HW2

* 1. Loading the data and plotting the sensor locations

A circle with dots and lines

Description automatically generated with medium confidence

* 1. Extract Oz channel data for each condition

A group of orange and blue lines

Description automatically generated

1. Compute the power spectral density (PSD) of the entire eyes open and eyes closed conditions for channel “Oz”:

A graph of different types of lines

Description automatically generated with medium confidence

1. For only the eyes-closed condition**, identify the peak alpha frequency** (i.e., 8-12 Hz) over channel “Oz” for each subject:

**peak\_alpha\_frequencies for subjects = [11, 10, 9, 11, 10]**

Plot the **topographies** for the eyes open and eyes closed conditions.

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

1. PSD for teeth clenched condition using channels “T7” and “T8:

A graph of a diagram

Description automatically generated with medium confidence

1. Plotting the teeth clenched topographies at 10, 25, 40, and 65 Hz for each subject:

A screenshot of a computer screen

Description automatically generated

1. Extract Fz channel data for blinking:

A group of blue lines

Description automatically generated

1. Compute the power spectral density (PSD) of the entire blinking conditions for channel “Fz” and computing the peak frequency:

A screenshot of a graph

Description automatically generated

1. I developed an eye blink detector using a sliding window approach on EEG data. Each window was of length 256 samples (equivalent to 1 second) with an overlap of 30%, ensuring a robust and dense coverage of the entire data sequence. Within each window, the data was squared to improve its resilience against minor fluctuations and to accentuate the amplitude of potential blinks. A dynamic threshold for peak detection was then computed for each windowed segment based on the formula: (mean of the squared data plus 2.5 times its standard deviation). Peaks, indicative of eye blinks, were subsequently identified within these windowed segments by determining points where the amplitude exceeded the computed threshold. To maintain accuracy and avoid counting the same blink from adjacent overlapping windows, a minimum distance :(length 128 samples (equivalent to 0.5 second)) constraint was enforced between successive peaks. After processing the entire dataset, the detected peaks from all the windows were collated, and any duplicates arising from the overlap mechanism were removed. This procedure was iteratively executed for each subject's data, resulting in a list of detected blink instances. By harnessing a blend of amplitude thresholding, peak detection, and the sliding window technique with its inherent overlap, the method offers a meticulous and precise identification of eye blinks.

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

1. To validate the performance of the developed blink detector, I randomly selected five instances from the detected blinks. Topographic maps for these instances are presented. In most of these topographic representations, I observed heightened activity in the frontal area. This prominent frontal activity serves as evidence supporting the accuracy and effectiveness of the developed detector.

A group of colorful circles with text

Description automatically generated

# Code Section:

HW2

The data consists of 4 runs related closed eyes, open eyes, blinking, and clenching tasks:  
each a 2 dimensional matrix: samples (2560) \* channels (16)  
Channels are as follows (ordered from 1 to 16) :  
1-FPz 2-Fz 3-T7 4-T8 5-C3 6-C4 7-C5 8-C6 9-CP3 10-CP4 11-Cz 12-CPz 13-Pz 14-PO7 15-PO8 16-Oz  
sampling rate: 256 Hz,  
task duration: 1 min,  
number of channels: 16

In [62]:

**import** numpy **as** np

**from** scipy.io **import** loadmat

**import** mne

**import** matplotlib.pyplot **as** plt

**from** scipy.signal **import** welch

**from** scipy.signal **import** find\_peaks

In [2]:

*# 1. Load the .mat file*

filenames **=** ["S1\_data.mat", "S2\_data.mat", "S3\_data.mat", "S4\_data.mat", "S5\_data.mat"]

matrices **=** [loadmat(filename) **for** filename **in** filenames]

print('type(matrices):', type(matrices), 'len(matrices):', len(matrices), 'type(matrices[0]):', type(matrices[0]))

*# type(matrices): <class 'list'> len(matrices): 5 type(matrices[0]): <class 'dict'>,*

*#So, matrices[i] refers to ith matrix*

*# Print the keys in the first loaded matrix*

print('matrices[0].keys():', matrices[0]**.**keys()) *# matrices[0].keys(): dict\_keys(['\_\_header\_\_', '\_\_version\_\_', '\_\_globals\_\_', 'data'])*

*#so matrices[i]['data'] refers to the data of the ith matrix*

print(type(matrices[0]['data']), matrices[0]['data']**.**shape, len(matrices[0]['data'][0, 0])) *#<class 'numpy.ndarray'> (1, 1) 4*

*#so, matrices[i]['data'][0, 0][j] refers to the jth array of the ith matrix with shape (15360, 16): EyesOpen(j=0), EyesClosed(j=1), Blink(j=2), Clench(j=3), i=0,... ,4*

*########################################################################################################################################################################*

EyesOpen**=**[]

EyesClosed**=**[]

Blink**=**[]

Clench**=**[]

**for** i **in** range( len(matrices)):

**for** j **in** range(len(matrices[0]['data'][0, 0])):

**if** j**==**0:

EyesOpen**.**append(matrices[i]['data'][0, 0][j])

**elif** j**==**1:

EyesClosed**.**append(matrices[i]['data'][0, 0][j])

**elif** j**==**2:

Blink**.**append(matrices[i]['data'][0, 0][j])

**else**:

Clench**.**append(matrices[i]['data'][0, 0][j])

print('len(EyesOpen):', len(EyesOpen), 'len(EyesClosed):', len(EyesClosed), 'len(Blink):', len(Blink), 'len(Clench):', len(Clench))

*#len(EyesOpen): 5 len(EyesClosed): 5 len(Blink): 5 len(Clench): 5*

EyesOpen\_np**=**np**.**array(EyesOpen)

EyesClosed\_np**=**np**.**array(EyesClosed)

Blink\_np**=**np**.**array(Blink)

Clench\_np**=**np**.**array(Clench)

print('EyesOpen\_np.shape:', EyesOpen\_np**.**shape, 'EyesClosed\_np.shape:', EyesClosed\_np**.**shape, 'Blink\_np.shape:', Blink\_np**.**shape, 'Clench\_np.shape:', Clench\_np**.**shape)

*#EyesOpen\_np.shape: (5, 15360, 16) EyesClosed\_np.shape: (5, 15360, 16) Blink\_np.shape: (5, 15360, 16) Clench\_np.shape: (5, 15360, 16)*

*######################################################################################################################################*

*# Read the eloc16C2.txt*

**with** open('eloc16C2.txt', 'r') **as** f:

lines **=** [line**.**strip() **for** line **in** f**.**readlines() **if** line**.**strip()] *# This removes any empty lines*

*# Check that you're only processing 16 lines*

**if** len(lines) **!=** 16:

print(f"Warning: Expected 16 lines but found {len(lines)} lines.")

**for** line **in** lines:

print(line) *# This will print out all lines so you can inspect them*

**else**:

*# Extract channel names, theta, and radius*

channel\_names **=** [line**.**split()[3]**.**replace('.', '') **for** line **in** lines]

theta **=** np**.**array([float(line**.**split()[1])**-**90 **for** line **in** lines[0:]]) **\*** np**.**pi **/** 180.0 *# Convert to radians*

radius **=** np**.**array([float(line**.**split()[2]) **for** line **in** lines[0:]])

*# Convert polar to Cartesian*

x **=** radius **\*** np**.**cos(theta)**/**5

y **=** **-**radius **\*** np**.**sin(theta)**/**5

z **=** np**.**zeros\_like(x) *# default z-coordinate for all channels*

ch\_pos **=** dict(zip(channel\_names, zip(x, y, z)))

montage **=** mne**.**channels**.**make\_dig\_montage(ch\_pos, coord\_frame**=**'head')

info **=** mne**.**create\_info(ch\_names**=**channel\_names, sfreq**=**256, ch\_types**=**'eeg')

info**.**set\_montage(montage)

*# Plot the montage*

montage**.**plot(show\_names**=True**)

type(matrices): <class 'list'> len(matrices): 5 type(matrices[0]): <class 'dict'>

matrices[0].keys(): dict\_keys(['\_\_header\_\_', '\_\_version\_\_', '\_\_globals\_\_', 'data'])

<class 'numpy.ndarray'> (1, 1) 4

len(EyesOpen): 5 len(EyesClosed): 5 len(Blink): 5 len(Clench): 5

EyesOpen\_np.shape: (5, 15360, 16) EyesClosed\_np.shape: (5, 15360, 16) Blink\_np.shape: (5, 15360, 16) Clench\_np.shape: (5, 15360, 16)

A circular object with black dots

Description automatically generated

Out[2]:

In [3]:

*# Extract Oz channel data for each condition*

eyes\_open\_oz **=** EyesOpen\_np[:, :, 15] *# The 16th channel (0-based indexing)*

eyes\_closed\_oz **=** EyesClosed\_np[:, :, 15]

n\_s **=** eyes\_closed\_oz**.**shape[0]

fig, axes **=** plt**.**subplots(n\_s, 1, figsize**=**(10, 2**\***n\_s))

**for** i **in** range(n\_s):

axes[i]**.**plot(eyes\_open\_oz[i], label**=**'Eyes Open')

axes[i]**.**plot(eyes\_closed\_oz[i], label**=**'Eyes Closed')

axes[i]**.**set\_title(f'Sunject {i **+** 1}')

axes[i]**.**set\_xlabel('Time Points')

axes[i]**.**set\_ylabel('Amplitude')

axes[i]**.**legend()

plt**.**tight\_layout()

plt**.**show()

A group of orange and blue lines

Description automatically generated

(a) Compute the power spectral density (PSD) of the entire eyes open and eyes closed conditions for channel “Oz”

In [4]:

*# a.*

*# Define the sampling frequency and parameters for the Welch method*

fs **=** 256 *# Given*

nperseg **=** fs *#256 samples for a 1-second window*

noverlap **=** nperseg **//** 2 *# 50% overlap*

*# Compute PSD for each subject*

frequencies, psd\_open **=** welch(eyes\_open\_oz, fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1)

\_, psd\_closed **=** welch(eyes\_closed\_oz, fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1)

*# Filtering the frequencies and corresponding PSD values to be within the 0-70 Hz range*

mask **=** (frequencies **>=** 0) **&** (frequencies **<=** 70)

filtered\_frequencies **=** frequencies[mask]

filtered\_psd\_open **=** psd\_open[:, mask]

filtered\_psd\_closed **=** psd\_closed[:, mask]

psd\_open\_log**=**10**\***np**.**log10(filtered\_psd\_open)

psd\_closed\_log**=**10**\***np**.**log10(filtered\_psd\_closed)

fig, axes **=** plt**.**subplots(n\_s, 1, figsize**=**(10, 2**\***n\_s))

**for** i **in** range(n\_s):

axes[i]**.**plot(psd\_open\_log[i], label**=**'Eyes Open')

axes[i]**.**plot(psd\_closed\_log[i], label**=**'Eyes Closed')

axes[i]**.**set\_title(f'Subject {i **+** 1} - Power Spectral Density (PSD) for Channel "Oz"')

axes[i]**.**set\_xlabel('Frequency (Hz)')

axes[i]**.**set\_ylabel('PSD (dB/Hz)')

axes[i]**.**legend()

axes[i]**.**grid(**True**, which**=**'both')

plt**.**tight\_layout()

plt**.**show()

A graph of different types of lines

Description automatically generated with medium confidence

In [5]:

*# b.*

alpha\_mask **=** (filtered\_frequencies **>=** 8) **&** (filtered\_frequencies **<=** 12)

alpha\_psd **=** psd\_closed\_log[:, alpha\_mask]

peak\_alpha\_indices **=** np**.**argmax(alpha\_psd, axis**=**1)

peak\_alpha\_frequencies **=** filtered\_frequencies[alpha\_mask][peak\_alpha\_indices]

*#peak\_alpha\_frequencies = array([11., 10., 9., 11., 10.])*

num\_subjects, num\_timepoints, num\_channels **=** EyesOpen\_np**.**shape *# (5, 15360, 16)*

*# Preallocate arrays for storing the PSD data*

psd\_open\_all **=** np**.**empty((num\_subjects, num\_channels, len(frequencies)))

psd\_closed\_all **=** np**.**empty((num\_subjects, num\_channels, len(frequencies)))

*# Calculate the PSD for all channels and subjects*

**for** channel **in** range(num\_channels):

\_, psd\_open\_all[:, channel, :] **=** welch(EyesOpen\_np[:, :, channel], fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1) *#shape: (5, 16, 129)*

\_, psd\_closed\_all[:, channel, :] **=** welch(EyesClosed\_np[:, :, channel], fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1) *#shape: (5, 16, 129)*

*# # Determine consistent color limits across subjects for each conditions*

*# global\_closed=[]*

*# global\_open=[]*

*# for subject in range(num\_subjects):*

*# peak\_freq = peak\_alpha\_frequencies[subject]*

*# # Extract PSD values at the peak alpha frequency for both eyes-open and eyes-closed conditions*

*# closed\_peak\_psd\_values = psd\_closed\_all[subject, :, frequencies == peak\_freq].squeeze()*

*# open\_peak\_psd\_values = psd\_open\_all[subject, :, frequencies == peak\_freq].squeeze()*

*# global\_closed.append(closed\_peak\_psd\_values)*

*# global\_open.append(open\_peak\_psd\_values)*

*# global\_vmin\_closed =np.min(global\_closed)*

*# global\_vmax\_closed =np.max(global\_closed)*

*# global\_vmin\_open =np.min(global\_open)*

*# global\_vmax\_open =np.max(global\_open)*

*# global\_vmin\_closed\_log =10\*np.log10(global\_vmin\_closed)*

*# global\_vmax\_closed\_log =10\*np.log10(global\_vmax\_closed)*

*# global\_vmin\_open\_log =10\*np.log10(global\_vmin\_open)*

*# global\_vmax\_open\_log =10\*np.log10(global\_vmax\_open)*

open**=**[]

closed**=**[]

**for** subject **in** range(num\_subjects):

*# Extract the peak alpha frequency for this subject*

peak\_freq **=** peak\_alpha\_frequencies[subject]

*# Extract PSD values at the peak alpha frequency for both eyes-open and eyes-closed conditions*

closed\_peak\_psd\_values **=** psd\_closed\_all[subject, :, frequencies **==** peak\_freq]**.**squeeze()

open\_peak\_psd\_values **=** psd\_open\_all[subject, :, frequencies **==** peak\_freq]**.**squeeze()

closed**.**append(closed\_peak\_psd\_values)

open**.**append(open\_peak\_psd\_values)

*# Determine consistent color limits across conditions for each sunject*

global\_vmin **=** min(np**.**min(closed\_peak\_psd\_values), np**.**min(open\_peak\_psd\_values))

global\_vmax **=** max(np**.**max(closed\_peak\_psd\_values), np**.**max(open\_peak\_psd\_values))

*# # Extract 10log10 PSD values at the peak alpha frequency for both eyes-open and eyes-closed conditions*

*# closed\_peak\_psd\_values\_log=10\*np.log10(closed\_peak\_psd\_values)*

*# open\_peak\_psd\_values\_log=10\*np.log10(open\_peak\_psd\_values)*

*# # Determine consistent color limits across conditions for each sunject*

*# global\_vmin\_log = min(np.min(closed\_peak\_psd\_values\_log), np.min(open\_peak\_psd\_values))*

*# global\_vmax\_log = max(np.max(closed\_peak\_psd\_values\_log), np.max(open\_peak\_psd\_values))*

fig, axes **=** plt**.**subplots(1, 2, figsize**=**(6,2.5))

*# Eyes-closed condition*

im1, \_ **=** mne**.**viz**.**plot\_topomap(closed\_peak\_psd\_values, info, cmap**=**'jet', axes**=**axes[0], show**=False**)

im1**.**set\_clim(global\_vmin, global\_vmax)

axes[0]**.**set\_title(f'Subject {subject **+** 1} - Eyes Closed\nPeak Alpha at {peak\_freq:.2f} Hz')

plt**.**colorbar(im1, ax**=**axes[0])

*# Eyes-open condition*

im2, \_ **=** mne**.**viz**.**plot\_topomap(open\_peak\_psd\_values, info, cmap**=**'jet', axes**=**axes[1], show**=False**)

im2**.**set\_clim(global\_vmin, global\_vmax)

axes[1]**.**set\_title(f'Subject {subject **+** 1} - Eyes Open\nPeak Alpha at {peak\_freq:.2f} Hz')

plt**.**colorbar(im2, ax**=**axes[1])

*# # Eyes-closed condition log*

*# im1, \_ = mne.viz.plot\_topomap(closed\_peak\_psd\_values\_log, info, cmap='jet', axes=axes[2], show=False)*

*# im1.set\_clim(global\_vmin\_log, global\_vmax\_log)*

*# axes[2].set\_title(f'Subject {subject + 1} - 10log10 Eyes Closed\nPeak Alpha at {peak\_freq:.2f} Hz')*

*# plt.colorbar(im1, ax=axes[2])*

*# # Eyes-open condition log*

*# im2, \_ = mne.viz.plot\_topomap(open\_peak\_psd\_values\_log, info, cmap='jet', axes=axes[3], show=False)*

*# im2.set\_clim(global\_vmin\_log, global\_vmax\_log)*

*# axes[3].set\_title(f'Subject {subject + 1} - 10log10 Eyes Open\nPeak Alpha at {peak\_freq:.2f} Hz')*

*# plt.colorbar(im2, ax=axes[3])*

plt**.**tight\_layout()

plt**.**show()

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

A close-up of a globe

Description automatically generated

In [6]:

*# c.*

*# Extract channels “T7” and “T8 (2 and 3) data for each condition*

Clench\_np\_T7 **=** Clench\_np[:, :, 2]

Clench\_np\_T8 **=** Clench\_np[:, :, 3]

fs **=** 256

nperseg **=** fs *#256 samples for a 1-second window*

noverlap **=** nperseg **//** 2 *# 50% overlap*

*# Compute PSD for each subject*

frequencies, psd\_Clench\_T7 **=** welch(Clench\_np\_T7, fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1)

\_, psd\_Clench\_T8 **=** welch(Clench\_np\_T8, fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1)

*# Filtering the frequencies and corresponding PSD values to be within the 0-70 Hz range*

mask **=** (frequencies **>=** 0) **&** (frequencies **<=** 70)

filtered\_frequencies **=** frequencies[mask]

filtered\_psd\_Clench\_T7 **=** psd\_Clench\_T7[:, mask]

filtered\_psd\_Clench\_T8 **=** psd\_Clench\_T8[:, mask]

psd\_Clench\_T7\_log**=**10**\***np**.**log10(filtered\_psd\_Clench\_T7)

psd\_Clench\_T8\_log**=**10**\***np**.**log10(filtered\_psd\_Clench\_T8)

fig, axes **=** plt**.**subplots(n\_s, 1, figsize**=**(10, 2**\***n\_s))

**for** i **in** range(n\_s):

axes[i]**.**plot(psd\_Clench\_T7\_log[i], label**=**'psd\_Clench\_T7')

axes[i]**.**plot(psd\_Clench\_T8\_log[i], label**=**'psd\_Clench\_T8')

axes[i]**.**set\_title(f'Subject {i **+** 1} - Power Spectral Density (PSD) for Clench task')

axes[i]**.**set\_xlabel('Frequency (Hz)')

axes[i]**.**set\_ylabel('PSD (dB/Hz)')

axes[i]**.**legend()

axes[i]**.**grid(**True**, which**=**'both')

plt**.**tight\_layout()

plt**.**show()

A graph of a diagram

Description automatically generated with medium confidence

In [7]:

*#d.*

*# Preallocate arrays for storing the PSD data*

PSD\_Clench **=** np**.**empty((num\_subjects, num\_channels, len(frequencies)))

*# psd\_Clench\_T8 = np.empty((num\_subjects, num\_channels, len(frequencies)))*

clench\_freq**=**[10, 25, 40, 65]

*# Calculate the PSD for all channels and subjects*

**for** channel **in** range(16):

\_, PSD\_Clench[:, channel, :] **=** welch(Clench\_np[:, :, channel], fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1) *#shape: (5, 16, 129)*

fig, axes **=** plt**.**subplots(5, len(clench\_freq), figsize**=**(3 **\*** len(clench\_freq), 1.5 **\*** 5))

**for** subject **in** range(5):

*# Using a list comprehension to gather the PSD values for each frequency in clench\_freq*

psd\_values\_for\_freqs **=** [PSD\_Clench[subject, :, frequencies **==** freq]**.**squeeze() **for** freq **in** clench\_freq]

*# Get global vmin and vmax for colorbar scaling for the current subject across all frequencies*

global\_vmin **=** np**.**min(psd\_values\_for\_freqs)

global\_vmax **=** np**.**max(psd\_values\_for\_freqs)

**for** idx, frequency **in** enumerate(clench\_freq):

clenched\_psd\_values **=** psd\_values\_for\_freqs[idx]

im, \_ **=** mne**.**viz**.**plot\_topomap(clenched\_psd\_values, info, cmap**=**'jet', axes**=**axes[subject, idx], show**=False**)

im**.**set\_clim(global\_vmin, global\_vmax)

axes[subject, idx]**.**set\_title(f'Subject {subject **+** 1} - Frequency {frequency} Hz')

*# Add individual colorbar for each subject at the rightmost column of the subplot array*

fig**.**colorbar(im, ax**=**axes[subject, **-**1], orientation**=**'vertical', pad**=**0.1)

plt**.**tight\_layout()

plt**.**show()

A screenshot of a computer screen

Description automatically generated

In [8]:

*# 2.*

*# Extract Fz channel data for each condition*

eyes\_blink\_Fz **=** Blink\_np[:, :, 1] *# The 16th channel (0-based indexing)*

n\_s **=** eyes\_blink\_Fz**.**shape[0]

fig, axes **=** plt**.**subplots(n\_s, 1, figsize**=**(10, 2**\***n\_s))

**for** i **in** range(n\_s):

axes[i]**.**plot(eyes\_blink\_Fz[i], label**=**'Eyes Open')

axes[i]**.**set\_title(f'Sunject {i **+** 1}')

axes[i]**.**set\_xlabel('Time Points')

axes[i]**.**set\_ylabel('Amplitude')

axes[i]**.**legend()

plt**.**tight\_layout()

plt**.**show()

A group of blue lines

Description automatically generated

In [65]:

*# 2.*

*# Define the sampling frequency and parameters for the Welch method*

fs **=** 256 *# Given*

nperseg **=** fs *#256 samples for a 1-second window*

noverlap **=** nperseg **//** 2 *# 50% overlap*

*# Compute PSD for each subject*

frequencies, psd\_blink\_Fz **=** welch(eyes\_blink\_Fz, fs**=**fs, nperseg**=**nperseg, noverlap**=**noverlap, axis**=**1)

*# Filtering the frequencies and corresponding PSD values to be within the 0-70 Hz range*

mask **=** (frequencies **>=** 0) **&** (frequencies **<=** 70)

filtered\_frequencies **=** frequencies[mask]

filtered\_psd\_blink\_FZ **=** psd\_blink\_Fz[:, mask]

psd\_blink\_log\_FZ**=**10**\***np**.**log10(filtered\_psd\_blink\_FZ)

*# calculate the peach frequency*

blink\_mask **=** (filtered\_frequencies **>=** 1) **&** (filtered\_frequencies **<=** 15)

blink\_psd **=** psd\_blink\_log\_FZ[:, blink\_mask]

peak\_blink\_indices **=** np**.**argmax(blink\_psd, axis**=**1)

peak\_blink\_frequencies **=** filtered\_frequencies[blink\_mask][peak\_blink\_indices]

print('peak\_blink\_frequencies:', peak\_blink\_frequencies)

*# peak\_blink\_frequencies: [3. 2. 3. 3. 2.]*

*# Now modify your plotting loop:*

fig, axes **=** plt**.**subplots(n\_s, 1, figsize**=**(10, 2**\***n\_s))

**for** i **in** range(n\_s):

axes[i]**.**plot(filtered\_frequencies, psd\_blink\_log\_FZ[i], label**=**'FZ')

*# Add a red dotted line for the peak frequency*

peak\_freq **=** peak\_blink\_frequencies[i]

peak\_power **=** psd\_blink\_log\_FZ[i][filtered\_frequencies **==** peak\_freq]

axes[i]**.**axvline(x**=**peak\_freq, color**=**'red', linestyle**=**'--')

*# Annotate the point with frequency and power value*

annotation\_text **=** f'({peak\_freq:.2f} Hz, {peak\_power[0]:.2f} dB/Hz)'

axes[i]**.**annotate(annotation\_text, xy**=**(peak\_freq, peak\_power[0]), xycoords**=**'data',

xytext**=**(**-**100, 10), textcoords**=**'offset points',

arrowprops**=**dict(arrowstyle**=**"->"))

axes[i]**.**set\_title(f'Subject {i **+** 1} - Power Spectral Density (PSD) for Channel "Fz"')

axes[i]**.**set\_xlabel('Frequency (Hz)')

axes[i]**.**set\_ylabel('PSD (dB/Hz)')

axes[i]**.**legend()

axes[i]**.**grid(**True**, which**=**'both')

plt**.**tight\_layout()

plt**.**show()

peak\_blink\_frequencies: [3. 2. 3. 3. 2.]

A screenshot of a graph

Description automatically generated

The eye blink detector was developed using a sliding window approach on EEG data. Each window was of length 256 samples (equivalent to 1 second) with an overlap of 30%, ensuring a robust and dense coverage of the entire data sequence. Within each window, the data was squared to improve its resilience against minor fluctuations and to accentuate the amplitude of potential blinks. A dynamic threshold for peak detection was then computed for each windowed segment based on the formula: (mean of the squared data plus 2.5 times its standard deviation). Peaks, indicative of eye blinks, were subsequently identified within these windowed segments by determining points where the amplitude exceeded the computed threshold. To maintain accuracy and avoid counting the same blink from adjacent overlapping windows, a minimum distance :(length 128 samples (equivalent to 0.5 second)) constraint was enforced between successive peaks. After processing the entire dataset, the detected peaks from all the windows were collated, and any duplicates arising from the overlap mechanism were prudently removed. This procedure was iteratively executed for each subject's data, resulting in a list of detected blink instances. By harnessing a blend of amplitude thresholding, peak detection, and the sliding window technique with its inherent overlap, the method offers a meticulous and precise identification of eye blinks.

In [66]:

**def** compute\_threshold(eeg\_data, k**=**2.5):

"""

Computes the amplitude threshold for blink detection.

Parameters:

- eeg\_data: 1D array representing EEG data.

- k: constant multiplier for the standard deviation.

Returns:

- Threshold for blink detection.

"""

**return** np**.**mean(eeg\_data) **+** k **\*** np**.**std(eeg\_data), *#std: Compute the standard deviation*

**def** detect\_blinks(eeg\_data, threshold):

"""

Detects eye blinks based on amplitude thresholding.

Parameters:

- eeg\_data: 1D array representing EEG data.

- threshold: amplitude threshold for detecting blinks.

Returns:

- blink\_times: indices where blinks are detected.

"""

blink\_times **=** np**.**where(np**.**abs(eeg\_data) **>** threshold)[0]

**return** blink\_times

In [72]:

distance\_between\_peaks **=** 128 *# Half of one second between peaks; adjust as needed*

window\_size **=** 256 *# 1 seconds window size*

window\_step **=** int(window\_size **\*** 0.7) *# 30% overlap*

blink\_times\_all\_subjects **=** [] *# Master list to store blink times for all subjects*

*# Process each epoch of the data*

**for** subject\_num, epoch\_data **in** enumerate(eyes\_blink\_Fz, start**=**1):

squared\_data **=** epoch\_data**\*\***2 *# Square the data for more resilience*

threshold **=** compute\_threshold(squared\_data) *# Compute threshold for the squared data*

all\_peaks **=** []

*# Slide the window through the data*

**for** start **in** range(0, len(squared\_data) **-** window\_size **+** 1, window\_step):

end **=** start **+** window\_size

windowed\_data **=** squared\_data[start:end]

*# Detect peaks in the windowed data*

peaks, \_ **=** find\_peaks(windowed\_data, height**=**threshold, distance**=**distance\_between\_peaks)

peaks **=** peaks **+** start *# Adjust peak indices for the entire data*

all\_peaks**.**extend(peaks)

*# Remove duplicate peaks*

all\_peaks **=** list(set(all\_peaks))

all\_peaks**.**sort()

*# Append the blink times for this epoch to the master list*

blink\_times\_all\_subjects**.**append(all\_peaks)

*# Print the number of detected blinks*

*# print(f"Detected {len(all\_peaks)} blinks in this epoch.")*

*# Visualize detected blinks*

plt**.**figure(figsize**=**(15, 3))

plt**.**plot(squared\_data)

plt**.**plot(all\_peaks, squared\_data[all\_peaks], "ro")

*# Set title to indicate the subject number*

plt**.**title(f"Detected Blinks for Subject {subject\_num}")

*# Annotate the number of peaks detected*

plt**.**annotate(f"Detected Blinks for Subject\_{subject\_num}: {len(all\_peaks)}",

xy**=**(0.95, 0.95), xycoords**=**'axes fraction',

fontsize**=**10,

xytext**=**(**-**5, **-**5), textcoords**=**'offset points',

ha**=**'right', va**=**'top')

plt**.**show()

*# Convert the master list to an array for convenience*

blink\_times\_array **=** np**.**array(blink\_times\_all\_subjects)

print(blink\_times\_array)

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

A graph with red dots

Description automatically generated

[list([196, 427, 2372, 2457, 2937, 3044, 3146, 3664, 3947, 4150, 4353, 4566, 5166, 5430, 5464, 5680, 6195, 6269, 6416, 6907, 7421, 7935, 8230, 8891, 9076, 9264, 9546, 9579, 9976, 10111, 10251, 10612, 10662, 10815, 11036, 11202, 11344, 11373, 11555, 11710, 11844, 11997, 12197, 12384, 12749, 12885, 12976, 13447, 13652, 13730, 13900, 14107, 14257, 14321, 14512, 15106])

list([71, 242, 640, 1040, 1505, 1766, 1838, 2013, 2729, 2881, 3096, 3138, 3683, 3893, 4395, 4611, 5261, 5477, 5581, 6381, 6748, 6981, 7659, 8169, 8382, 8531, 8663, 8671, 8777, 9185, 9335, 9959, 10107, 10255, 11380, 11739, 12157, 12420, 12823, 13245, 13703, 13897, 14461, 14668, 14750])

list([13, 249, 491, 727, 961, 1188, 1425, 1661, 1898, 2149, 2384, 2603, 2860, 2898, 3093, 3286, 3319, 3549, 3824, 4006, 4191, 4196, 4396, 4651, 4783, 4933, 5136, 5357, 5554, 5726, 5953, 5994, 6368, 6599, 6768, 6960, 7150, 7517, 7594, 8180, 8485, 8663, 9080, 9239, 9326, 9519, 9641, 9846, 10148, 10733, 11513, 12036, 12516, 14637, 14846, 15066, 15275])

list([194, 477, 541, 1144, 1475, 1723, 2000, 2158, 2315, 2340, 2542, 2757, 2930, 3168, 3340, 3530, 3752, 3963, 4177, 4370, 4392, 4555, 4767, 4836, 5024, 5380, 5597, 5795, 6016, 6392, 6594, 6642, 6862, 7224, 7258, 7589, 7630, 8236, 8372, 8423, 8793, 8963, 9340, 9700, 10090, 10326, 10443, 10661, 10807, 11021, 11169, 11349, 11560, 11755, 12013, 12184, 12652, 13047, 13299, 13533, 13740, 13801, 13958, 14216, 14252, 14627, 14792, 14987, 15217])

list([4, 1029, 2376, 2455, 2835, 2872, 3231, 3630, 3942, 4293, 4683, 5013, 5414, 5730, 6091, 6365, 6603, 6796, 6835, 7307, 7529, 7768, 7994, 8166, 8427, 8660, 9191, 9328, 9393, 9784, 10118, 10382, 10808, 11073, 11603, 11651, 12079, 12311, 12563, 12816, 13055, 13320, 13353, 13557, 13825, 14116, 14262, 14456, 14707])]

C:\Users\tnlab\AppData\Local\Temp\ipykernel\_24380\1085501306.py:47: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.

blink\_times\_array = np.array(blink\_times\_all\_subjects)

To validate the performance of the developed blink detector, we randomly selected five instances from the detected blinks. Topographic maps for these instances are presented. In most of these topographic representations, we observed heightened activity in the frontal area. This prominent frontal activity serves as evidence supporting the accuracy and effectiveness of our developed detector.

In [75]:

*# Validation*

*# Reshape the data to (n\_epochs, n\_channels, n\_times)*

Blink **=** np**.**transpose(Blink\_np, (0, 2, 1))

*# Create an EpochsArray object using your provided info*

events **=** np**.**array([[i, 0, 1] **for** i **in** range(Blink**.**shape[0])])

epochs **=** mne**.**EpochsArray(Blink, info, events**=**events)

*# Precompute the global minimum and maximum across subjects and blink times for consistent colormap scaling*

global\_vmin **=** np**.**inf

global\_vmax **=** **-**np**.**inf

**for** i **in** range(Blink**.**shape[0]):

random\_blink\_times **=** np**.**random**.**choice(blink\_times\_array[i], 5, replace**=False**)

**for** blink\_time **in** random\_blink\_times:

time\_point **=** blink\_time **/** epochs**.**info['sfreq']

data **=** epochs[i]**.**average()**.**data[:, epochs**.**time\_as\_index([time\_point])[0]]

global\_vmin **=** min(global\_vmin, data**.**min())

global\_vmax **=** max(global\_vmax, data**.**max())

fig, axs **=** plt**.**subplots(Blink**.**shape[0], 5, figsize**=**(15, 10))

**for** i **in** range(Blink**.**shape[0]):

*# Randomly select 5 blink times for this subject*

random\_blink\_times **=** np**.**random**.**choice(blink\_times\_array[i], 5, replace**=False**)

*# Convert sample points to time in seconds*

random\_time\_points **=** random\_blink\_times **/** epochs**.**info['sfreq']

*# Get data values for the randomly selected times and determine their min and max*

subject\_data **=** epochs[i]**.**average()**.**data *# This Evoked object contains the grand average (averaged over trials or epochs) of the data for each channel.*

subject\_values\_at\_times **=** [subject\_data[:, int(tp **\*** epochs**.**info['sfreq'])] **for** tp **in** random\_time\_points]

subject\_vmin **=** min([np**.**min(val) **for** val **in** subject\_values\_at\_times])

subject\_vmax **=** max([np**.**max(val) **for** val **in** subject\_values\_at\_times])

*# For each of these blink times*

**for** j, time\_point **in** enumerate(random\_time\_points):

*# Plot the topomap for this time point*

data\_at\_time **=** subject\_data[:, int(time\_point **\*** epochs**.**info['sfreq'])]

im1, \_ **=** mne**.**viz**.**plot\_topomap(data\_at\_time, epochs**.**info, cmap**=**'jet', axes**=**axs[i, j], show**=False**)

im1**.**set\_clim(subject\_vmin, subject\_vmax)

axs[i, j]**.**set\_title(f"Subject {i**+**1}, Time: {time\_point:.2f}s")

fig**.**colorbar(im1, ax**=**axs[i, 4], orientation**=**'vertical', pad**=**0.05)

plt**.**tight\_layout()

plt**.**show()

Not setting metadata

5 matching events found

No baseline correction applied

0 projection items activated

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

NOTE: pick\_channels() is a legacy function. New code should use inst.pick(...).

A group of colorful circles with text

Description automatically generated