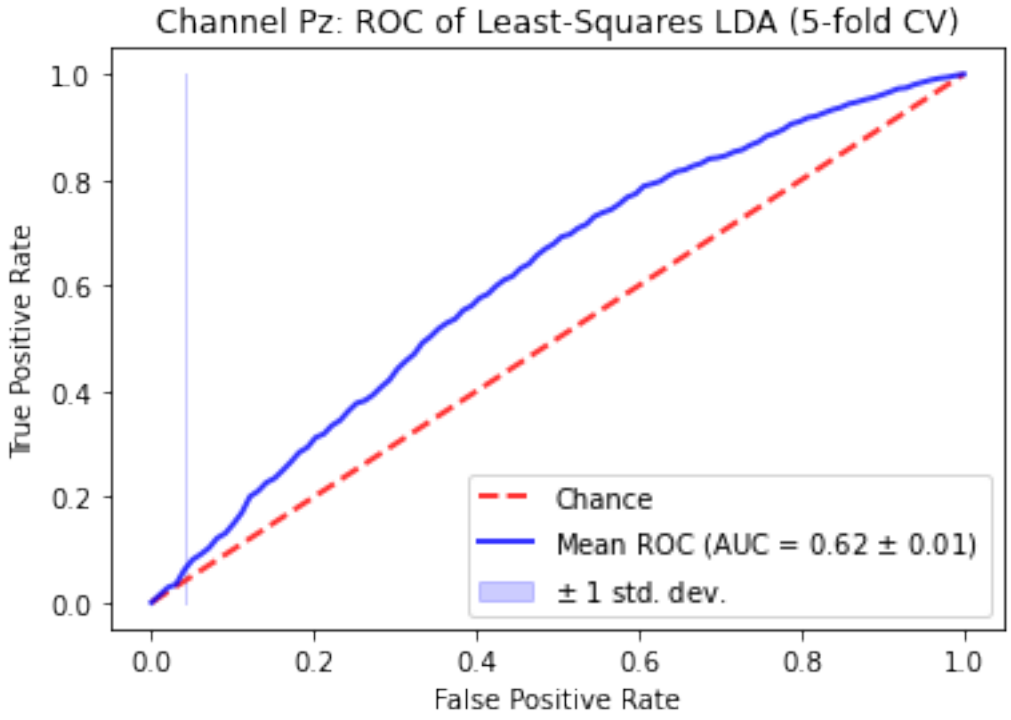
```
    plt.ylim([-0.05, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Channel ' + channels[i] + ': ROC of Least-Squares LDA (' +␣
↪str(nfolds) + '-fold CV)')
    plt.legend(loc="lower right")
    plt.show()
```

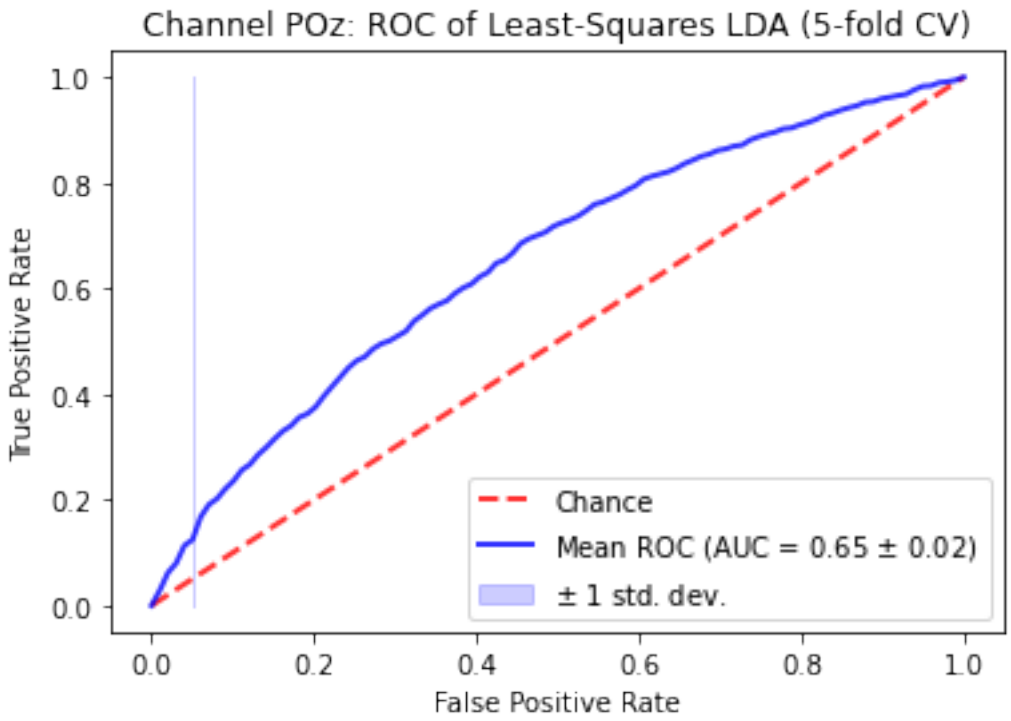



Channel FCz: ROC of Least-Squares LDA (5-fold CV)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Chance
Mean ROC (AUC = 0.50 ± 0.02)
± 1 std. dev.


Channel Cz: ROC of Least-Squares LDA (5-fold CV)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Chance
Mean ROC (AUC = 0.57 ± 0.02)
± 1 std. dev.


Channel CPz: ROC of Least-Squares LDA (5-fold CV)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Chance
Mean ROC (AUC = 0.60 ± 0.02)
± 1 std. dev.


Channel Pz: ROC of Least-Squares LDA (5-fold CV)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Chance
Mean ROC (AUC = 0.62 ± 0.01)
± 1 std. dev.



[26]:
```
# Set clf to clf_lsqrs for convenience
clf = clf_lsqrs.fit(x_train, y)

# Make predictions on our dataset
conf = clf.decision_function(x_test) # predicted confidence score
pred = clf.predict(x_test)           # predicted label (we won't actually use␣
↪this)

auc_score = roc_auc_score(y_test, pred)
print(auc_score)
```

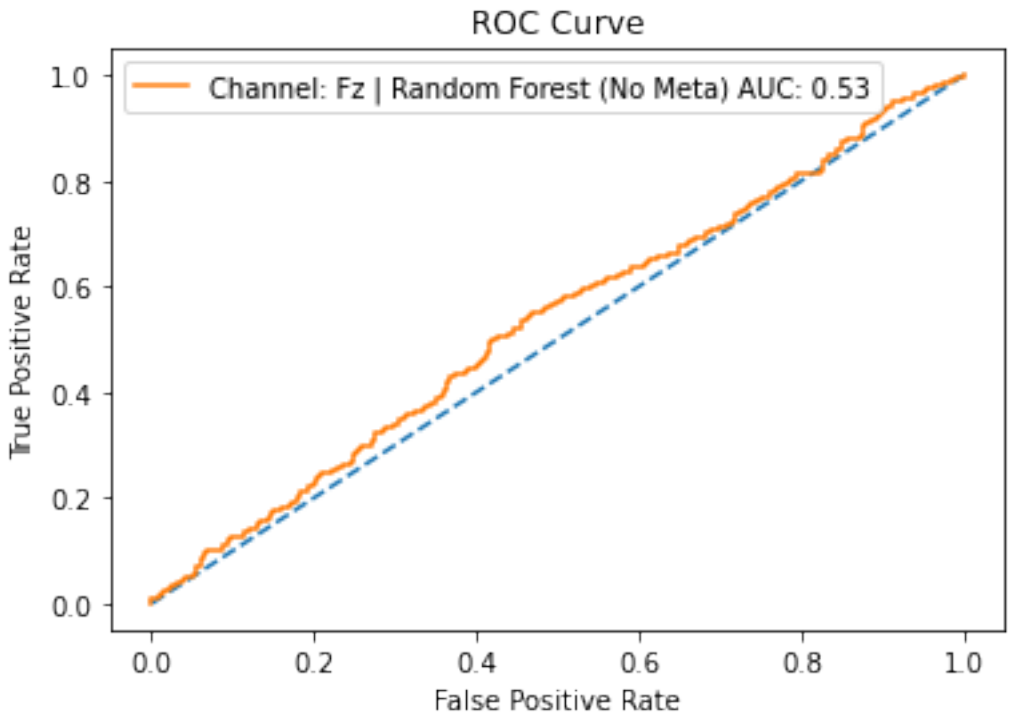
```
0.5470994293959468
```

[27]:
```
def plot_roc(y, preds, model):
    fpr, tpr, thresholds = roc_curve(y, preds)
    optimal_idx = np.argmax(tpr - fpr)
    optimal_threshold = thresholds[optimal_idx]
    print("Optimal threshold: ", optimal_threshold)
    l1, = plt.plot([0, 1], [0, 1], '--')
    l2, = plt.plot(fpr, tpr, label = 'Random Forest')
    auc = roc_auc_score(y, preds)
    l = plt.legend([l2], [model+str(' AUC: %.2f' % auc)])
    plt.xlabel('False Positive Rate')
```

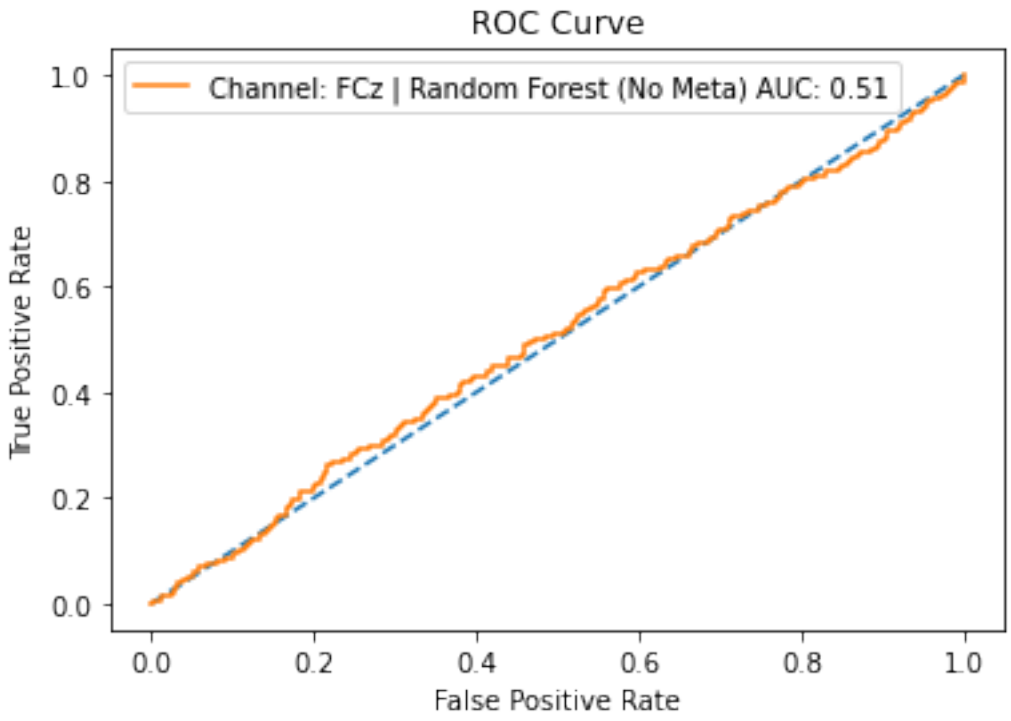
```
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.show()
```

[30]:
```
clf = ensemble.RandomForestClassifier(n_jobs = -1, n_estimators=500,␣
↪min_samples_split=10,
                                      min_samples_leaf=4, max_depth = 10,␣
↪random_state=42)

for i in range(len(c_values)):
    x_train, x_test, y_train, y_test = train_test_split(train_filtered[:,i,:],␣
↪train_labels, test_size=0.2)
    clf.fit(x_train, y_train)
    preds = clf.predict_proba(x_test)[:,1]
    plot_roc(y_test, preds, 'Channel: {} | Random Forest (No Meta)'.
↪format(channels[i]))
```

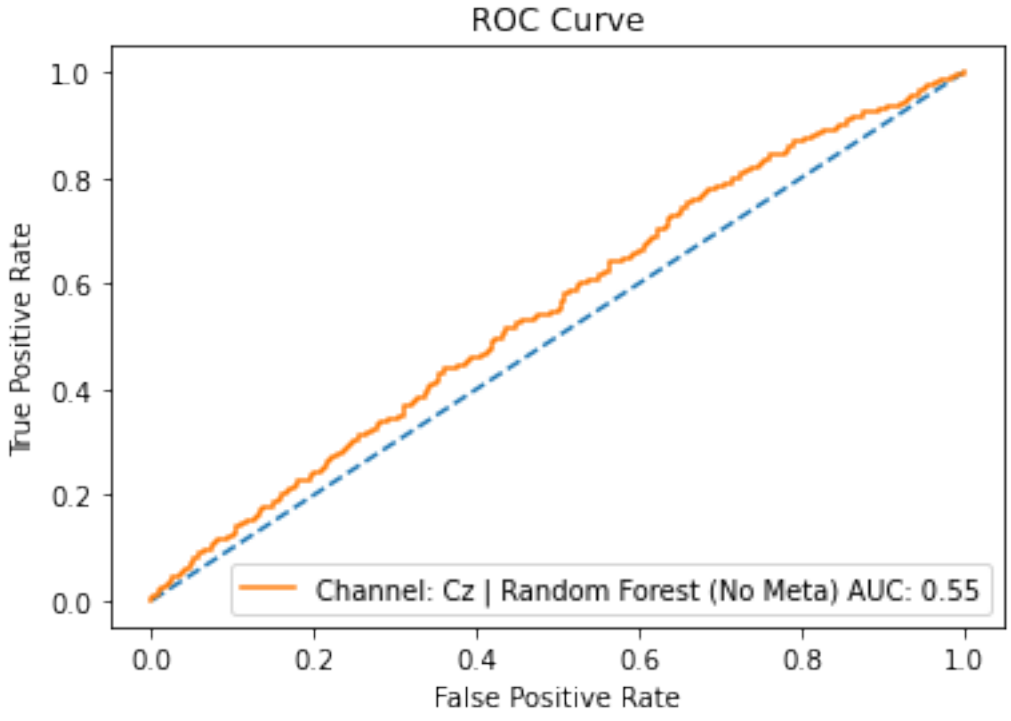
```
Optimal threshold:  0.7139832898302843
```



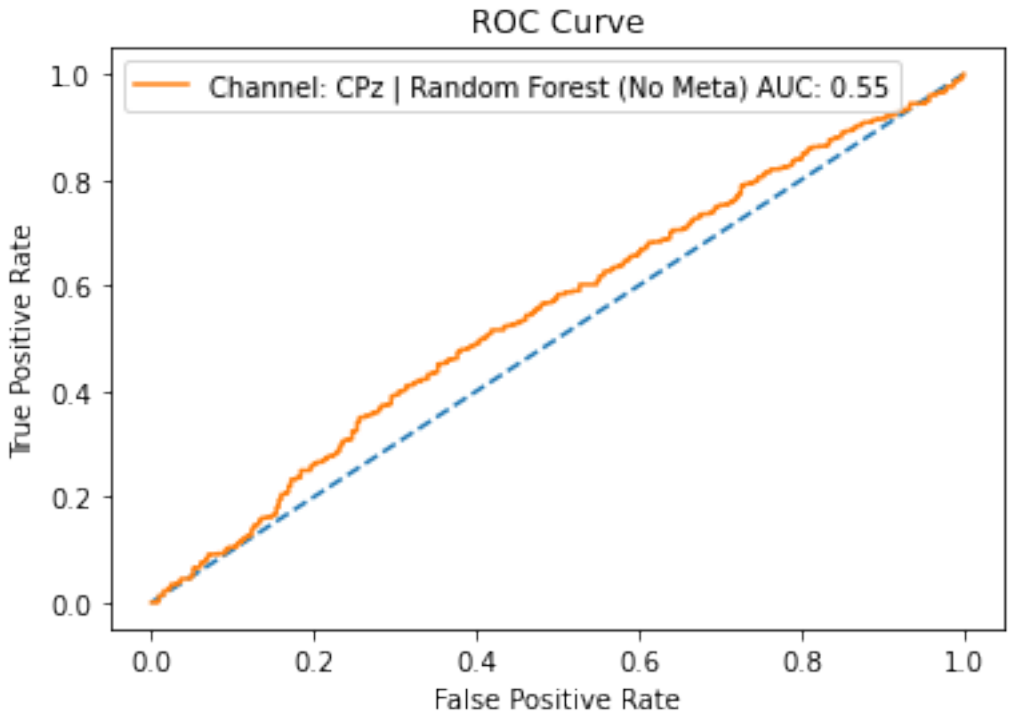

```
Optimal threshold:  0.7456524202147459
```



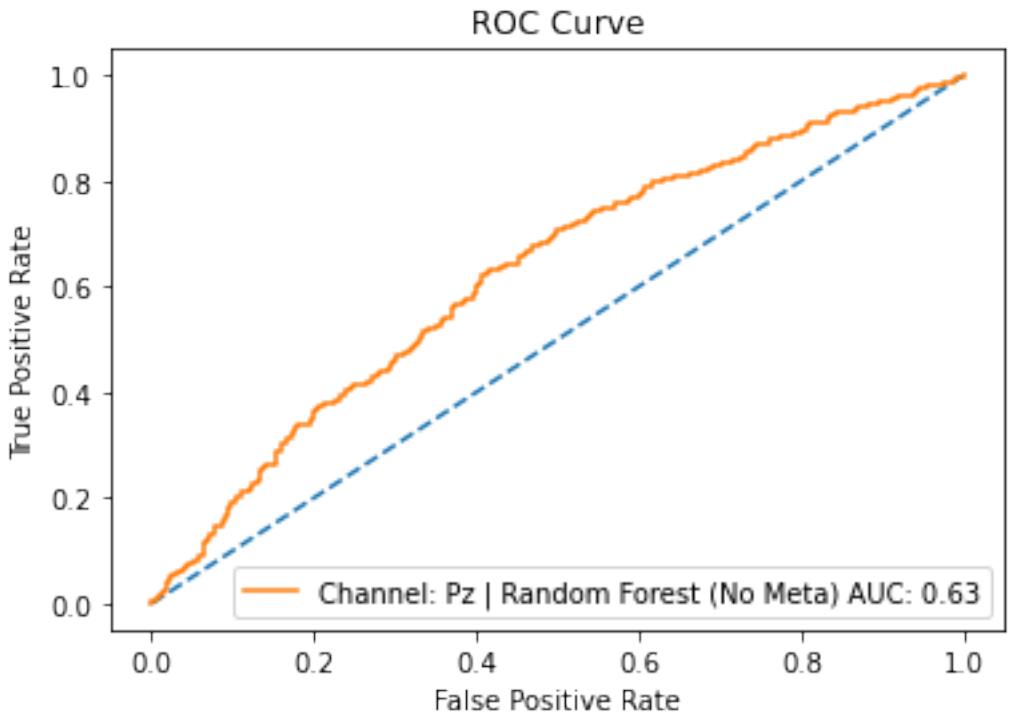

```
Optimal threshold:  0.6871062897304994
```



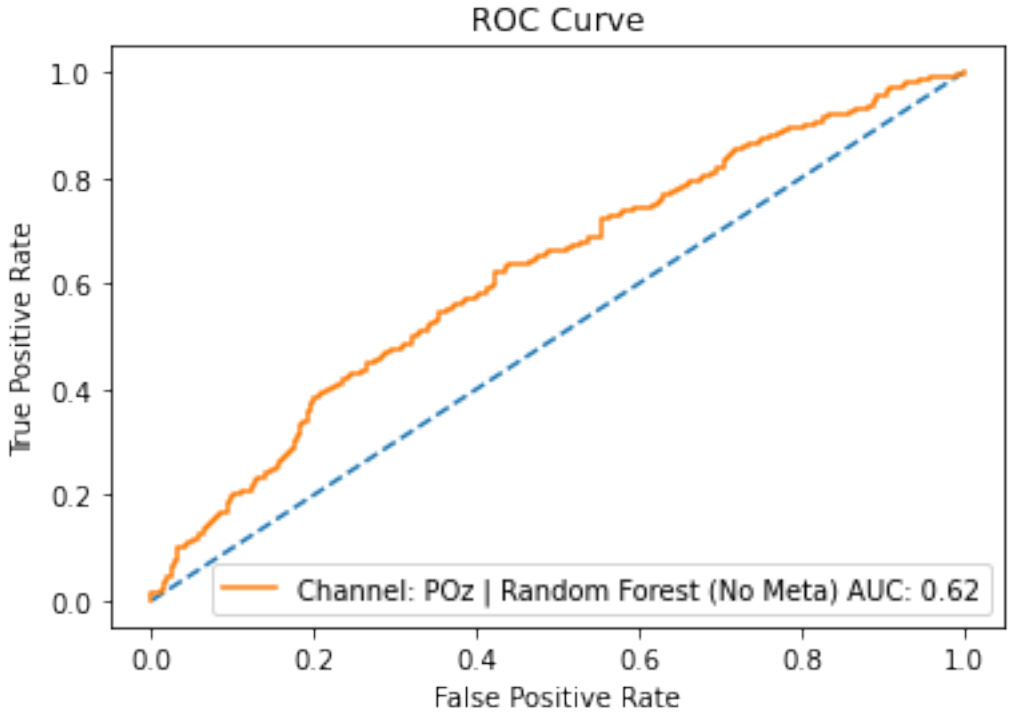
```
Optimal threshold:  0.7219935476574952
```




```
Optimal threshold:  0.707596371194472
```




```
Optimal threshold:  0.7013070749193551
```



1.10 Results & Discussion

Accuracy by Electrodes & Model

- C-Support Vector Classification (SVC)
  - Fz electrode score: 0.48846405423083794
  - FCz electrode score: 0.4958808189079858
  - Cz electrode score: 0.49725396497064
  - CPz electrode score: **0.5162611197803382**
  - Pz electrode score: 0.5
  - POz electrode score: 0.513825285376175
- Random Guesser
  - Fz electrode score: 0.5141091192837635
  - FCz electrode score: 0.49842441330126874
  - Cz electrode score: **0.5324666098323388**
  - CPz electrode score: 0.4857942349878501
  - Pz electrode score: 0.501171875
  - POz electrode score: 0.5272782694440012
- LDA (lsqr auto)
  - Fz, accuracy: 0.7056526386634466
  - FCz, accuracy: **0.710937619594336**
  - Cz, accuracy: 0.7042733283186191
  - CPz, accuracy: 0.7070314211436187
  - Pz, accuracy: 0.702664396848648
  - POz, accuracy: 0.699907623685287
- LDA (lsqr)
  - Fz, accuracy: 0.6985293690697706
  - FCz, accuracy: 0.7031241669635906
  - Cz, accuracy: 0.692095754648508
  - CPz, accuracy: 0.7090985391345659
  - Pz, accuracy: 0.7102527152038218
  - POz, accuracy: **0.7127782836480726**
- LDA (svd)
  - Fz, accuracy: 0.7040439711258034
  - FCz, accuracy: 0.7033543159533896
  - Cz, accuracy: 0.7022059463953443
  - CPz, accuracy: 0.7038146139329875
  - Pz, accuracy: 0.7051910210222099
  - POz, accuracy: **0.7077218681130159**

As shown and discussed in our methods section, SVC models were shown to perform poorly on average, with some electrodes scoring lower than random guesser model. We believe that this is due to the fact that our preprocessing was not as great as we intended to be. The bandpassing and filtering seem to achieve little to no effect on clearing up the noise in our data. Regardless, our LDA models achieved on average 70% accuracy in the testing environment, which is relatively acceptable given the noise and that past submissions to this challenge are not too far from it.

Despite our unmentioned efforts to clear up the data, BCI and machine learning are difficult tasks, with our best model achieving 71% accuracy. There is clearly room for improvement, but some important observations were made. The linear discriminant analysis is suitable for high dimension dataset, which almost all BCI datasets are, due to its excellence in its ability to find linear combinations of features that characterize/separate 2+ classes of events. This translated to our finding such that LDA models achieved the highest accuracy.

To conclude, it was important to acknowledge that there is no "one-classifier-fits-all" classification approach to the multi-dimensional dataset explored in this challenge. The uniqueness of every dataset makes it necessary to try multiple classifiers and find the best hyperparameters through optimization, ultimately solving the question at hand.