EEG Motor Imagery Classification and Clustering Project

Amirhossein Yousefvand, 810199516

Electrical and Computer Engineering Faculty

College of Engineering

University of Tehran

Contents

Data Processing and Feature Extraction

1. Loading Data

Note: Datast A is “BCICIV_calib_ds1a.mat” and Dataset B is “BCICIV_calib_ds1d.mat”.

The dataset contains continuous EEG signals with a sampling frequency of 100 Hz. For the calibration data, there are markers indicating the time points of stimulus presentation and the corresponding target classes. The data is provided in BCICIV_calib_ds1a.mat.mat and BCICIV_calib_ds1d.mat file format with the following variables:

1. cnt: This contains the EEG signals. It is a matrix of size [time x channels] stored in the INT16 data type. To convert these values to microvolts (uV), we use the following formula: cnt = 0.1 * double(cnt)
2. mrk: This contains information about the target markers with the following fields (the evaluation data files do not include this variable):
    - ❖ pos: Positions of the markers in the EEG signal given in sample units.
    - ❖ y: Target classes (-1 for class one or 1 for class two).
3. nfo: This contains additional information with the following fields:
    - ❖ fs: Sampling frequency.
    - ❖ clab: A cell array of channel labels.
    - ❖ classes: A cell array of class names for motor imagery.
    - ❖ xpos: X-coordinates of electrode positions in a 2D projection.
    - ❖ ypos: Y-coordinates of electrode positions in a 2D projection.

For each selected file, we use the pos vector to get the start point of each time window.

We loaded the data using scipy.io.loadmat. Then we extracted the EEG signal from the ‘cnt’ column and converted them to uV values. Next, we extracted the marker structure (position and classes), if possible, and extracted the additional information structure, consisting of nfo, sampling rate (fs), channel labels (channel_name), motor imaginary classes (classes), X-positions of electrodes (xpos), and Y-positions of electrodes (ypos).

Here we have printed out the extracted information:

DATASET1

```
Continuous EEG signals (cnt): (190594, 59)
Positions of cues (pos): [  2091   2891   3691   4491   5291   6091   6891   7692   8492   9292
  10092  10892  11692  12492  13292  16294  17094  17894  18694  19494
  20294  21094  21894  22694  23494  24294  25094  25894  26694  27494
  30495  31295  32095  32895  33695  34495  35295  36095  36895  37695
  38495  39295  40095  40895  41695  44696  45496  46296  47096  47896
  48696  49496  50296  51096  51896  52696  53496  54296  55096  55896
  58895  59695  60495  61295  62095  62895  63695  64495  65295  66095
  66895  67695  68495  69295  70095  73094  73894  74694  75495  76295
  77095  77895  78695  79495  80295  81095  81895  82695  83495  84295
  87294  88094  88894  89694  90494  91294  92094  92894  93694  94494
  97385  98185  98985  99785 100585 101385 102185 102985 103785 104585
 105385 106185 106985 107785 108585 111584 112384 113184 113984 114784
 115584 116384 117184 117984 118785 119585 120385 121185 121985 122785
 125787 126587 127387 128187 128987 129787 130587 131387 132187 132987
 133787 134587 135387 136187 136987 139986 140786 141586 142386 143186
 143986 144786 145586 146386 147186 147986 148786 149586 150386 151186
 154185 154985 155785 156585 157385 158185 158985 159785 160586 161386
 162186 162986 163786 164586 165386 168385 169185 169985 170785 171585
```

```
 172385 173185 173985 174785 175585 176385 177185 177985 178785 179585
 182584 183384 184184 184984 185784 186584 187384 188184 188984 189784]
Target classes (y): [ 1  1 -1  1  1  1  1 -1 -1  1 -1  1 -1 -1 -1  1 -1 -1  1 -1  1  1 -1 -1
 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1 -1 -1  1 -1  1 -1  1  1
  1  1  1  1  1 -1  1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1  1 -1 -1 -1
  1 -1 -1 -1  1 -1  1 -1 -1 -1  1  1  1 -1  1  1  1 -1 -1  1  1  1  1 -1
  1  1 -1  1 -1  1  1  1 -1  1  1 -1  1 -1  1 -1 -1 -1 -1  1  1 -1  1  1
  1  1 -1  1 -1 -1  1  1 -1 -1  1  1  1  1  1  1  1 -1 -1 -1 -1  1 -1 -1
  1 -1 -1 -1  1  1 -1 -1  1  1 -1  1  1  1  1 -1  1 -1 -1  1 -1 -1 -1 -1
 -1 -1  1  1  1  1 -1 -1 -1 -1 -1  1  1 -1  1 -1 -1 -1  1 -1  1 -1  1 -1
 -1  1 -1 -1 -1  1  1  1]
Sampling rate (fs): 100
Channel labels (clab): ['AF3', 'AF4', 'F5', 'F3', 'F1', 'Fz', 'F2', 'F4', 'F6', 'FC5', 'FC3', 'FC1', 'FCz',
'FC2', 'FC4', 'FC6', 'CFC7', 'CFC5', 'CFC3', 'CFC1', 'CFC2', 'CFC4', 'CFC6', 'CFC8', 'T7', 'C5', 'C3', 'C1',
'Cz', 'C2', 'C4', 'C6', 'T8', 'CCP7', 'CCP5', 'CCP3', 'CCP1', 'CCP2', 'CCP4', 'CCP6', 'CCP8', 'CP5', 'CP3',
'CP1', 'CPz', 'CP2', 'CP4', 'CP6', 'P5', 'P3', 'P1', 'Pz', 'P2', 'P4', 'P6', 'PO1', 'PO2', 'O1', 'O2']
Motor imagery classes: ['left', 'foot']
X-positions of electrodes (xpos): [-0.20109028  0.20109028 -0.48547489 -0.32894737 -0.16535231  0.
  0.16535231  0.32894737  0.48547489 -0.60591541 -0.39919579 -0.19765935
  0.          0.19765935  0.39919579  0.60591541 -0.74834683 -0.52472976
 -0.30963911 -0.10226303  0.10226303  0.30963911  0.52472976  0.74834683
 -0.87719298 -0.64569058 -0.421549   -0.20773757  0.          0.20773757
  0.421549    0.64569058  0.87719298 -0.74834683 -0.52472976 -0.30963911
 -0.10226303  0.10226303  0.30963911  0.52472976  0.74834683 -0.60591541
 -0.39919579 -0.19765935  0.          0.19765935  0.39919579  0.60591541
 -0.48547489 -0.32894737 -0.16535231  0.          0.16535231  0.32894737
  0.48547489 -0.10395865  0.10395865 -0.17113186  0.17113186]
Y-positions of electrodes (ypos): [ 0.68656518  0.68656518  0.52547424  0.46520183  0.43208641  0.421549
  0.43208641  0.46520183  0.52547424  0.27165704  0.23384348  0.21394494
  0.20773757  0.21394494  0.23384348  0.27165704  0.15177169  0.12553103
  0.11086096  0.10426648  0.10426648  0.11086096  0.12553103  0.15177169
  0.          0.          0.          0.          0.          0.
  0.          0.          0.         -0.15177169 -0.12553103 -0.11086096
 -0.10426648 -0.10426648 -0.11086096 -0.12553103 -0.15177169 -0.27165704
 -0.23384348 -0.21394494 -0.20773757 -0.21394494 -0.23384348 -0.27165704
 -0.52547424 -0.46520183 -0.43208641 -0.421549   -0.43208641 -0.46520183
 -0.52547424 -0.65583812 -0.65583812 -0.86033797 -0.86033797]
```

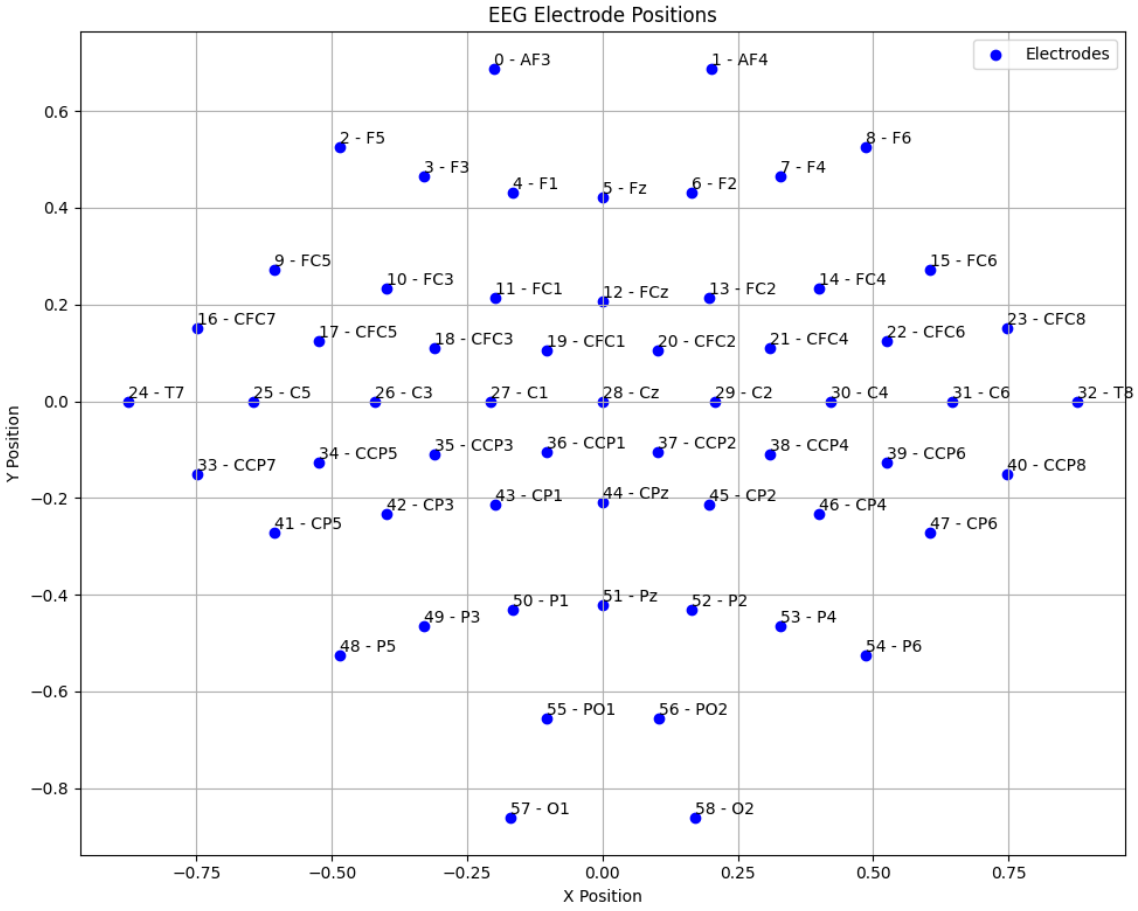
DATASET B

```
Continuous EEG signals (cnt): (190473, 59)
Positions of cues (pos): [  2095   2895   3695   4495   5295   6095   6895   7695   8495   9295
  10095  10895  11695  12495  13295  16294  17094  17894  18694  19494
  20294  21094  21894  22694  23494  24295  25095  25895  26695  27495
  30494  31294  32094  32894  33694  34494  35294  36094  36894  37694
  38494  39294  40094  40894  41694  44693  45493  46293  47093  47893
  48693  49493  50293  51093  51893  52693  53493  54293  55093  55893
  58892  59692  60492  61292  62092  62892  63692  64492  65292  66093
  66893  67693  68493  69293  70093  73092  73892  74692  75492  76292
  77092  77892  78692  79492  80292  81092  81892  82692  83492  84292
  87291  88091  88891  89691  90491  91291  92091  92891  93691  94491
  97292  98092  98892  99692 100492 101292 102092 102892 103692 104492
 105292 106092 106892 107692 108492 111491 112291 113091 113891 114691
 115491 116291 117091 117891 118691 119492 120292 121091 121891 122692
 125691 126491 127291 128091 128891 129691 130491 131291 132091 132891
 133691 134491 135291 136091 136891 139890 140690 141490 142290 143090
 143890 144690 145490 146290 147090 147890 148690 149490 150290 151090
 154089 154889 155689 156489 157289 158090 158890 159690 160490 161290
 162090 162890 163690 164490 165290 168289 169089 169889 170689 171489
 172289 173089 173889 174689 175489 176289 177089 177890 178689 179489
 182488 183288 184088 184888 185688 186488 187288 188088 188888 189688]
Target classes (y): [ 1  1  1 -1  1  1  1 -1  1  1  1 -1  1 -1 -1 -1  1 -1 -1 -1  1  1  1 -1
  1 -1 -1 -1  1 -1  1  1  1 -1  1  1 -1 -1 -1 -1  1 -1  1  1  1  1  1 -1
 -1 -1  1 -1 -1 -1 -1  1  1 -1 -1  1 -1 -1 -1  1  1  1  1 -1 -1  1 -1 -1
  1  1 -1  1 -1  1  1 -1  1  1 -1 -1  1  1 -1  1  1 -1  1  1 -1 -1 -1  1
 -1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1  1  1 -1 -1 -1  1  1  1  1 -1 -1  1 -1
  1 -1 -1  1  1  1  1  1 -1  1  1 -1  1 -1 -1  1 -1  1 -1 -1 -1 -1  1  1
 -1  1 -1  1 -1 -1 -1  1  1 -1  1 -1  1 -1 -1 -1  1  1  1  1 -1  1 -1 -1
  1 -1 -1  1 -1  1  1  1 -1  1  1 -1 -1  1  1 -1  1  1 -1 -1 -1  1  1  1
  1  1 -1 -1  1  1 -1 -1]
Sampling rate (fs): 100
Channel labels (clab): ['AF3', 'AF4', 'F5', 'F3', 'F1', 'Fz', 'F2', 'F4', 'F6', 'FC5', 'FC3', 'FC1', 'FCz',
'FC2', 'FC4', 'FC6', 'CFC7', 'CFC5', 'CFC3', 'CFC1', 'CFC2', 'CFC4', 'CFC6', 'CFC8', 'T7', 'C5', 'C3', 'C1',
'Cz', 'C2', 'C4', 'C6', 'T8', 'CCP7', 'CCP5', 'CCP3', 'CCP1', 'CCP2', 'CCP4', 'CCP6', 'CCP8', 'CP5', 'CP3',
'CP1', 'CPz', 'CP2', 'CP4', 'CP6', 'P5', 'P3', 'P1', 'Pz', 'P2', 'P4', 'P6', 'PO1', 'PO2', 'O1', 'O2']
Motor imagery classes: ['left', 'right']
X-positions of electrodes (xpos): [-0.20109028  0.20109028 -0.48547489 -0.32894737 -0.16535231  0.
  0.16535231  0.32894737  0.48547489 -0.60591541 -0.39919579 -0.19765935
  0.          0.19765935  0.39919579  0.60591541 -0.74834683 -0.52472976
 -0.30963911 -0.10226303  0.10226303  0.30963911  0.52472976  0.74834683
```

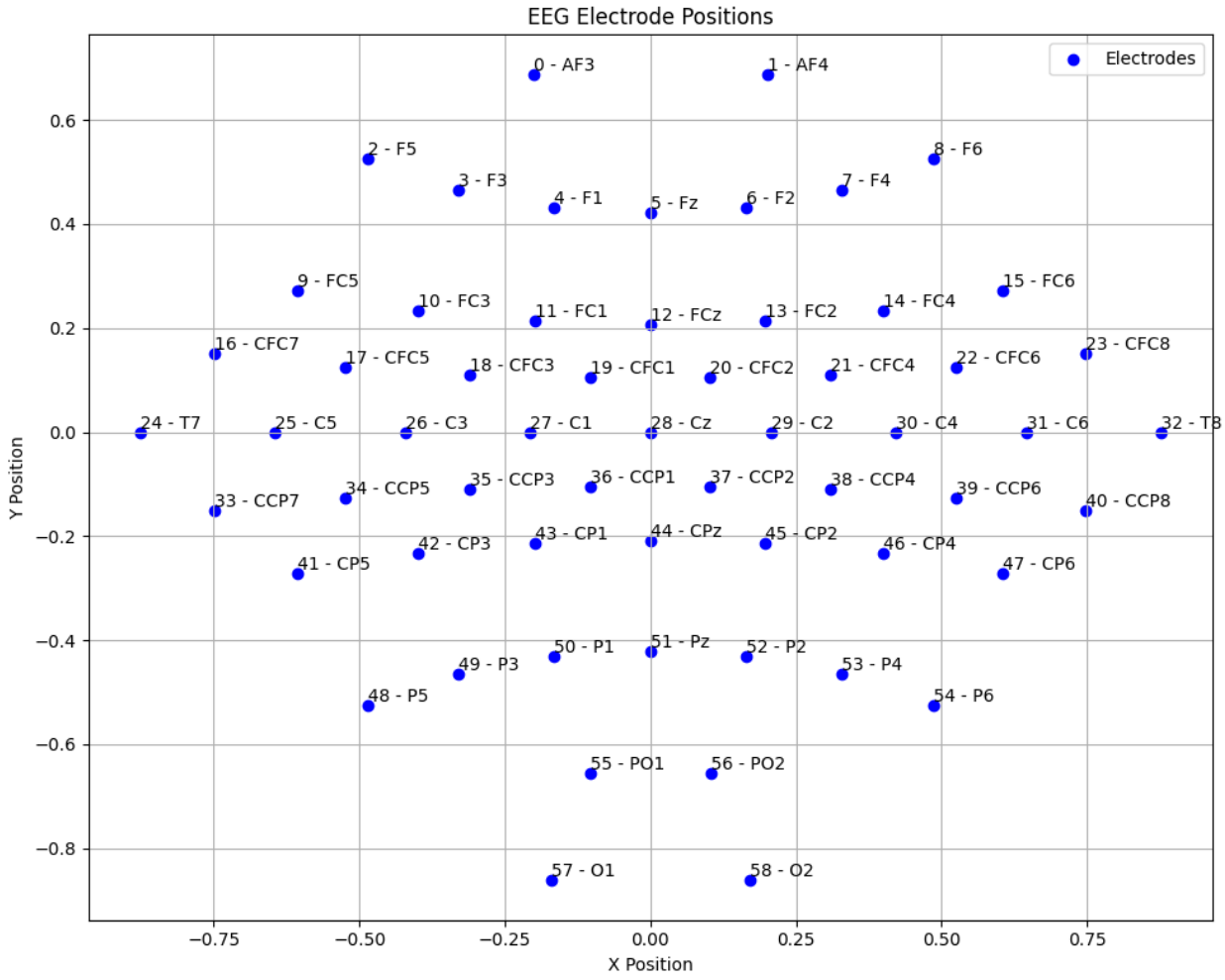
```
 -0.87719298 -0.64569058 -0.421549   -0.20773757  0.          0.20773757
  0.421549    0.64569058  0.87719298 -0.74834683 -0.52472976 -0.30963911
 -0.10226303  0.10226303  0.30963911  0.52472976  0.74834683 -0.60591541
 -0.39919579 -0.19765935  0.          0.19765935  0.39919579  0.60591541
 -0.48547489 -0.32894737 -0.16535231  0.          0.16535231  0.32894737
  0.48547489 -0.10395865  0.10395865 -0.17113186  0.17113186]
Y-positions of electrodes (ypos): [ 0.68656518  0.68656518  0.52547424  0.46520183  0.43208641  0.421549
  0.43208641  0.46520183  0.52547424  0.27165704  0.23384348  0.21394494
  0.20773757  0.21394494  0.23384348  0.27165704  0.15177169  0.12553103
  0.11086096  0.10426648  0.10426648  0.11086096  0.12553103  0.15177169
  0.          0.          0.          0.          0.          0.
  0.          0.          0.         -0.15177169 -0.12553103 -0.11086096
 -0.10426648 -0.10426648 -0.11086096 -0.12553103 -0.15177169 -0.27165704
 -0.23384348 -0.21394494 -0.20773757 -0.21394494 -0.23384348 -0.27165704
 -0.52547424 -0.46520183 -0.43208641 -0.421549   -0.43208641 -0.46520183
 -0.52547424 -0.65583812 -0.65583812 -0.86033797 -0.86033797]
```

Here we have plotted the electrode positions, with their respective channel labels.

DATASET A



DATASET B



2. Bandpass Filtering

We used the butter and filtfilt functions from scipy.signal to filter our signal.

'butter' function is used to design a Butterworth filter. It returns the filter coefficients.

'filtfilt' function applies the filter to a data sequence using forward and backward filtering to ensure zero-phase distortion.

Butterworth Bandpass Filter Function:

```
def butter_bandpass(lowcut, highcut, fs, order=3):
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='band')
    return b, a
```

- lowcut: The lower cutoff frequency of the bandpass filter.
- highcut: The upper cutoff frequency of the bandpass filter.
- fs: The sampling frequency of the signal.
- order: The order of the filter (default is 3). Higher order means a steeper roll-off.
- nyquist: The Nyquist frequency, which is half the sampling rate. It's the highest frequency that can be accurately represented at a given sampling rate.
- low and high: These are the normalized cutoff frequencies. Normalization is done by dividing the actual cutoff frequencies by the Nyquist frequency.
- butter(order, [low, high], btype='band'): This designs a Butterworth bandpass filter with the specified order and cutoff frequencies. It returns the filter coefficients b and a.

Bandpass Filter Function:

```
def bandpass_filter(data, lowcut, highcut, fs, order=3):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = filtfilt(b, a, data, axis=0)
    return y
```

- data: The input signal to be filtered.
- lowcut: The lower cutoff frequency of the bandpass filter.
- highcut: The upper cutoff frequency of the bandpass filter.
- fs: The sampling frequency of the signal.
- order: The order of the filter (default is 3).
- butter_bandpass(lowcut, highcut, fs, order): Calls the previously defined function to get the filter coefficients b and a.
- filtfilt(b, a, data, axis=0): Applies the Butterworth filter to the data. filtfilt performs forward and backward filtering, which eliminates phase distortion. The axis=0 argument specifies that filtering is applied along the first axis (commonly used for time-series data). Returns the filtered signal y.

knowing that mu band is (8, 12) and beta band is (13, 30), we filter the EEG signal using the bandpass filter function, resulting in EEG_mu and EEG_beta. It's not necessary to divide mu and beta from each other, hence we have created the EEG_mu_beta signal consisting of the two signal bands, i.e. the (8, 30) band. The results' shapes are:

DATASET A

```
Original EEG signals (EEG): (190594, 59)
Mu band EEG signals (EEG_mu): (190594, 59)
Beta band EEG signals (EEG_beta): (190594, 59)
mu+beta EEG signals (EEG_mu_beta): (190594, 59)
```

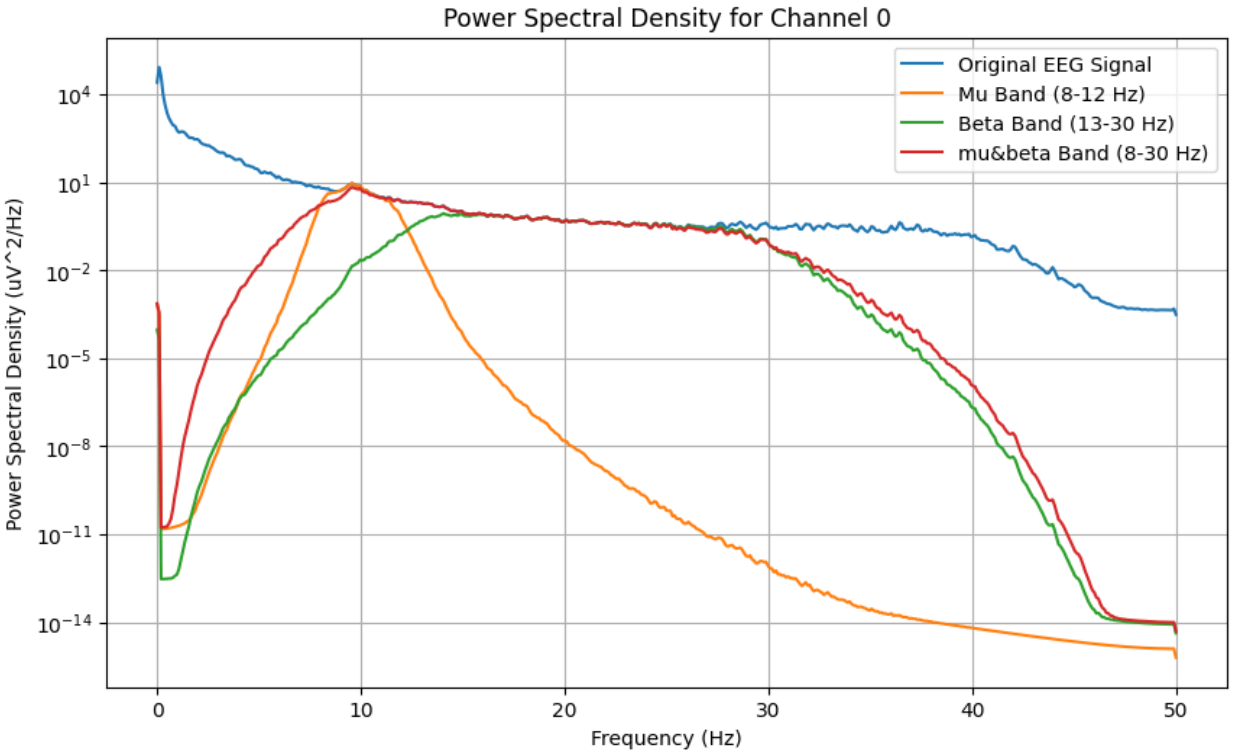
DATASET B

```
Original EEG signals (EEG): (190473, 59)
Mu band EEG signals (EEG_mu): (190473, 59)
Beta band EEG signals (EEG_beta): (190473, 59)
```

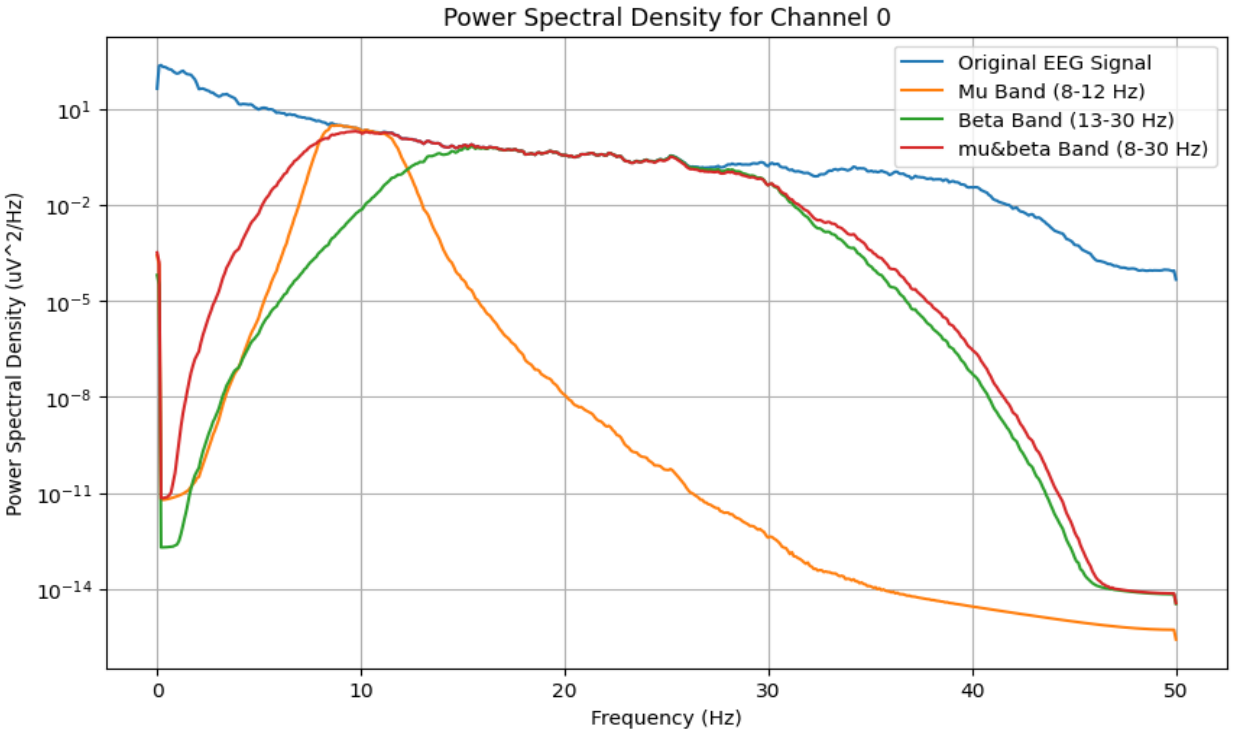
```
mu+beta EEG signals (EEG_mu_beta): (190473, 59)
```

We have plotted the power spectral density of the original EEG signal and the filtered ones for channel 0.

DATASET A



DATASET B



As we can see in this plot, the mu band is (8, 12), the beta band is (13, 30), and the combination of both have a band of (8, 30), which proves that our bandpass filter has worked as expected. The band is where we have (higher) density.

3. Spatial Filtering

Spatial filtering is a technique used in signal processing to enhance desired signal components and suppress noise or artifacts by manipulating data across different spatial locations, such as electrodes in EEG recordings. The primary purposes of spatial filtering include noise reduction, signal enhancement, and source localization. It operates by applying mathematical transformations to the signals recorded from multiple sensors or channels.

Types of Spatial Filters:

1. Common Average Reference (CAR): This filter re-references each channel to the average of all channels, thereby reducing common noise and enhancing the differences between individual channels.
2. Laplacian Filter: This filter enhances local differences by subtracting the average of neighboring electrodes from the signal at each electrode, which emphasizes focal brain activity and helps in detecting localized neural events.

3. Principal Component Analysis (PCA): PCA is a statistical method that transforms data into a new coordinate system where the greatest variance lies along the first principal component, the second greatest variance along the second component, and so on. This transformation helps in reducing dimensionality and highlighting the main sources of variance in the data.

4. Independent Component Analysis (ICA): ICA separates mixed signals into statistically independent components, which is particularly useful for isolating and removing artifacts like eye blinks or muscle activity from EEG data.
5. Beamforming: This technique focuses on signals coming from a specific direction while suppressing signals from other directions, making it useful for source localization in brain imaging and communication systems.

In this project, we have applied CAR and Laplacian filtering.

3.1. CAR Filtering

Common Average Reference (CAR) filtering is a technique used in EEG signal processing to reduce noise and artifacts by re-referencing each channel to the average of all channels. This method helps in improving the signal-to-noise ratio by removing common noise components that are present across all channels.

We have coded this function for this task:

```
def apply_car_filter(data):
    mean_signal = np.mean(data, axis=1, keepdims=True)
    car_filtered = data - mean_signal
    return car_filtered
```

- mean_signal = np.mean(data, axis=1, keepdims=True): Computes the average signal across all channels for each time point. The keepdims=True argument ensures that the result has the same number of dimensions as the input, which is necessary for correct broadcasting in the next step.
- car_filtered = data - mean_signal: Subtracts the average signal from each channel's signal at each time point. This effectively re-references each channel to the common average, removing the common noise.

We have applied the CAR filter to mu signal, beta signal, and the combination of them (mu_beta). The results' shapes are:

DATASET A

```
CAR-filtered Mu band EEG signals (EEG_mu_car): (190594, 59)
CAR-filtered Beta band EEG signals (EEG_beta_car): (190594, 59)
CAR-filtered mu+Beta band EEG signals (EEG_mu_beta_car): (190594, 59)
```

DATASET B

```
CAR-filtered Mu band EEG signals (EEG_mu_car): (190473, 59)
CAR-filtered Beta band EEG signals (EEG_beta_car): (190473, 59)
CAR-filtered mu+Beta band EEG signals (EEG_mu_beta_car): (190473, 59)
```

3.2. Laplacian Filtering

Laplacian filtering is a spatial filtering technique used primarily in EEG signal processing to emphasize local activity by subtracting the average of neighboring electrodes from each electrode's signal. This method enhances focal brain activity and helps in identifying localized neural events.

We have coded this function for this task:

```
def apply_laplacian_filter(data, electrode_locations, radius=0.25):
    filtered_data = np.zeros_like(data)
    n_channels = data.shape[0]
    n_samples = data.shape[1]

    for i in range(n_channels):
        channel_data = data[i]  # Update the indexing of channel_data
        channel_location = electrode_locations[i]
        distance = np.linalg.norm(electrode_locations - channel_location,
axis=1)
        neighbors = np.nonzero(distance <= radius)
        filtered_data[i] = channel_data - np.mean(channel_data[neighbors])
    return filtered_data
```

- electrode_locations: A 2D array with the coordinates (x, y) of each electrode.
- radius: The radius within which neighboring electrodes are considered for averaging. This can be assumed like a hyperparameter. We should test and find the best radius.
- filtered_data = np.zeros_like(data): Creates an array of the same shape as data to store the filtered signals.
- n_channels = data.shape[0]: The number of channels (electrodes).
- n_samples = data.shape[1]: The number of time samples.
- channel_data = data[i]: Extract the signal for the current channel.
- channel_location = electrode_locations[i]: Get the location of the current channel.
- distance = np.linalg.norm(electrode_locations - channel_location, axis=1): Calculate the Euclidean distance from the current channel to all other channels.
- neighbors = np.nonzero(distance <= radius): Identify neighboring channels within the specified radius.
- filtered_data[i] = channel_data - np.mean(data[neighbors], axis=0): Subtract the mean signal of the neighboring channels from the current channel's signal to get the Laplacian-filtered signal.

We have created a 2D array called electrode_locations which combines the x and y coordinates of the electrodes.

After using the filter, we transposed the Laplacian-filtered signals back to the original shape.

We have applied the Laplacian filter to mu signal, beta signal, and the combination of them (mu_beta). The results' shapes are:

DATASET A

```
Laplacian-filtered Mu band EEG signals (EEG_mu_laplacian): (190594, 59)
Laplacian-filtered Beta band EEG signals (EEG_beta_laplacian): (190594, 59)
Laplacian-filtered mu&Beta band EEG signals (EEG_mu_beta_laplacian): (190594, 59)
```

DATASET B

```
Laplacian-filtered Mu band EEG signals (EEG_mu_laplacian): (190473, 59)
```

```
Laplacian-filtered Beta band EEG signals (EEG_beta_laplacian): (190473, 59)
Laplacian-filtered mu&Beta band EEG signals (EEG_mu_beta_laplacian): (190473, 59)
```

4. Data Visualization

This section shows the methods and results of visualizing the EEG data to better understand the effects of our filtering techniques.

4.1. Functions for Plotting Signals

First, we define functions for plotting the signals and comparing their characteristics before and after filtering.

The functions we defined here are:

- plot_signals: This function plots the original and filtered signals for a specified channel. It is useful for comparing the effects of the filters.
- amplitude_plot: This function plots the signals from multiple channels for the original, CAR-filtered, and Laplacian-filtered EEG data. It helps in visualizing the differences across channels.
- amplitude_scatter_plot: This function creates scatter plots of the signal amplitudes at a specific time point, showing the spatial distribution of the signals before and after filtering.

4.2. Functions for Applying t-SNE

t-SNE (t-distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique that helps in visualizing high-dimensional data.

The functions we defined here are:

- apply_tsne: This function applies t-SNE to the EEG data, reducing its dimensionality for visualization.
- plot_tsne: This function plots the t-SNE results for the original, CAR-filtered, and Laplacian-filtered EEG data, allowing for visual comparison.
- plot_tsne_seperated: This function plots the t-SNE results for a single dataset, providing a clear view of the clustering and structure in the data.

4.3. Plot Signals

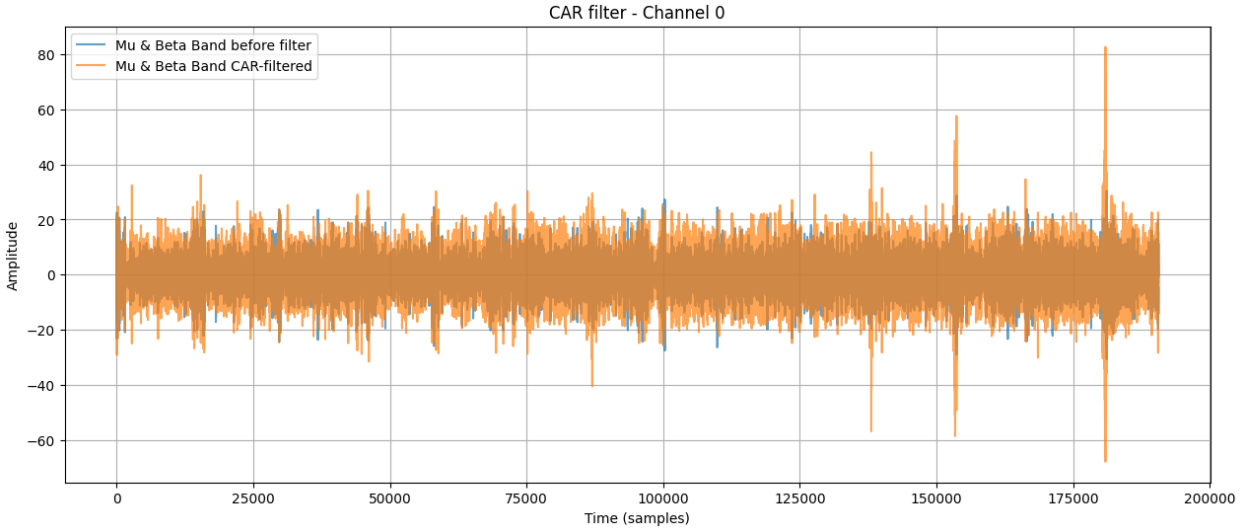
The selected channel for these plots is channel 0, and the selected band is mu and beta band.

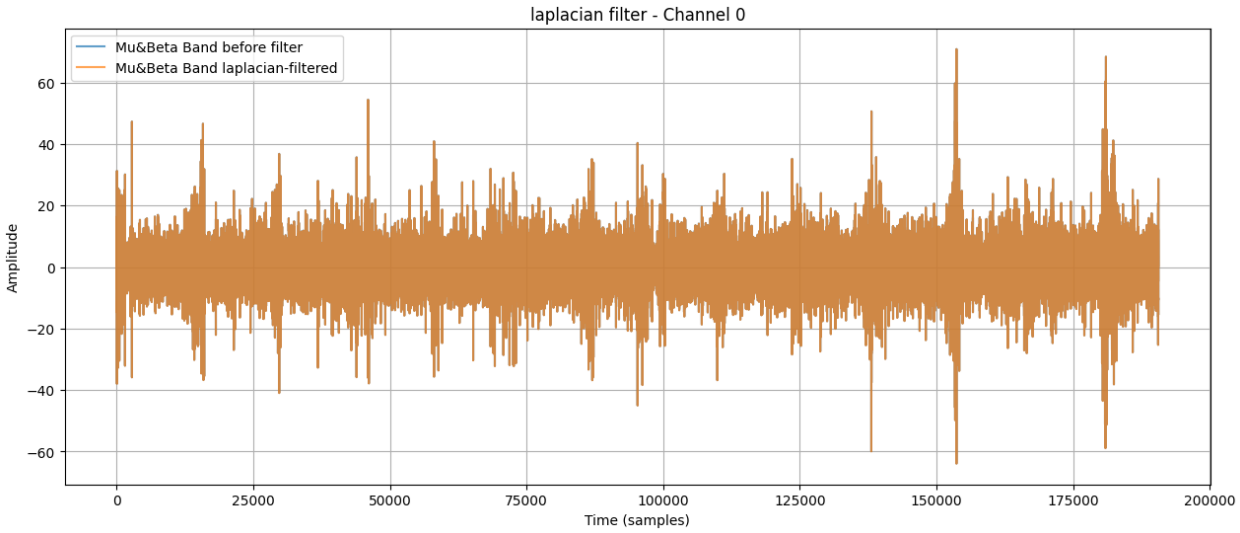
4.3.1. Plot Filter Effects

These plots compare the original and filtered signals for the Mu+Beta band, highlighting the changes introduced by the CAR and Laplacian filters.

```
channel_to_plot = 0
# Plot the Mu+beta band signals before and after CAR filtering
plot_signals(EEG_mu, EEG_mu_beta_car, 'Mu & Beta Band', 'CAR',
channel_to_plot)
# Plot the Mu&beta band signals before and after laplacian filtering
plot_signals(EEG_mu_beta, EEG_mu_beta_laplacian,'Mu&Beta Band',
'laplacian', channel_to_plot)
```

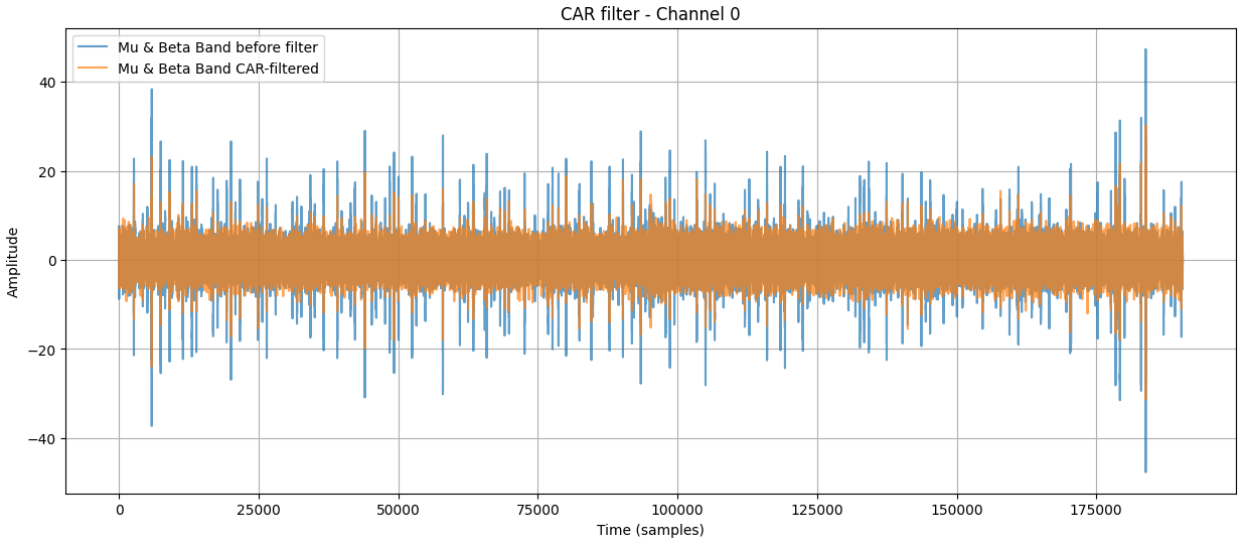
DATASET A

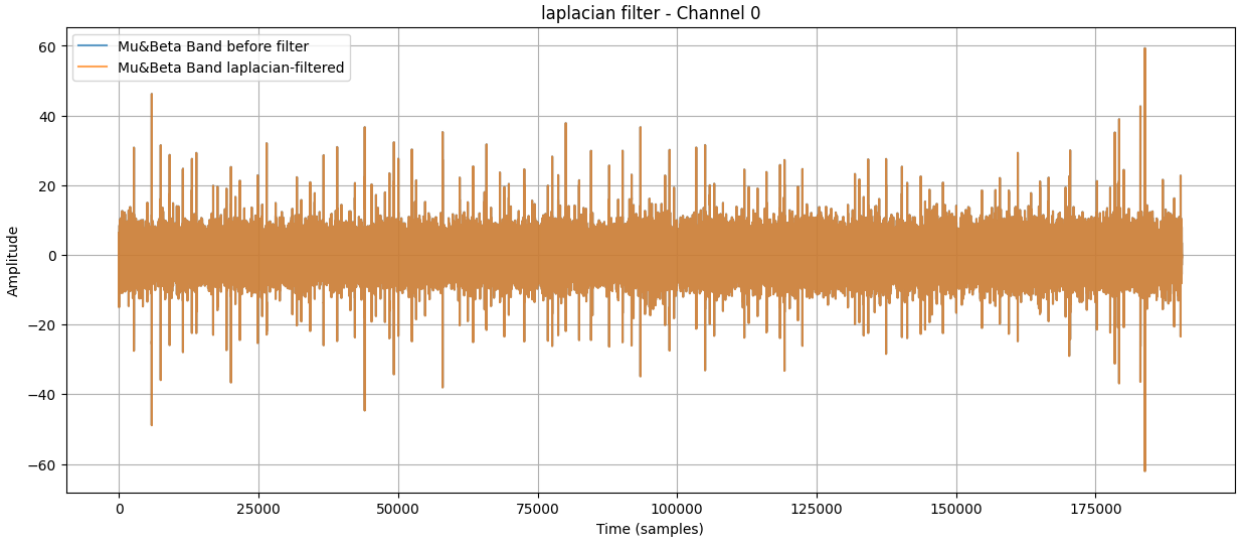




The original signals have a higher similarity with Laplacian-filtered signals and less similarity with CAR-filtered signals.

DATASET B





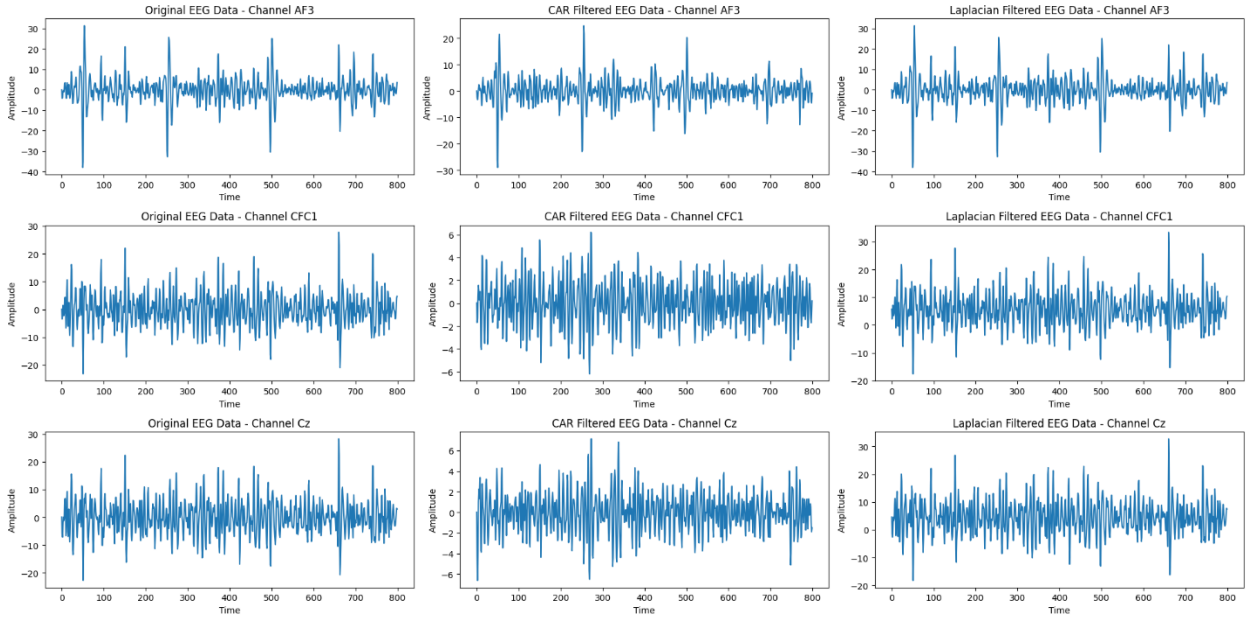
4.3.2. Plot Different Channels

These plots visualize the original, CAR-filtered, and Laplacian-filtered signals for multiple channels, providing a comprehensive view of the filtering effects across different bands.

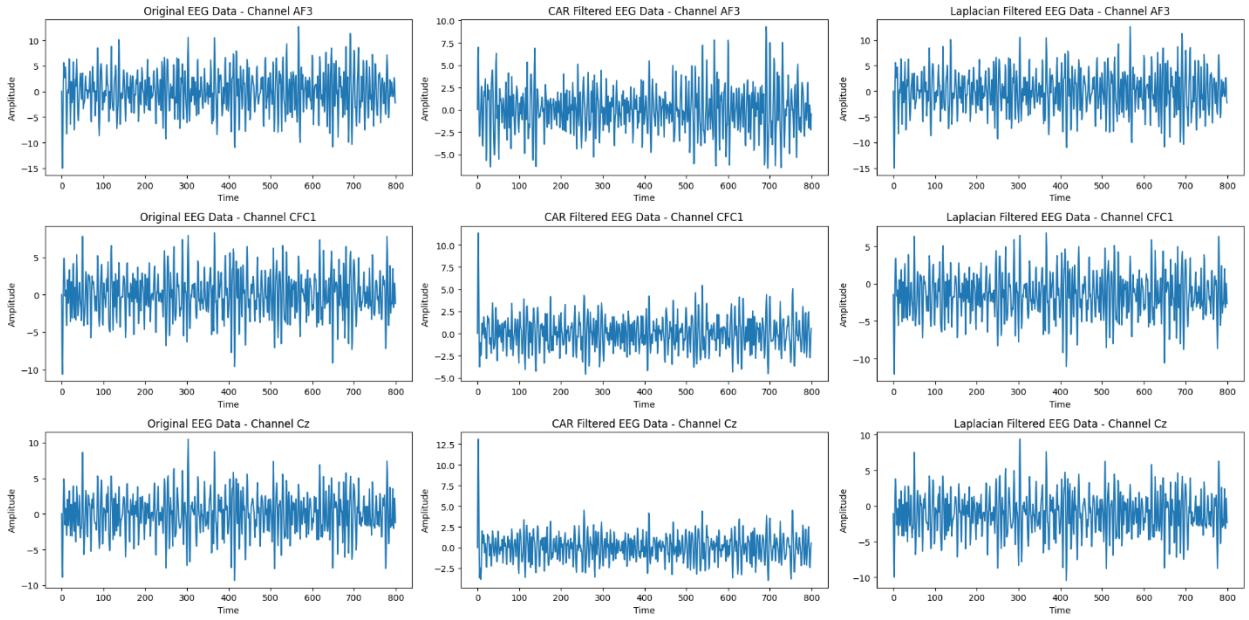
Mu+Beta Band:

```
amplitude_plot(EEG_mu_beta, EEG_mu_beta_car, EEG_mu_beta_laplacian)
```

DATASET A



DATASET B

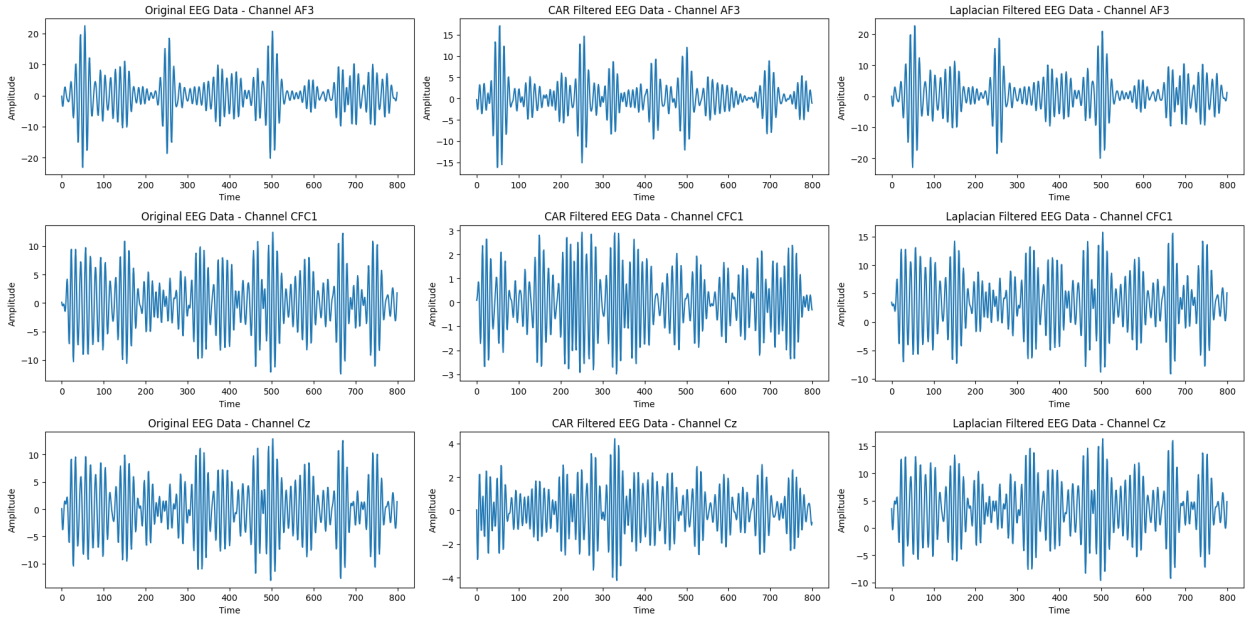


As we can see again the original signals shapes have a higher similarity with Laplacian-filtered signals and less similarity with CAR-filtered signals.

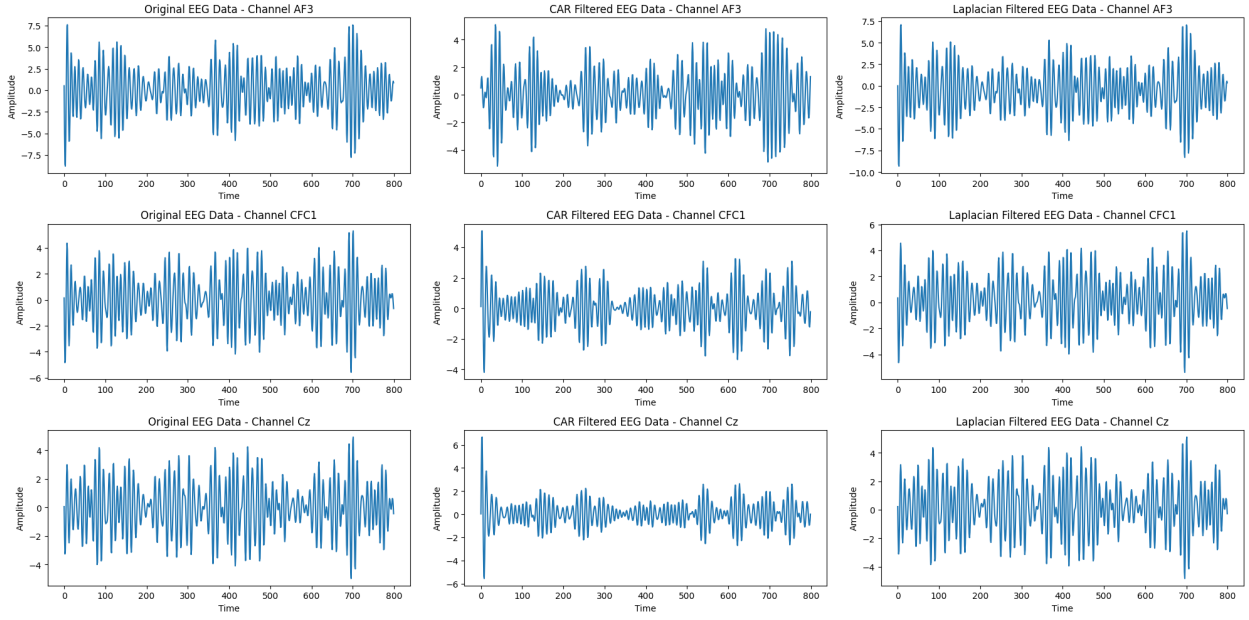
Mu Band:

```
amplitude_plot(EEG_mu, EEG_mu_car, EEG_mu_laplacian)
```

DATASET A



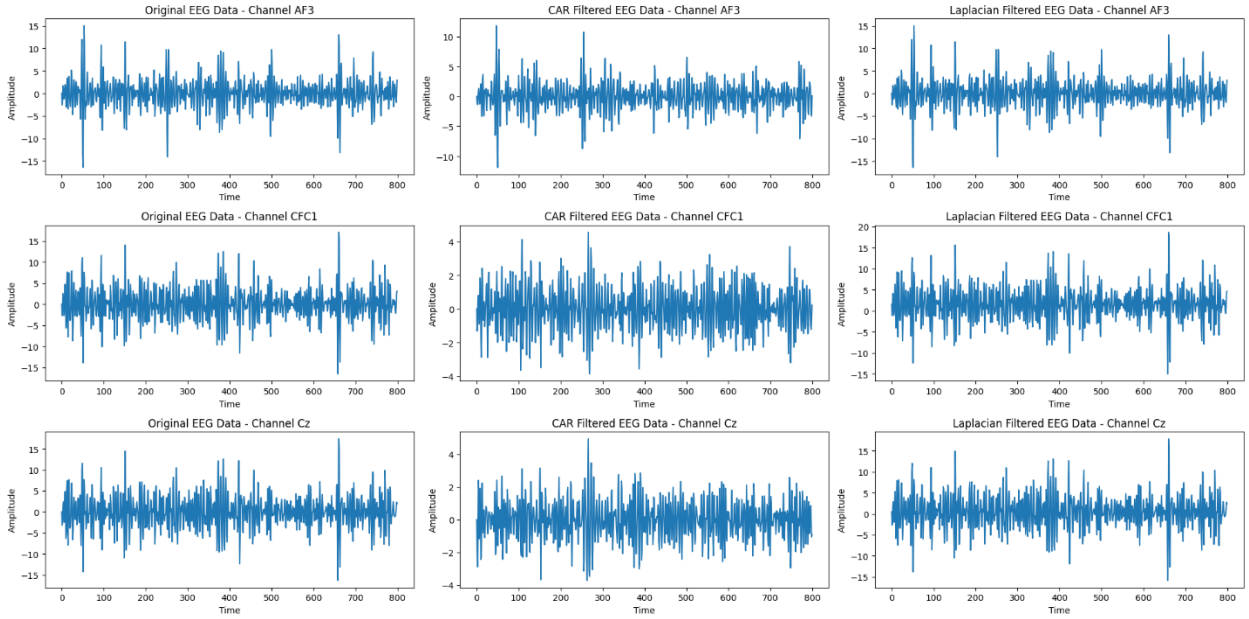
DATASET B



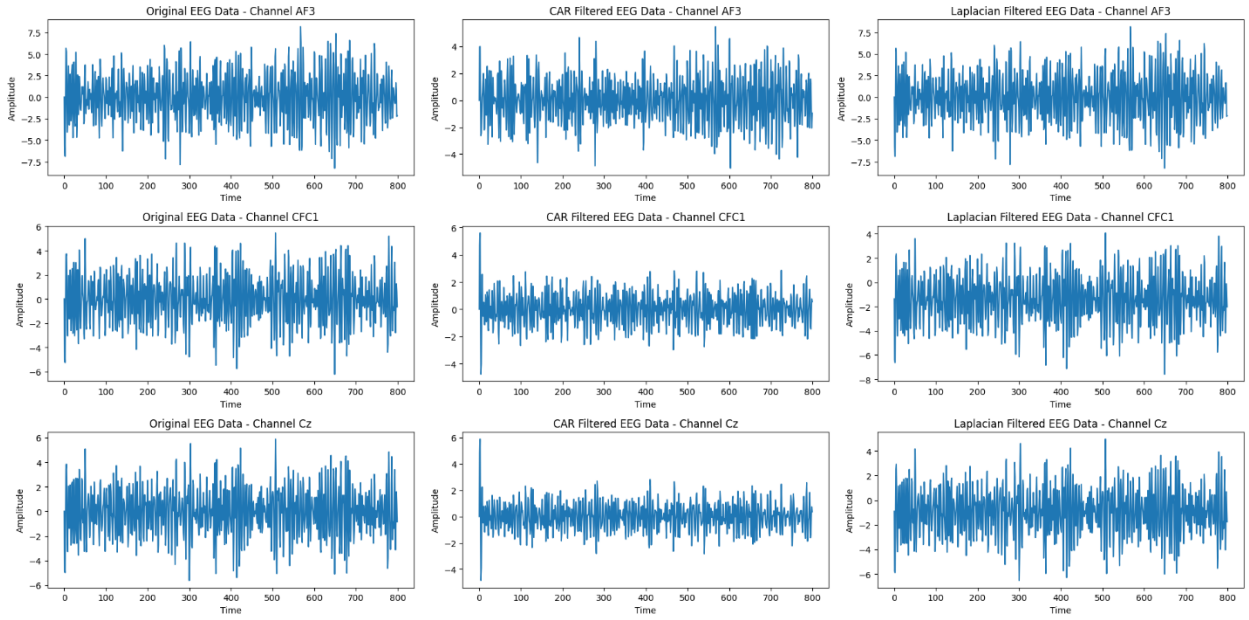
Beta Band:

```
amplitude_plot(EEG_beta, EEG_beta_car, EEG_beta_laplacian)
```

DATASET A



DATASET B



The frequency of Beta signal is higher than MU signal.

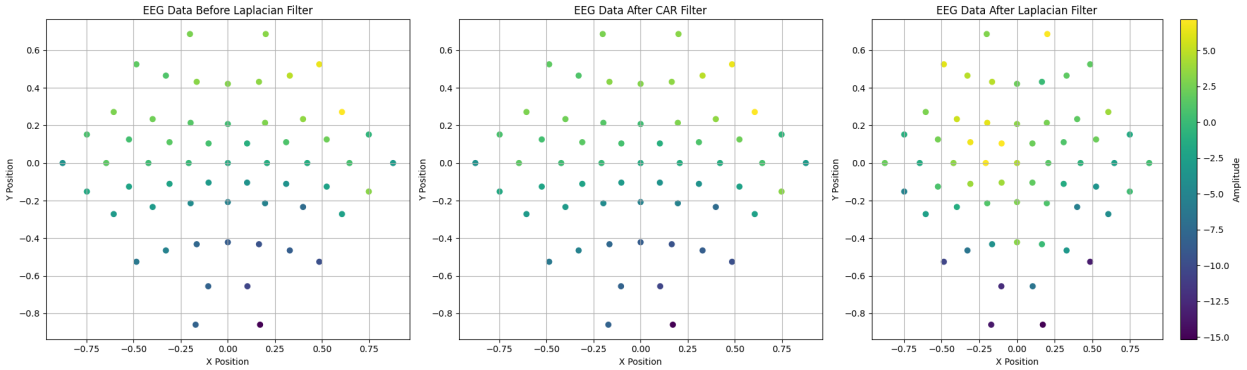
4.3.3. Scatter Plot Electrodes

These scatter plots show the spatial distribution of signal amplitudes across the electrode array at a specific time point, before and after filtering.

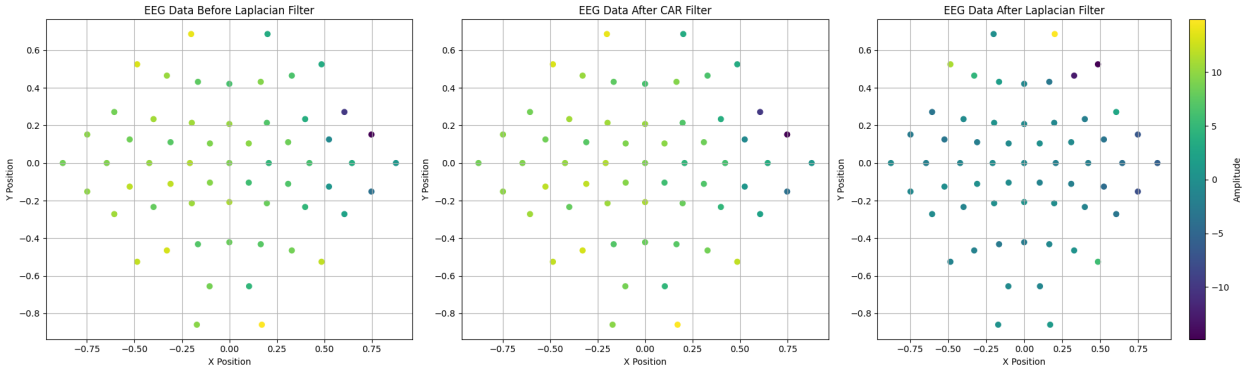
Mu+Beta Band:

```
amplitude_scatter_plot(EEG_mu_beta, EEG_mu_beta_car,
EEG_mu_beta_laplacian)
```

DATASET A



DATASET B

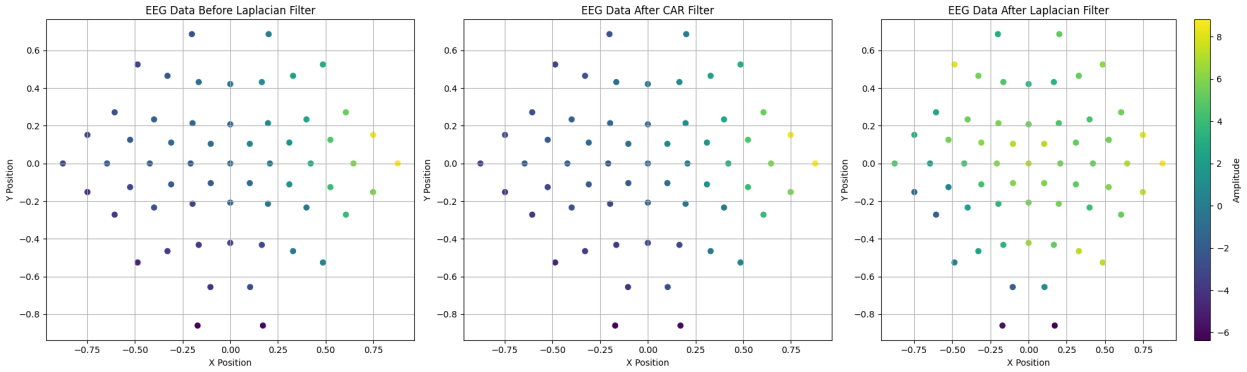


In dataset B amplitudes after Laplacian filter are more negative.

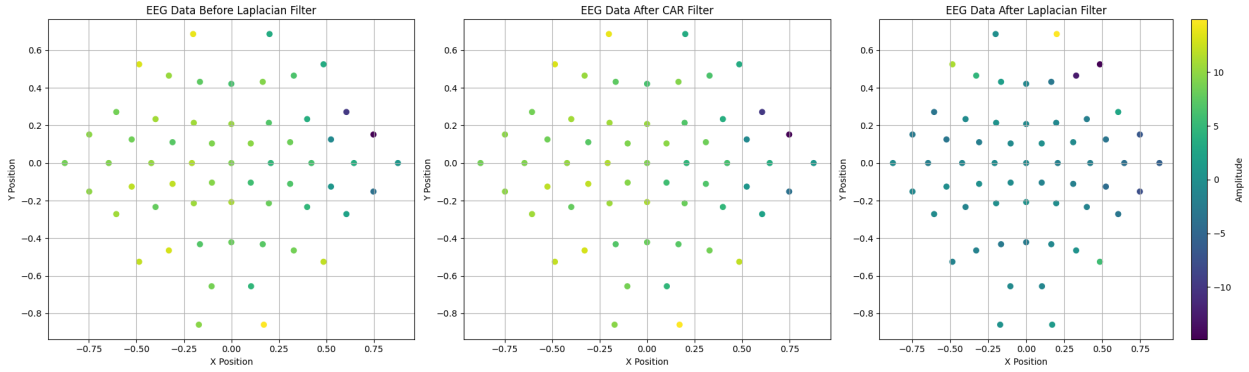
Mu Band:

```
amplitude_scatter_plot(EEG_mu, EEG_mu_car, EEG_mu_laplacian)
```

DATASET A



DATASET B

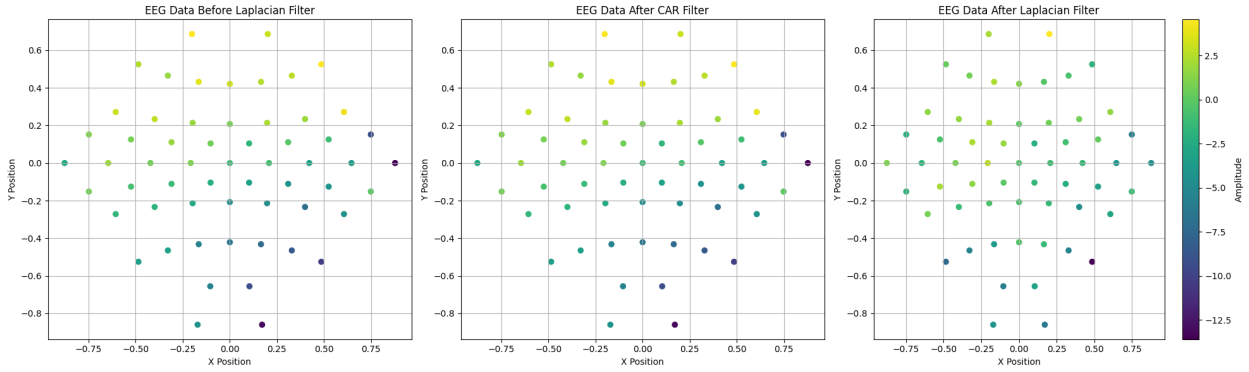


In dataset B amplitudes after Laplacian filter are more negative.

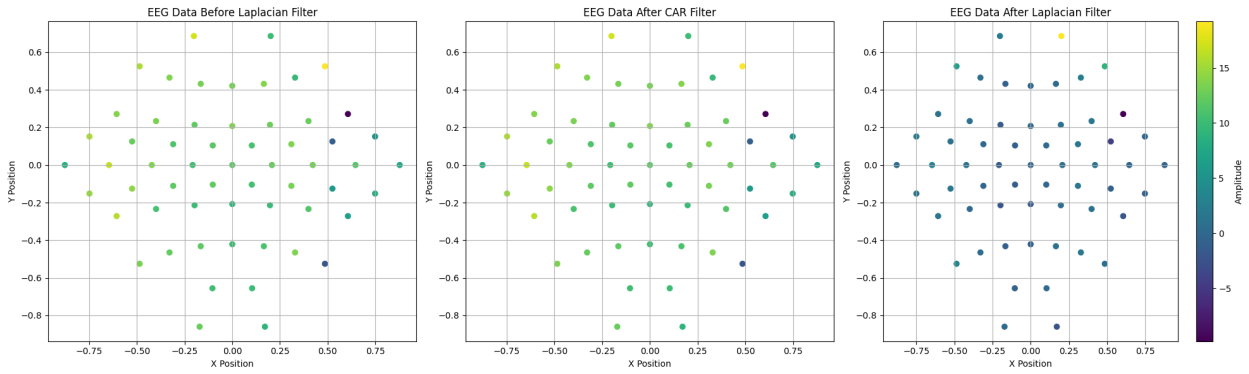
Beta Band:

```
amplitude_scatter_plot(EEG_beta, EEG_beta_car, EEG_beta_laplacian)
```

DATASET A



DATASET B



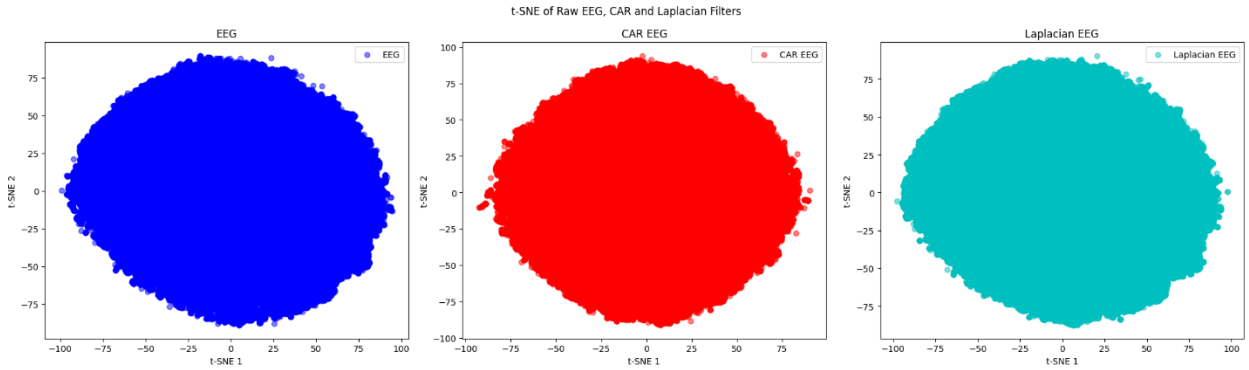
4.4. Apply and Plot t-SNE

This section applies t-SNE to the original, CAR-filtered, and Laplacian-filtered Mu band EEG data and plots the results. This visualization technique helps in understanding the structure and clustering of the high-dimensional EEG data after different filtering methods.

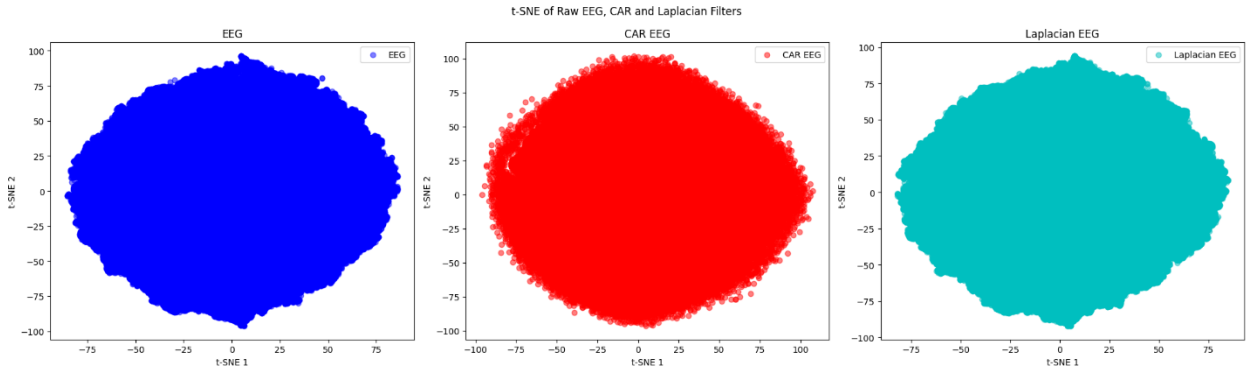
Using these visualizations, we can gain insights into the effects of different filtering techniques on EEG signals, which aids in the analysis and interpretation of neural data.

```
tsne_mu_beta = apply_tsne(EEG_mu_beta)
tsne_mu_beta_car = apply_tsne(EEG_mu_beta_car)
tsne_mu_beta_laplacian = apply_tsne(EEG_mu_beta_laplacian)

plot_tsne(tsne_mu_beta, tsne_mu_beta_car, tsne_mu_beta_laplacian, 't-SNE
of Raw EEG, CAR and Laplacian Filters'))
```

DATASET A



DATASET B



As we can see again the original signals t-SNE have a higher similarity with Laplacian-filtered signals and less similarity with CAR-filtered signals.

5.Feature Extraction

Before performing classification and clustering, it is essential to extract and reduce features to achieve the best accuracy. We employed three different methods for feature extraction: CSP, PCA and LDA

We compared the results of these methods to determine which provides the best features for subsequent classification and clustering.

Data preparation

Before applying feature extraction methods, we need to prepare our data to be suitable for the functions.

Reshape and transpose:

CSP requires 3D data in the format (samples, channels, window data). Our dataset contains samples with a frequency of 100 Hz, resulting in 190,473 data points from 59 channels over time. We have 200 starting positions for each class. We need to separate each window based on their starting positions. To determine the window size, we find the minimum distance between two consecutive window positions

```
window_size = min(window_pos[x + 1] - window_pos[x] for x in
range(window_pos.size - 1))
window_size = min(window_size, EEG_mu_beta.shape[0]-window_pos[-1])
print("window_size = ",window_size)
```

```
For dataset A, this value is 800, and for dataset B, this value is
785. We then extract each window. This process is repeated for the
original data, car-filtered data, and Laplacian-filtered data:
```

```
EEG_3D_mu_beta = np.array([EEG_mu_beta[start:start + window_size, :] for
start in window_pos])
EEG_3D_mu_beta_T = EEG_3D_mu_beta.transpose(0,2,1)

EEG_3D_mu_beta_car = np.array([EEG_mu_beta_car[start:start + window_size,
:] for start in window_pos])
EEG_3D_mu_beta_car_T = EEG_3D_mu_beta_car.transpose(0,2,1)

EEG_3D_mu_beta_laplacian = np.array([EEG_mu_beta_laplacian[start:start +
window_size, :] for start in window_pos])
EEG_3D_mu_beta_laplacian_T = EEG_3D_mu_beta_laplacian.transpose(0,2,1)

print(EEG_mu_beta_car.shape)
print(EEG_3D_mu_beta_car.shape)
print(EEG_3D_mu_beta_car_T.shape)
```

Results for dataset A:

1. Original data shape: (190,473, 59)
2. Reshaped data shape: (200, 800, 59)
3. Transposed data shape: (200, 59, 800)

Results for dataset B:

- Original data shape: (190,473, 59)
- Reshaped data shape: (200, 785, 59)
- Transposed data shape: (200, 59, 785)

Train & Test split:

Before applying feature extraction methods, it is crucial to split the data into training and testing sets. This ensures that the model is evaluated on unseen data, providing a better measure of its performance.

The train_test_split function from sklearn.model_selection is used to split the data. We stratify the split based on the class labels to maintain the same proportion of classes in both the training and testing sets.

Here is the code to perform the split:

```
from sklearn.model_selection import train_test_split

EEG_3D_mu_beta_T_train, EEG_3D_mu_beta_T_test, y_mu_beta_train,
y_mu_beta_test = train_test_split(EEG_3D_mu_beta_T, window_label,
test_size=0.25, stratify=window_label, random_state=30)
print(EEG_3D_mu_beta_T_train.shape)

EEG_3D_mu_beta_car_T_train, EEG_3D_mu_beta_car_T_test,
y_mu_beta_car_train, y_mu_beta_car_test =
train_test_split(EEG_3D_mu_beta_car_T, window_label, test_size=0.25,
stratify=window_label, random_state=30)
print(EEG_3D_mu_beta_car_T_train.shape)

EEG_3D_mu_beta_laplacian_T_train, EEG_3D_mu_beta_laplacian_T_test,
y_mu_beta_laplacian_train, y_mu_beta_laplacian_test =
train_test_split(EEG_3D_mu_beta_laplacian_T, window_label, test_size=0.25,
stratify=window_label, random_state=30)
print(EEG_3D_mu_beta_laplacian_T_train.shape)
```

Results after splitting on datatset A:

- For the original data:
  - Training data shape: `(150, 59, 800)`
  - Testing data shape: `(50, 59, 800)`
- For the car-filtered data:
  - Training data shape: `(150, 59, 800)`
  - Testing data shape: `(50, 59, 800)`
- For the Laplacian-filtered data:
  - Training data shape: `(150, 59, 800)`
  - Testing data shape: `(50, 59, 800)`

For data set b is the same with 785 samples in each window.

Using Common Spatial Patterns (CSP)

Install mne

CSP tries to maximize the variance between two classes and minimize variance within each class. CSP was applied to the segmented EEG epochs to extract spatial filters that discriminate between different motor imagery tasks.

To perform feature extraction using CSP (Common Spatial Patterns), we first install the `mne` library:

```
pip install mne
```

Apply CSP

We tested different values for `n_components`, which represent the number of features after extraction. We determined that setting `n_components` to the number of channels minus one (58) gave us the best results in classification.

We applied the CSP function to our different filtered datasets:

- `EEG_3D_mu_beta_T_train` (original data)
- `EEG_3D_mu_beta_car_T_train` (car-filtered data)
- `EEG_3D_mu_beta_laplacian_T_train` (Laplacian-filtered data)

For 2D visualization, we used 2 components. When applying CSP, we fit and transform on the training data and then only transform (not fit) on the test data to avoid data leakage.

Here is the code for applying CSP with 58 components:

```
from mne.decoding import CSP

csp58 = CSP(n_components=58, reg='ledoit_wolf', log=True, cov_est='epoch')

EEG_3D_mu_beta_T_train_csp58 = csp58.fit_transform(EEG_3D_mu_beta_T_train,
y_mu_beta_train)
EEG_3D_mu_beta_T_test_csp58 = csp58.transform(EEG_3D_mu_beta_T_test)
```

Other datasets are fitted and transformed like above code.

CSP parameters:

1. **`n_components=58`**:
   - This parameter specifies the number of components to extract using CSP. In this case, it is set to 58, which is the number of channels minus one. This means the CSP will extract 58 spatial filters or features from the EEG data.
2. **`reg='ledoit_wolf'`**:

- The `reg` parameter sets the type of regularization to apply to the covariance matrix. Regularization helps to make the covariance estimation more robust, especially when dealing with high-dimensional data.
- `'ledoit_wolf'` is a type of shrinkage estimator that improves the covariance matrix estimate by reducing its variance. This is particularly useful for EEG data, which can be noisy and high-dimensional.

3. **`log=True`**:
   - The `log` parameter indicates whether to apply a logarithmic transformation to the features after CSP filtering. Setting `log=True` means that the logarithm of the variances of the CSP-filtered signals will be taken.
   - This transformation often helps to stabilize variance and makes the data more suitable for subsequent classification tasks.
4. **`cov_est='epoch'`**:
   - The `cov_est` parameter specifies how the covariance matrices should be estimated.
   - `'epoch'` means that the covariance matrices will be computed from the epochs (individual time windows) of the EEG data. This approach is suitable for data that is segmented into epochs, such as trials in an EEG experiment.

Concatenate Results

After applying CSP to the EEG data, we want to visualize and check the results. To do this, we need to combine the CSP-transformed training and test data into a single dataset. This allows us to plot and compare the features extracted from both sets.

```
EEG_3D_mu_beta_T_csp58 = np.concatenate((EEG_3D_mu_beta_T_train_csp58,
EEG_3D_mu_beta_T_test_csp58), axis=0)
train_test_labels = np.concatenate((y_mu_beta_train, y_mu_beta_test),
axis=0)
print("EEG CSP feature Shape",EEG_3D_mu_beta_T_csp58.shape)
```

```
EEG CSP feature Shape =(200, 58)
```

Plot results

We utilized three distinct methods to visualize the effects of CSP. Given that our features are multidimensional and we aimed to represent them in a two-dimensional space, these methods were chosen for their ability to effectively depict the CSP-transformed data.

Plotting using the First Two Components of CSP

Code:

```
import matplotlib.pyplot as plt
def plot_csp(csp_features, labels , title ):
  plt.figure(figsize=(8, 6))
  n_components = EEG_3D_mu_beta_car_T_csp58.shape[1]

  if n_components == 1:
      plt.scatter(csp_features[:, 0], np.zeros_like(csp_features[:, 0]),
c=labels, cmap='viridis', edgecolor='k')
      plt.xlabel('CSP Component 1')
```

```
        plt.yticks([])
    else:
        plt.scatter(csp_features[:, 0], csp_features[:, 1], c=labels,
cmap='viridis', edgecolor='k')
        plt.xlabel('CSP Component 1')
        plt.ylabel('CSP Component 2')

    plt.title('CSP Features ' + title)
    plt.colorbar(label='Class Label')
    plt.show()
```

Description:

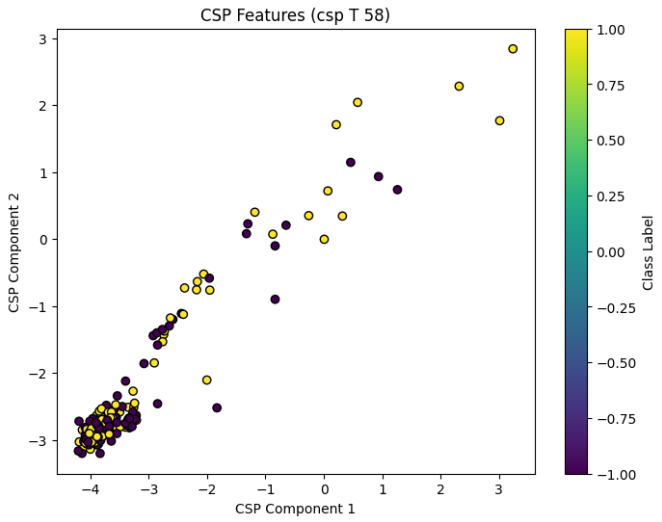
This plot visualizes the CSP features using the first two components derived from the Common Spatial Patterns (CSP) algorithm. The function checks the number of components extracted (`n_components`). If only one component is found, it plots this component against zeros for clarity. Otherwise, it plots the first component on the x-axis and the second component on the y-axis. The color of each point represents its corresponding class label, as indicated by the color bar.

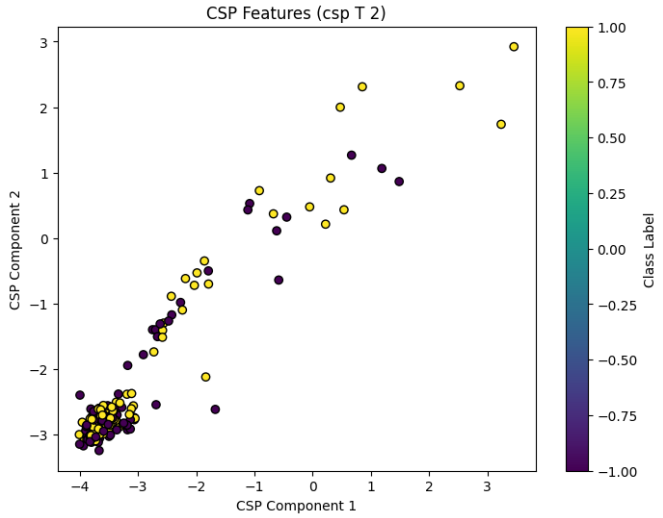
Plots:

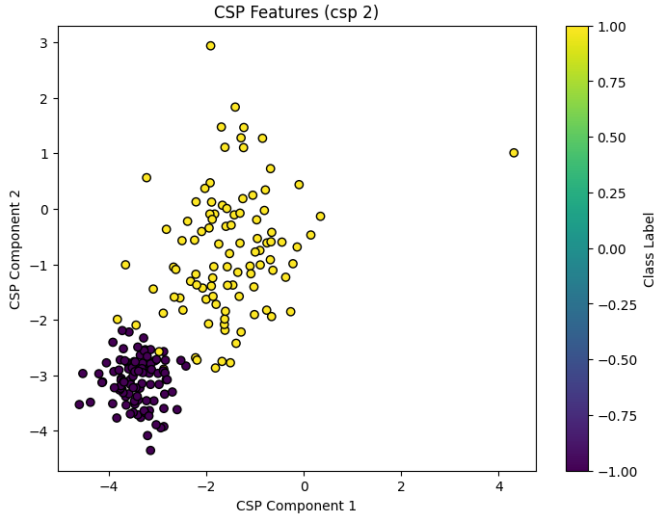
We employed three visualization methods to explore the effects of CSP. The first plot showcases CSP with 58 features after applying the car filter. Plots using Laplacian and no filter yielded similar results to the car filter, so we focused solely on the car-filtered data for this report. The second plot depicts CSP with 2 features after the car filter. The third plot shows CSP with 2 features after the car filter, using a representation assuming 800 channels with each channel containing 59 data points.

In dataset B, CSP feature extraction demonstrated superior performance, effectively separating different classes based on their features. Conversely, in dataset A, the results were more favorable for the not-transformed data, while the performance on the 59-channel data was less pronounced.

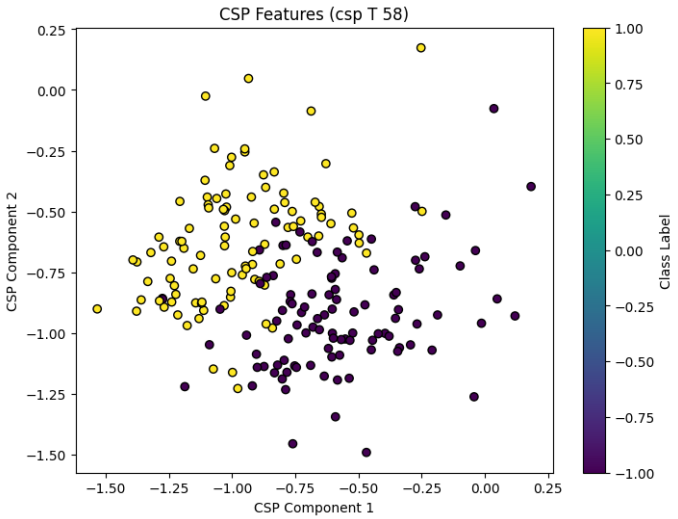
Dataset A:

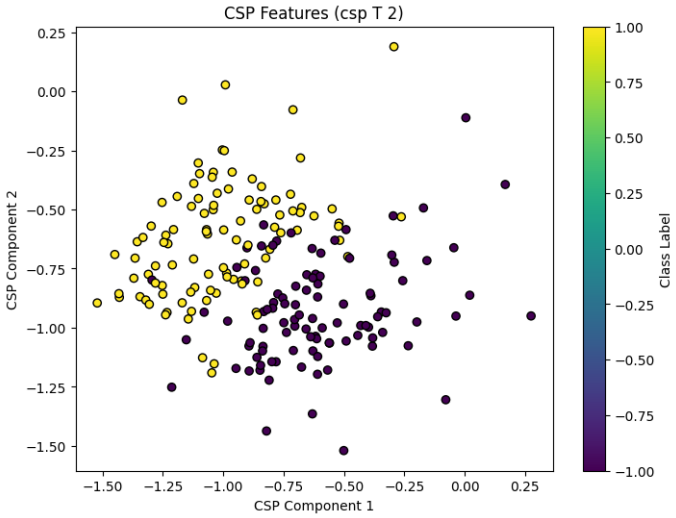


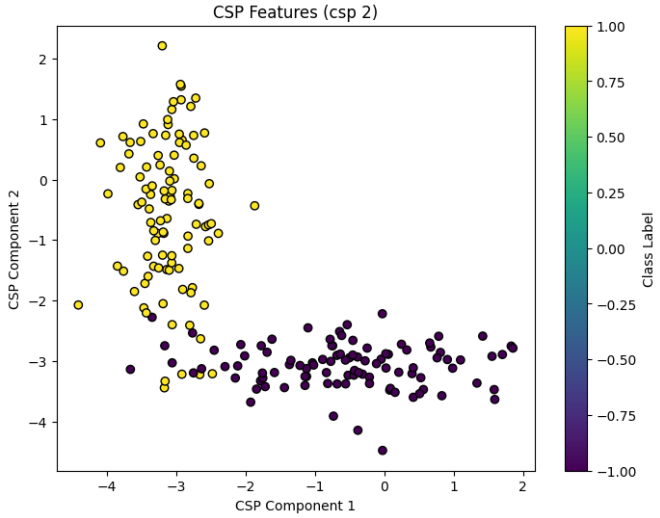

CSP Features (csp T 2)
CSP Component 2
CSP Component 1
Class Label


CSP Features (csp 2)
CSP Component 2
CSP Component 1
Class Label

Dataset B:


CSP Features (csp T 58)
CSP Component 2
CSP Component 1
Class Label





Plotting using PCA (Principal Component Analysis)

Code:

```
from sklearn.decomposition import PCA

def plot_csp_using_pca(EEG, labels, text):

  pca = PCA(n_components=2)
  reduced_features = pca.fit_transform(EEG)

  plt.figure(figsize=(8, 6))
  scatter = plt.scatter(reduced_features[:, 0], reduced_features[:, 1],
c=labels, cmap='viridis', alpha=0.7)
  plt.title('PCA of CSP Features'+ text)
  plt.xlabel('Principal Component 1')
  plt.ylabel('Principal Component 2')
```

```
plt.colorbar(scatter, label='Label')
plt.show()
```

This function initializes a PCA object with two components and fits it to EEG data. It transforms the CSP-transformed features into two principal components using PCA.

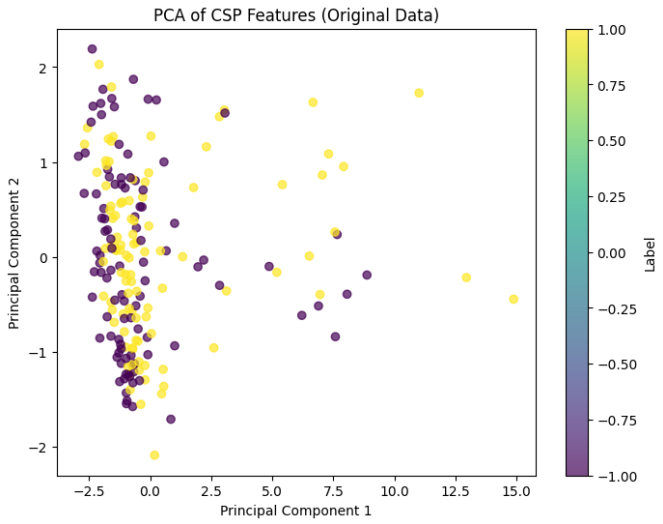
The resulting reduced features are plotted as a scatter plot, where each point represents an EEG sample.

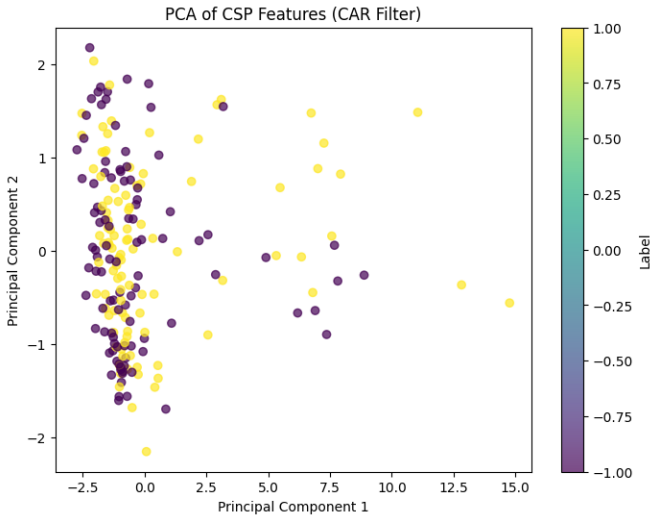
Points are colored according to their class labels, facilitating visual assessment of class separability in the reduced feature space.

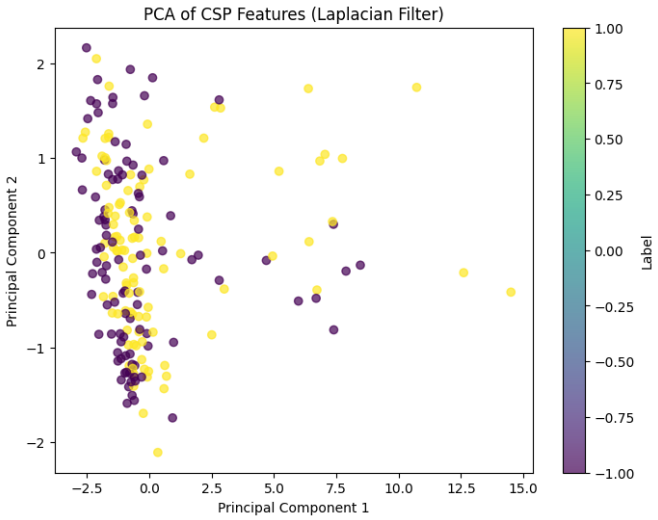
The plot includes labels for the principal components, a title reflecting the use of PCA on CSP features, and a color bar indicating class labels.

Plots:

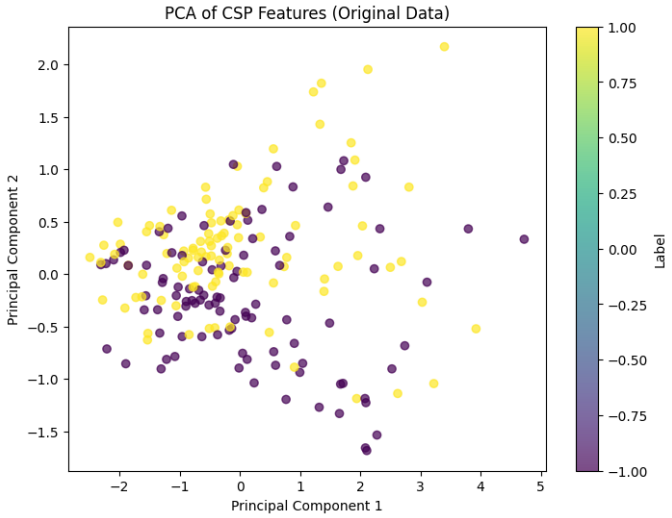
Dataset A:

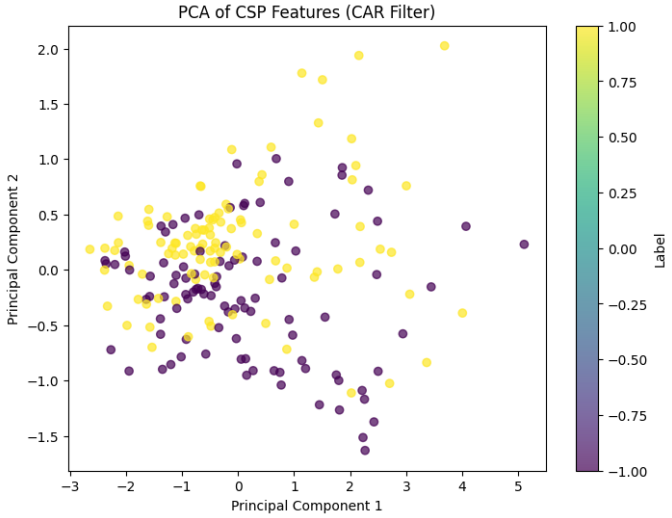


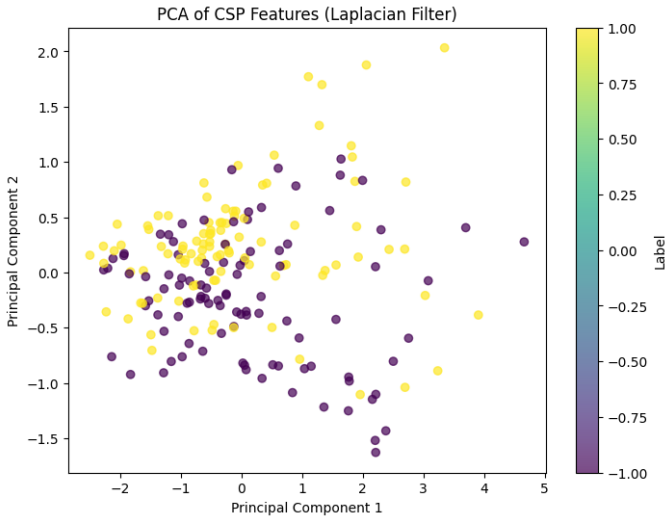

PCA of CSP Features (CAR Filter)
Principal Component 2
Principal Component 1
Label
2
1
0
−1
−2
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


PCA of CSP Features (Laplacian Filter)
Principal Component 2
Principal Component 1
Label
2
1
0
−1
−2
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

Dataset B:


PCA of CSP Features (Original Data)
Principal Component 2
Principal Component 1
Label


PCA of CSP Features (CAR Filter)
Principal Component 2
Principal Component 1
Label


PCA of CSP Features (Laplacian Filter)
Principal Component 2
Principal Component 1
Label

Results:

We plotted the results for the original data, as well as data filtered using the car and Laplacian methods. However, using different spatial filters didn't significantly change how the data looked when plotted. Therefore, we decided to focus on using the results from the car filter. However, this way of showing the data didn't work very well. This led us to believe that using PCA didn't have much effect on improving the visualization of the data.

plot using TSNE

The `plot_csp_using_tsne` function employs t-Distributed Stochastic Neighbor Embedding (t-SNE) to visualize CSP-transformed EEG features in a lower-dimensional space. It takes three parameters: `EEG`, which represents the CSP-transformed features; `labels`, an array indicating the class labels for each feature vector; and `text`, additional text used in the plot title.

code:

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42)
reduced_features_tsne = tsne.fit_transform(EEG_3D_mu_beta_car_T_csp58)

plt.figure(figsize=(10, 8))
scatter = plt.scatter(reduced_features_tsne[:, 0],
reduced_features_tsne[:, 1], c=train_test_labels, cmap='viridis',
alpha=0.7)
plt.title('t-SNE of CSP Features')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.colorbar(scatter, label='Label')
plt.show()
```

Description:

This function initializes a t-SNE object with two components and a fixed random state for reproducibility.

It applies t-SNE to transform the CSP-transformed features into a two-dimensional space.

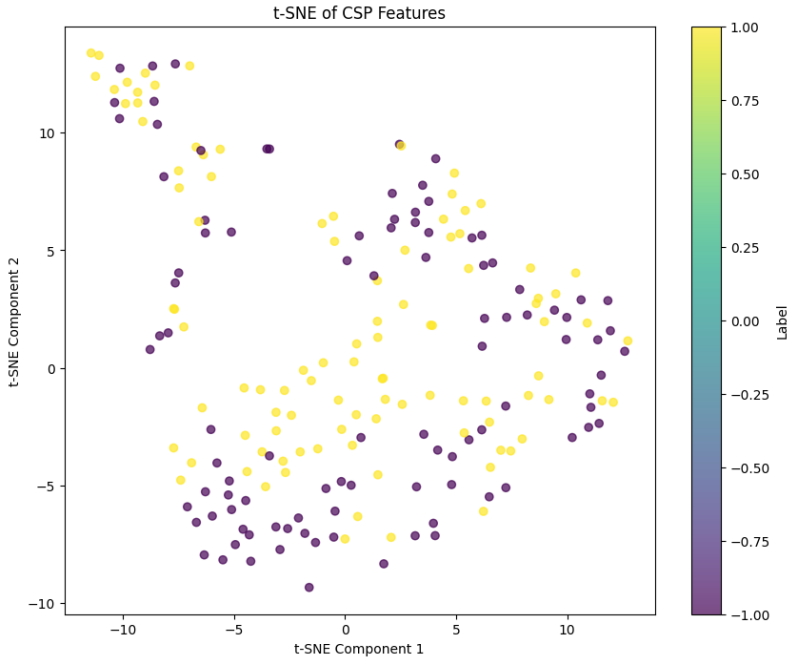
The transformed features are plotted as a scatter plot, where each point represents an EEG sample.

Points are colored according to their class labels, allowing for visual inspection of class distribution in the transformed feature space.

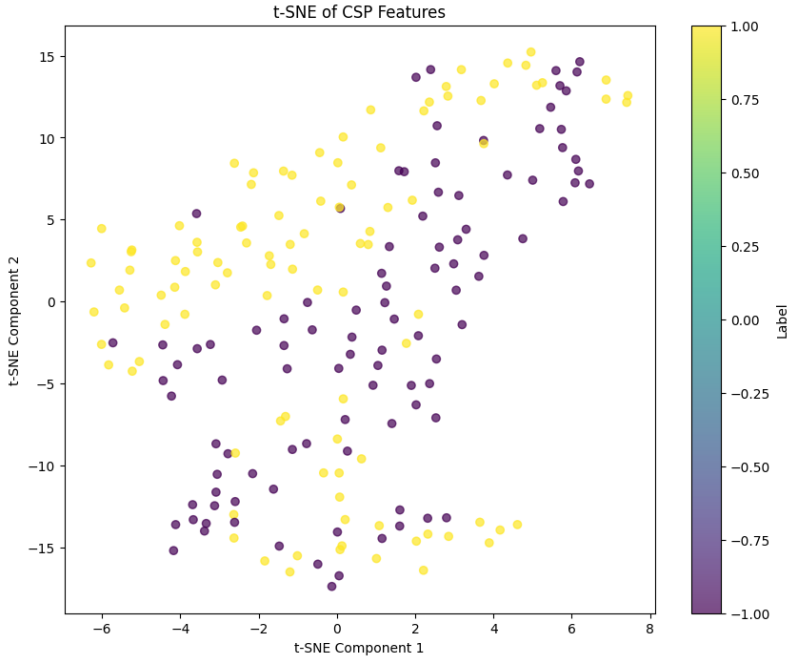
The plot includes labels for the t-SNE components, a title reflecting the use of t-SNE on CSP features, and a color bar indicating class labels.

Plot:

Dataset A:



Dataset B:



We exclusively tested this method on the car-filtered dataset. Similar to previous methods, the results were more favorable for dataset B.

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a technique used for dimensionality reduction in datasets. It identifies orthogonal components that explain the maximum variance in the data, allowing for a simplified representation.

We applied PCA in two scenarios: first, after extracting 58 features using CSP, and second, directly on the original car-filtered EEG data.

Find best n-component:

To determine the optimal number of PCA components that explain at least 95% of the variance, we utilized the following code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca = PCA()
pca.fit(EEG_3D_mu_beta_car_T_train_csp58)

plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance by Number of PCA Components')
plt.grid(True)
plt.show()
```

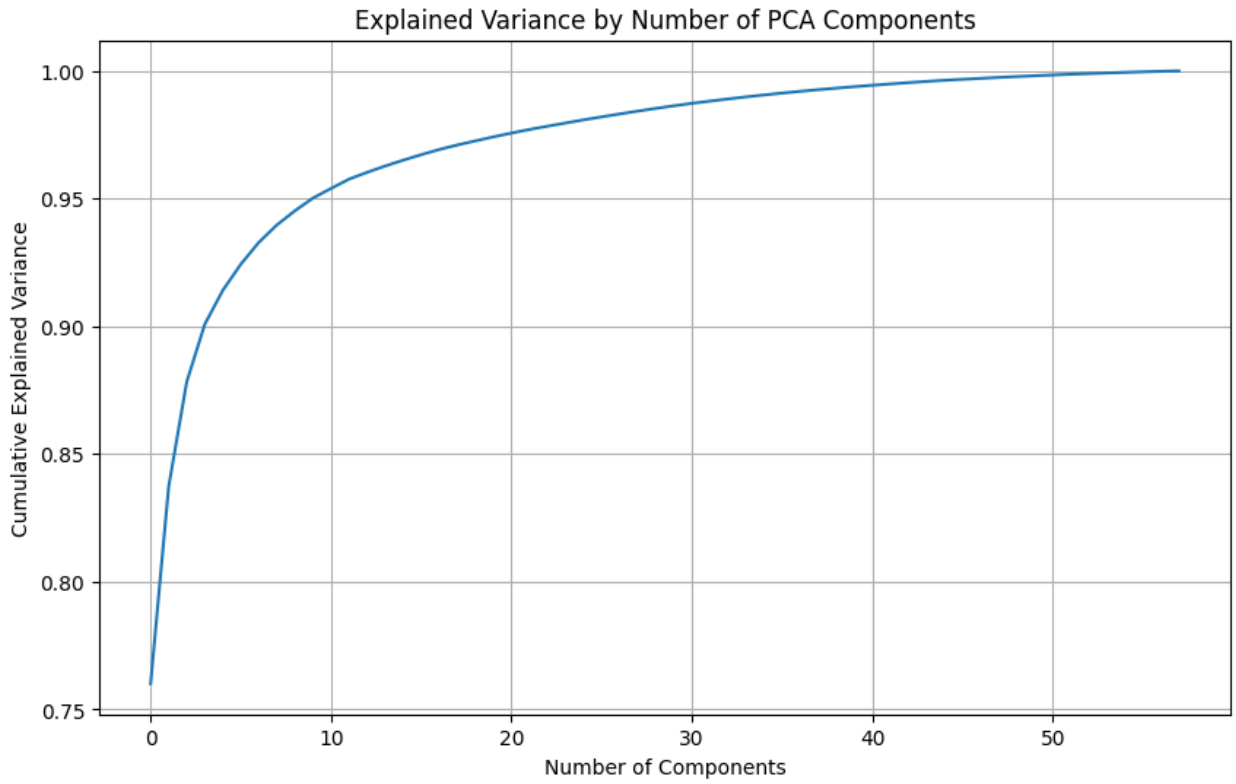
for applying it on original data we used this code before each pca to flatten 3-D data.

```
n_trials, n_channels , n_samples= EEG_3D_mu_beta_car.shape
EEG_3D_mu_beta_car_reshaped = EEG_3D_mu_beta_car.reshape(n_trials, -1)
print(EEG_3D_mu_beta_car_reshaped.shape)
```

the shape is `(200, 47200)` for dataset A and `(200, 46315)` for dataset B because of different window size.

Best number of components is 10 for dataset A after csp, and 161 on original data. For dataset B it is25 and 171 accorgingly.

Sample output for dataset A after csp:




```
Number of components to explain 95.0% variance: 10
```

Applying PCA with Optimal Components:

After determining the optimal number of components (n_components), we applied PCA as follows:

PCA after CSP:

```
pca = PCA(n_components=10)
EEG_3D_mu_beta_car_T_train_csp58_pca10 =
pca.fit_transform(EEG_3D_mu_beta_car_T_train_csp58)
EEG_3D_mu_beta_car_T_test_csp58_pca10
=  pca.transform(EEG_3D_mu_beta_car_T_test_csp58)
print("train set shape:",EEG_3D_mu_beta_car_T_train_csp58_pca10.shape)
print("test set shape:",EEG_3D_mu_beta_car_T_test_csp58_pca10.shape)
```

Description**:** This code snippet applies PCA to the CSP-transformed EEG data using the determined optimal number of components (`n_components`). It transforms both the training and test datasets to reduce dimensionality while preserving important variance.

results for dataset A:

```
train set shape: (150, 10)
test set shape: (50, 10)
```

results for dataset B:

```
train set shape: (150, 25)
test set shape: (50, 25)
```

PCA on Original Car-Filtered Data:

For the original car-filtered EEG data, we applied PCA using the following approach:

For best n-components:

```
n_trials, n_channels , n_samples= EEG_3D_mu_beta_car.shape
EEG_3D_mu_beta_car_reshaped = EEG_3D_mu_beta_car.reshape(n_trials, -1)
print(EEG_3D_mu_beta_car_reshaped.shape)

n_components = 160

pca = PCA(n_components=n_components)
EEG_3D_mu_beta_car_pca160 = pca.fit_transform(EEG_3D_mu_beta_car_reshaped)

print(EEG_3D_mu_beta_car_pca160.shape)
```

for 2-components:

```
n_components = min(EEG_3D_mu_beta_car_reshaped.shape[1], 2)

pca = PCA(n_components=n_components)
EEG_3D_mu_beta_car_pca2 = pca.fit_transform(EEG_3D_mu_beta_car_reshaped)

print(EEG_3D_mu_beta_car_pca2.shape)
```

Description: This section reshapes the car-filtered EEG data into a format suitable for PCA. It then applies PCA with a larger number of components (160) to capture detailed variance and with a minimal number (2) for 2D visualization purposes. These transformations aim to reduce the dimensionality of the data while preserving essential variance information.

Plot PCA

Code:

```
import numpy as np
from sklearn.decomposition import PCA
def plot_pca(pca_features, labels, title) :
  plt.figure(figsize=(8, 6))

  if n_components == 1:
      plt.scatter(pca_features[:, 0], np.zeros_like(pca_features[:, 0]),
c=labels, cmap='viridis', edgecolor='k')
```

```
        plt.xlabel('Principal Component 1')
        plt.yticks([])
    else:
        plt.scatter(pca_features[:, 0], pca_features[:, 1], c=labels,
cmap='viridis', edgecolor='k')
        plt.xlabel('Principal Component 1')
        plt.ylabel('Principal Component 2')

    plt.title(title)
    plt.colorbar(label='Class Label')
    plt.show()
```

description:

The plot_pca function is designed to visually represent the results of Principal Component Analysis (PCA), a technique used for reducing the dimensionality of data while preserving its variance. This function accepts PCA-transformed features (pca_features) and corresponding class labels (labels) to plot them in either one-dimensional or two-dimensional space.

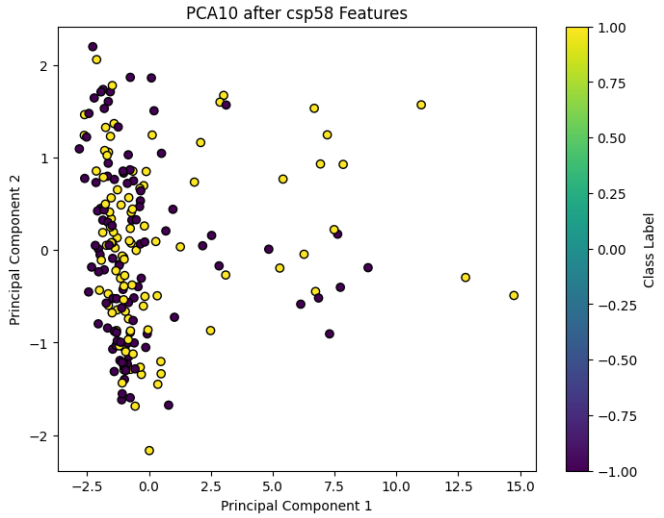
If the pca_features have only one principal component (n_components == 1), the function creates a scatter plot along the x-axis, labeled as 'Principal Component 1'. The y-axis is suppressed (yticks=[]), and each data point is color-coded according to its class label using the 'viridis' colormap.

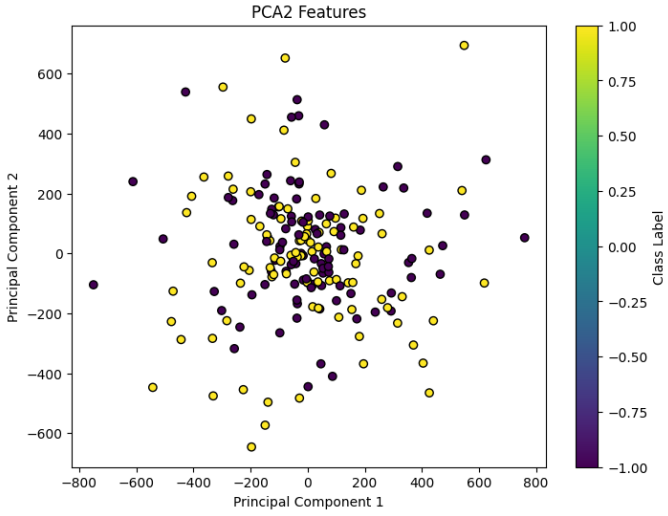
In the case of two principal components (n_components == 2), the function plots the data points in a two-dimensional scatter plot, with 'Principal Component 1' on the x-axis and 'Principal Component 2' on the y-axis. Again, each point is colored based on its class label.

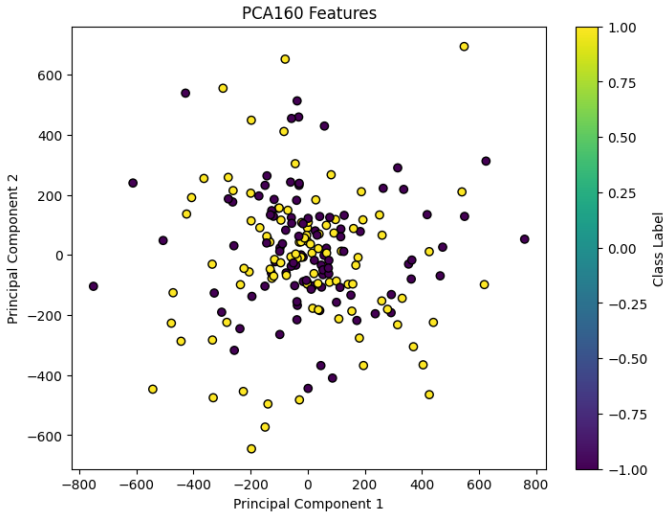
The title of the plot (title) provides context or a description related to the PCA visualization. A color bar is included in the plot, indicating the corresponding class labels of the data points.

Plots:

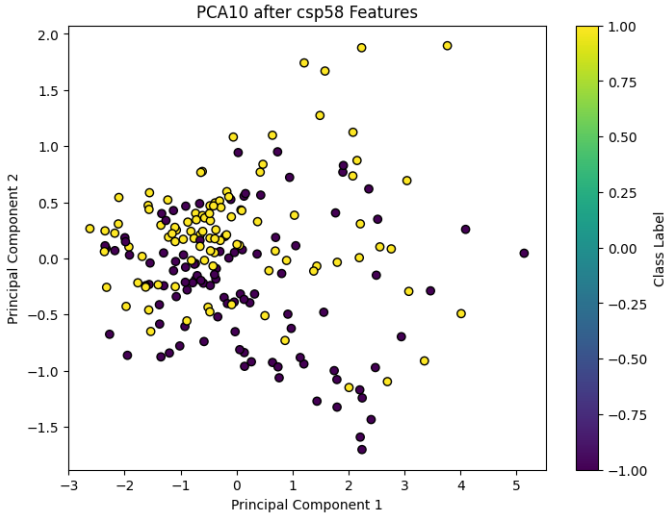
Dataset A:

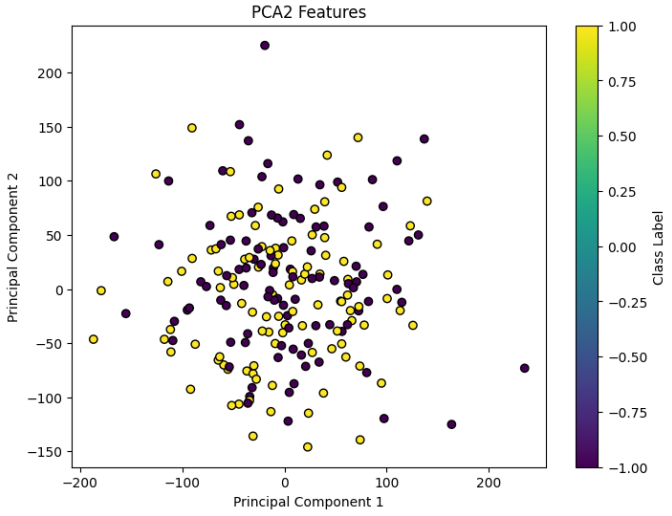

PCA10 after csp58 Features
Principal Component 2
Principal Component 1
Class Label

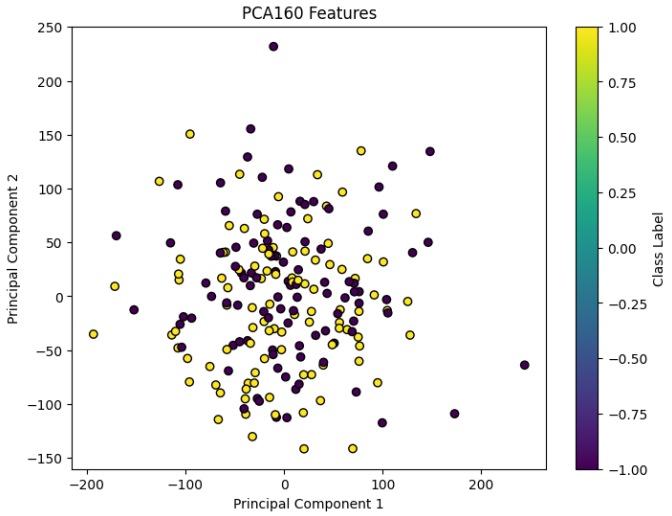

PCA2 Features
Principal Component 2
Principal Component 1
Class Label


PCA160 Features
Principal Component 2
Principal Component 1
Class Label

Dataset B:


PCA10 after csp58 Features
Principal Component 2
Principal Component 1
Class Label


PCA2 Features
Principal Component 2
Principal Component 1
Class Label


PCA160 Features
Principal Component 2
Principal Component 1
Class Label

The first plot displays PCA applied after CSP feature extraction. The second plot shows PCA applied directly on the original data with n-components set to 2. The third plot illustrates PCA applied on the original data, focusing on its two principal components.

In comparison to CSP results, PCA did not effectively extract discriminative features. The plots indicate that different classes are not well-separated using PCA, suggesting limited effectiveness in distinguishing between motor imagery classes based on the extracted features.

Using Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is a dimensionality reduction technique that seeks to maximize the separation between multiple classes in the data by projecting it onto a lower-dimensional space.

Similar to PCA, we applied LDA in two ways. Firstly, we applied it after CSP feature extraction:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
LDA

n_classes = len(np.unique(window_label))
n_features = EEG_3D_mu_beta_car_T_train_csp58.shape[1]
n_components = min(n_features, n_classes - 1)
lda = LDA(n_components=n_components)
EEG_3D_mu_beta_car_T_train_csp58_lda =
lda.fit_transform(EEG_3D_mu_beta_car_T_train_csp58, y_mu_beta_car_train)
EEG_3D_mu_beta_car_T_test_csp58_lda =
lda.transform(EEG_3D_mu_beta_car_T_test_csp58)
print(EEG_3D_mu_beta_car_T_train_csp58_lda.shape)
```

This code applies LDA on CSP features to project the data into a discriminative subspace suitable for classification.

Number of classes is the number of unique labels and number of features are the second dimension of data. Suitable n-componets for LDA is equal to “Number of classes -1” if it is less than current features. We fit and transform LDA on out data.

Next, we applied LDA directly on the original data:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
LDA

n_trials, n_samples, n_channels = EEG_3D_mu_beta_car.shape
EEG_3D_mu_beta_car_reshaped = EEG_3D_mu_beta_car.reshape(n_trials, -1)
n_classes = len(np.unique(window_label))
n_features = EEG_3D_mu_beta_car_reshaped.shape[1]
n_components = min(n_features, n_classes - 1)
lda = LDA(n_components=n_components)
EEG_3D_mu_beta_car_lda = lda.fit_transform(EEG_3D_mu_beta_car_reshaped,
window_label)
```

```
print(EEG_3D_mu_beta_car_lda.shape)
```

In this code we used not transformed 3-D reshaped data into 2 dimention to be appropriate for LDA. Then we applied LDA on it.

Since our dataset consists of only two classes (left and right/foot), the output feature of LDA is one-dimensional in both datasets.

To visualize the results of LDA, we used the following plotting function:

```
def plot_lda(pca_features, labels, title):
  plt.figure(figsize=(8, 6))

  if n_components == 1:
      plt.scatter(pca_features[:, 0], np.zeros_like(pca_features[:, 0]),
c=labels, cmap='viridis', edgecolor='k')
      plt.xlabel('Linear Discriminant 1')
      plt.yticks([])
  else:
      plt.scatter(lda_features[:, 0], pca_features[:, 1], c=labels,
cmap='viridis', edgecolor='k')
      plt.xlabel('Linear Discriminant 1')
      plt.ylabel('Linear Discriminant 2')

  plt.title('LDA Features'+ title)
  plt.colorbar(label='Class Label')
  plt.show()
```

This function is similar to plot-pca that is mentioned before.

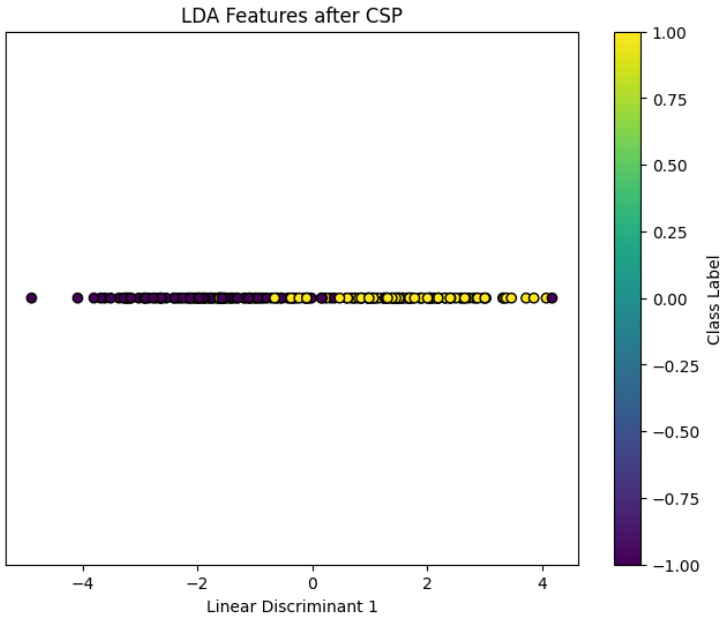
We concatenated test and train results of lda and then plotted the results:

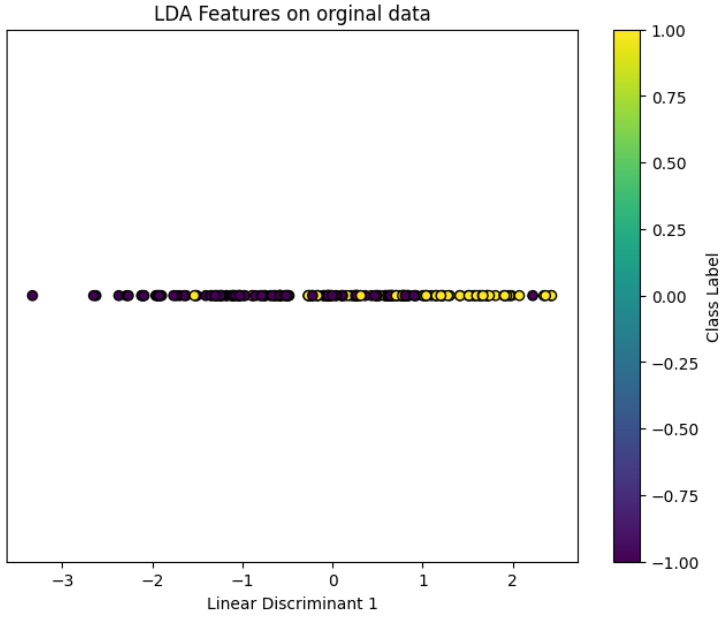
```
EEG_3D_mu_beta_car_T_csp58_lda =
np.concatenate((EEG_3D_mu_beta_car_T_train_csp58_lda,
EEG_3D_mu_beta_car_T_test_csp58_lda), axis=0)
car_train_test_labels = np.concatenate((y_mu_beta_car_train,
y_mu_beta_car_test), axis=0)
print("EEG LDA feature Shape",EEG_3D_mu_beta_car_T_csp58_lda.shape)
```

For both datasets:

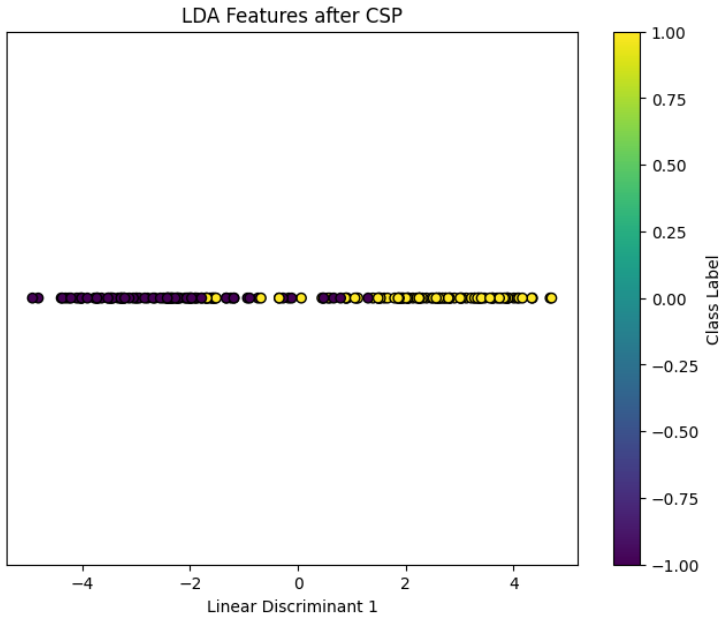
```
EEG LDA feature Shape (200, 1)
```

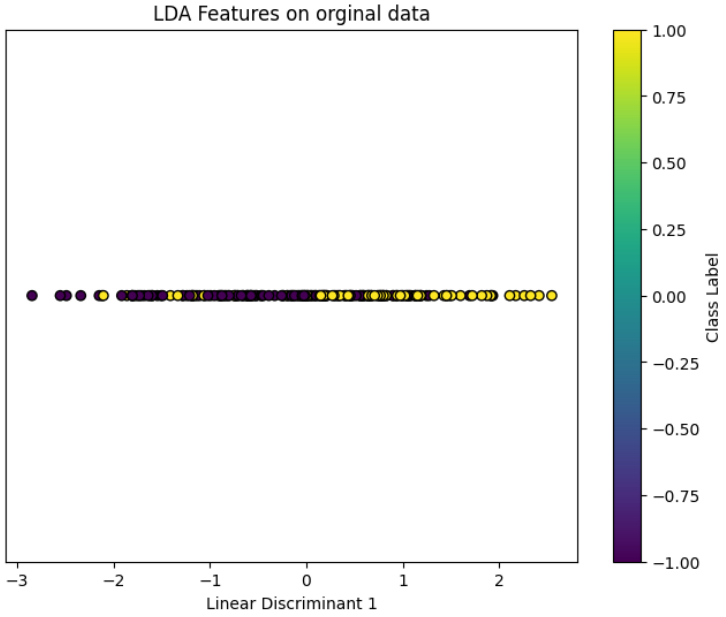
Plots for dataset A:





Plots for dataset B:





LDA effectively separates data in both original and CSP-extracted datasets, showing superior performance when applied after CSP preprocessing. Interestingly, unlike PCA and CSP, it performs better on dataset A compared to dataset B.

Visual Comparisons

CSP, PCA, and LDA offer distinct advantages depending on the dataset characteristics and classification goals. CSP proved effective in spatially separating motor imagery classes, while PCA provided insights into variance distribution, but it didn't perform well in extracting distinctive features.

LDA, although primarily utilized in 1D due to dataset constraints, highlighted challenges in dimensionality reduction for EEG data.

So, between these feature extraction methods, CSP proved to be the best one.

Classification

In this section, after applying various filters, we use the filtered signals named "EEG_3D_mu_beta_car_T_csp58_pca10" to perform classification. For this, according to the code below, we fit 8 different classifiers on the train data and then obtain the evaluation metrics using the test data.

```
def classification(X_train, y_train, X_test, y_test):
  models = {
      'Logistic Regression': LogisticRegression(),
      'Decision Tree': DecisionTreeClassifier(),
      'Random Forest': RandomForestClassifier(),
      'SVM': SVC(),
      'KNN': KNeighborsClassifier(),
      'Gradient Boosting': GradientBoostingClassifier(),
      'AdaBoost': AdaBoostClassifier(),
      'Naive Bayes': GaussianNB(),
      'MLP Neural Network': MLPClassifier()
  }

  results = {}
  for name, model in models.items():
      print()
      print(f"\033[38;5;208m{name}\033[0m")
      model.fit(X_train, y_train)
      y_pred = model.predict(X_test)
      accuracy = accuracy_score(y_test, y_pred)
```

DATASET A

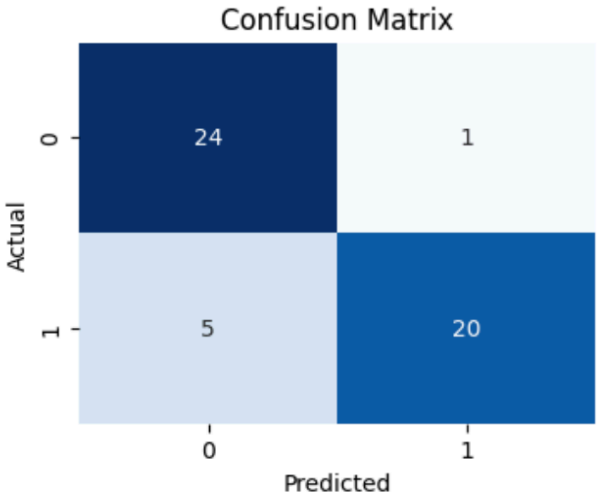
The results of applying the classifiers are presented in the table below.

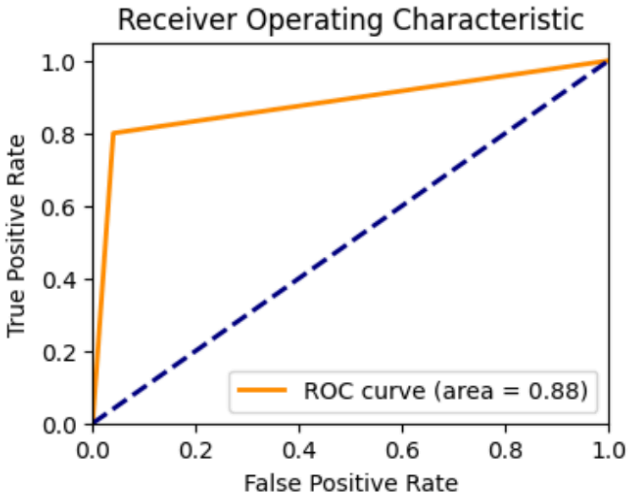
This tables is for data with CAR filter and applying PCA:

classifier	Accuracy	Precision	Recall	F1-SCORE
Logistic Regression	82	94	68	79
Decision Tree	72	79	60	68
Random Forest	80	100	60	75
SVM	84	90	76	83
KNN	68	76	52	62
Gradient Boosting	82	94	68	79
AdaBoost	78	94	60	73
Naïve Bayes	80	94	64	76
MLP	88	95	80	87

Table 1 – Evaluation Matrix of Classifiers with PCA

The ROC Curve and Confusion Matrix for all classifiers have been plotted. For example, the following results were obtained for the MLP:





This tables is for data with CAR filter and applying LDA:

classifier	Accuracy	Precision	Recall	F1-SCORE
Logistic Regression	74	80	64	71
Decision Tree	72	79	60	68
Random Forest	72	79	60	68
SVM	74	80	64	71
KNN	78	89	64	74
Gradient Boosting	72	79	60	68
AdaBoost	72	79	60	68
Naïve Bayes	74	80	64	71
MLP	74	80	64	71

Table 2 – Evaluation Matrix of Classifiers with LDA

As we can see results of classification with PCA data are much better than classification with LDA data.

DATASET B

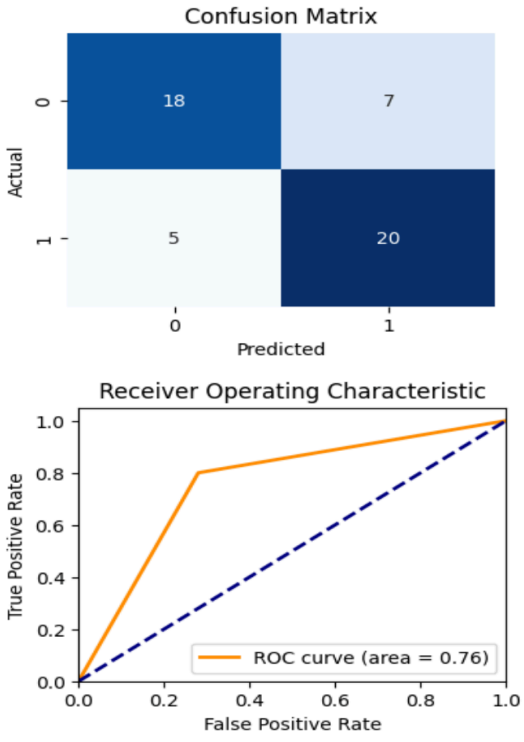
The results of applying the classifiers are presented in the table below.

This tables is for data with CAR filter and applying PCA:

classifier	Accuracy	Precision	Recall	F1-SCORE
Logistic Regression	84	81	88	85
Decision Tree	70	71	68	69
Random Forest	74	73	76	75
SVM	80	83	76	79
KNN	78	75	84	79
Gradient Boosting	74	75	72	73
AdaBoost	66	68	60	64
Naïve Bayes	82	83	80	82
MLP	76	74	80	77

Table 3 – Evaluation Matrix of Classifiers with PCA

The ROC Curve and Confusion Matrix for all classifiers have been plotted. For example, the following results were obtained for the MLP:



This tables is for data with CAR filter and applying LDA:

classifier	Accuracy	Precision	Recall	F1-SCORE
Logistic Regression	82	83	80	82
Decision Tree	78	82	72	77
Random Forest	78	82	72	77
SVM	80	80	80	80
KNN	82	83	80	82
Gradient Boosting	78	82	72	77
AdaBoost	78	82	72	77
Naïve Bayes	82	83	80	82
MLP	82	83	80	82

Table 4 – Evaluation Matrix of Classifiers with LDA

Average accuracy of data using PCA is 76 and average accuracy of data using LDA is 80. In this data set accuracy of classification using LDA is more than classification using PCA.

Then, we tested some of the classifiers (with PCA data)that had good accuracy with different parameters to achieve better accuracy.

MLP

DATASET A

By adjusting the parameters of the MLP classifier, we achieved an accuracy of 90% with the following evaluation metrics:

Precision: 95%, Recall: 84%, F1-Score: 89%

MLP Parameters:

```
mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=200,
random_state=42)
```

Gradient Boosting Classifier

Gradient Boosting Classifier (GBC) is an ensemble learning method that combines the predictions of several weak learners (typically decision trees) to improve predictive accuracy. It sequentially fits new models to provide a more accurate prediction in areas where previous models performed poorly. Here, we use GBC to classify EEG data after feature extraction.

Code:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

def gradient_boosting(X_train, y_train, X_test, y_test):

  param_grid = {
```

```
        'n_estimators': [100, 200],
        'learning_rate': [0.01, 0.1],
        'max_depth': [3,4,5],
        'min_samples_split': [6],
        'min_samples_leaf': [4],
        'subsample': [0.9],
        'max_features': ['log2']
    }

    gbc = GradientBoostingClassifier()
    grid_search = GridSearchCV(gbc, param_grid, cv=5, scoring='accuracy',
verbose=2)
    grid_search.fit(X_train, y_train)

    best_params = grid_search.best_params_
    print(f'Best parameters: {best_params}')
    best_gbc = GradientBoostingClassifier(**best_params)
    best_gbc.fit(X_train, y_train)

    y_pred = best_gbc.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    print(f'Accuracy with best parameters: {accuracy}')
    return best_gbc
```

This function `gradient_boosting` utilizes GridSearchCV to find the optimal hyperparameters for the Gradient Boosting Classifier. It then trains the classifier with these parameters on the training data (`X_train`, `y_train`) and evaluates its performance on the test data (`X_test`, `y_test`). The accuracy of the classifier with the best parameters is printed, and the trained model (`best_gbc`) is returned for further evaluation or prediction.

For optimizing the Gradient Boosting Classifier (GBC) parameters, we performed a grid search using a GPU for efficient computation. Initially, we explored a wide range of hyperparameters to find the best configuration.

Grid Search Parameters:

n_estimators: Number of boosting stages (100, 200, 300).

learning_rate: Step size for each boosting iteration (0.01, 0.1, 0.2).

max_depth: Maximum depth of the individual trees (3, 4, 5).

min_samples_split: Minimum number of samples required to split an internal node (2, 5, 10).

min_samples_leaf: Minimum number of samples required to be at a leaf node (1, 2, 4).

subsample: Fraction of samples used for fitting each tree (0.8, 0.9, 1.0).

max_features: Maximum number of features to consider for splitting a node ('auto', 'sqrt', 'log2').

After identifying the best performing parameters, we refined our grid to focus on the most effective combinations, aiming to reduce computational overhead while maintaining optimal performance.

This approach ensures that our GBC model is well-tuned to handle the complexities of our EEG data classification task efficiently.

We apply gradient boosting on reults of csp 58 + pca 10 and csp 58 + lda.

Results:

Dataset A:

using pca:

```
Best parameters: {'learning_rate': 0.1, 'max_depth': 4, 'max_features':
'log2', 'min_samples_leaf': 4, 'min_samples_split': 6, 'n_estimators':
100, 'subsample': 0.9}
Accuracy with best parameters: 0.78
```

Using lda:

```
Best parameters: {'learning_rate': 0.01, 'max_depth': 3, 'max_features':
'log2', 'min_samples_leaf': 4, 'min_samples_split': 6, 'n_estimators':
200, 'subsample': 0.9}
Accuracy with best parameters: 0.74
```

datasetB:

using pca:

```
Best parameters: {'learning_rate': 0.1, 'max_depth': 3, 'max_features':
'log2', 'min_samples_leaf': 4, 'min_samples_split': 6, 'n_estimators':
200, 'subsample': 0.9}
Accuracy with best parameters: 0.68
```

Using lda:

```
Best parameters: {'learning_rate': 0.01, 'max_depth': 3, 'max_features':
'log2', 'min_samples_leaf': 4, 'min_samples_split': 6, 'n_estimators':
100, 'subsample': 0.9}
Accuracy with best parameters: 0.8
```

In Gradient Boosting Classification PCA works better on Dataset A and lda works better on dataset B. that is unlike vitual results in feature extraction.

Although many things depend on seeds and different seeds may have different results.

AdaBoost Classifier

AdaBoost (Adaptive Boosting) is an ensemble learning technique that combines multiple weak classifiers to create a strong classifier. It sequentially trains weak learners, where each subsequent learner pays more attention to instances misclassified by previous learners. This iterative process allows AdaBoost to focus on difficult cases, improving overall accuracy.

Code:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

def ada_boost(X_train, y_train, X_test, y_test):


  weak_classifier = GaussianNB()
  ada_boost = AdaBoostClassifier(base_estimator=weak_classifier,
n_estimators=100, learning_rate=.01, random_state=42)
  ada_boost.fit(X_train, y_train)
  y_pred = ada_boost.predict(X_test)

  accuracy = accuracy_score(y_test, y_pred)
  print(f"Accuracy: {accuracy:.2f}")
ada_boost(EEG_3D_mu_beta_car_T_train_csp58_pca10, y_mu_beta_car_train,
EEG_3D_mu_beta_car_T_test_csp58_pca10, y_mu_beta_car_test)
```

Parameters:

base_estimator: The base estimator to be boosted. In this case, GaussianNB() is used as the weak classifier.

n_estimators: Number of boosting rounds or weak learners to train (100 in this case).

learning_rate: Weight applied to each classifier's contribution during training (0.01 in this case).

random_state: Seed for random number generation, ensuring reproducibility.

The ada_boost function implements AdaBoost with a Gaussian Naive Bayes (GaussianNB) as the weak classifier. It trains the AdaBoost classifier using n_estimators=100 rounds of boosting, each learner being influenced by misclassifications weighted according to learning_rate=0.01. After training, it predicts labels for the test set and calculates the accuracy of predictions using accuracy_score.

Parameter setting was done using a grid search on different parameters and implementing with the best parameters available in grid search.

Results:

Dataset A:

csp+pca:

```
Accuracy: 0.80
```

```
Csp + lda:
```

```
Accuracy: 0.74
```

```
Dataset B:
```

csp+pca:

```
Accuracy: 0.82
```

```
Csp + lda:
```

```
Accuracy: 0.82
```

K-Nearest Neighbors (KNN) Classifier

K-Nearest Neighbors (KNN) is a non-parametric, instance-based learning algorithm used for classification tasks. It classifies data points based on the majority class among their k nearest neighbors in the feature space. The choice of k affects the model's bias-variance trade-off: smaller values make the model more sensitive to noise, while larger values smooth the decision boundary.

Code:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score

def KNN_classifier(X_train, y_train, X_test, y_test,k):

  param_grid = {'n_neighbors': np.arange(1, 31)}
  knn = KNeighborsClassifier()
  grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
  grid_search.fit(X_train, y_train)
  best_k = grid_search.best_params_['n_neighbors']

  print(f'The best value of k is: {best_k}')

  best_knn = KNeighborsClassifier(n_neighbors=best_k)
  best_knn.fit(X_train, y_train)

  y_pred = best_knn.predict(X_test)

  accuracy = accuracy_score(y_test, y_pred)
  print(f'Accuracy with k={k}: {accuracy}')
```

If the k parameter is not provided manually, the function automatically selects the best n_neighbors found through grid search. The function then predicts labels for the test data (X_test) and evaluates the accuracy of predictions using accuracy_score. This approach ensures that the KNN model is tuned to perform optimally on the given EEG dataset, leveraging the strengths of neighbor-based classification for accurate motor imagery classification tasks.

Results :

Dataset A:

csp+pca:

```
The best value of k is: 24
Accuracy with k=24: 0.72
```

```
Csp + lda:
```

```
The best value of k is: 8
Accuracy with k=8: 0.74
```

```
Dataset B:
```

csp+pca:

```
The best value of k is: 5
Accuracy with k=0: 0.78
```

```
Csp + lda:
```

```
The best value of k is: 1
Accuracy with k=0: 0.78
```

We expected that KNN trained with fewer features would perform better. Previously, there was about a 10% difference in accuracy between LDA and PCA, with LDA performing better using only one feature. However, after changing the seed, we no longer observed this difference, and both methods are performing almost equally well.

Clustering

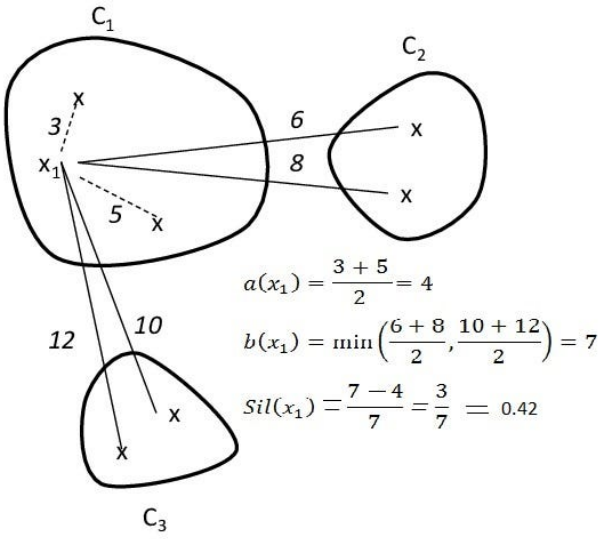
Silhouette Score and Plot

The silhouette score is a metric used to evaluate the quality of clustering. It measures how similar an object is to its own cluster compared to other clusters.

First calculate the average distance to all other points in the same cluster. This is known as a_i. Then calculate the average distance to all points in the nearest neighboring cluster (b_i).
The silhouette score (si) for each point is then given:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

An example of silhouette score:



A silhouette plot is a graphical representation of the silhouette scores of individual data points within clusters. Here's how to determine the best silhouette plot:

1. **High Average Silhouette Score**: The best silhouette plot will have a high average silhouette score across all clusters, ideally close to +1. This indicates that clusters are well-separated and data points are appropriately clustered.
2. **Consistent Width of Silhouettes**: The silhouette widths should be consistent across clusters. Large variations in silhouette widths suggest that some clusters are more compact than others, which can indicate uneven clustering quality.
3. **Positive Silhouette Values**: Most data points should have positive silhouette values, indicating they are closer to their own cluster than to neighboring clusters. A large number of negative values can suggest misclassification.
4. **Balanced Cluster Sizes**: The plot should show clusters of approximately similar sizes. A good clustering solution doesn't have one cluster much larger or smaller than the others unless there's a justified reason for it.

5. **Minimal Overlap**: There should be minimal or no overlap between silhouettes of different clusters. Overlapping silhouettes indicate that clusters are not well-separated.

In summary, the best silhouette plot will have high, positive, and consistent silhouette scores across all clusters with minimal overlap and balanced cluster sizes.

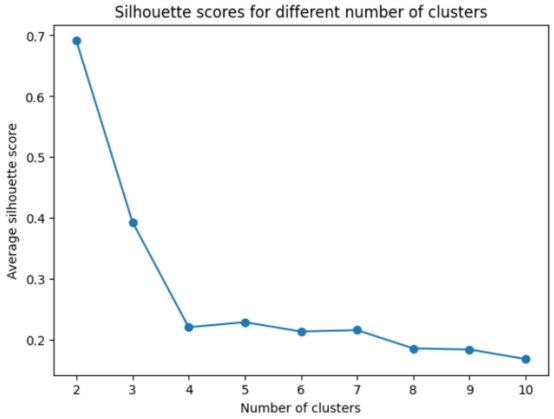
To determine the appropriate number of clusters, we first obtain the silhouette scores for different numbers of clusters and plot their silhouette plots. Then, based on the explanations given above, we select the optimal number of clusters.

To find the appropriate number of clusters, we perform clustering using the K-means method for the number of clusters from 2 to 10, according to the code below. We then obtain their silhouette scores and silhouette plots.

```
def plot_silhouette(data, max_clusters=10):
    range_n_clusters = range(2, max_clusters + 1)
    silhouette_avg_scores = []

    for n_clusters in range_n_clusters:
        fig, ax1 = plt.subplots(1, 1)
        fig.set_size_inches(4, 3)

        # Initialize the clusterer with n_clusters value and a random
generator seed for reproducibility
        clusterer = KMeans(n_clusters=n_clusters, random_state=10)
        cluster_labels = clusterer.fit_predict(data)

        # The silhouette_score gives the average value for all the
samples.
        silhouette_avg = silhouette_score(data, cluster_labels)
        silhouette_avg_scores.append(silhouette_avg)
```

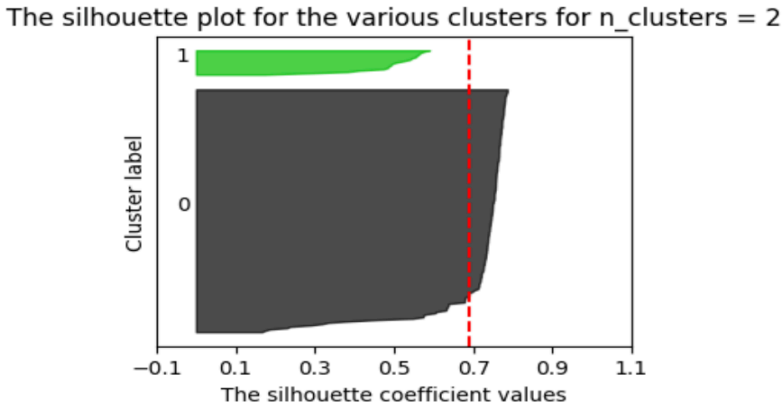
DATASET A

The highest silhouette score is obtained with 2 clusters:



Silhouette plot for 2 cluster:

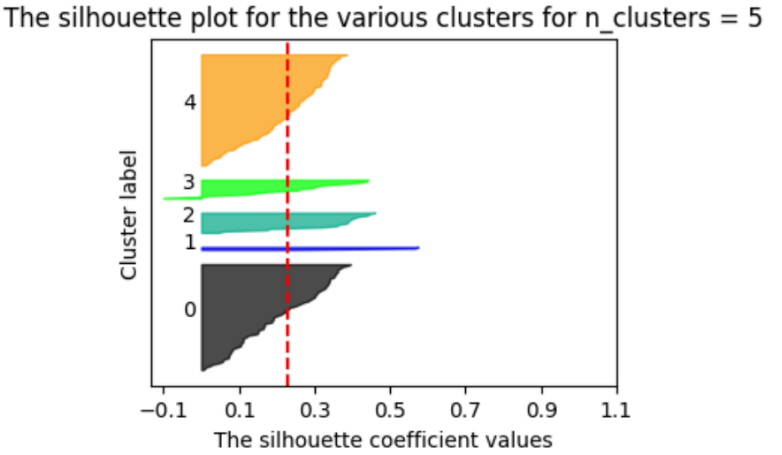
```
For n_clusters = 2, the average silhouette_score is : 0.6914144956357596
```



As can be seen, one of the clusters is much larger than the other, and the clusters are imbalanced.

Silhouette plot for 5 clusters:

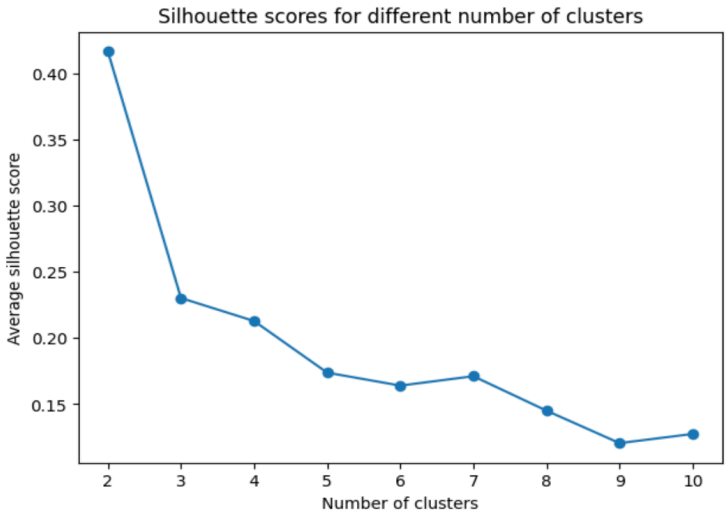
```
For n_clusters = 5, the average silhouette_score is : 0.22917696041580043
```



Clusters are imbalanced and silhouette score is lower than score of 2 clusters.

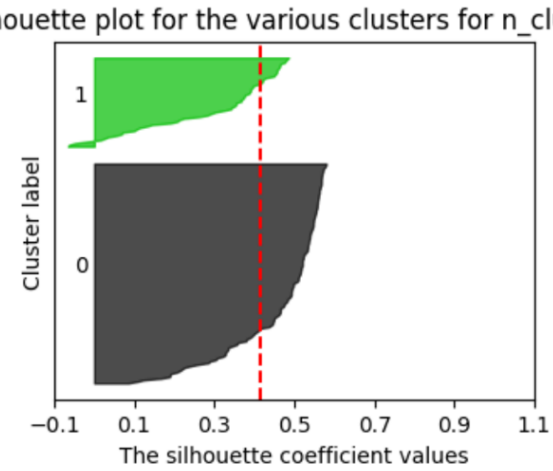
DATASET B

The highest silhouette score is obtained with 2 clusters:



Silhouette plot for 2 cluster:

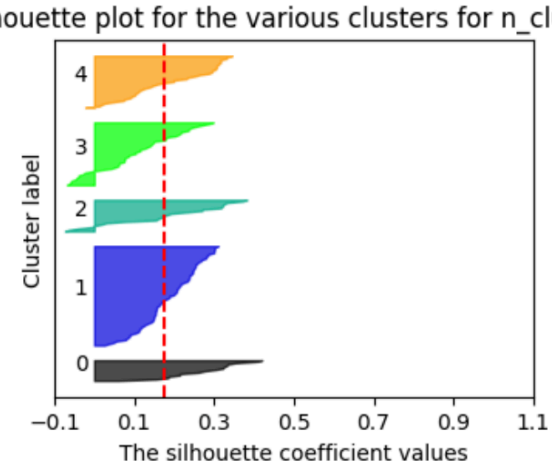
```
For n_clusters = 2, the average silhouette_score is : 0.4166689924158162
```



As can be seen, one of the clusters is much larger than the other, and the clusters are imbalanced.

Silhouette plot for 5 clusters:

```
For n_clusters = 5, the average silhouette_score is : 0.17382354907673617
```



Clusters are imbalanced and silhouette score is lower than score of 2 clusters.

K-Means

Using the code below, we apply the K-means method to find the labels and centers of the clusters.

```
def apply_kmeans(data, n_clusters):


    kmeans = KMeans(n_clusters=n_clusters, random_state=10)

    cluster_labels = kmeans.fit_predict(data)

    cluster_centers = kmeans.cluster_centers_


    return cluster_labels, cluster_centers
```

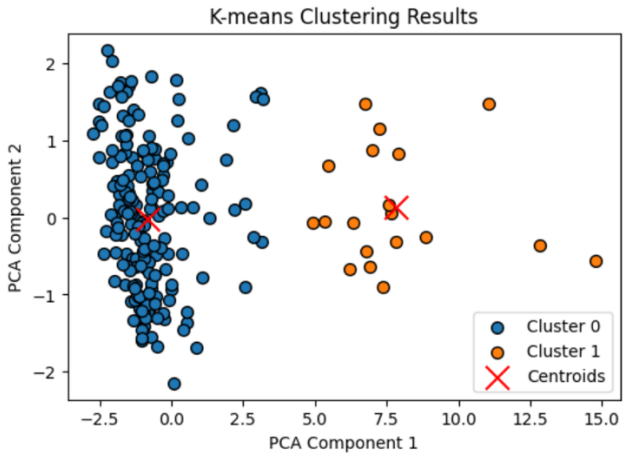
Then, using the subsequent code, we reduce the dimensionality of the data to two dimensions with PCA in order to plot them.

```
def plot_clusters(data, cluster_labels, cluster_centers):


    pca = PCA(n_components=2)

    reduced_data = pca.fit_transform(data)


    plt.figure(figsize=(6, 4))


    # Plotting the clustered data points
```

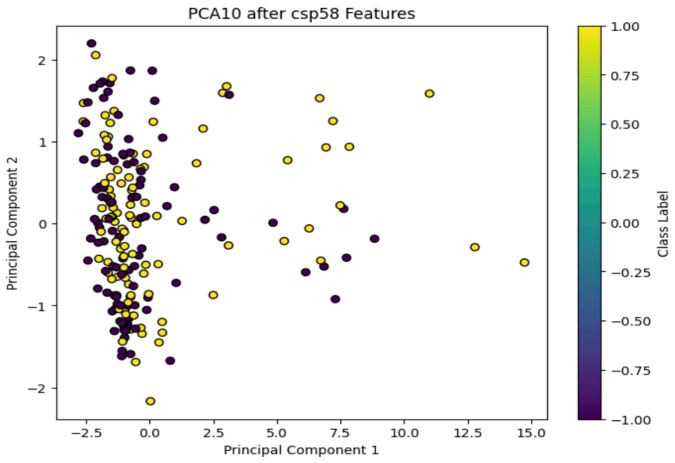
```
    unique_labels = np.unique(cluster_labels)
    colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
    for k, col in zip(unique_labels, colors):
        class_member_mask = (cluster_labels == k)
        xy = reduced_data[class_member_mask]
        plt.scatter(xy[:, 0], xy[:, 1], label=f'Cluster {k}',
edgecolor='k', s=50
```

DATASET A

plot of 2 clusters using k-means:

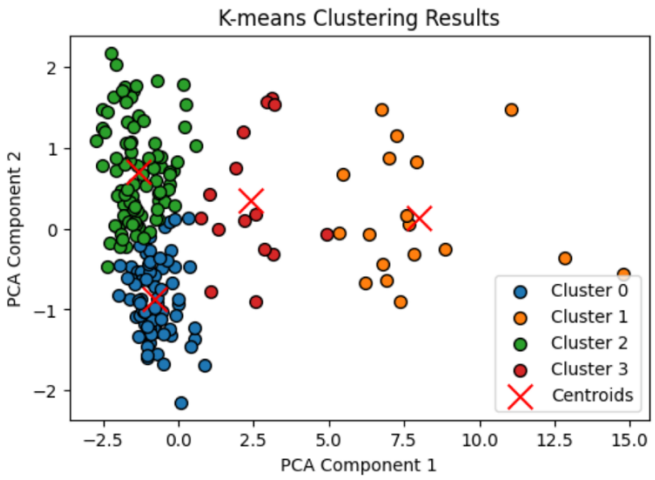


Real labels of data:

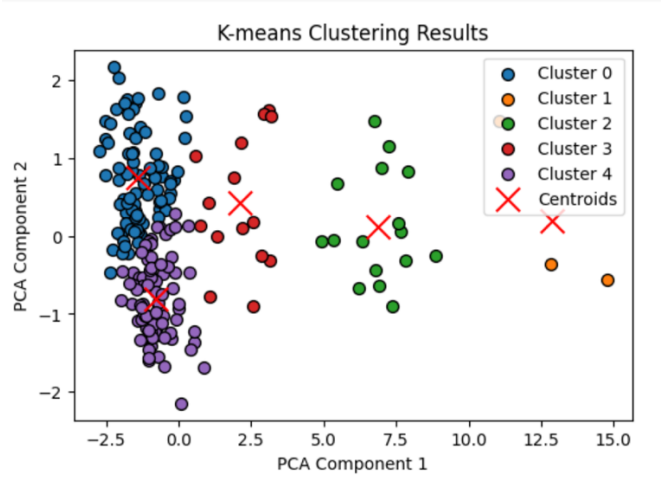


As we can see Due to the nature of the data, it is not possible to separate the data using this clustering method. The data from different clusters are intermixed.

4 clusters:

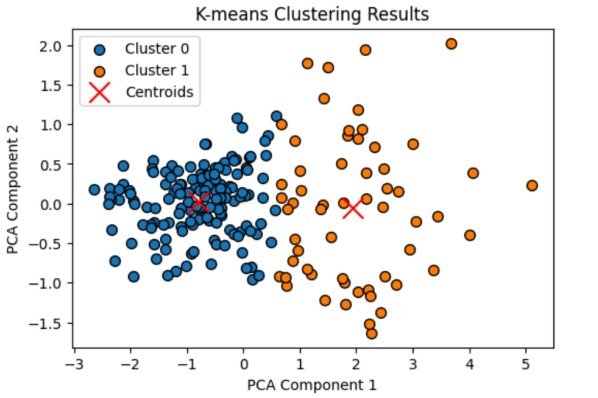


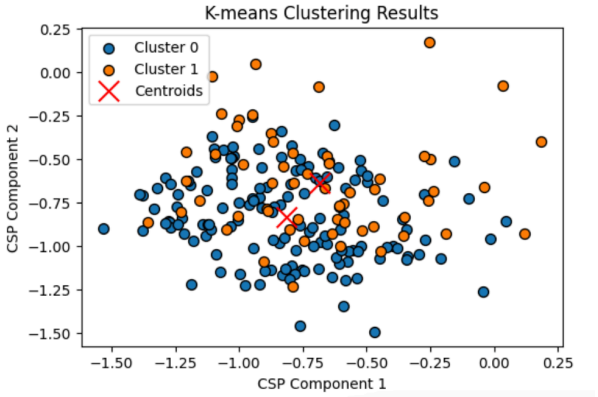
5 clusters:



DATASET B

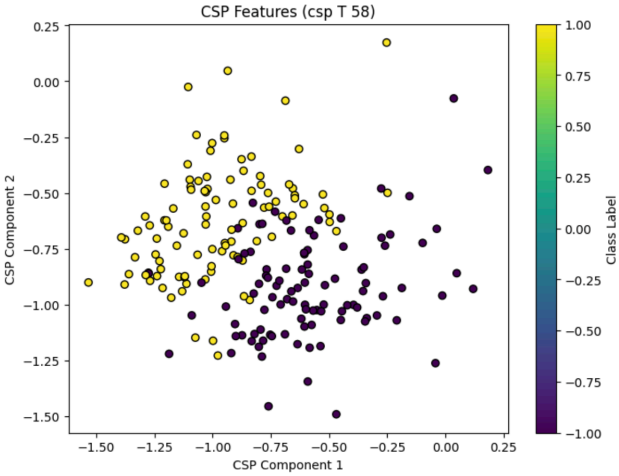
plot of 2 clusters using k-means with 2 different methods PCA and CSP components:



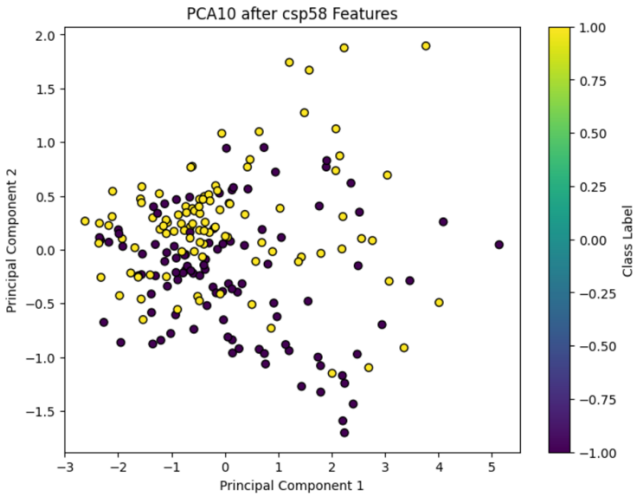


Real labels of data:

CSP:

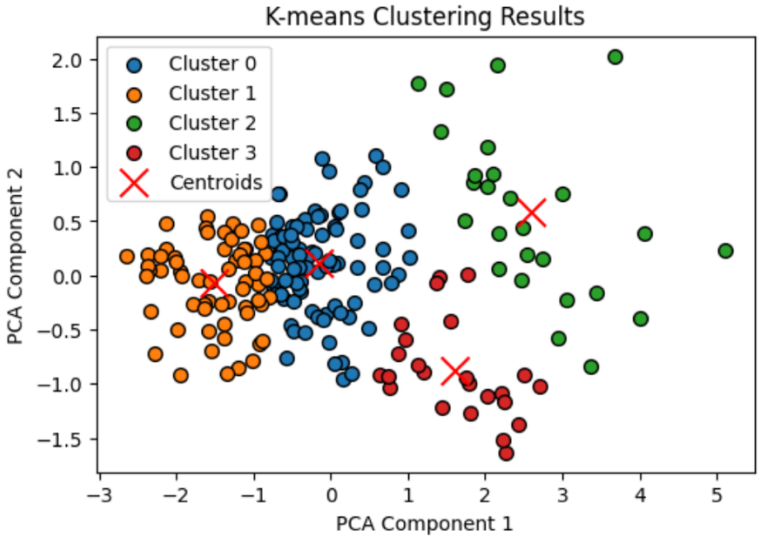


PCA:

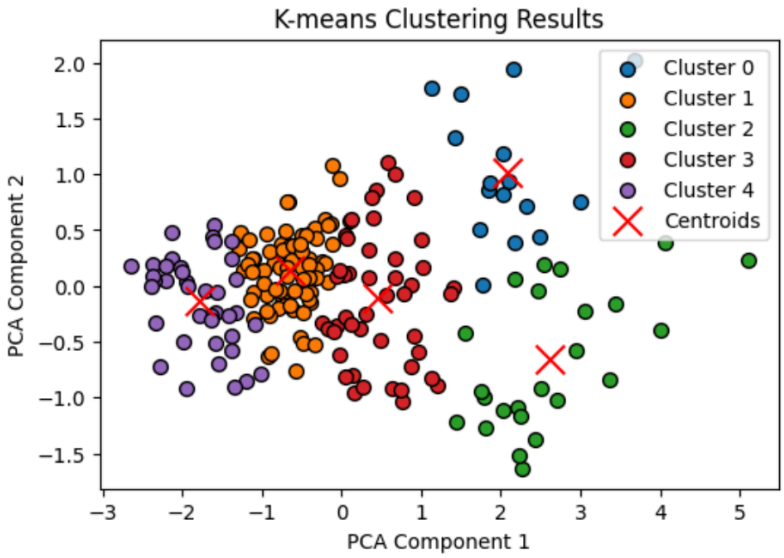


As we can see Due to the nature of the data, it is not possible to separate the data using this clustering method. The data from different clusters are intermixed.

4 clusters:



5 clusters:



DBSCAN

Using the code below, we apply the DB Scan method to find the labels and centers of the clusters.

```
def apply_dbscan(data, eps=0.5, min_samples=5):


    dbscan = DBSCAN(eps=eps, min_samples=min_samples)

    cluster_labels = dbscan.fit_predict(data)


    return cluster_labels
```

To find the optimal parameters for the DBSCAN method, we use grid search.

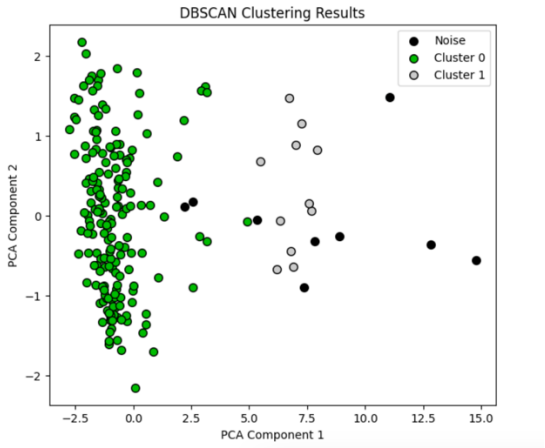
```
epss = [1.3,1.4,1.5,1.6,1.7,1.8,1.9,2,3,4,5]

mins = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

for i in range(11):

  for j in range(20):

    cluster_labels = apply_dbscan(EEG_3D_mu_beta_car_T_csp58, epss[i],
mins[j])

    plot_dbscan_clusters(EEG_3D_mu_beta_car_T_csp58, cluster_labels)
```

```
    print("eps: ", epss[i])

    print("mins: ", mins[j])
```

DATASET A

After performing grid search, the parameters that cluster the data effectively are:

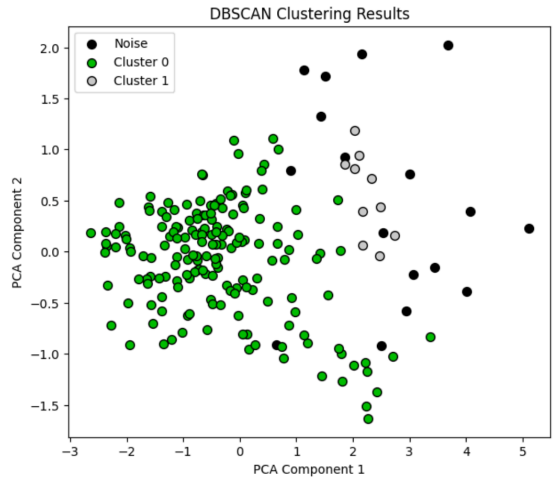
Epsilon = 3, min samples =12



DATASET B

After performing grid search, the parameters that cluster the data effectively are:

Epsilon = 1.3, min samples = 9

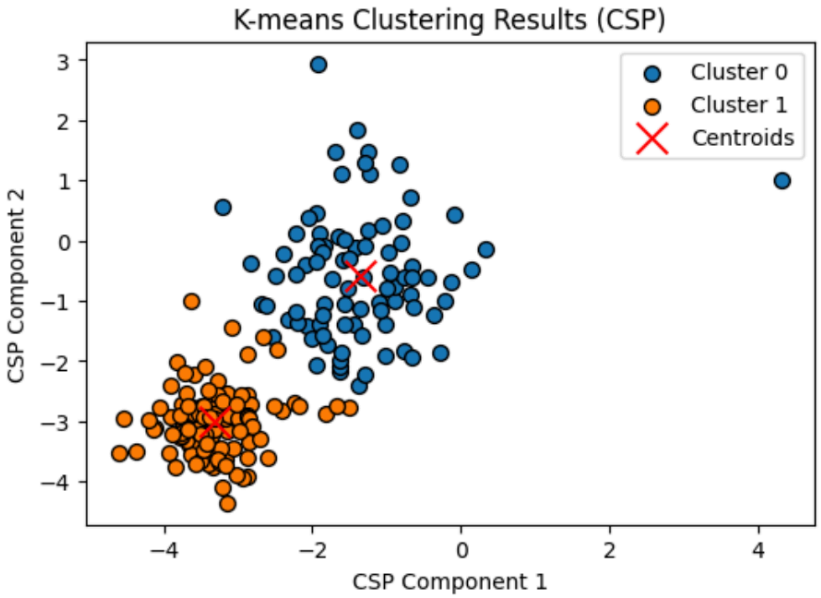



```
eps:  1.3
mins:  9
```

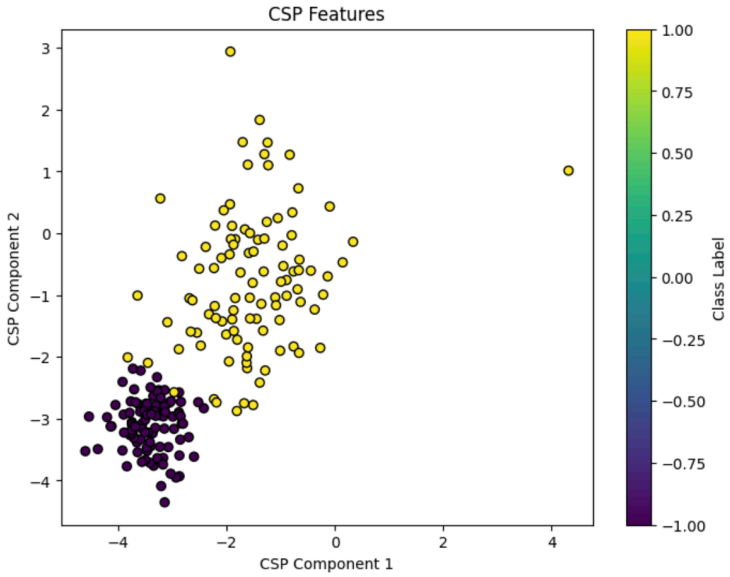
DATASET A

As we know, in CSP, we first specify the number of windows, then the number of channels, and finally the number of samples per window. However, we accidentally discovered that if we change the number of channels and the number of samples per window in CSP, the data separate well for clustering.

Clustering result:



Real labels:



As we can see clustering applied successfully.