Assignment 2

November 5, 2024

s234806

s234843

1 Assignment 2 EEG Data Analysis

This assignment is an exploration into a dataset of EEG data on patients. The best description given by the authors of the original data is "Resting state EEG with closed eyes and open eyes in females from 60 to 80 years old", found here: 'https://openneuro.org/datasets/ds005420/versions/1.0.0'. The data contains EEG measurements using 20 electrodes where the subjects where asked to keep their eyes open (oa) and then closed (oc) for around 4-5 minutes. For the purposes of this exercise we have taken a subset of 8 subjects using only 200 seconds of each of oc and oc to align the sizes.

The main aim is to give you an application of the Discrete Fourier Transform on real world data. As this step can be done in a single function a lot of the time spent on this exercise will be on visually inspecting the data and trying different classifiers on the data you have produced. If you are lucky enough to draw this assignment for your exam you should be prepared to answer mainly theoretical questions about the Fourier Transform as covered in the exercises and lectures from week 5&6.

Apart from the usual libraries you have been using in this course, you will have to install mne via pip or conda.

[1]:

```
import os
import mne
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import scipy.signal as signal

from os.path import join

import pandas as pd
from tqdm.notebook import tqdm
```

[2]:

```
path_data = "csv"
data_oa_original = []
data_oc_original = []
sample_rate = 50
```

```
sample_rate_original = 250

for folder in tqdm(os.listdir(path_data), desc="Loading data"):
    for filename in os.listdir(join(path_data, folder)):
        df = pd.read_csv(join(path_data, folder, filename), sep=",", index_col=0)
        if "oa" in filename:
            data_oa_original.append(df.values)
        else:
            data_oc_original.append(df.values)
```

```
Loading data:   0%|          | 0/8 [00:00<?, ?it/s]
```

[3]:

```
print(F"Loaded data from {len(data_oa_original)} patients")
```

```
Loaded data from 8 patients
```

1.1 Exercise 1

1.1.1 1. Let's take a look at the data by investigating the shape of it. What is the maximum frequency we can represent given the length?

Hint: Remember to factor in the sample rate

[4]:

```
for idx, (d_oc, d_oa) in enumerate(zip(data_oc_original, data_oa_original)):
    print(f"Shape of OC: {d_oc.shape}, Shape of OA: {d_oa.shape}")
```

```
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
Shape of OC: (20, 50000), Shape of OA: (20, 50000)
```

The shape of the data-matrices indicate that there are 50.000 frequencies for each of the 20 electrodes, for both scenarios (oa and oc). Thus we can calculate the sampling rate by dividing the amount of frequencies with the observation period of 200 seconds.

$\frac{50000}{200\text{ s}} = 250$ Hz

Then the maximum frequency we can capture according to the Shannon Nyquist theorem is half the sampling frequency, which is 125 Hz.

Now we want to center all the data around the same value (0), in eeg data analysis terms this is called referencing, and down sample in order to have a manageable size that a normal pc with 16 Gb of RAM can manipulate. Here we will apply a simple average referencing

1.1.2 2. Implement the following function:

[5]:
```
def set_reference_and_downsample(eeg_data, ref_channels, sample_rate=50):
    """Take the mean of the reference channels and subtract that from the␣
  ↪data"""
    eeg_masked = eeg_data[ref_channels] # samples 1/5 of the frequencies (50␣
  ↪instead of 250)
    eeg_down_sampled = signal.resample(eeg_masked, eeg_masked.shape[1] //␣
  ↪(sample_rate_original // sample_rate), axis=1)

    ref_signal = np.mean(eeg_down_sampled, axis=0)
    re_referenced_data = eeg_down_sampled - ref_signal
    return re_referenced_data
```

We will reference and down sample each patient's data individually:

[6]:
```
data_oc = []
data_oa = []
for i in range(len(data_oa_original)):
    data_oc.append(set_reference_and_downsample(data_oc_original[i], np.
  ↪arange(20), sample_rate=sample_rate))
    data_oa.append(set_reference_and_downsample(data_oa_original[i], np.
  ↪arange(20), sample_rate=sample_rate))
```

[7]:
```
for idx, (d_oc, d_oa) in enumerate(zip(data_oc, data_oa)):
    print(f"Shape of OC: {d_oc.shape}, Shape of OA: {d_oa.shape}")
```

```
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
Shape of OC: (20, 10000), Shape of OA: (20, 10000)
```

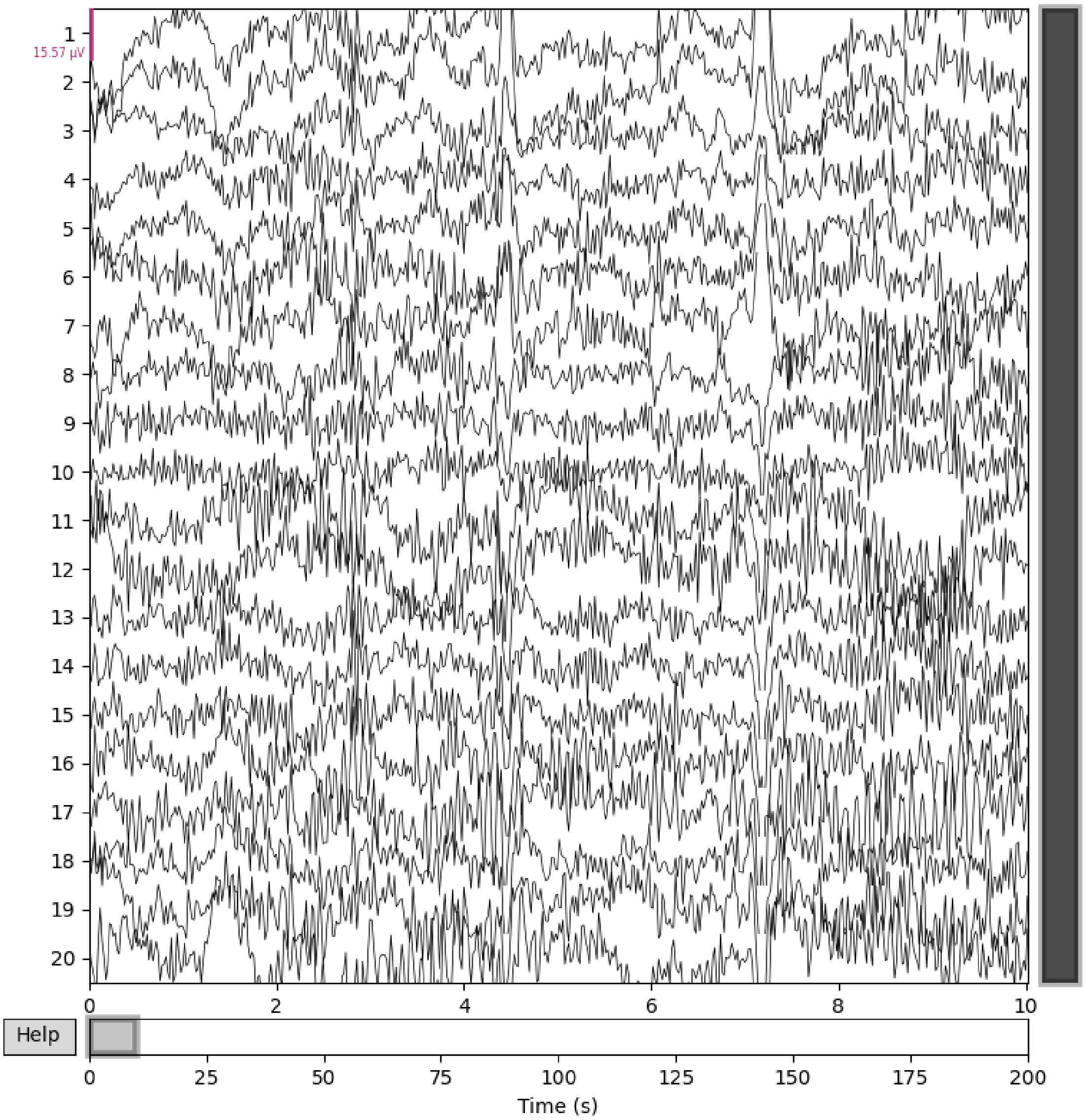
1.1.3 3. Using the following plotting function, have a look at the data and comment on the quality of it:

[8]:
```
def plot_eeg_mne_style(eeg_data, ch_names, sampling_rate):
    """
    Plot EEG data in an MNE-style plot for each channel.

    Parameters:
    eeg_data : np.ndarray
        The EEG data with shape (n_channels, n_points).
    ch_names : list of str
        List of channel names (must be of length n_channels).
    sampling_rate : float
        The sampling rate of the EEG data (in Hz).
    """
    # Create MNE info structure with channel names and sampling rate
    info = mne.create_info(ch_names=ch_names, sfreq=sampling_rate, ch_types='eeg')

    # Create MNE RawArray object with the data
    raw = mne.io.RawArray(eeg_data, info)

    # Plot the EEG data with MNE's built-in plot function
    raw.plot(scalings='auto', show=True, block=True)
    plt.show()
```

[9]:

```
channel_names = np.arange(1,21,1).astype(str).tolist()
plot_eeg_mne_style(data_oa[4], ch_names=channel_names, sampling_rate=sample_rate)
```

```
Creating RawArray with float64 data, n_channels=20, n_times=10000
    Range : 0 … 9999 =      0.000 …   199.980 secs
Ready.
Using matplotlib as 2D backend.
```



Our first impression of the plot is that it appears very messy, and it was not immediately apparent what was being displayed. After some initial inspection, we found that the plot shows the time-domain signals for all the electrodes. The vertical scale represents different channels or electrode placements (labeled from 1 to 20), while the x-axis represents time. This means each line represents the signal amplitude (or "voltage") for a particular channel over time, rather than individual frequencies as in a Spectrogram (in the frequency-domain). However from our interpretation of the plot, the horizontal range of 0-10, denoted on the x-axis, represents a fixed 10-second window reference for which the eeg-data is displayed.

This means we would need to issue multiple plots to see the rest of the data - or apply an interactive panning feature as indicated by the grey bar in the current plot. This fact alone makes the plot less than ideal for obtaining a quick overview of the entire period. Additionally, some parts of the signals are cut off in the bottom and in the top of the plot, which indicates that there could be missing values (although insignificant).

In regards to the waveform content, the frequencies are implicit in the signal but are not individually displayed as they would be in a spectrogram or Fourier transform, where the signal is decomposed into its frequency components. A spectrogram would allow analysts to visualize how different frequency bands change over time, whereas this plot only shows raw amplitude data for each channel.

This plot gives a good overview of activity across channels, which can help in recognizing certain time-based patterns or synchronized events across channels. However, it does not appear very effective for identifying patterns at specific frequencies without further processing, and it does not alone, in it's current state, give an overview of the entire 200 second period.

Inspecting the contents of the plot, we see that the rate of amplitude fluctuations increases briefly around 3 seconds, which makes the waveform look darker. This implies that some event happened around that time, causing specific brain activities which were captured by multiple of the electrodes.

1.2 Exercise 2 Discrete Fourier Transform

Given the sample rate and duration of the EEG signals we have just loaded, complete this exercise using the signal from the first channel of the first OC patient.

1.2.1 1. Using the formula for fourier coefficients: $c_k = \sum_{n=0}^{N-1} x(n)e^{-i2\pi\frac{kn}{N}}$, calculate the fourier coefficient for the signal corresponding to $k = 25$

[10]:
```
data_oc[0][0]
```

[10]:
```
array([7.43743201e-06, 1.12972772e-05, 6.10790679e-06, …,
       4.59394583e-06, 7.48215372e-06, 5.54689676e-06])
```

[11]:
```
N = 10000
fs = 50
k = 25
coeff_k = 0
for n, x_n in enumerate(data_oc[0][0]):
    coeff_k += x_n * np.exp(-2j*np.pi*k*n/N)
print(f"Fourier coefficient {k} = {coeff_k:.3}")
print(f"Absolute value = {np.abs(coeff_k):.3}")
```

```
Fourier coefficient 25 = (0.00156+0.00306j)
Absolute value = 0.00343
```

This means that the Fourier coefficient $c_{25} = 0.00343$

1.2.2 2. Calculate the frequency that the fourier coefficient for $k = 25$ corresponds to

To find the frequency of the 25th frequency bin, we can use the general formula $f_k = k \cdot \frac{f_s}{N}$

[12]:
```
f25 = k*fs/N
f25
```

```
[12]: 0.125
```

1.2.3 3. Calculate the amplitude of the frequency that corresponds to the fourier coefficient for $k = 1$

The magnitude of the Fourier coefficient corresponds to the contribution of each frequency, hence we can begin by just computing the magnitude of the coefficient c_1.

[13]:

```
N = 10000
fs = 50
k = 1
coeff_k = 0
for n, x_n in enumerate(data_oc[0][0]):
    coeff_k += x_n * np.exp(-2j*np.pi*k*n/N)
c1 = abs(coeff_k)
c1
```

```
[13]: 0.00012174365460098814
```

However since we have N frequency bins, we need to normalize the magnitude of the coefficient by dividing by N, to obtain the actual amplitude of the frequency corresponding to $k = 1$.

[14]:

```
a1 = c1/N
a1
```

```
[14]: 1.2174365460098813e-08
```

1.3 Exercise 3 Frequency Domain and Filtering

In this exercise you will take a time domain filter, plot its frequency response and use it to filter the data. Let's create the orthogonal basis matrix, but here in a vectorised function because otherwise it'd take too long.

[15]:

```
def create_basis(N, fs):
    # Ensure N is odd
    assert N % 2 == 1

    ns = np.arange(N)
    n_pairs = int((N - 1) / 2)

    freqs = np.arange(int(0.5 * N) + 1) * fs / N

    # Create cosine and sine terms for all frequencies at once
    ks = np.arange(1, n_pairs + 1)
    angles = 2 * np.pi * np.outer(ns, ks) / N

    cosines = np.sqrt(2) * np.cos(angles)
    sines = np.sqrt(2) * np.sin(angles)
    # Create basis matrix with first column of ones, then interleave cosines and sines
    V = np.ones((N, N))
    V[:, 1::2] = cosines  # Fill odd-indexed columns with cosines
    V[:, 2::2] = sines    # Fill even-indexed columns with sines

    # Normalize by sqrt(N)
    V /= np.sqrt(N)

    return freqs, V

def compute_spectrum(y, V, scale=True):
    z = V.T@y

    z_cos = z[2::2]
    z_sin = z[1::2]
    amp_spectrum = np.sqrt(z_cos**2 + z_sin**2)
    amp_spectrum = np.hstack((np.abs(z[0]), amp_spectrum))

    if scale:
        scales = np.ones(len(amp_spectrum))
        scales[0]  = 1
        scales[1:] = np.sqrt(2)


        return amp_spectrum/np.sqrt(len(y))*scales
    else:
        return amp_spectrum
```

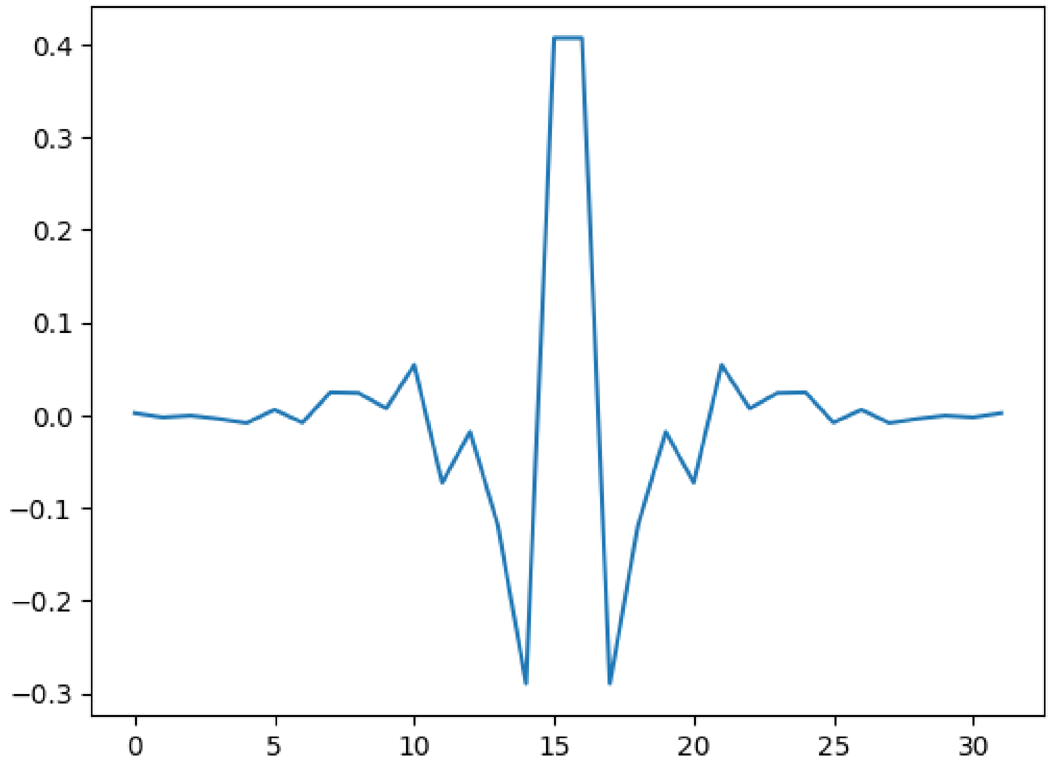
[16]:

```
N_odd = N-1
# prepare basis matrix V
freqs, V = create_basis(N-1, sample_rate)
```

1.3.1 1. Below you are given a time domain filter, plot it and comment on its shape. How do you think it does its filtering?

```
[17]: h = np.array([0.00206512, -0.00246738, -0.00061028, -0.00412665, -0.00860358, 0.
        ↪00576032,
                        -0.00815993, 0.02454267, 0.0238489, 0.00708215, 0.05402169, -0.
        ↪07304967,
                        -0.01784375, -0.11967235, -0.28945082, 0.40677716, 0.40677716,␣
        ↪-0.28945082,
                        -0.11967235, -0.01784375, -0.07304967, 0.05402169, 0.00708215,␣
        ↪0.0238489,
                        0.02454267, -0.00815993, 0.00576032, -0.00860358, -0.00412665,␣
        ↪-0.00061028,
                        -0.00246738, 0.00206512])
```

```
[18]: plt.plot(h)
      plt.show()
```



Since we are dealing with a time-domain filter, we know that time is on the x-axis and the signal amplitude is on the y-axis. We see that the filters average value $\bar{h} \approx 0$ and only has non-zero values within the range [0:30].

```
[ ]: #mean value
     h.mean()
```

```
[ ]: 7.10000000000207e-06
```

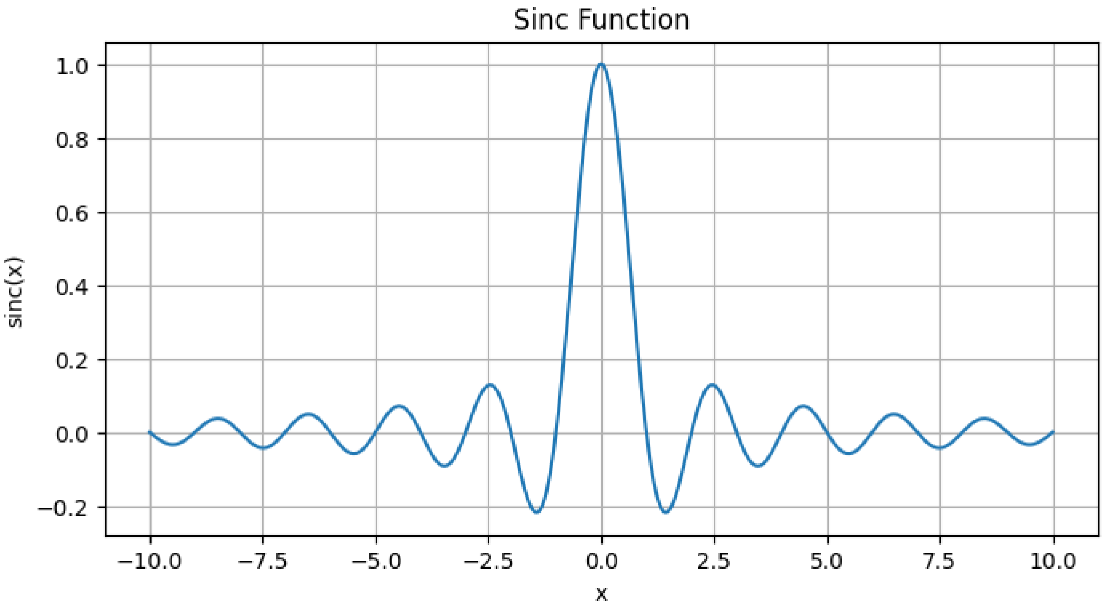
This means that when the filter is applied to a signal using convolution, the filter's influence at each point in time decays to near zero outside this range. Beyond that, the effect of the impulse diminishes.

As for the effect of the filter, high-pass filters tend to have more rapid changes, alternating around the center, to cancel out low-frequency components, and band-pass filters are designed to pass a certain band of frequencies, and typically also show a central peak with oscillations, though usually more pronounced. Whereas low-pass filters tend to have a large central peak with gradually diminishing oscillations on either side, which aligns more with the plot above.

The impulse response of our filter h also resembles something very similar to a a sinc function, which is a popular low-pass filter.

$sinc(x) = \frac{sin(x)}{x}$

```
[ ]: x = np.linspace(-10, 10, 1000)
     y = np.sinc(x)  # np.sinc(x) is normalized, representing sin(pi*x)/(pi*x)

     plt.figure(figsize=(8, 4))
     plt.plot(x, y)
     plt.title("Sinc Function")
     plt.xlabel("x")
     plt.ylabel("sinc(x)")
     plt.grid(True)
     plt.show()
```



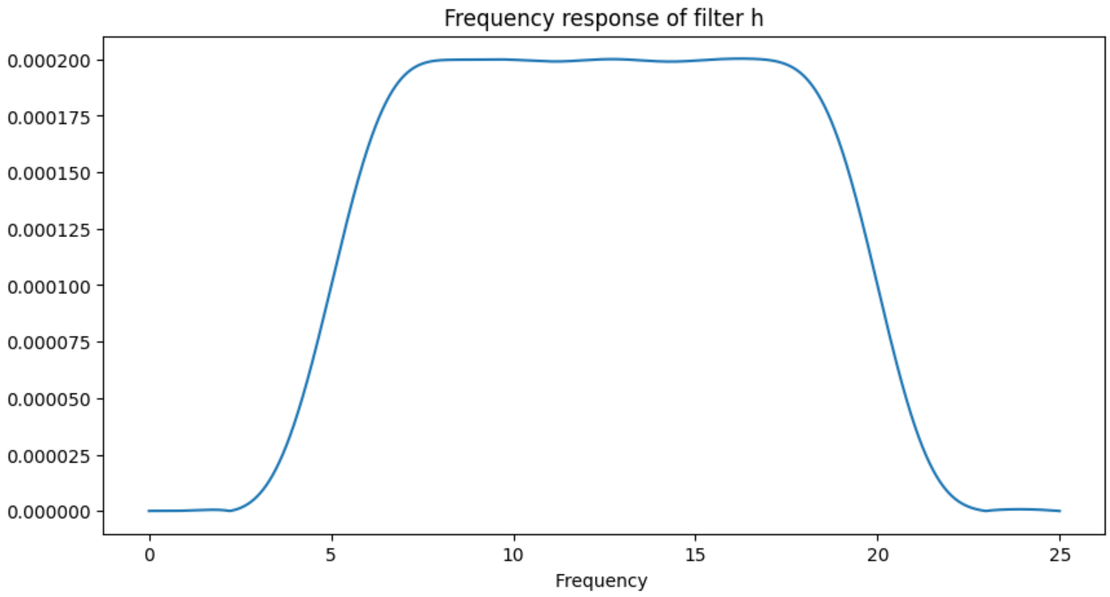
Therefore we believe the filter h is a low-pass filter, that emphasizes low frequencies and supresses higher frequencies during the convolution.

1.3.2 2.Using the function that computes the spectrum of a signal to look at the frequency response of the filter? How do you make the dimensions match between the filter and the orthogonal basis matrix?

Hint: use padding

Since the basis matrix is of shape $N \times N$, the filter has to be of length N as well. Therefore we apply zero-padding to construct the $\tilde{h}$ filter.

[19]:
```python
htilde = np.zeros(N-1)
htilde[0:len(h)] = h

plt.figure(figsize=(10, 5))
plt.plot(freqs, compute_spectrum(htilde, V))
plt.xlabel('Frequency ')
plt.title('Frequency response of filter h')
plt.show()
```



⋆ *In the following cell we the time domain filter, this is outside the scope of the course, but a very interesting topic to look into if you have the time. In addition there is a simpler function for calculating the frequency response:)*

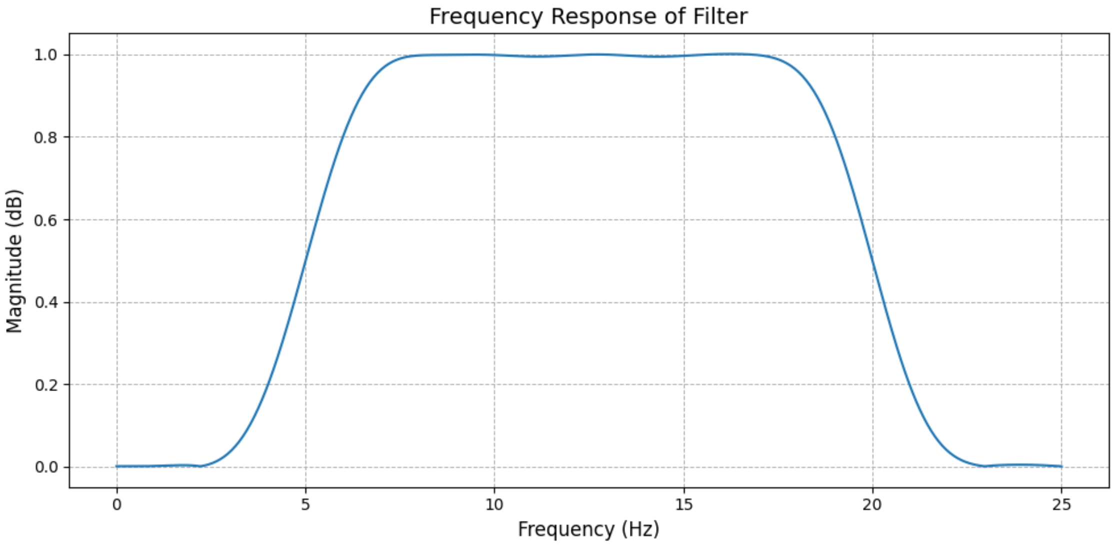
[20]:
```python
def create_bandpass_kernel(lowcut, highcut, fs, numtaps=32):
    nyquist = 0.5 * fs  # Nyquist frequency
    low = lowcut / nyquist
    high = highcut / nyquist
    # Design a bandpass FIR filter
    kernel = signal.firwin(numtaps, [low, high], pass_zero=False)

    return kernel

def frequency_response(filter_kernel, fs, padding=1000):
    # Compute the FFT of the filter kernel (rfft for real inputs)
    freq_response = np.abs(np.fft.rfft(filter_kernel, padding))

    # Compute the corresponding frequency axis
    freqs = np.abs(np.fft.rfftfreq(padding, 1/fs))

    return freqs, freq_response
print(create_bandpass_kernel(5, 20, fs, numtaps=32))
```

```
[ 0.00206512 -0.00246738 -0.00061028 -0.00412665 -0.00860358  0.00576032
 -0.00815993  0.02454267  0.0238489   0.00708215  0.05402169 -0.07304967
 -0.01784375 -0.11967235 -0.28945082  0.40677716  0.40677716 -0.28945082
 -0.11967235 -0.01784375 -0.07304967  0.05402169  0.00708215  0.0238489
  0.02454267 -0.00815993  0.00576032 -0.00860358 -0.00412665 -0.00061028
 -0.00246738  0.00206512]
```

And here is a function to plot the frequency response of the filter using the above function:

[21]:

```
freqs_h, freq_response_h = frequency_response(h, sample_rate)

plt.figure(figsize=(10, 5))
plt.plot(freqs_h, freq_response_h)
plt.xlabel('Frequency (Hz)', fontsize=12)
plt.ylabel('Magnitude (dB)', fontsize=12)
plt.title('Frequency Response of Filter', fontsize=14)
plt.grid(True, which='both', linestyle='--', linewidth=0.7)
plt.tight_layout()
plt.show()
```



1.3.3 2. Implement a function that filters a patients eeg signal in the time domain:

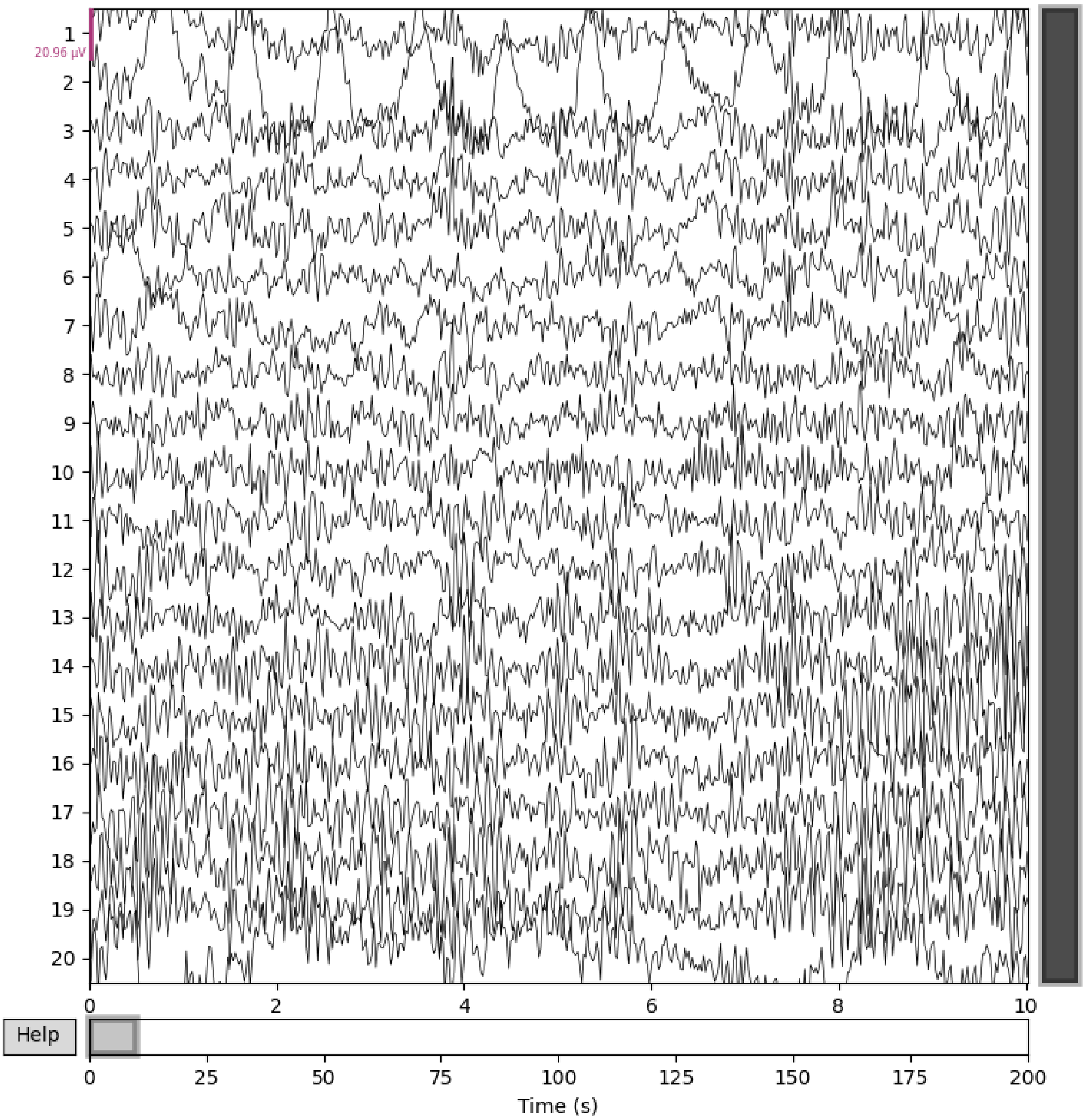
- Compare the data before and after filtering
- Try different thresholds like (1 to 3) and (40 to 70) and comment on the effects

The bandpass_filter function takes data from a single patient as input, along with the padded filter $\tilde{h}$. This means that the following function applies the filter to every frequency channel of a single patients data, using convolution.

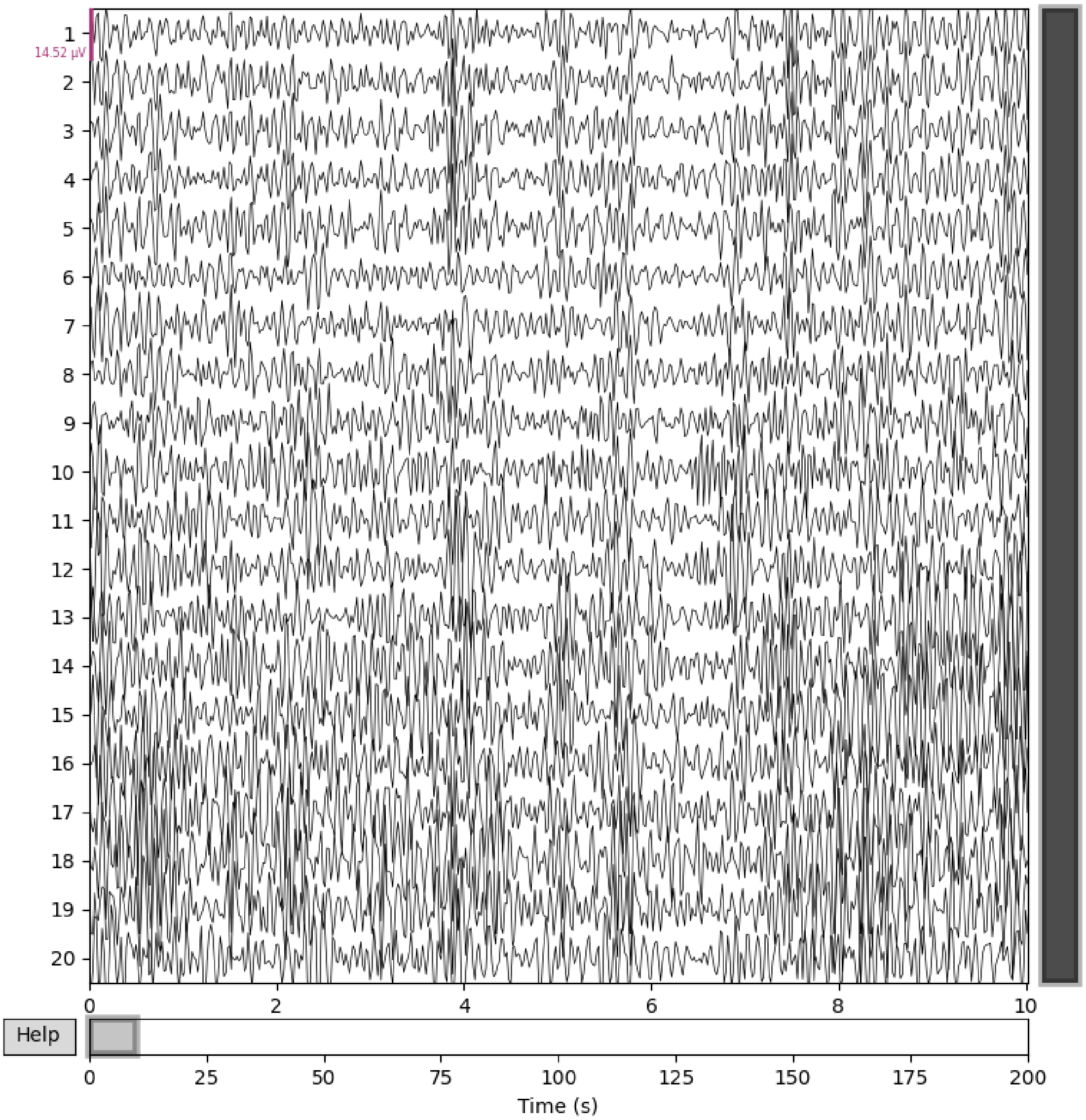
[22]:
```
def bandpass_filter(signal_data, kernel):
    output = []
    for channel in signal_data:
        filtered_signal = np.convolve(channel, kernel, mode='same')  # Convolve␣
 ↪with the signal
        output.append(filtered_signal)
    output = np.array(output)
    return output
```

[23]:
```
filtered_channels = bandpass_filter(data_oc[0], h)
plot_eeg_mne_style(data_oc[0], ch_names=channel_names,␣
 ↪sampling_rate=sample_rate)
plot_eeg_mne_style(np.array(filtered_channels), ch_names=channel_names,␣
 ↪sampling_rate=sample_rate)
```

```
Creating RawArray with float64 data, n_channels=20, n_times=10000
    Range : 0 … 9999 =      0.000 …   199.980 secs
Ready.
```



```
Creating RawArray with float64 data, n_channels=20, n_times=10000
    Range : 0 … 9999 =      0.000 …   199.980 secs
Ready.
```



⋆ *The function below does what we have just implemented in two separate functions, feel free to use it:)*

[24]:

```python
def bandpass_filter_butter(data, lowcut, highcut, fs, order=4):
    nyquist = 0.5 * fs  # Nyquist frequency
    low = lowcut / nyquist
    high = highcut / nyquist

    b, a = signal.butter(order, [low, high], btype='band')
    filtered_data = signal.filtfilt(b, a, data)

    return filtered_data
```

1.3.4 3. Complete the following function which takes a signal and computes its power spectrum and removes frequencies outside the min and max in order to remove redundant information after filtering

- Discuss the difference between computing the power spectrum of a signal and the frequency response of a filter

The power spectrum displays how the power of the signal is distributed amongst the frequencies present.

For a specific frequency bin k, the power at that frequency is simply $P(k) = |X(k)|^2$.

If we needed the total power across all frequencies, we could sum over the power values for each frequency bin:

$P_{\text{total}} = \sum_i |X(f_i)|^2$

[ ]:

```python
def compute_power_spectrum(signal, sampling_rate, min_freq=None, max_freq=None):
    """
    Computes the power spectrum for each channel in the signal.

    Parameters:
    signal : np.ndarray
        Input signal with shape (n_channels, n_points)
    sampling_rate : float
        The sampling rate of the signal (in Hz).

    Returns:
    freqs : np.ndarray
        The frequencies corresponding to the power spectrum.
    power_spectrum : np.ndarray
        Power spectrum of each channel with shape (n_channels, n_points//2).
    """
    n_channels, n_points = signal.shape

    # Perform the FFT for each channel
    fft_vals = np.fft.rfft(signal, axis=1)

    # Compute the power spectrum (squared magnitude of the FFT)
    power_spectrum = np.abs(fft_vals) ** 2

    # Compute the corresponding frequencies
    freqs = np.fft.rfftfreq(n_points, 1 / sampling_rate)

    if min_freq is not None:
        mask = freqs >= min_freq
        power_spectrum = power_spectrum[:,mask]
        freqs = freqs[mask]

    if max_freq is not None:
        mask = freqs <= max_freq
        power_spectrum = power_spectrum[:,mask]
        freqs = freqs[mask]

    return freqs, power_spectrum
```

This next function is just for plotting

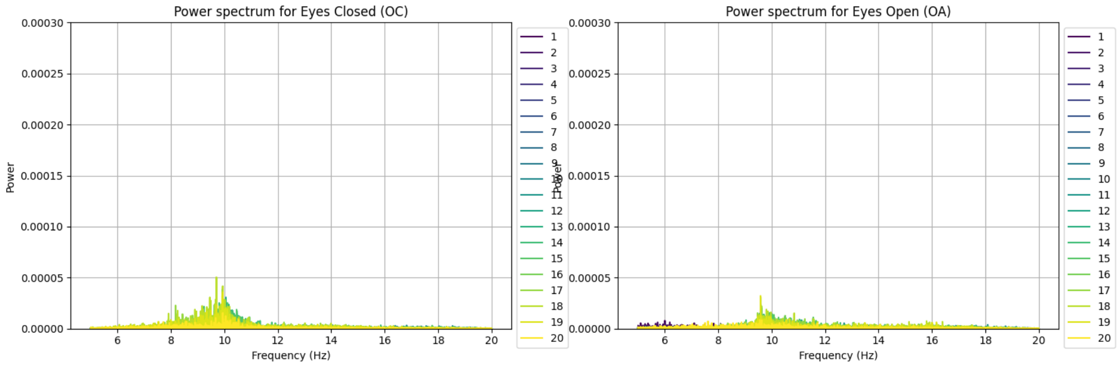
[26]:

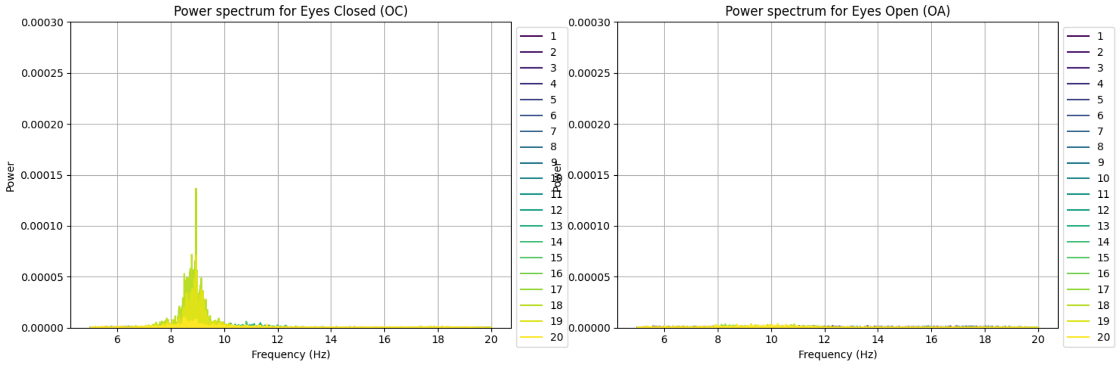
```
import matplotlib.cm as cm

def plot_power_spectrum_overlay(freqs, power_spectrum, ch_names, show=False,␣
 ↪title="", show_label=False, show_color=False):
    """
    Plot the power spectrum for all channels on the same plot, with brain wave␣
 ↪frequency bands highlighted.

    Parameters:
    freqs : np.ndarray
        Array of frequency values.
    power_spectrum : np.ndarray
        Power spectrum data with shape (n_channels, n_freqs).
    ch_names : list of str
        List of channel names.
    show : bool
        Whether to display the plot immediately.
    title : str
        Title of the plot.
    """


    # Plot the power spectrum for each channel
    colors = cm.viridis(np.linspace(0, 1, len(ch_names)))
    for i, ch_name in enumerate(ch_names):
        plt.plot(freqs, power_spectrum[i], color=colors[i], label=(ch_name if␣
 ↪show_label else ""))

    if show_color:
        # Define brain wave frequency bands and their corresponding colors
        bands = {
            'Theta (4-8 Hz)': (4, 8, 'green', 0.1),
            'Alpha (8-13 Hz)': (8, 13, 'yellow', 0.1),
            'Beta (13-30 Hz)': (13, 20, 'orange', 0.1)
        }

        # Highlight the brain wave frequency bands
        for band, (start_freq, end_freq, color, alpha) in bands.items():
            plt.axvspan(start_freq, end_freq, color=color, alpha=alpha,␣
 ↪label=band)

    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Power')
    plt.legend(bbox_to_anchor=(1, 1), loc='upper left')
    plt.grid(True)
    plt.title('Power Spectrum for EEG Channels' if not title else title)
    plt.tight_layout()
    if show:
        plt.show()
```

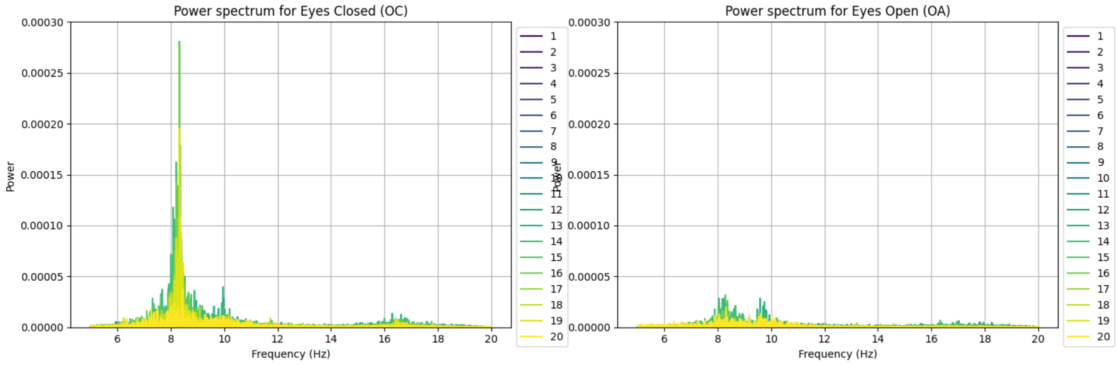
We have been advised by Ivana (Human Cognition lecturer and EEG researcher), that we should take a look at the frequencies between 5 and 20 Hz in order to get a distinction between OC and OA. Here we plot all the power spectra from each channel on top of one each other. Feel free to investigate other indexes.

[27]:

```
patient_indexes = [0, 3, 5]
min_freq=5
max_freq=20
power_spectra_oc = []
power_spectra_oa = []
y_max = 3e-4

for segment in patient_indexes:
    plt.subplots(1, 2, figsize=(15, 5))

    data_oc_time_filtered = bandpass_filter(data_oc[segment], h)
    data_oa_time_filtered = bandpass_filter(data_oa[segment], h)
    # Compute power spectra for both conditions
    freqs_oc, power_spectrum_oc = compute_power_spectrum(data_oc_time_filtered,␣
 ↪sampling_rate=sample_rate, min_freq=min_freq, max_freq=max_freq)
    freqs_oa, power_spectrum_oa = compute_power_spectrum(data_oa_time_filtered,␣
 ↪sampling_rate=sample_rate, min_freq=min_freq, max_freq=max_freq)
    power_spectra_oc.append((freqs_oc, power_spectrum_oc))
    power_spectra_oa.append((freqs_oa, power_spectrum_oa))

    plt.subplot(1, 2, 1)
    plot_power_spectrum_overlay(freqs_oc, power_spectrum_oc, channel_names,␣
 ↪title=f"Power spectrum for Eyes Closed (OC)", show_label = True)
    plt.ylim([0, y_max])
    plt.subplot(1, 2, 2)
    plot_power_spectrum_overlay(freqs_oa, power_spectrum_oa, channel_names,␣
 ↪title=f"Power spectrum for Eyes Open (OA)", show_label = True)
    plt.ylim([0, y_max])
    plt.show()
```







Using the same color coding as above, we have divided the frequencies from 4 to 20 hz into three ranges, according to the table below:

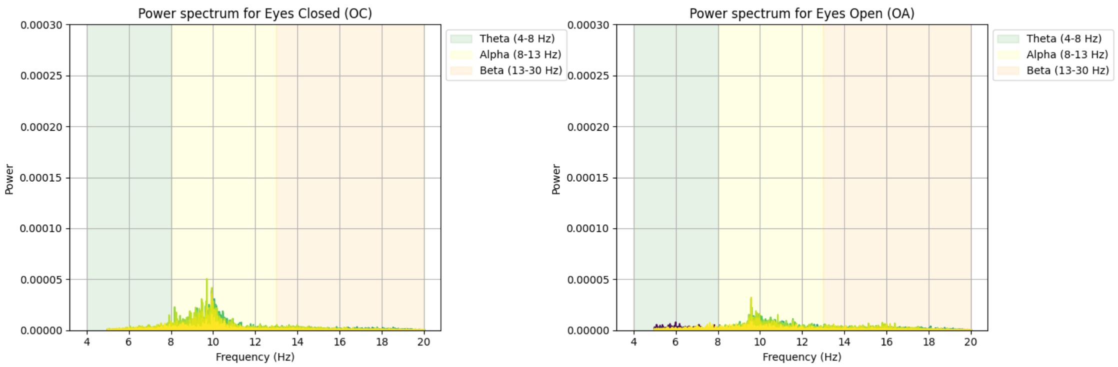
Some sources suggest these ranges:

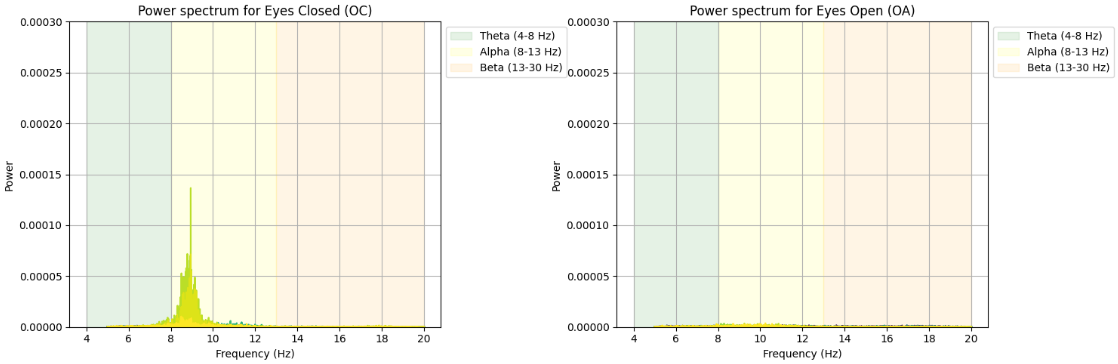
Brain Wave	Frequency Range	Characteristics
Delta ( )	0.3 - 4 Hz	Deep sleep, unconscious mind, restorative sleep
Theta ( )	4 - 8 Hz	Light sleep, relaxation, creativity, meditation

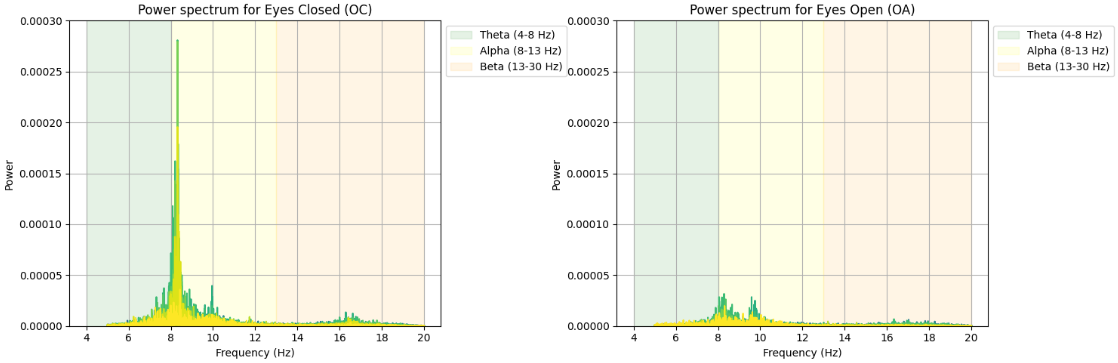
Brain Wave	Frequency Range	Characteristics
Alpha ( )	8 - 13 Hz	Calm, relaxed but alert, quiet focus
Beta ( )	13 - 30 Hz	Active thinking, focus, problem-solving
Gamma ( )	30 Hz and above	High-level information processing, cognitive functioning

[28]:

```python
patient_indexes = [0, 3, 5]
min_freq=5
max_freq=20
power_spectra_oc = []
power_spectra_oa = []
y_max = 3e-4

for segment in patient_indexes:
    plt.subplots(1, 2, figsize=(15, 5))

    data_oc_time_filtered = bandpass_filter(data_oc[segment], h)
    data_oa_time_filtered = bandpass_filter(data_oa[segment], h)
    # Compute power spectra for both conditions
    freqs_oc, power_spectrum_oc = compute_power_spectrum(data_oc_time_filtered, sampling_rate=sample_rate, min_freq=min_freq, max_freq=max_freq)
    freqs_oa, power_spectrum_oa = compute_power_spectrum(data_oa_time_filtered, sampling_rate=sample_rate, min_freq=min_freq, max_freq=max_freq)
    power_spectra_oc.append((freqs_oc, power_spectrum_oc))
    power_spectra_oa.append((freqs_oa, power_spectrum_oa))

    plt.subplot(1, 2, 1)
    plot_power_spectrum_overlay(freqs_oc, power_spectrum_oc, channel_names, title=f"Power spectrum for Eyes Closed (OC)", show_color = True)
    plt.ylim([0, y_max])
    plt.subplot(1, 2, 2)
    plot_power_spectrum_overlay(freqs_oa, power_spectrum_oa, channel_names, title=f"Power spectrum for Eyes Open (OA)", show_color = True)
    plt.ylim([0, y_max])
    plt.show()
```







1.3.5 2. Using the loop above take a look at the spectrograms for eyes open and eyes closed.

- Is there a noticeable difference between the two? How could we use this difference for classification?
- Discuss whether the conditions of our fft are sufficient to avoid spectral leakage

The above three plots show the power spectrum for the eeg measurement for patient 0, 3 and 5.

In the three plots when the patients' eyes were closed, the most common frequencies were alpha and waves. This is expected as alpha waves are common in states of calm, as displayed in the table above.

When the patients opened their eyes, their frequency range didn't change much, however the power of the alpha waves was significantly reduced. This is again expected, as their state of relaxation was interrupted.

We entirely believe it would be possible to classify whether a person has open or closed eyes based on these data, as there appears to be a significant difference in the power of the frequencies correlated with their physical state, depending on their true state. This could probably be done in various different ways, both using logistic regression and classification trees as well as more complex models, such as neural networks.

Since our signal has an integer number of cycles ($N = 10.000$) within the FFT window, the signal will not be subject to spectral leakage. This phenomenon only occurs when the number of cycles are non-integer, as the Fourier analysis assumes the signal repeats discontinuously, creating artificial jumps at the window edges. These discontinuities introduce additional frequency components that spread out around the main frequency.

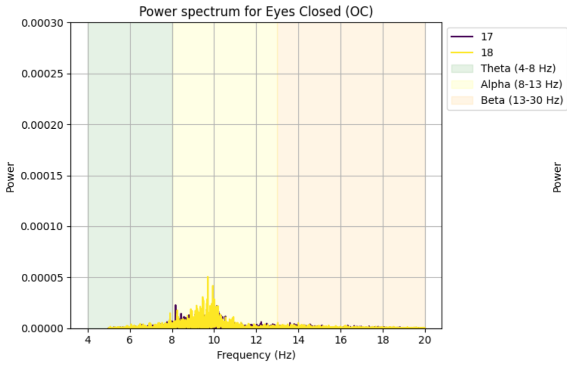
1.3.6 3. The channels at index 17 and 18 are known to be correlated to have open vs closed eyes, plot these:

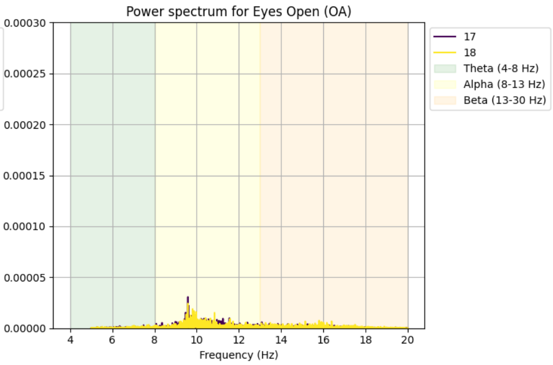
[29]:

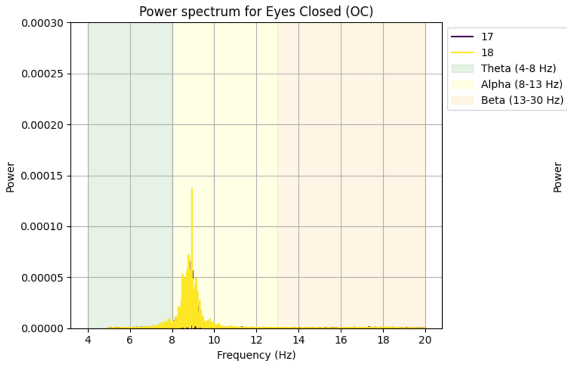
```
optical_channel_slice = slice(16,18)     # Look it up if in doubt as to how a␣
 ↪slice works
data_oc_optical_channels = [arr[optical_channel_slice] for arr in data_oc]
data_oa_optical_channels = [arr[optical_channel_slice] for arr in data_oa]
```

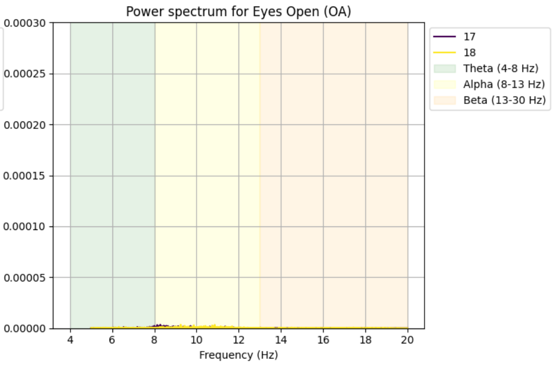
[30]:

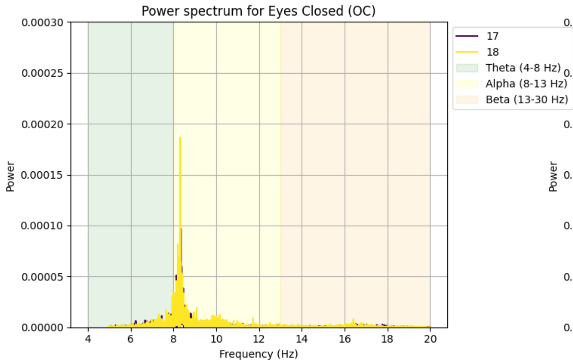
```
patient_indexes = [0, 3, 5]
min_freq=5
max_freq=20

for segment in patient_indexes:
    plt.subplots(1, 2, figsize=(15, 5))

    data_oc_time_filtered = bandpass_filter(data_oc_optical_channels[segment],␣
 ↪h)
    data_oa_time_filtered = bandpass_filter(data_oa_optical_channels[segment],␣
 ↪h)
    # Compute power spectra for both conditions
    freqs_oc, power_spectrum_oc = compute_power_spectrum(data_oc_time_filtered,␣
 ↪sampling_rate=sample_rate, min_freq=min_freq, max_freq=max_freq)
    freqs_oa, power_spectrum_oa = compute_power_spectrum(data_oa_time_filtered,␣
 ↪sampling_rate=sample_rate, min_freq=min_freq, max_freq=max_freq)

    plt.subplot(1, 2, 1)
    plot_power_spectrum_overlay(freqs_oc, power_spectrum_oc,␣
 ↪channel_names[optical_channel_slice], title=f"Power spectrum for Eyes Closed␣
 ↪(OC)", show_label = True, show_color = True)
    plt.ylim([0, y_max])
    plt.subplot(1, 2, 2)
    plot_power_spectrum_overlay(freqs_oa, power_spectrum_oa,␣
 ↪channel_names[optical_channel_slice], title=f"Power spectrum for Eyes Open␣
 ↪(OA)", show_label = True, show_color = True)
    plt.ylim([0, y_max])
    plt.show()
```

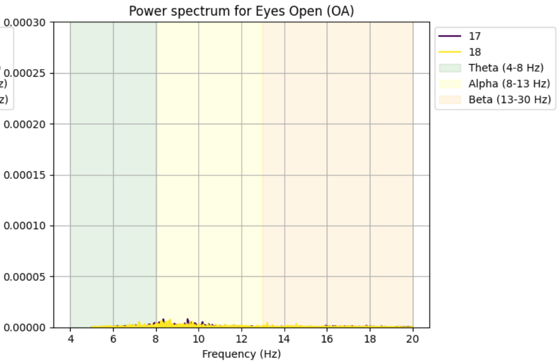

Power spectrum for Eyes Closed (OC)
Power
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
4
6
8
10
12
14
16
18
20
Frequency (Hz)
17
18
Theta (4-8 Hz)
Alpha (8-13 Hz)
Beta (13-30 Hz)


Power spectrum for Eyes Open (OA)
Power
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
4
6
8
10
12
14
16
18
20
Frequency (Hz)
17
18
Theta (4-8 Hz)
Alpha (8-13 Hz)
Beta (13-30 Hz)


Power spectrum for Eyes Closed (OC)
Power
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
4
6
8
10
12
14
16
18
20
Frequency (Hz)
17
18
Theta (4-8 Hz)
Alpha (8-13 Hz)
Beta (13-30 Hz)


Power spectrum for Eyes Open (OA)
Power
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
4
6
8
10
12
14
16
18
20
Frequency (Hz)
17
18
Theta (4-8 Hz)
Alpha (8-13 Hz)
Beta (13-30 Hz)


Power spectrum for Eyes Closed (OC)
Power
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
4
6
8
10
12
14
16
18
20
Frequency (Hz)
17
18
Theta (4-8 Hz)
Alpha (8-13 Hz)
Beta (13-30 Hz)

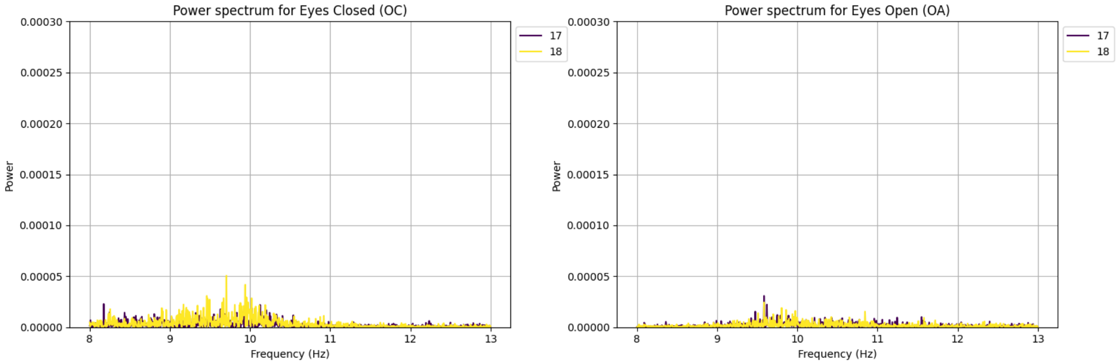

Power spectrum for Eyes Open (OA)
Power
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
4
6
8
10
12
14
16
18
20
Frequency (Hz)
17
18
Theta (4-8 Hz)
Alpha (8-13 Hz)
Beta (13-30 Hz)

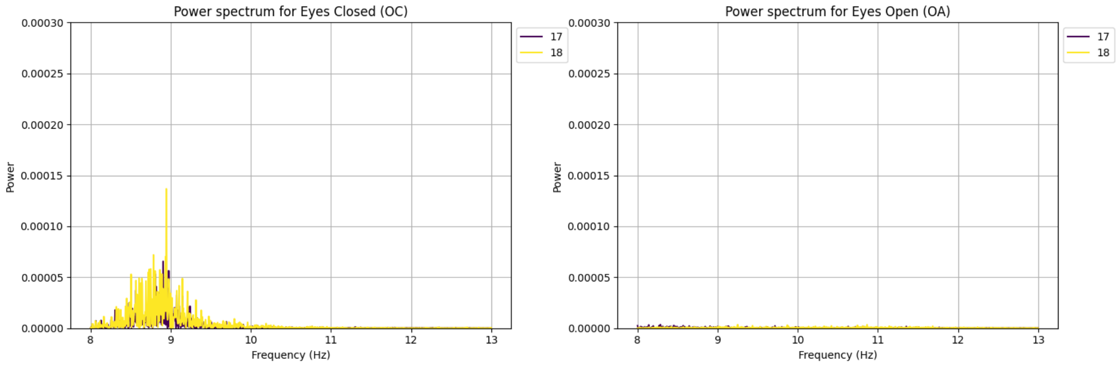
1.3.7 4. Now get the power spectra from the range that seems to be the most suited for classifying the difference between OC and OA based on the visual inspection you did above:

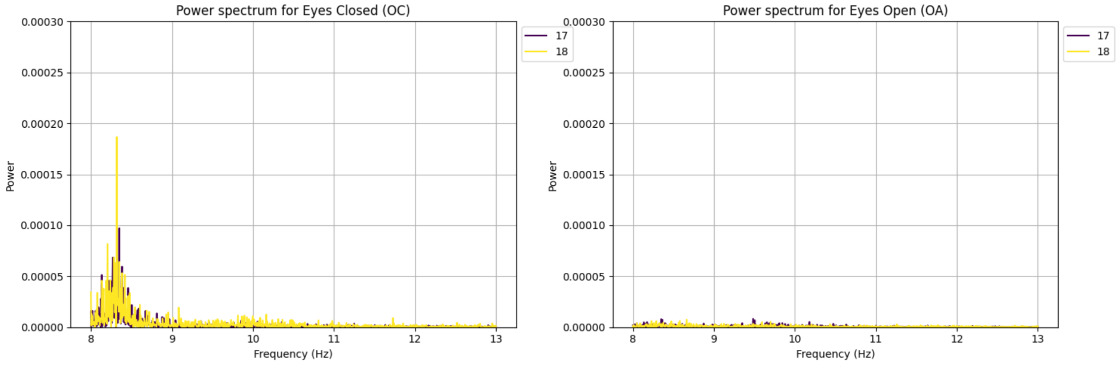
[31]:
```
min_freq = 8
max_freq = 13

power_spectra_oc = [compute_power_spectrum(arr, sampling_rate=sample_rate,␣
 ↪min_freq=min_freq, max_freq=max_freq) for arr in data_oc_optical_channels]
power_spectra_oa = [compute_power_spectrum(arr, sampling_rate=sample_rate,␣
 ↪min_freq=min_freq, max_freq=max_freq) for arr
                    in data_oa_optical_channels]
```

Let's plot this smaller spectrum

[32]:
```
for idx in patient_indexes:
    freqs_oc, spectrum_oc = power_spectra_oc[idx]
    freqs_oa, spectrum_oa = power_spectra_oa[idx]
    plt.subplots(1, 2, figsize=(15, 5))
    plt.subplot(1, 2, 1)
    plot_power_spectrum_overlay(freqs_oc, spectrum_oc, ch_names=[17, 18,],␣
 ↪title=f"Power spectrum for Eyes Closed (OC)", show_label = True)
    plt.ylim([0, y_max])

    plt.subplot(1, 2, 2)
    plot_power_spectrum_overlay(freqs_oa, spectrum_oa, ch_names=[17, 18,],␣
 ↪title=f"Power spectrum for Eyes Open (OA)", show_label = True)
    plt.ylim([0, y_max])
    plt.show()
```







1.3.8 5. Now let's compute the energies in this range, typically referred to as the alpha band, and plot them

Hint: recall that the energy of a signal is given by $E = \sum_{n=0}^{N} f(n)^2$

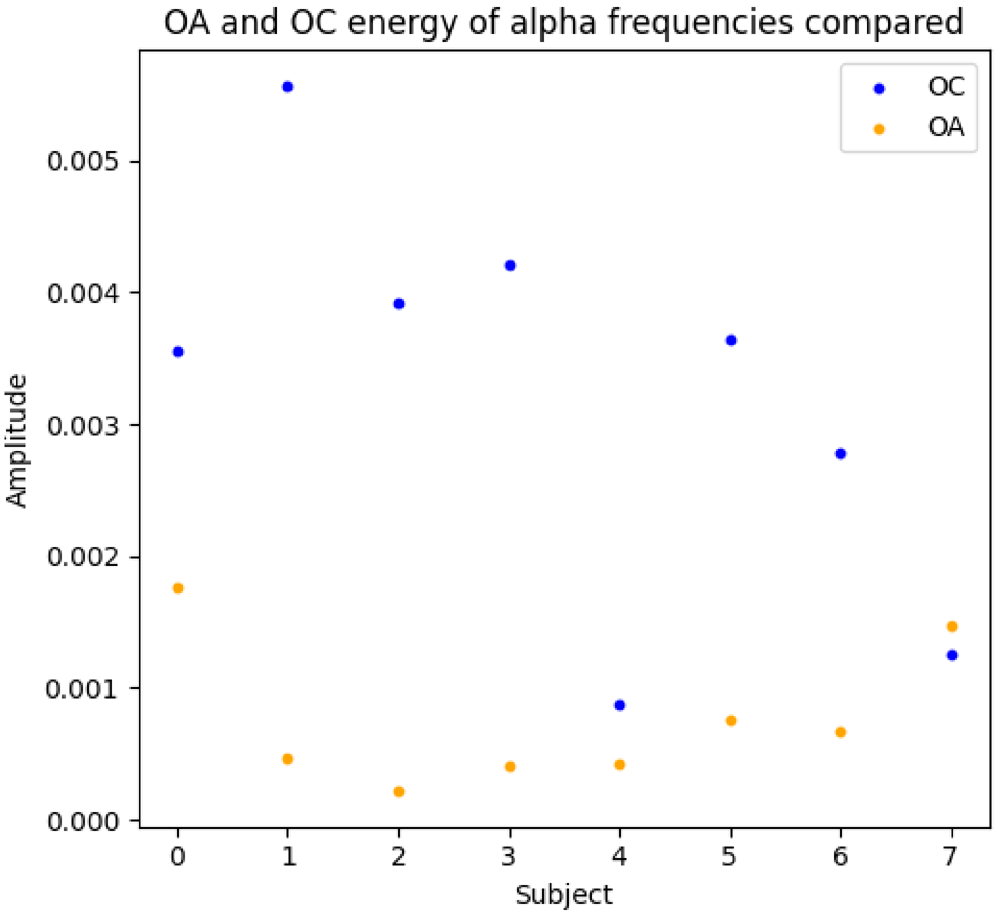
[202]:
```
alphas_oc = []
alphas_oa = []
for (freqs_oc, spectrum_oc), (freqs_oa, spectrum_oa) in zip(power_spectra_oc,␣
 ↪power_spectra_oa):

    alpha_oc = sum(spectrum_oc[1])
    alpha_oa = sum(spectrum_oa[1])

    alphas_oc.append(alpha_oc)
    alphas_oa.append(alpha_oa)

alphas_oc = np.array(alphas_oc)
alphas_oa = np.array(alphas_oa)
```

[203]:

```
def plot_alphas(alphas_oc, alphas_oa, show=False, s = 10):
    fig = plt.figure(figsize=(10, 5))
    plt.subplot(1,2,1)
    plt.scatter(range(len(alphas_oc)), alphas_oc, label='OC', color='blue', s=s)
    plt.scatter(range(len(alphas_oc)), alphas_oa, label='OA', color='orange',␣
 ↪s=s)
    # The following line makes a line which you can change to illustrate a␣
 ↪decision boundary:
    plt.title('OA and OC energy of alpha frequencies compared')
    plt.xlabel('Subject')
    plt.ylabel('Amplitude')
    plt.legend()

    plt.tight_layout()
    if show:
        plt.show()
```

[204]:

```
plot_alphas(alphas_oc, alphas_oa, show=True)
```



Since we already computed the amplitudes from the fourier coefficients in the compute_power_spectrum function and squared the amplitudes to obtain the power, we just need to take the sum of the power to obtain the energy.

1.3.9 6. Now that you have processed the data using the dft and inspected the energies of the two different cases, discuss how well you believe this data could be used to create a classifier that classifies whether a person has their eyes open or closed:

As can be seen in the plot above, the different states seem to form distinct clusters. When the participants have open eyes, they emit a significantly smaller energy (orange dots) than when they have their eyes closed (blue dots). This is ofcourse the case because we have narrowed in on the alpha band, disregarding any other wavelengths.

From the plot above, it seems that it would be entirely possible to perform classification using advanced methods or even simpler methods such as logistic regression, with a sigmoid activation function.

It seems that the classification would be likely to misclassify observations a small portion of the time, however obtaining a decent accuracy seems feasible.

1.4 Exercise 4 Classification

This exercise is here to ties what you learned in weeks 1-3 together with signals processing. Firstly we will use a Naive Bayes classifier and then we will implement a CNN to do classification on the EEG data. Here you can spend as much time and effort as you wish refining the classification or coming up with other techniques, but be warned EEG data is notoriously noisy.

1.4.1 1. Implement a Naive Bayes classifier using the entire dataset to plot the posterior distribution.

A naive Bayes classifier assumes that the features used for classification are normally distributed according to their empirical mean and variance. The steps for classification are as follows: 1. Find the mean and variance of the features of each class for the calculation of the likelihood $P(\alpha|C_x) = \mathcal{N}(\mu_{\alpha_x}, \sigma^2_{\alpha_x})$ - In our case the features are just the alpha energy 2. Define a prior distribution for each class given the features - Here we use a uniform prior representing the proportion of each class in the dataset, why do you think this makes sense? $Prior(C) = 0.5$ 3. Given a new value, calculate the posterior probability of belonging to OC (class 0) using Bayes theorem:

$$P(C_0|\alpha) = \frac{P(\alpha|C_0) \cdot Prior(C_0)}{P(\alpha|C_0) \cdot Prior(C_0) + P(\alpha|C_1) \cdot Prior(C_1)}$$

$$P(C_1|\alpha) = 1 - P(C_0|\alpha)$$

1 Compute the means and variances:

[ ]:

```
m0, v0 = ?
m1, v1 = ?
```

2 Define the priors along with a helper function for calculating the pdf of a normal distribution:

[ ]:
```python
def normal_pdf(x, m, v):
    return np.exp(-(x - m) ** 2 / (2 * v)) / np.sqrt(2 * np.pi * v)

def naive_bayes_prior_0(alpha_energy):
    return ?

def naive_bayes_prior_1(alpha_energy):
    return ?
```

3 We will define a function to compute the posterior distribution and one to use that for classification:

[ ]:
```python
def compute_posterior(alpha_energy, m0, v0, m1, v1):
    likelihood_0 = ?
    likelihood_1 = ?

    prior_0 = ?
    prior_1 = ?

    return ?

def classify_naive_bayes(alpha_energy, m0, v0, m1, v1):
    return ? >= 0.5
```

Now we plot the posterior probability of a given observation being OC according to our classifier. Make sure you understand what this plot represents, do ask if in doubt.

[ ]:
```python
xs = np.linspace(1e-10, 3e-5, 100)
plt.figure(figsize=(10, 6))
plt.plot(xs, compute_posterior(xs, m0, v0, m1, v1))
plt.xlabel('Alpha energy')
plt.title('Posterior probability of eyes closed (OC)')
plt.show()
```

The following is a helper function for the cross-validation loop we are about to do:

[ ]:
```python
def cross_validation_split(data_oc, data_oa, hold_out_val, i, device,␣
 ↪as_arrays=False):
    """
    Performs cross-validation split for one of time, frequency, or alpha data.

    Args:
        data_oc (list or array): Optical channels (oc) data.
        data_oa (list or array): Optical amplifiers (oa) data.
        hold_out_val (int): Number of elements to hold out in each fold.
        i (int): Current fold index.

    Returns:
        train_current (list or array): Training set for current fold.
        val_current (list or array): Validation set for current fold.
    """
    # Define slices for validation and training sets
    slice_val = slice(i * hold_out_val, (i + 1) * hold_out_val)
    before_slice = slice(0, i * hold_out_val)
    after_slice = slice((i + 1) * hold_out_val, None)
    # Validation sets
    val_current_oc = data_oc[slice_val]
    val_current_oa = data_oa[slice_val]

    # Training sets (excluding validation slices)
    train_current_oc = data_oc[before_slice] + data_oc[after_slice]
    train_current_oa = data_oa[before_slice] + data_oa[after_slice]

    # Combine oc and oa for the current fold
    val_current = val_current_oc + val_current_oa
    train_current = train_current_oc + train_current_oa

    val_targets = [0 for _ in val_current_oc] + [1 for _ in val_current_oa]
    train_targets = [0 for _ in train_current_oc] + [1 for _ in␣
↪train_current_oa]
    if as_arrays:
        return train_current_oc, train_current_oa, val_current_oc,␣
↪val_current_oa
    else:
        train_set = SimpleDataset(train_current, train_targets, device)
        val_set = SimpleDataset(val_current, val_targets, device)

        return train_set, val_set
```

1.4.2 2. Define the parameters of the CNN

If you want to use the LazyLinear layer you need the newest version of Pytorch

[ ]:

```
import torch
import torch.nn as nn
from tqdm.notebook import tqdm

class CNN(torch.nn.Module):
    def __init__(self, num_classes, in_channels=1, lr=0.001):
        super().__init__()
        self.num_classes = num_classes
        self.in_channels = in_channels
        self.lr = lr
        self.define_network()

    def define_network(self):
        # Define layers as a torch.nn.Sequential object
        # This is pretty nice, since we can just go layers(input) to get output
        # Rather than having a bunch of functions in the forward function
        self.layers = torch.nn.Sequential(
            nn.Conv2d(in_channels=self.in_channels, out_channels=?,␣
↪kernel_size=3, padding=1), # dim = in
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.ReLU(),
            nn.Flatten(),
            nn.LazyLinear(out_features=?),    # Automatically infers the input␣
↪dimension
            nn.ReLU(),
            nn.Linear(in_features=?, out_features=?),
            nn.ReLU(),
            nn.Linear(in_features=?, out_features=self.num_classes)
        ).to(device)

        # Loss function and optimizer, as you know, Adam is the meta
        self.criterion = nn.CrossEntropyLoss()
        self.optim = torch.optim.Adam(self.layers.parameters(), lr=self.lr)

    def forward(self, x):
        return self.layers(x)

    def train(self, train_dataloader, epochs=1, val_dataloader=None,␣
↪run_filepath=""):

        # To hold accuracy during training and testing
        train_accs = []
        test_accs = []
        for epoch in tqdm(range(epochs), desc=f'Training'):

            path_epoch = join(run_filepath, f"{epoch}.pth")
            os.makedirs(os.path.dirname(path_epoch), exist_ok=True)
            if os.path.exists(path_epoch):
                self.load_model(path_epoch)
                print(f"Loaded model from {path_epoch}. Skipping epoch.")

            epoch_acc = 0

            for inputs, targets in tqdm(train_dataloader):
                logits = self(inputs)
                loss = self.criterion(logits, targets)
                loss.backward()

                self.optim.step()
                self.optim.zero_grad()
                # Not actually used for training, just for keeping track of accuracy
                epoch_acc += (torch.argmax(logits, dim=1) == targets).sum().item()

            self.save_model(path_epoch)
            train_accs.append(epoch_acc / len(train_dataloader.dataset))
            print(f"Epoch {epoch} training accuracy: {epoch_acc / len(train_dataloader.dataset)}")

            # If we have val dataloader, we can evaluate after each epoch
            if val_dataloader is not None:
                acc = self.eval(val_dataloader)
                test_accs.append(acc)
                print(f"Epoch {epoch} validation accuracy: {acc}")

        return train_accs, test_accs

    def eval(self, test_dataloader):

        total_acc = 0

        for input_batch, label_batch in test_dataloader:
            # Get predictions
            logits = self(input_batch)

            # Remember, outs are probabilities (so there's 10 for each input)
            # The classification the network wants to assign, must therefore be the probability with the larget value
            # We find that using argmax (dim=1, because dim=0 would be across batch dimension)
            classifications = torch.argmax(logits, dim=1)
            total_acc += (classifications == label_batch).sum().item()

        total_acc = total_acc / len(test_dataloader.dataset)

        return total_acc

    def save_model(self, path):
        torch.save({
            'model_state_dict': self.state_dict(),
            'optimizer_state_dict': self.optim.state_dict()
        }, path)
        print(f"Model saved to {path}")

    # Load model method
    def load_model(self, path):
        checkpoint = torch.load(path)
        self.load_state_dict(checkpoint['model_state_dict'])
        self.optim.load_state_dict(checkpoint['optimizer_state_dict'])
        print(f"Model loaded from {path}")
```

1.4.3 3. Define the parameters of a Feed Forward Neural Network

[ ]:

```
class FFNN(CNN):
    def __init__(self, num_classes, in_channels=1, lr=0.001):
        super().__init__(num_classes, in_channels, lr)

    def define_network(self):
        self.layers = torch.nn.Sequential(
            nn.Flatten(),
            nn.LazyLinear(out_features=?),
            nn.ReLU(),
            nn.Linear(in_features=?, out_features=?),
            nn.ReLU(),
            nn.Linear(in_features=?, out_features=self.num_classes)
        ).to(device)
        self.criterion = nn.CrossEntropyLoss()
        self.optim = torch.optim.Adam(self.layers.parameters(), lr=self.lr)
```

We also need a boilerplate torch dataset definition in order to use our data:

[ ]:

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

class SimpleDataset(Dataset):
    def __init__(self, time_data, targets, device, dtype_data=torch.float32,␣
 ↪dtype_targets=torch.long):
        """
        Args:
            time_data (list or array): Training time data.
            targets (list or array): Corresponding target values (labels).
        """
        self.data = time_data
        self.targets = targets
        self.device = device
        self.dtype_data = dtype_data
        self.dtype_targets = dtype_targets

    def __len__(self):
        # Return the total number of samples
        assert len(self.data) == len(self.targets)
        return len(self.data)
    def __getitem__(self, idx):
        # Retrieve a single sample (time data and corresponding target)
        sample = self.data[idx]
        target = self.targets[idx]
        sample = torch.tensor(sample).to(dtype=self.dtype_data).to(device)
        target = torch.tensor(target).to(dtype=self.dtype_targets).to(device)
        if sample.ndim == 2:
            sample = sample.unsqueeze(0)

        # Optionally convert to torch tensors if needed
        return sample, target
```

1.4.4 4. Now use the models you have defined above in cross-validation loops, where we will lazily omit the testing part and rely solely on validation, that each use the EEG data in different ways. Here we have set up the following 4 loops, but feel free to experiment with other configurations (Feel free to try other combinations):

- Naive Bayes on alpha energy
- CNN on frequency domain
- CNN on Time domain
- FFNN on frequency domain

The pytorch models will be saved in a folder named 'models' and can be loaded from there. The test loop automatically loads a model if it has the same name as the one it is about to train thus skipping the epoch.

[ ]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using {device}")

hold_out_val = 1
hold_out_test = 0

# Assuming the oc and oa data are already concatenated, split them back to␣
 ↪separate.
train_set_time_oc = data_oc_optical_channels[hold_out_test:]
train_set_time_oa = data_oa_optical_channels[hold_out_test:]

train_set_frequency_oc = [el[1] for el in power_spectra_oc[hold_out_test:]]
train_set_frequency_oa = [el[1] for el in power_spectra_oa[hold_out_test:]]

train_set_alpha_oc = alphas_oc[hold_out_test:]
train_set_alpha_oa = alphas_oa[hold_out_test:]

k_folds = len(train_set_alpha_oc) // hold_out_val
```

[ ]:

```
run_name = join(os.getcwd(), "models", "testing-freq-naive-bayes")

val_accuracies_naive_bayes = []
for i in tqdm(range(k_folds), desc=f"Cross-validation {os.path.
 ↪basename(run_name)}"):
    # Combine oc and oa for the current fold
    train_oc, train_oa, val_oc, val_oa = cross_validation_split(alphas_oc.
 ↪tolist(), alphas_oa.tolist(), hold_out_val, i, device=device, as_arrays=True)

    m0, v0 = np.mean(train_oc), np.var(train_oc)
    m1, v1 = np.mean(train_oa), np.var(train_oa)
    acc_oc = sum([classify_naive_bayes(el, m0, v0, m1, v1) == 1 for el in␣
 ↪val_oc])
    acc_oa = sum([classify_naive_bayes(el, m0, v0, m1, v1) == 0 for el in␣
 ↪val_oa])
    acc = (acc_oc + acc_oa) / (len(val_oc) + len(val_oa))

    val_accuracies_naive_bayes.append(acc)
```

[ ]:

```
run_name = join(os.getcwd(), "models", "testing-time-CNN")
val_accuracies_cnn_time = []
epochs_CNN_time = 1
# Cross-validation loop
for i in tqdm(range(k_folds), desc=f"Cross-validation {os.path.
 ↪basename(run_name)}"):
    # Combine oc and oa for the current fold
    train_set_current, val_set_current =␣
 ↪cross_validation_split(train_set_time_oc, train_set_time_oa, hold_out_val,␣
 ↪i, device=device)

    CNN_time = CNN(num_classes=2)
    CNN_time.train(DataLoader(train_set_current), epochs=epochs_CNN_time,␣
 ↪run_filepath=f"{run_name}-{i}")
    val_cnn_time = CNN_time.eval(DataLoader(val_set_current))
    val_accuracies_cnn_time.append(val_cnn_time)
```

[ ]:

```
run_name = join(os.getcwd(), "models", "testing-freq-CNN")
val_accuracies_cnn_freq = []
epochs_CNN_freq = 10

for i in tqdm(range(k_folds), desc=f"Cross-validation {os.path.
 ↪basename(run_name)}"):
    # Combine oc and oa for the current fold
    train_set_current, val_set_current =␣
 ↪cross_validation_split(train_set_frequency_oc, train_set_frequency_oa,␣
 ↪hold_out_val, i, device=device)
```

```
    CNN_freq = CNN(num_classes=2)
    CNN_freq.train(DataLoader(train_set_current), epochs=epochs_CNN_freq,␣
 ↪run_filepath=f"{run_name}-{i}")
    val_cnn_freq = CNN_freq.eval(DataLoader(val_set_current))
    val_accuracies_cnn_freq.append(val_cnn_freq)
```

[ ]:

```
run_name = join(os.getcwd(), "models", "testing-freq-FFNN")
epochs_FFNN = 2
val_accuracies_fnn_freq = []
for i in tqdm(range(k_folds), desc=f"Cross-validation {os.path.
 ↪basename(run_name)}"):
    # Combine oc and oa for the current fold
    train_set_current, val_set_current =␣
 ↪cross_validation_split(train_set_frequency_oc, train_set_frequency_oa,␣
 ↪hold_out_val, i, device=device)

    FFNN_freq = FFNN(num_classes=2)
    FFNN_freq.train(DataLoader(train_set_current), epochs=epochs_FFNN,␣
 ↪run_filepath=f"{run_name}-{i}")
    val_cnn_freq = FFNN_freq.eval(DataLoader(val_set_current))
    val_accuracies_fnn_freq.append(val_cnn_freq)
```

[ ]:

```
# Calculate means
# Small value to prevent division by zero
epsilon = 1e-8

# Calculate means
mean_cnn_time = [sum(val_accuracies_cnn_time) / (len(val_accuracies_cnn_time) +␣
 ↪epsilon)] * len(val_accuracies_cnn_time)
mean_cnn_freq = [sum(val_accuracies_cnn_freq) / (len(val_accuracies_cnn_freq) +␣
 ↪epsilon)] * len(val_accuracies_cnn_freq)
mean_fnn_freq = [sum(val_accuracies_fnn_freq) / (len(val_accuracies_fnn_freq) +␣
 ↪epsilon)] * len(val_accuracies_fnn_freq)
mean_threshold = [sum(val_accuracies_naive_bayes) /␣
 ↪(len(val_accuracies_naive_bayes) + epsilon)] *␣
 ↪len(val_accuracies_naive_bayes)

# Plotting
# Plotting
plt.figure(figsize=(8, 6))
cnn_time_color = 'blue'  # Example color for CNN Time
cnn_freq_color = 'orange'  # Example color for CNN Freq
fnn_freq_color = 'green'  # Example color for FNN Freq
threshold_color = 'red'  # Example color for Threshold
```

```
plt.plot(val_accuracies_cnn_time, label='CNN Time', marker='o', color=cnn_time_color)
plt.plot(val_accuracies_cnn_freq, label='CNN Freq', marker='s', color=cnn_freq_color)
plt.plot(val_accuracies_fnn_freq, label='FNN Freq', marker='^', color=fnn_freq_color)
plt.plot(val_accuracies_naive_bayes, label='Naive Bayes', marker='d', color=threshold_color)

# Plot means with low alpha, matching colors
plt.plot(mean_cnn_time, label='Mean CNN Time', linestyle='--', alpha=0.3, color=cnn_time_color)
plt.plot(mean_cnn_freq, label='Mean CNN Freq', linestyle='--', alpha=0.3, color=cnn_freq_color)
plt.plot(mean_fnn_freq, label='Mean FNN Freq', linestyle='--', alpha=0.3, color=fnn_freq_color)
plt.plot(mean_threshold, label='Mean Naive Bayes', linestyle='--', alpha=0.3, color=threshold_color)

plt.title('Validation Accuracies with Means')
plt.xlabel('Index')
plt.ylabel('Validation Accuracy')
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.grid(True)
plt.ylim((0, 1.1))
plt.tight_layout()
plt.show()
```

1.4.5 5. How did it go?