Machine Learning
University of Tehran
Instructors: Dr. Tavassolipour, Dr. A. Dehaqani

Final Project Report

EEG-Based Brain-Computer Interface Using Machine Learning Methods

— Spring 2024 —

Project Collaborators:

Name	Student ID
Mobin Roohi	610300060
Abolfazl Khodayari	610399207
Ali Rajabzadeh	610398214

Abstract

This project focuses on the development of an EEG-based Brain-Computer Interface (BCI) using various machine learning methods. The goal is to perform classification of motor imagery tasks and clustering using electroencephalogram (EEG) data. More specifically, we will use two distinct datasets, which contain EEG recordings of subjects imagining movements of their left hand or feet, and left or right hand, respectively.

The project consists of multiple stages of signal and data pre-processing, analysis, and feature extraction before the data are given to the classification and clustering models. For signal pre-processing, we use methods such as band-pass filtering, common average reference (CAR) and more for purposes of noise reduction and data cleaning. A significant stage then is to use the common spatial patterns (CSP) algorithm to extract features usable for the models. Additionally other methods are analyzed and compared to the CSP as well.

Finally, the pre-processed and extracted features are used in classification and clustering models to detect motor imagery intention for BCI uses and also grouping together similar data as a means to further analyze and understand similar motor imagery actions.

Overall, this project demonstrates the application of machine learning methods in EEG signal processing for developing an effective BCI system. The comparative analysis of different pre-processing and feature extraction techniques provides insights into optimizing BCI performance.

Contents

Initial Report:

1 Motor Imagery and EEG Signals

Motor Imagery

Motor imagery (MI) is a cognitive process where an individual mentally simulates a movement without any actual physical execution. This mental practice involves imagining the sensations and experiences of performing a motor task, engaging similar neural processes as those used in actual movement. MI is a powerful tool in both cognitive neuroscience and applied fields such as sports psychology and rehabilitation. [1]

Motor imagery activates motor-related brain regions, including the primary motor cortex, supplementary motor area, and parietal cortex, which are also involved in planning and executing real movements. This overlap in neural activation suggests that MI shares common pathways with motor execution, making it a valuable method for enhancing motor learning and performance. [2]

Applications of Motor Imagery in General

1. Neurorehabilitation

Doctors and physiotherapists, often encourage their recovering patients to imagine movement as a way to help them recover their motor abilities.

2. Sports

Mental practice, when combined with physical practice, can be beneficial to beginners learning a sport, but even more helpful to professionals looking to enhance their skills. Physical practice generates the physical feedback necessary to improve, while mental practice creates a cognitive process physical practice cannot easily replicate. [3]

3. Medicine

When surgeons and other medical practitioners mentally rehearse procedures along with their physical practice, it produces the same results as physical rehearsal, but costs much less. But unlike its use in sports, to improve a skill, mental practice is used in medicine as a form of stress reduction before operations. [4]

Applications of Motor Imagery in Brain-Computer Interfaces (BCIs)

Motor imagery (MI) is extensively used in brain-computer interfaces (BCIs) to enable communication and control through brain activity. Here are the key applications:

1. Assistive Technology for People with Disabilities

- **Post-Stroke Recovery:** MI-BCIs are used in rehabilitation programs to promote motor recovery in stroke patients. By imagining movements, patients can activate motor-related brain areas, which can enhance neuroplasticity and aid in the recovery of motor functions.[5]

- **Control of Assistive Devices**: Amputees can use MI-BCIs to control prosthetic hands or limbs. By imagining the movement of their missing limb, they can generate control signals that drive the prosthetic device, enabling more natural and intuitive movements. Users may also control other devices such as wheelchairs or other assistive device. [6]
- **Communication Devices**: MI-based BCIs can help individuals with severe motor impairments (e.g., amyotrophic lateral sclerosis, spinal cord injuries) to communicate by translating their imagined movements into text or speech output.

2. Gaming and Virtual Reality

- **Immersive Experiences**: MI-BCIs can enhance gaming and virtual reality experiences by allowing users to interact with virtual environments through imagined movements, creating a more immersive and engaging experience.
- **Rehabilitation Games**: Combining MI-BCIs with serious games designed for rehabilitation can motivate patients to perform motor imagery tasks, making therapy more enjoyable and effective.

3. Research and Cognitive Neuroscience

- **Brain Function Studies**: Researchers use MI-BCIs to study brain functions related to motor control and cognitive processes. By analyzing the brain's response to motor imagery, scientists gain insights into neural mechanisms underlying movement and imagination.
- **Neurofeedback Training**: MI-BCIs are used in neurofeedback protocols to train individuals to modulate their brain activity. This can be beneficial for enhancing cognitive functions, reducing symptoms of neurological disorders, and improving mental health.

4. Adaptive and Intelligent Systems

- **Smart Environments**: MI-BCIs can be integrated into smart home systems, allowing users to control home appliances, lighting, and other devices through imagined movements, enhancing accessibility and convenience.
- **Human-Computer Interaction**: MI-BCIs facilitate more natural and intuitive human-computer interaction by enabling users to control computers and other digital devices through their thoughts, bypassing traditional input methods.

EEG signals

Electroencephalography (EEG), is a method used to record electrical activity of the brain. EEG signals represent the electrical impulses produced by neural activity in the brain. These signals are captured using electrodes placed on the scalp, which detect the tiny electrical charges that result from the activity of neurons within the brain.[7] Now let us see how this process works:

Electrode Placement

Electrodes are typically placed on the scalp according to a standardized system, such as the 10-20 system, which ensures consistent and reproducible positioning across different recording sessions and subjects.

Signal Acquisition

The electrical activity detected by the electrodes is amplified and recorded. This raw EEG data consists of continuous voltage fluctuations over time.

Signal Processing

The recorded signals are often filtered to remove noise and artifacts. Advanced techniques such as Fourier Transform or Wavelet Transform are used to analyze the frequency components of the signals.

Data Interpretation

The processed signals are interpreted to understand brain function or to diagnose conditions. For instance, specific patterns of brain waves might indicate different stages of sleep or the presence of epileptic activity. In particular, EEG is the gold standard diagnostic procedure to confirm epilepsy. [8]

Categorization Based on Frequency

The EEG signals are categorized into various waveforms or frequency bands based on their frequency. The table [9] below shows these frequency bands along with their relevance.

Band Name	Frequency in Hz	Cognitive Activity
Delta (δ)	$0.5 - 4$	Associated with sleep stages
Theta (θ)	$4 - 8$	Emotion recognition, Deep meditation and olfactory perception
Alpha (α)	$8 - 13$	Determination of eye-closed relax condition and Drowsiness detection
Mu (μ)	$8 - 13$	Motor planning, Motor imagery
Beta (β)	$13 - 30$	Motor activity
Gamma (γ)	$30 - 100$	Attention, memory, Visual perception, learning

In the case of EEG signals used in MI-based BCI applications, the mu and beta bands are particularly important.

MI-BCI Based on EEG

To create an MI-BCI, the BCI needs to be conveyed what the brain's intentions are, i.e. it should receive the MI. One popular way of doing this, is using EEG to convey the MI to the BCI. This is what we are going to do in this project.

2 Challenges Associated with MI

Challenges:

1. Individual Differences

- **Variability in MI Ability**: People vary significantly in their ability to perform motor imagery. Some individuals can vividly imagine movements, while others struggle, which can affect the effectiveness of MI-based interventions.
- **Assessment**: Reliable assessment of an individual's motor imagery ability is challenging. Self-report questionnaires, such as the Kinesthetic and Visual Imagery Questionnaire (KVIQ), may not always accurately reflect actual MI proficiency.

2. Neurophysiological Challenges

- **EEG Signal Quality**: In BCIs, the quality of EEG signals during MI can be affected by noise and artifacts from muscle movements, eye blinks, and other physiological activities.
- **Inter- and Intra-Subject Variability**: There is considerable variability in EEG patterns both between different individuals and within the same individual over time. This variability makes it difficult to develop robust algorithms for MI detection and classification.

3. Training and Learning

- **Learning Curve**: Users often require extensive training to improve their motor imagery skills and to generate consistent and distinguishable EEG patterns. This process can be time-consuming and may not yield uniform results across users.
- **Feedback Mechanisms**: Providing effective and immediate feedback is crucial for training. However, designing appropriate feedback mechanisms that enhance learning without causing frustration or fatigue is challenging.

4. Cognitive Load and Fatigue

- **Mental Fatigue**: Engaging in motor imagery tasks can be mentally exhausting, leading to decreased performance over time. Managing cognitive load and preventing fatigue are important for sustained MI practice.
- **Attention and Focus**: Successful MI requires sustained attention and focus. Distractions and lapses in concentration can significantly impact the quality of motor imagery and the corresponding neural signals.

5. Technological and Computational Challenges

- **Signal Processing**: Extracting meaningful features from EEG signals for MI is complex. Advanced signal processing techniques are required to filter noise and artifacts while preserving relevant information.

- **Algorithm Development**: Developing accurate and real-time algorithms for classifying MI-related EEG patterns is challenging due to the non-stationary nature of EEG signals and the need for computational efficiency.

6. Integration with Other Modalities

- **Multimodal Approaches**: Combining MI with other modalities such as motor execution, observation, or haptic feedback can enhance outcomes. However, integrating these modalities requires sophisticated experimental designs and data analysis techniques.

7. Clinical and Practical Applications

- **Personalization**: MI-based interventions need to be tailored to individual needs, capabilities, and goals. Creating personalized training protocols and rehabilitation programs can be resource-intensive.

- **Transfer to Real-World Tasks**: Ensuring that improvements gained through MI practice transfer to actual motor performance and real-world tasks is an ongoing challenge in both sports and rehabilitation contexts.

Solution to Challenges:

1. Individual Differences

- **Personalized Training Programs**: Develop tailored training programs that cater to individual abilities and progress. Use baseline assessments to customize the training intensity and content.

- **Objective Assessments**: Utilize objective measures such as neuroimaging or physiological markers in conjunction with self-report questionnaires to more accurately assess motor imagery ability.

2. Neurophysiological Challenges

- **Artifact Removal Techniques**: Implement advanced signal processing methods to remove noise and artifacts from EEG signals, such as Independent Component Analysis (ICA) and adaptive filtering.

- **Standardized Protocols**: Develop and adhere to standardized protocols for data collection to reduce variability. Include consistent electrode placement and environmental conditions.

3. Training and Learning

- **Incremental Training**: Use incremental training approaches that gradually increase task complexity and difficulty, allowing users to build their skills over time.

- **Enhanced Feedback**: Provide immediate and multimodal feedback (visual, auditory, haptic) to help users understand and improve their motor imagery performance.

4. Cognitive Load and Fatigue

- **Balanced Training Sessions**: Design training sessions with appropriate breaks and varied tasks to prevent mental fatigue and maintain user engagement.
- **Attention-Enhancing Techniques**: Incorporate techniques such as mindfulness and mental exercises to improve focus and reduce distractions during motor imagery practice.

5. Technological and Computational Challenges

- **Advanced Signal Processing**: Use machine learning and deep learning techniques to improve the accuracy of signal processing and classification of MI-related EEG patterns.
- **Real-Time Processing**: Develop efficient algorithms and use high-performance computing resources to enable real-time processing and feedback in MI applications.

6. Integration with Other Modalities

- **Multimodal Training**: Combine MI with physical practice, motor observation, or virtual reality environments to enhance the overall effectiveness of training programs.
- **Haptic Feedback**: Integrate haptic feedback devices to provide tactile sensations that can enhance the realism and effectiveness of motor imagery training.

7. Clinical and Practical Applications

- **Interdisciplinary Collaboration**: Foster collaboration between neuroscientists, psychologists, engineers, and clinicians to design comprehensive MI-based interventions.
- **Translational Research**: Conduct research that focuses on translating MI training improvements to real-world tasks, ensuring that gains in the lab transfer to practical applications.

---

3 Pre-processing of EEG Signals

Relevance

As discussed before, in MI-based BCI's, the MI is expressed through EEG signals. To have a robust BCI, the MI must be conveyed to the computer correctly, thus the EEG signals must accurately communicate MI. However, raw EEG signals are inherently noisy and may contain various types of artifacts. These artifacts include muscle movements, eye blinks, and environmental interferences.
These noise and artifacts limit the accuracy of the EEG data conveying MI. Consequently, when pre-processing EEG signals, in addition to the regular pre-processing such as missing data replacement and normalization, it is necessary to carry out pre-processing methods to reduce noise, eliminate artifacts, and even- tually enhance the SNR[1] of the EEG data. [10] [9]

[1] SNR stands for Signal-to-Noise Ratio. SNR refers to the dimensionless ratio between signal and noise power.

MI-BCI systems mainly rely on a temporal and spatial filtering approach to achieve this pre-processing, which we will discuss next.

Temporal Filtering

Temporal filtering is the most common of the pre-processing approaches used for EEG signals. Temporal filters are usually either low-pass or band-pass filters that are used to extract signals that are in the the frequency band where the relevant neurophysiological information lies. In the context of MI application, as discussed previously, the waves forms most relevant are the mu and beta wave forms. Thus using filtering such as a Butterworth or Chebyshev bandpass filters of 8–30 Hz frequency, mu and beta frequency bands are extracted. [11] [12]

Spatial Filtering

Spatial filtering refers to the process of enhancing the signal-to-noise ratio and extracting relevant information from EEG recordings by manipulating the spatial distribution of electrode signals.

3.1 Common Average Reference (CAR)

In the common average reference (CAR) method, the average value across all electrodes is subtracted from the value of the specific channel being analyzed. CAR approach offers EEG recordings that are nearly reference-free by placing importance on components present in a significant portion of the electrode array. By reducing these common components, the CAR effectively acts as a high-pass spatial filter that reduces noise quite a bit.
The CAR is computed according to the formula,

$$V_i^{CAR} = V_i^{ER} - \frac{1}{n}\sum_{j=1}^{n} V_j^{ER}$$

where V_i^{ER} is the potential between the ith electrode and the reference and n is the number of electrodes in the montage [13].
CAR is computationally simple, and therefore is robust and efficient to both on-chip and real-time applications. CAR is commonly used in EEG pre-processing, where it is used to identify small signal sources in very noisy recordings and increases SNR [9][14].

3.2 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a technique used in data analysis, machine learning, and artificial intelligence to reduce the dimensionality of datasets while retaining important information. Steps of PCA are as follows:

1. Standardizing

We ensure that all initial variables have the same scale $(\mu = 0, \sigma = 1)$.

2. Computing the Covariance Matrix

We calculate the covariance between each pair of initial variables.

3. Computing the Eigenvalues and Eigenvectors of the Covariance Matrix

Notice that eigenvectors represent the directions of maximum variance (principal directions) and Eigenvalues indicate the amount of variance explained by each eigenvector.

4. Creating a Feature Vector to Decide Which Principal Components to Keep

At the first, We should sort the eigenvectors by their corresponding eigenvalues (in descending order). Then we select the top k eigenvectors (where k is the desired reduced dimensionality).

5. Recast the Data Along the Principal Components Axes

Now we have disired axes. At the first we project the original data onto the selected eigenvectors. These new axes (principal components) are uncorrelated and capture most of the information [15][16].

In summary use cases of PCA are as follows:

1. Dimensionality Reduction

- **Data Compression**: PCA reduces the number of variables in large datasets while retaining most of the variance. This helps in compressing data, which is useful for storage and transmission.
- **Visualization**: By projecting high-dimensional data onto a lower-dimensional space (typically 2D or 3D), PCA helps in visualizing the structure and patterns in the data, making it easier to understand and interpret.

2. Noise Reduction

- **Signal Processing**: In applications like image and speech processing, PCA can be used to reduce noise by retaining only the principal components that capture the most significant features of the data, filtering out the noise components.

3. Feature Extraction and Engineering

- **Preprocessing for Machine Learning**: PCA is often used to transform the features of a dataset before feeding it into machine learning models, helping to improve performance and reduce computational costs by eliminating redundant and irrelevant features.
- **Pattern Recognition**: In fields like handwriting recognition, facial recognition, and bioinformatics, PCA helps in extracting the most informative features from the raw data, aiding in more accurate classification and pattern detection.

4. Financial Modeling

- **Risk Management**: PCA is used in finance to identify and model the underlying factors that drive the movements of financial markets. By analyzing the principal components, investors and analysts can better understand and manage risks.
- **Portfolio Optimization**: PCA helps in reducing the complexity of the covariance matrix in portfolio optimization, making it easier to identify the key drivers of portfolio returns and construct diversified portfolios.

5. Bioinformatics

- **Gene Expression Analysis**: PCA is used to analyze gene expression data, reducing the dimensionality of the data to identify patterns and clusters of genes that exhibit similar expression profiles, which can be critical for understanding biological processes and diseases.

6. Image Processing

- **Face Recognition**: PCA, often referred to as Eigenfaces in this context, is used to reduce the dimensionality of facial image data and to extract the most important features for face recognition algorithms.
- **Image Compression**: PCA can be used to compress images by reducing the number of pixels needed to represent the image while maintaining the essential features, leading to efficient storage and transmission.

7. Environmental Science

- **Climate Data Analysis**: PCA is employed to analyze climate data by identifying patterns and trends in temperature, precipitation, and other climate variables, helping researchers understand and predict climate change.

8. Marketing and Customer Insights

- **Market Segmentation**: PCA helps in reducing the dimensionality of customer data, making it easier to identify distinct market segments based on purchasing behavior, preferences, and demographics.
- **Sentiment Analysis**: PCA is used to analyze text data from customer reviews and feedback, extracting the key components that reflect customer sentiments and opinions [15][16].

3.3 Independent Component Analysis (ICA)

Independent Component Analysis (ICA) is a computational technique used in machine learning and artificial intelligence to separate a multivariate signal into additive, independent components. It is particularly useful when the observed signals are assumed to be generated by a mixture of independent sources. There are two main assumptions in ICA:

1. Independent sources

ICA assumes that the observed signals are generated by a linear combination of independent source signals. The goal is to recover these source signals from the observed mixture, even though the mixing process itself may not be known.

2. Statistical Independence

Unlike other dimensionality reduction techniques such as Principal Component Analysis (PCA), which seeks uncorrelated components, ICA aims to find statistically independent components. Independence implies that the components are as unrelated as possible, capturing different underlying phenomena.

Underlying model

Mathematically, ICA can be represented as $X = AS$, where X is the observed mixture of signals, A is the mixing matrix, ans S is the matrix of independent source signals. The goal of ICA is to estimate the unmixing matrix W such that $S = WX$, where $W^\circ = A^{-1}$.

The steps of ICA are as follows:

1. Data Preprocessing

- **Normalization**: Scale the data to have zero mean and unit variance to ensure uniform influence of all features.
- **Centering**: Subtract the mean from each feature to center the data around zero, focusing the analysis on variations.

2. Whitening

- **Covariance Matrix Calculation**: Compute the covariance matrix of the centered data to understand its spread.
- **Eigenvalue Decomposition**: Decompose the covariance matrix to obtain eigenvalues and eigenvectors.
- **Whitening Transformation**: Use these eigenvalues and eigenvectors to decorrelate the data, ensuring that the covariance matrix of the transformed data is an identity matrix. This step helps to simplify the separation process.

3. Independent Component Estimation

- **Initial Random Weight Initialization**: Start with a randomly initialized unmixing matrix.
- **Iterative Optimization**: Apply an optimization algorithm to iteratively adjust the unmixing matrix. The goal is to maximize the statistical independence of the estimated components.

- **Nonlinearity Application**: Use a nonlinear function (e.g., tanh) to enhance non-Gaussianity, which is crucial for achieving independence.
- **Update Unmixing Matrix**: Continuously update the unmixing matrix based on the derivatives of the contrast function until a convergence criterion (like a minimal change in weights) is met.

4. Reconstruction

- **Recover Independent Components**: Multiply the final unmixing matrix by the preprocessed data to obtain the independent components, which represent the original source signals.

5. Post-processing (Optional)

- **Component Selection**: Choose the most relevant independent components based on their significance or specific criteria relevant to the analysis.
- **Scaling**: Adjust the scale of the components if necessary for interpretability or comparison purposes.

6. Interpretation and Analysis

- **Interpretation**: Analyze the independent components to understand the underlying sources or patterns in the data. For example, in a mixture of audio recordings, these components could represent distinct speakers.
- **Validation**: Validate the results through visualization, statistical tests, or domain-specific knowledge to ensure the components make sense and are useful.

These steps encapsulate the process of ICA, from initial data preparation to the extraction and interpretation of independent components, providing a clear path to uncovering hidden structures within multivariate data [17].

3.4 Minimum Norm Estimation (MNE)

Minimum Norm Estimation is a pre-processing technique used to estimate the underlying neural activity that is the source of the electrical readings seen in the EEG data. One of its uses is in source localization, which aims to determine the locations in brain that are the source of the electrical readings. The task of finding the neural source of the EEG readings is also called the *bioelectromagnetic inverse problem* and MNE is one possible approach to solving it.
The relationship between a given electrical source distribution in the brain and the electric potential or magnetic field measured at discrete points on or above the scalp surface (EEG signals) is linear. [18] [19] This relationship can be described by the following:

$$\mathbf{d} = \mathbf{L}\mathbf{s}$$

where, $\mathbf{d}$, is the vector of recorded data (e.g., EEG or MEG signals) at electrode or sensor locations; $\mathbf{L}$ is the "leadfield" matrix representing the relationship between the neural sources and the recorded data, and $\mathbf{s}$, is the vector of unknown source distribution representing neural activity in the brain.

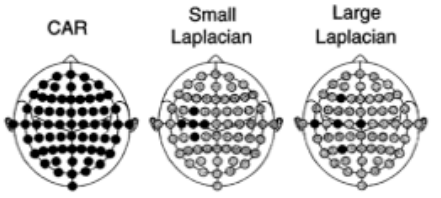
The aim of the inverse problem is to find the vector $\mathbf{s}$. However, this problem is inherently underdetermined because there are typically more unknowns (elements of $\mathbf{s}$) than equations (data points in $\mathbf{d}$). In other words, the number of neural sources are much more than the scalp electrodes reading the electrical pulses. This means there are multiple possible solutions that can explain the observed data and there is no unique solution.
One possible approach to solving the inverse problem and a pre-processing technique used on EEG data analysis is the MNE. This approach seeks the solution $\mathbf{s}$ that minimizes the norm (L_2 norm for instance) of $\mathbf{s}$, subject to the constraint that the solution explains the observed data. It assumes a minimal distribution of neural activity, looking for solutions with minimal overall power. One formulation of the MNE result is the MNP[2] or pseudoinverse matrix,

$$\hat{\mathbf{s}} = \mathbf{L}^T \left(\mathbf{L}\mathbf{L}^T\right)^{-1} \mathbf{d},$$

which minimizes this norm.
MNE is used is source localization, estimating the spatial distribution of neural sources in the brain based on the recorded EEG data. This allows researchers to identify the specific brain regions involved in the functions of the brain. Moreover, the localization that is obtained, may be used as features for a machine learning algorithm.
Mainly in our application, the MNE is a useful spatial filter that can project the EEG data onto the relevant brain regions or apply spatial filters to enhance the signal-to-noise ratio. MNE can also account for individual differences in brain anatomy and function, making it particularly useful for personalized MI-based BCI systems. [20]

3.5 Laplacian Filter

The Laplacian method calculates the second derivative of the spatial voltage distribution at each electrode location, making sure that activity originating from radial sources directly beneath the electrode. As a result, it is used as a high-pass spatial filter, increasing localized activity while reducing more diffuse activity. High spatial resolution is achieved by employing many electrodes distributed across the entire scalp. The Laplacian value at each electrode is found by combining its value with that of surrounding electrodes. The distances to these surrounding electrodes determine the spatial filtering characteristics of the Laplacian: shorter distances make it more sensitive to higher spatial frequencies and less sensitive to lower spatial frequencies.



To compute the Laplacian derivations, a finite difference method is used, which approximates the second derivative by subtracting the mean activity at surrounding electrodes from the channel of interest. The Laplacian is calculated using the following formula:

$$V_i^{\mathrm{LAP}} = V_i^{\mathrm{ER}} - \sum_{j \in S_i} g_{ij} V_j^{\mathrm{ER}}$$

[2]"minimumn-norm pseudoinvers" or "Moore – Penrose Inverse" matrix

where $g_{ij} = 1/d_{ij} / \sum_{j \in Si} 1/d_{ij}$.
and Si is the set of electrodes surrounding the i th electrode, and d_{ij} is the distance between electrodes i and j (where j is a member of Si). Just like many of the other spatial filters, this methods is also very robust for noise and artifact reduction. It is also quite simple to implement. [13].

---

4 Feature Extraction

4.1 Overview

Measuring motor imagery through an EEG leads to a large amount of data due to high sampling rate and electrodes. Even after pre-processing, this data would still be large and contain repetitive and non-discriminating data. To achieve the best possible performance it is necessary to work with small values that are capable of discriminating MI task activity from unintentional neural activity. These small values are the features and the task of feature extraction is the process of obtaining these features. More formally, feature extraction, transforms the pre-processed data into the features space that will be fed to the classifier. This feature space should contain all of the necessary discrimination information for a classifier to do its job. [12]
In this section, we provide brief information for a variety of feature extraction methods for MI-based BCI and then discuss CSP[3] as our main feature extraction method.
For MI-BCI, the feature extraction methods can be divided into six categories:

1. Time-based methods that use timing info in EEG signals.
2. Spectral methods that focus on frequency info in EEG signals.
3. Time-frequency methods that combine timing and frequency info.
4. Spatial methods analyze that signals from different electrodes.
5. Spatio-temporal methods that combine spatial and timing info.
6. Spatio-spectral methods that use both spatial and frequency info. [12]

Now we one popular approach, the CSP:

4.2 Common Spatial Pattern (CSP)

Introduction

The Common Spatial Pattern (CSP) method is widely recognized as one of the most popular techniques for identifying spatial patterns in multiclass data. It utilizes covariance and correlation of the data and reduces multi-channel data into a low-dimensional subspace through linear transformation. Compared to Principal Component Analysis (PCA), CSP is particularly effective for dimension reduction and relies on Eigen decomposition to extract features. The spatial filter produced by CSP is designed to finely separate different classes of EEG signals based on their band power features, thus increasing classification accuracy.

[3] Common Spatial Pattern

Table 1: List of some of the fearture extraction methods.[9]

Approaches	Advantages	Drawbacks	Domain
Autoregressive Model (AR)	Provides precise spectral estimations and improves frequency resolution for short segments.	The model's dependability depends on choosing the correct AR coefficients, and its slow performance makes it less suitable for real-time analysis.	Frequency domain
Fast Fourier Transform (FFT)	A good approach for analyzing stationary signals, good for narrowband signals such as sine waves.	Lacks the capability to effectively analyze nonstationary EEG signals and is too sensitive to noise.	Frequency domain
Common Spatial Pattern (CSP)	Can be used for analyzing multichannel data and obtains suitable outcomes for chaotic signals.	Achieving good results often requires a large number of electrodes, and the assumption of Gaussian distribution for each class may not be true for EEG data in some cases.	Spatial Filters
Short-time Fourier Transform (STFT)	Provides frequency information for signals at each time point.	Its fixed temporal resolution makes it less effective for nonstationary signals.	Time and frequency domain
Wavelet Transform (WT)	More appropriate for analyzing transient and sudden changes in signals, especially those that are non-stationary.	An appropriate mother wavelet must be chosen.	Time and frequency domain
Empirical Mode Decomposition (EMD)	Ideal for processing nonstationary and nonlinear signals.	EMD is prone to mode mixing and is sensitive to noise, which poses a significant challenge.	Time and frequency domain

CSP attempts to highlight and minimize the variance within multiclass data, increasing the variance of one class while reducing the variance of others. This process allows the features of different classes to have discriminative properties, leading to improved classification accuracy.
In EEG signals with high temporal sampling rates, there can be a wide range of similarities between signals, possibly leading to overfitting. To address this, CSP focuses on increasing class-to-class discrimination while considering the short time domain properties usually observed in EEG signals (usually within milliseconds).
Given the importance of feature extraction in EEG signal classification, CSP is a widely utilized approach due to its effectiveness in feature extration [21].

Algorithm

Step 1:
Using the results taken from two class distributions, CSP aims to find the spatial filters from a linear combination of a multichannel signal which maximizes variance for one class and minimizes variance for other one. To this end, spatial patterns are obtained with the simultaneous diagonalization of the covariance matrices of the EEG signals from data of each class which can be solved by maximizing the following criterion which is called the Raleigh criterion:

$$R(\mathbf{w}) = \frac{\mathbf{W^T \Sigma_1 W}}{\mathbf{W^T \{\Sigma_1 + \Sigma_2\} W}} \tag{1}$$

where Σ_1 and Σ_2 are normalized average covariance matrices of class 1 and class 2 respectively.

Step 2:
Equation (1) can be solved by reformulating it into the following constrained optimization problem:

$$\max_{\mathbf{w}} \mathbf{W}^{\mathrm{T}} \mathbf{\Sigma_1} \mathbf{W}$$

such that

$$\mathbf{W}^{\mathrm{T}} \{\mathbf{\Sigma}_1 + \mathbf{\Sigma_2}\} \mathbf{W} - \mathbf{I} = 0$$

Step 3:
For a given EEG trial X, the transformed matrix Z is given as:

$$\mathbf{Z} = \mathbf{WX}$$

Step 4:
Feature vector f_p are calculated using:

$$\mathbf{f_p} = \log\left(\frac{\mathrm{var}\left(\mathbf{Z_p}\right)}{\sum_{p=1}^{2r} \mathrm{var}\left(\mathbf{Z_p}\right)}\right)$$

where $\mathbf{Z}_p$ are the first and last r rows of transformed matrix $\mathbf{Z}$. It is shown that $r = 1$ or $r = 2$ is a good choice and including more spatial patterns does not lead to more improvement in accuracy. In this work, we have used $r = 1$. Hence, we obtain two features for each trial.

As a data-driven technique, CSP technique is one of highly successful spatial technique which helps in estimating spatial filters to analyze multichannel data in MI-BCI. The spatial filters obtained are applied to relevant frequency bands data to obtain features. If CSP spatial filter are applied to an EEG signal filtered using an irrelevant frequency band, the extracted features will give poor performance. Hence, frequency band selection is an important issue to be handled. Exhaustive search and manual tweaking can help in learning the best bands, but being time consuming, are not suggested. The major problem in CSP is tuning of BCI device for every subject as the rythmic patterns of selected frequency band rhythms varies from subject to subject. The existenceof artifacts and the non-stationary nature of an EEG trial data may further deteriorate the performance of CSP. More recently research has been focused on finding spatial patterns from a filter bank of non-overlapping fixed sized subbands [22].

For other variations of this method, building on it, we can mention to the following methods.

Subband CSP (SBCSP) has been used to analyze motor imagery data using different fixed sized subbands[23].

Filter bank CSP (FBCSP) allows analysis of CSP technique by applying it on different frequency bands filtered EEG based on maximal mutual information criterion[24].

4.3 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an algorithm that quickly computes the Discrete Fourier Transform (DFT) of a sequence, transforming a time-domain signal into its frequency components. The FFT reduces the computational complexity of the DFT from $O\left(n^2\right)$ to $O(n \log n)$.
The DFT of a sequence x_j is given by

$$X_k = \sum_{j=0}^{n-1} x_j \cdot e^{-i2\pi kj/n}$$

for $k = 0, 1, \ldots, n-1$, where X_k are the frequency components and i is the imaginary unit.
The FFT when applied to time series data such as EEG data can produce features that are fit for machine learning tasks. These features are in the frequency domain instead of the time domain and contains specific frequency band powers.

4.4 Wavelet Transform

The Fourier Transform provides frequency information of a signal that represents frequencies and their magnitude. It does not tell us when in time the frequencies exist. The transform is therefore ideal for stationary signals. Lacks capability to provide frequency information for a localized signal region in time. This is where we can use a wavelet or continuous wavelet transform to fix this issue.
The Continuous Wavelet Transform results in analyzing a signal into different frequencies at different resolutions, known as multiresolution analysis.

$$F(\tau, s) = \frac{1}{\sqrt{|s|}} \int_{-\infty}^{+\infty} f(t) \psi^* \left(\frac{t-\tau}{s} \right) dt$$

where τ is the translation parameter, which shifts the wavelet function in time, s is the scale parameter, $f(t)$ is the original signal or function being transformed, $\psi^*(\cdot)$ is the complex conjugate of the mother wavelet function $\psi(t)$ indicating which bases is used.

Implementation Report:

NOTE: We will implement the project using datasets A and C. However, dataset C will not be analyzed as thoroughly as dataset A, since they will share a lot of traits. Thus, we will only check the most important aspects of dataset C and leave more A lot of the similar and repetitive results obtained from data set C are omitted not to make the report longer than it already is! Most of the earlier plots and results for dataset C do not add anything of value and only the important results and plots for dataset C has been included in this report.

5 Data loading

5.1 Dateset

The datasets are provided by the Berlin BCI group, which includes the Berlin Institute of Technology (Machine Learning Laboratory), Fraunhofer FIRST (Intelligent Data Analysis Group), and the Neurophysics Group at Campus Benjamin Franklin of the Charité - University Medicine Berlin.

5.2 Data format

Data are provided in Matlab format (*.mat) containing variables:

- mrk: Structure containing target marker information with the following fields (note: the evaluation data file does not include this variable):
- pos: Vector of marker positions in the given EEG signals in sample units.
- y: Vector of target classes (-1 for class one or 1 for class two).
- nfo: Structure providing additional information with the following fields:
- fs: Sampling rate.
- clab: Cell array of channel labels.
- classes: Cell array of motor imagery class names.
- xpos: X positions of electrodes in a 2D projection.
- ypos: Y positions of electrodes in a 2D projection.

5.3 Import data

In the initial phase of the project, we will utilize the "BCICIV_calib_ds1a" dataset. Subsequently, we will employ the "BCICIV_calib_ds1c" dataset to perform all signal processing steps, followed by classification and clustering. Finally, we will compare the results of both datasets.
Load the BCICIV_calib_ds1a.mat file containing the EEG signals:

```
mat = scipy.io.loadmat("/kaggle/input/bciciv-calib-ds1a/BCICIV_calib_ds1a.mat")
print("Variables include: ", mat.keys())
```

Variables include dict_ keys: (['__header__', '__version__', '__globals__', 'mrk', 'cnt', 'nfo'])

Then we extracted variables and additional information:

```
        cnt = mat['cnt'].astype(np.float64) * 0.1  # Converted to microvolts
        pos = mat['mrk']['pos'][0][0].flatten()
        y = mat['mrk']['y'][0][0].flatten()
        nfo = mat['nfo']
        fs = nfo['fs'][0][0][0][0]
        clab = [str(c[0]) for c in nfo['clab'][0][0][0]]
        classes = [str(c[0]) for c in nfo['classes'][0][0][0]]
        xpos = nfo['xpos'][0][0][0]
        ypos = nfo['ypos'][0][0][0]
```

In the process, we log the number of time samples and channels, and list the classes:

```
        print(f"Number of time samples: {cnt.shape[0]}, Number of channels: {cnt.shape[1]}")
        print(f"The list of classes is {classes}")
```

Specifically, the number of time samples is 190,594, and the number of channels is 59. The list of classes includes 'left' and 'foot', representing the motor imagery of moving the left hand or the feet.

5.4 Explanatory Data Analysis

In our project, we defined a function to visualize EEG data using the MNE library. This function is utilized throughout the rest of the code to generate visual representations of the EEG signals.

```
def visualize_EEG(cnt, title, clab, fs, channel_num=20):
        '''Input EEG data and visualize it using mne library'''
        # Create info metavariable
        info = mne.create_info(ch_names=clab, sfreq=fs, ch_types='eeg')
        # Create RawArray object
        raw = mne.io.RawArray(cnt.T, info)
        # Plot raw EEG data
        raw.plot(n_channels=channel_num, scalings='auto', title=title, bgcolor='white',
        color='darkblue')
```

The visualize_ EEG function accepts several parameters: cnt (EEG data), title (title for the plot), clab (channel labels), fs (sampling frequency), and channel_ num (number of channels to display, default is 20).

The mne.create_ info function is used to create an info object that contains metadata about the EEG data. Then, the mne.io.RawArray function creates a RawArray object from the EEG data (cnt.T) and the info object. This RawArray object is essential for handling and visualizing EEG data in MNE. Finally, the raw.plot function is used to plot the raw EEG data. It takes several parameters, including the number of channels to display (n_ channels), scaling options (scalings), plot title (title), background color (bgcolor), and line color (color).

By using this function, we can easily visualize the EEG data, which helps in analyzing and interpreting the signals.

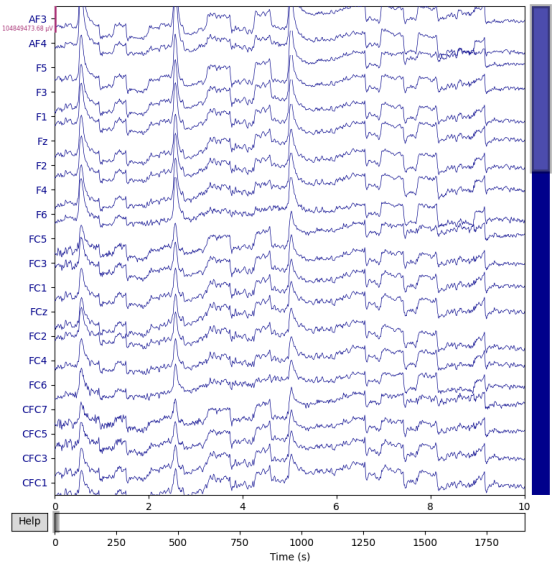


Figure 1: Raw EEG data (Dataset A)

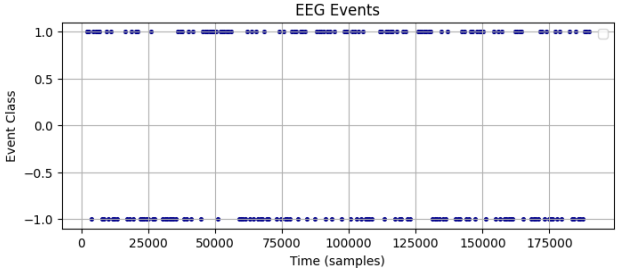


Figure 2: Events (Dataset A)

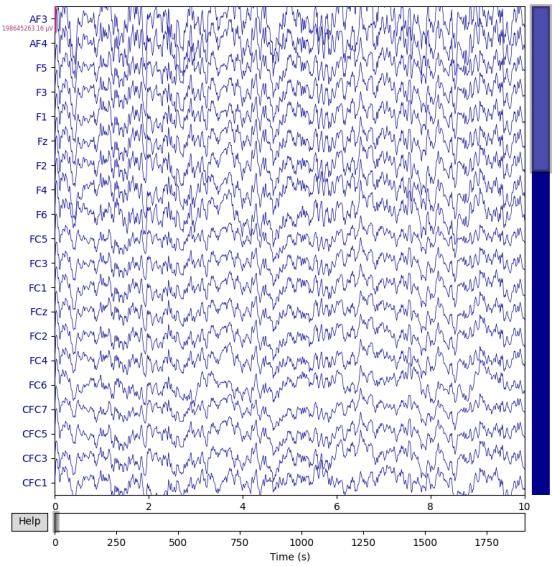


Figure 3: Raw EEG data (Dataset C)

6 Data preprocessing

In the data preprocessing phase of our project, we meticulously prepared the EEG data to ensure its quality and suitability for subsequent analysis. This section outlines the comprehensive steps we undertook to preprocess the data, enhancing its clarity and reliability. Note that, all the methods used here were clearly explained in the Initial Report of the project, so in this section, we focus on explaining the implementation. Our preprocessing pipeline included the following key tasks:

- **Bandpass Filtering**: We applied bandpass filters to isolate specific frequency bands crucial for our analysis, such as the beta and mu bands. This step involved using a Butterworth filter of order 3 within the frequency range of 8 to 30 Hz.

- **Spatial Filtering**:

  - To further refine the data, we employed the Common Average Reference (CAR) technique for spatial filtering. CAR is particularly effective for EEG preprocessing because it reduces the influence of noise and artifacts by averaging the signals from all electrodes and subtracting this average from each electrode's signal. This method enhances the signal-to-noise ratio and minimizes the impact of localized noise sources, making it a preferred choice over Laplacian filters for many EEG applications.
  - We explored additional preprocessing methods, including Principal Component Analysis (PCA) and Independent Component Analysis (ICA), to further clean the data and remove artifacts.

- **Data Visualization**: Visualization played a crucial role in our preprocessing workflow. We visualized the EEG data before and after applying spatial filters, using plots to compare raw and filtered signals. For high-dimensional data, we utilized t-SNE to gain a better understanding of the data structure.

By following these steps, we ensured that our EEG data was thoroughly preprocessed, setting a solid foundation for the subsequent signal processing, classification, and clustering tasks in our project. This rigorous preprocessing phase was essential for achieving accurate and meaningful results in our analysis.

6.1 Band-Pass Filtering

In the Band-Pass Filtering step, we utilized the signal.scipy module to isolate specific frequency bands crucial for our analysis. Below is the code we implemented:

```
def bandpass_filter(cnt, lo, hi, fs, order=3):
        tmp = 0.5 * fs
        low = lo / tmp
        high = hi / tmp
        b, a = scipy.signal.butter(order, [low, high], btype='band')
        result = scipy.signal.lfilter(b, a, cnt, axis=0)
        return result
```

The bandpass_ filter function is designed to take several parameters: cnt (EEG data), lo (low cutoff frequency), hi (high cutoff frequency), fs (sampling frequency), and order (order of the filter, default is 3). To normalize the cutoff frequencies, we set the variable tmp to half the sampling frequency (0.5 * fs). The low and high cutoff frequencies are then normalized by dividing them by tmp. We use the scipy.signal.butter function to create a Butterworth band-pass filter with the specified order and normalized cutoff frequencies, which returns the filter coefficients b and a. The scipy.signal.lfilter function is then applied to the EEG data (cnt) along the specified axis (axis=0), and the filtered data is stored in the variable result.

```
        bandpass_cnt = bandpass_filter(cnt, 8, 30, fs)
```

Finally, the bandpass_ filter function is called with the parameters cnt, 8, 30, and fs to apply a band-pass filter in the frequency range of 8 to 30 Hz, with the filtered data stored in bandpass_ cnt. Here is the visualized bandpass filtered EEG data:

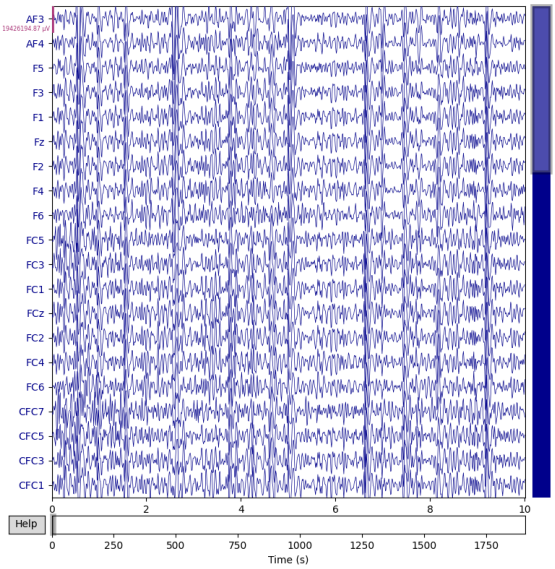


Figure 4: Bandpass filtered EEG data (Dataset A)

6.2 Spatial Filtering

6.2.1 Common Average Reference (CAR)

Next, we apply the Common Average Reference (CAR) technique as a spatial filter to reduce noise and enhance the quality of the dataset signals. Below is the code that implements the method explained in the initial project report.

```
def common_average_reference(cnt):
        '''Calculate and return the CAR of given EEG data'''
        return cnt - np.mean(cnt, axis=1, keepdims=True)
```

Here is the result:

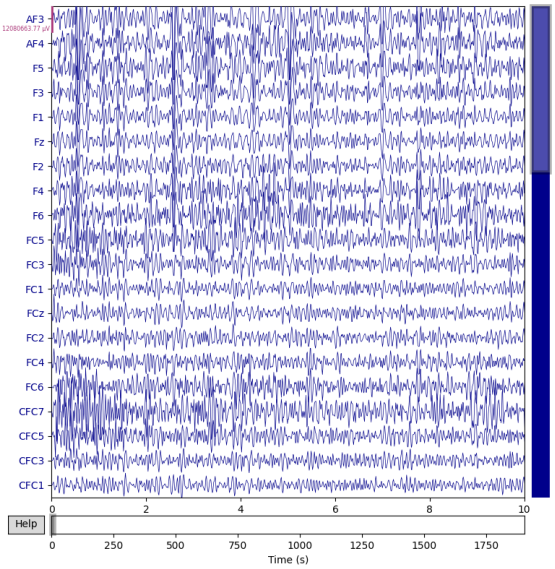


Figure 5: CAR Applied EEG (Dataset A)

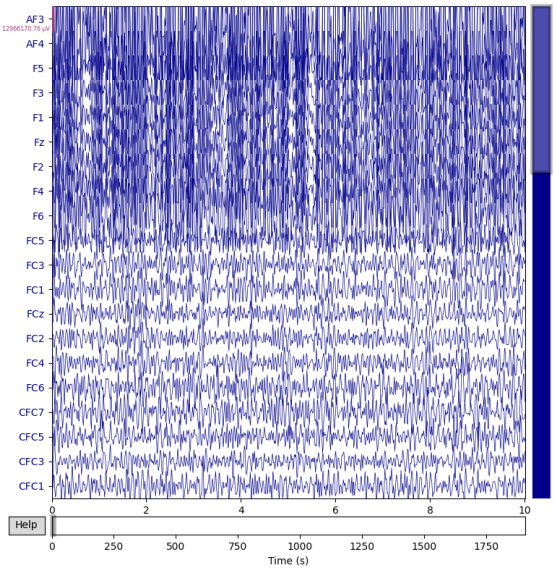


Figure 6: CAR Applied EEG (Dataset C)

To better understand the impact of these preprocessing methods on the data, we will now examine a segment of a single channel. This approach allows us to closely inspect the effects of the filters. We have three lists to compare: the raw signal, the band-pass filtered signal, and the CAR filtered signal. Specifically, we are visualizing the first 500 samples of a single channel for each type of signal.

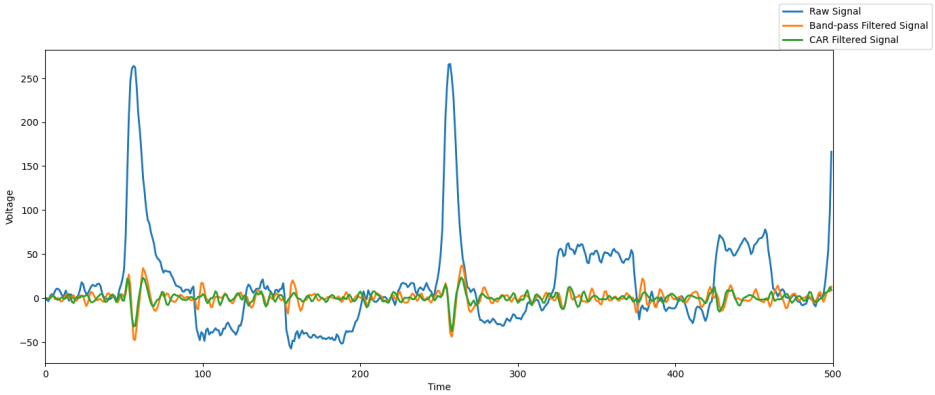


Figure 7: Different Stages (Dataset A)

To further enhance our visualization, we can use t-SNE to represent the data in higher dimensions (more channels). Please note that the following code block may take a bit longer to execute.

```
#Apply t-SNE to 2D
tsne = TSNE(n_components=2)
tsne_cnt1 = tsne.fit_transform(CAR_cnt[:500, :])
tsne_cnt2 = tsne.fit_transform(bandpass_cnt[:500, :])
tsne_cnt3 = tsne.fit_transform(cnt[:500, :])
```

Here is the results:

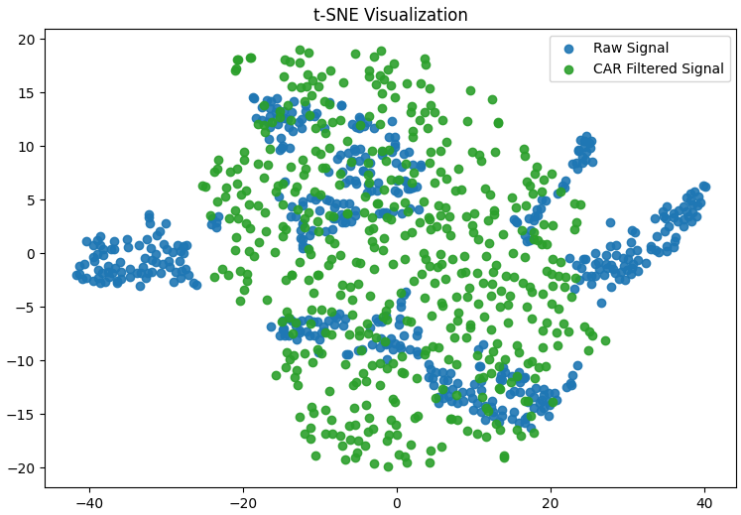


Figure 8: t-SNE Visualization (Dataset A)

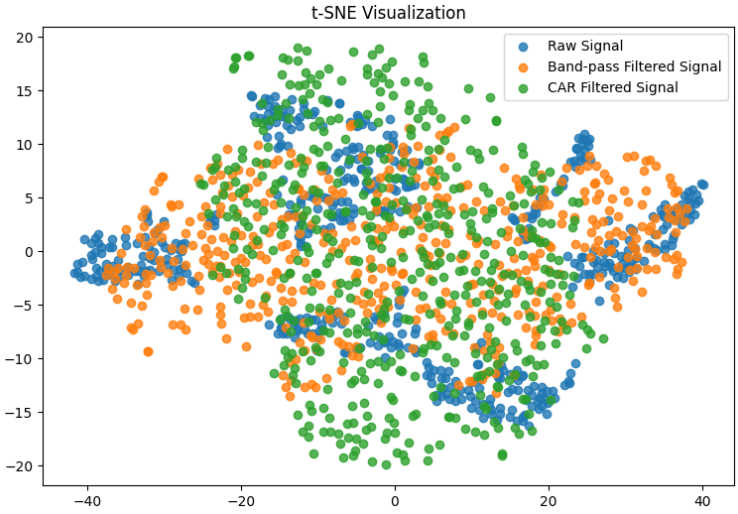


Figure 9: t-SNE Visualization (Dataset A)

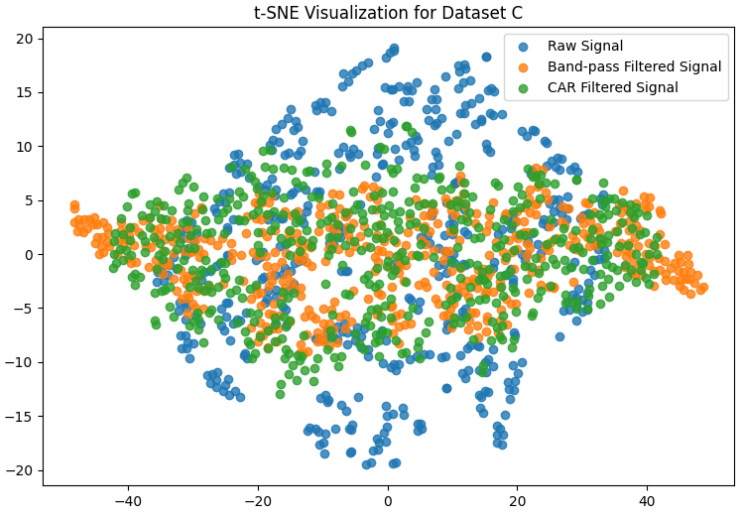


Figure 10: t-SNE Visualization (Dataset C)

6.2.2 Principal Component Analysis (PCA)

To further clean and denoise the data, we implemented and analyzed the Principal Component Analysis (PCA) dimensionality reduction method. Below is the code and an explanation of how it works:

```
def principal_component_analysis(data, n_components='mle', verbose=False):
        '''Perform PCA to reduce noise and reduce dimensionality.'''

        # Initialize PCA
        pca = PCA(n_components=n_components)

```

```
        # Reshape the data for applying PCA
        n_epochs, n_times, n_channels = data.shape
        data = data.reshape(-1, n_channels)

        # Apply PCA
        principal_components = pca.fit_transform(data)

        # Reshape it back
        principal_components = principal_components.reshape(n_epochs, n_times, -1)

        # Variance info
        if verbose:
        print(principal_components.shape)
        print(pca.explained_variance_ratio_)

        return principal_components
```

The principał_ component_ analysis function is designed to perform PCA for noise reduction and dimensionality reduction. It takes in the parameters data (EEG data), n_ components (number of principal components to keep, default is 'mle' which lets the algorithm decide), and verbose (a flag to print additional information). The PCA object is initialized with the specified number of components using PCA(n_ components = n_ components). The EEG data is reshaped to a 2D array where the number of epochs and time samples are combined into one dimension, and the number of channels remains the same. This reshaping is necessary for applying PCA, which operates on 2D data. PCA is applied to the reshaped data using pca.fit_ transform(data), which reduces the dimensionality and noise of the data. The resulting principal components are stored in principal_ components. The principal components are then reshaped back to the original 3D shape (epochs, time samples, channels) using principal_ components.reshape(n_ epochs, n_ times, -1). If the verbose flag is set to True, the function prints the shape of the principal components and the explained variance ratio, which indicates how much variance is captured by each principal component.

6.2.3 Independent Component Analysis (ICA)

This method is particularly effective for identifying and removing artifacts from EEG data, such as eye blinks, muscle activity, and other noise sources1. By transforming the data into a set of statistically independent components, ICA enhances the quality of the EEG signals, making them more suitable for further analysis. Below is the code we implemented for ICA and an explanation of how it works:

```
def independent_component_analysis(data, n_components=50):
        '''Perform ICA for more noise reduction'''

        # Initialize ICA
        ica = FastICA(n_components=n_components, random_state=42)

        # Reshape the data for applying ICA
        n_epochs, n_times, n_channels = data.shape
        data = data.reshape(-1, n_channels)

        # Apply PCA
        independent_components = ica.fit_transform(data)

        # Reshape it back
```

```
        independent_components = independent_components.reshape(n_epochs, n_times, -1)

        return independent_components
```

The independent_component_analysis function is designed to perform ICA for noise reduction. It takes in the parameters data (EEG data) and n_components (number of independent components to keep, default is 50). The ICA object is initialized with the specified number of components using FastICA(n_components=n_components, random_state=42), with the random_state parameter ensuring reproducibility of the results. The EEG data is reshaped to a 2D array where the number of epochs and time samples are combined into one dimension, and the number of channels remains the same. This reshaping is necessary for applying ICA, which operates on 2D data. ICA is then applied to the reshaped data using ica.fit_transform(data), which separates the mixed signals into independent components. The resulting independent components are stored in independent_components. Finally, the independent components are reshaped back to the original 3D shape (epochs, time samples, channels) using independent_components.reshape(n_epochs, n_times, -1). By implementing ICA, we can effectively reduce noise and artifacts in the EEG data, enhancing the signal quality.

6.3 Segmenting the Dataset

To segment the dataset into different epochs around the cue positions provided, we will divide the data into segments or epochs around these cue positions for classification purposes.

```
def segment_data(cnt, y, a=0, b=1000):
        # Segment start and end parameters in milliseconds.
        segment_start = a
        segment_end = b

        # Create the segments.
        segments = []
        segment_targets = []

        # Go through every cue position.
        for i, cue_pos in enumerate(pos):
        left = cue_pos + segment_start
        right = cue_pos + segment_end

        # Make sure no out of bounds.
        if left >= 0 and right <= cnt.shape[0]:
        segments.append(cnt[left:right,:])
        segment_targets.append(y[i])

        # Return as NumPy arrays
        segments_np = np.array(segments)
        segment_targets_np = np.array(segment_targets)

        return segments_np, segment_targets_np
```

The function segment_data is designed to segment the EEG data into different epochs around the cue positions. It takes in the parameters cnt (EEG data), y (target variables), a (start of the segment in milliseconds), and b (end of the segment in milliseconds). The parameters include cnt, a 2D NumPy array of size ((m, n)) where (m) is the number of time samples and (n) is the number of EEG channels; y, a 1D NumPy array con-

taining the target variables for each of the cue positions; a, the number of milliseconds after cue positions to choose as the start of the segment; and b, the number of milliseconds after cue positions to choose as the end of the segment. The variables segment_ start and segment_ end are set to the values of a and b, respectively. We initialize empty lists segments and segment_ targets to store the segmented data and their corresponding target values. We iterate through each cue position in pos, calculate the left and right boundaries of the segment, and ensure they are within bounds. If the boundaries are valid, we append the segment of data and the corresponding target value to the lists.

```
        # Segment the data to be used for model training
        seg_data, labels = segment_data(CAR_cnt, y, -50, 350)
        seg_data.shape
```

Finally, the segments and their targets are converted to NumPy arrays and returned.

```
output:
(200, 400, 59)
```

7 Feature Extraction

7.1 CSP

Using the MNE library the code for a function implementing CSP is the following:

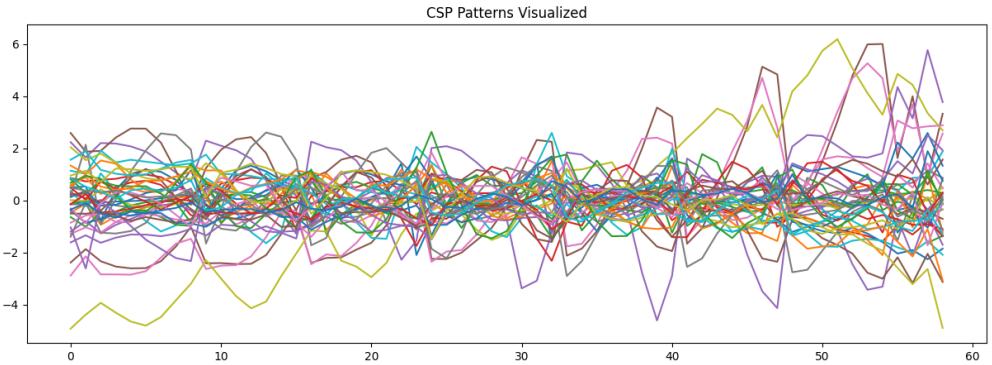
```

def common_spatial_patterns(data, labels, component_num=45, reg=None, plot=False):
    """
    Apply CSP using given data and their labels.

    Parameters
    ----------
    data : numpy.ndarray
        A 3D NumPy array of shape (n_epochs, n_times, n_channels)
        representing the data the CSP will be applied to.
        The order of n_times and n_channels here is according to
        the dataset.

    labels: numpy.ndarray
        A 1D array containig the labels for each of the epochs.

    component_num : int
        Representing the number of components (features) to be
        extracted using CSP.

    reg : None or string or float
        Type of regularization chosen for CSP.

    plot : bool
        If true, plot the CSP patterns. Otherwise do not.

    Returns
    -------
```

```
    csp_data : numpy.ndarray
        A 2D array of shape (n_epochs, n_components) containig
        the CSP applied data.

    """

    # Transpose 2nd and 3rd components to be compatible with the
    # MNE library function calls: (n_epochs, n_channels, n_times)
    csp_ready_data = data.transpose(0, 2, 1)

    # Initialize CSP for feature extraction.
    csp_obj = CSP(n_components=component_num, reg=reg, log=True)

    # Fit and apply the CSP using labels.
    csp_data = csp_obj.fit_transform(csp_ready_data, labels)

    # Plot CSP patterns if plot=True
    if plot:
        plot_csp(csp_obj, component_num)

    # Return CSP applied data
    return csp_data


def plot_csp(csp_obj, component_num):
    '''Plot the CSP features'''
    plt.figure(figsize=(15, 5))
    for i in range(component_num):
        plt.plot(csp_obj.patterns_[i], label=f'CSP Pattern {i+1}')
    plt.title('CSP Patterns Visualized')
    plt.show()
```

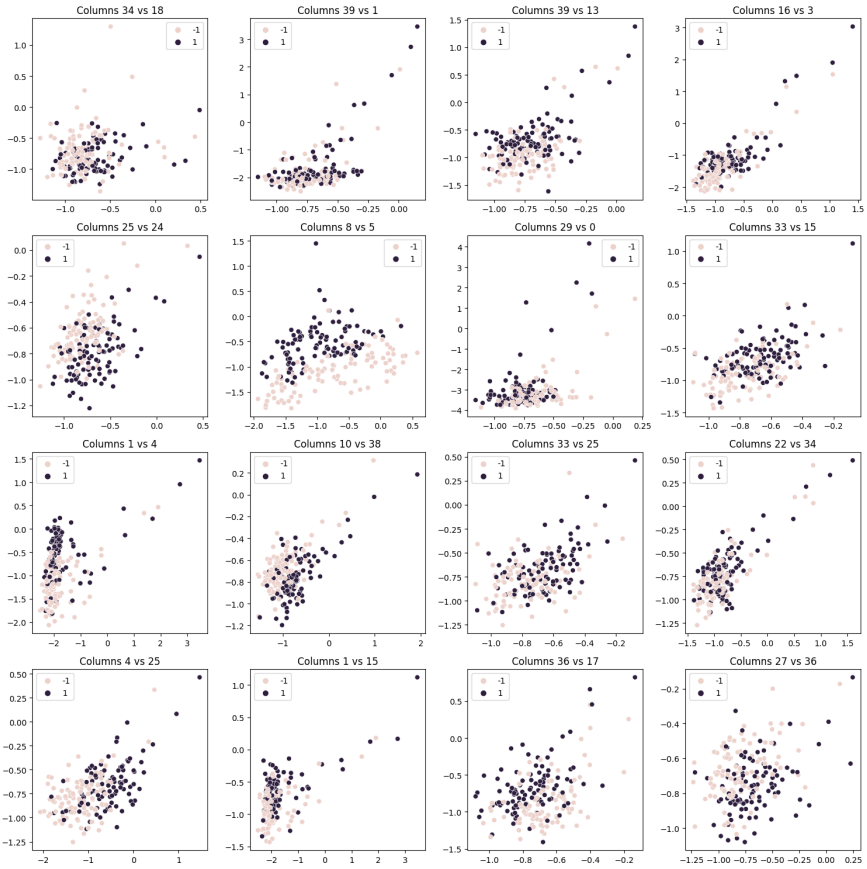
There is enough commenting to explain how the code works.
By choosing the number components to be 41 and running:

```
# Apply CSP
csp_data = common_spatial_patterns(seg_data, labels
                                   , component_num=41, plot=True)
```

The CSP produces the following patterns for dataset A:

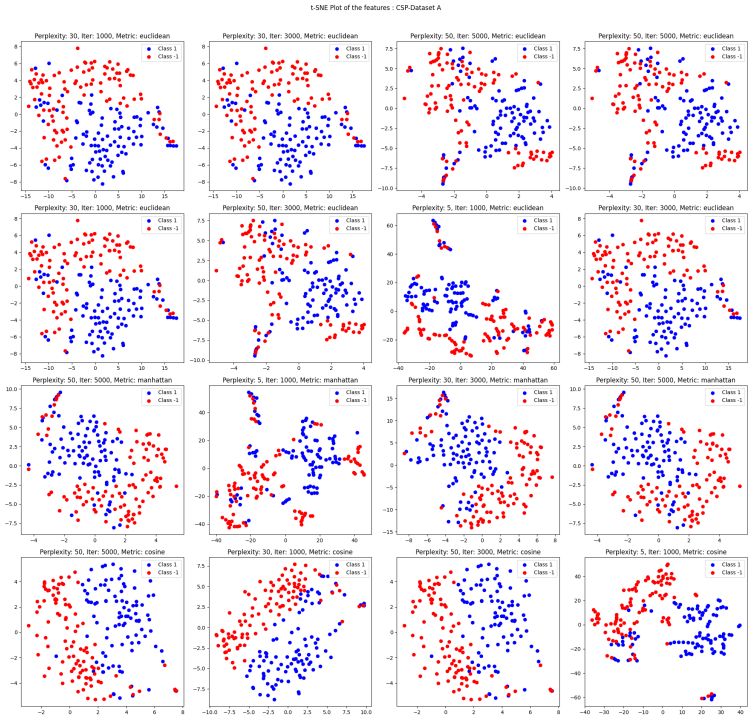


Moreover we can plot the feature extracted from the CSP in two ways. One is the following set of plots that show the scatter plot of two randomly selected features from dataset A.

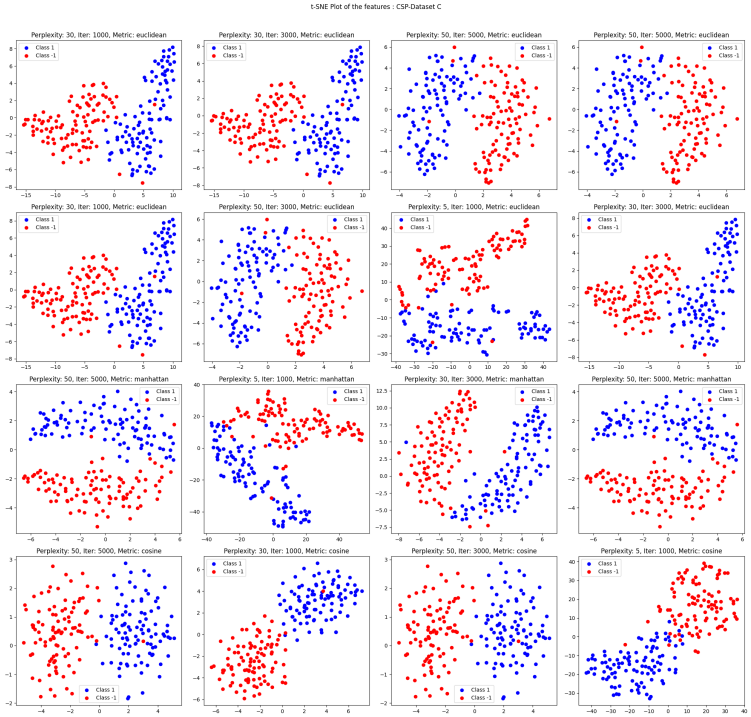


The other way is to again use t-SNE to plot high dimensional data that is present here. The following set of plots display the t-SNE visualization given 16 different sets of hyperparameters.

First for dataset A:



And then for dataset C:



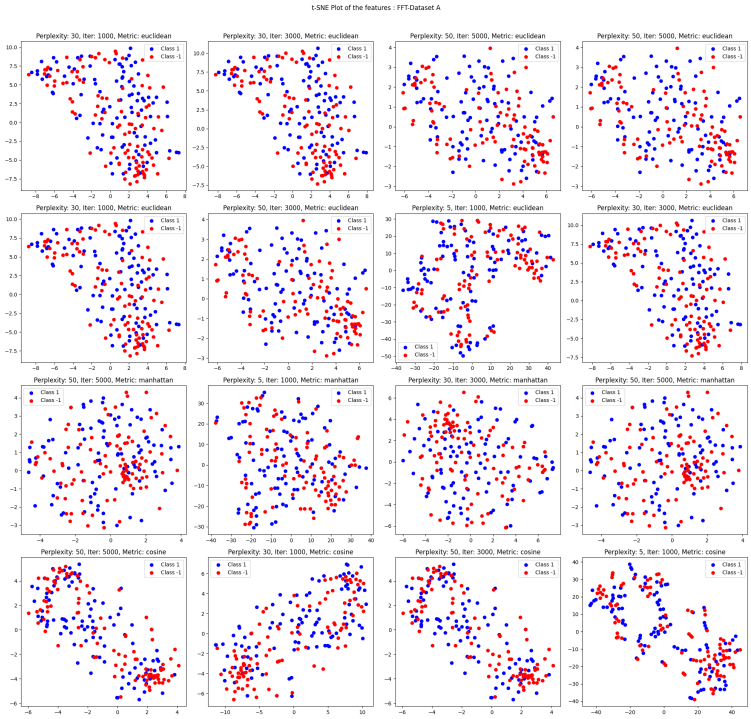
We can see that the CSP has managed to produce a good separation between the classes showing that is it appropriate feature extraction method here.

7.2 FFT

Code:

```
def fast_fourier_transform(data, fs):
    """
    Calcuate discrete Fourier transform using the FFT algorithm.

    parameters
    ----------
    data : numpy.ndarray
        The input data in the shape of (n_epochs, n_times, n_channels).

    fs : int
        The sampling frequency.

    returns
    -------
    dft : numpy.ndarray
        The resulting DFT. To use it or its magnitude as features for a
        classifiers to learn from, flatten its shape of (n_epochs, n_channels,
        n_times // 2 + 1) into (n_epochs, n_channels * (n_times // 2 + 1)).

    frequencies : numpy.ndarray
        Frequency values of the FFT.
    """

    # Prepare the data
    data = data.transpose(0, 2, 1)

    # Get the dimensions
    n_epochs, n_channels, n_times = data.shape

    # Array with complex data types to store the discrete fourier
    # transform for each epoch and channel pair. Only use the
    # n_times // 2 + 1 due to the symmetry in the DFT.
    dft = np.zeros((n_epochs, n_channels, n_times // 2 + 1), dtype=complex)

    # Go through every epoch channel combination and perform fft
    for i in range(n_epochs):
        for j in range(n_channels):
            dft[i, j, :] = fft(data[i, j, :])[:n_times // 2 + 1]

    # Calculate frequencies
    frequencies = np.fft.fftfreq(n_times, 1 / fs)[:n_times//2 + 1]

    return dft, frequencies

# Apply FFT
fft_results, frequencies = fast_fourier_transform(seg_data, fs)
abs_fft_data = np.abs(fft_results)

# Flatten to feed into classifiers
fft_data = abs_fft_data.reshape(fft_results.shape[0], -1)
```

Explained by the comments.
This time the t-SNE of the features for dataset A (we ommit dataset C for methods that are not optimal so as to not make the report too long) look like this:



The separation with this method is not too good. This is later supported by feeding its data to the classifiers. The range of accuracy of our classifiers given FFT feature extracted data is **48-64%**. The results can be seen in the notebook, we do not bring the details of the performance so as to not prolong the report too much.

7.3 CWT

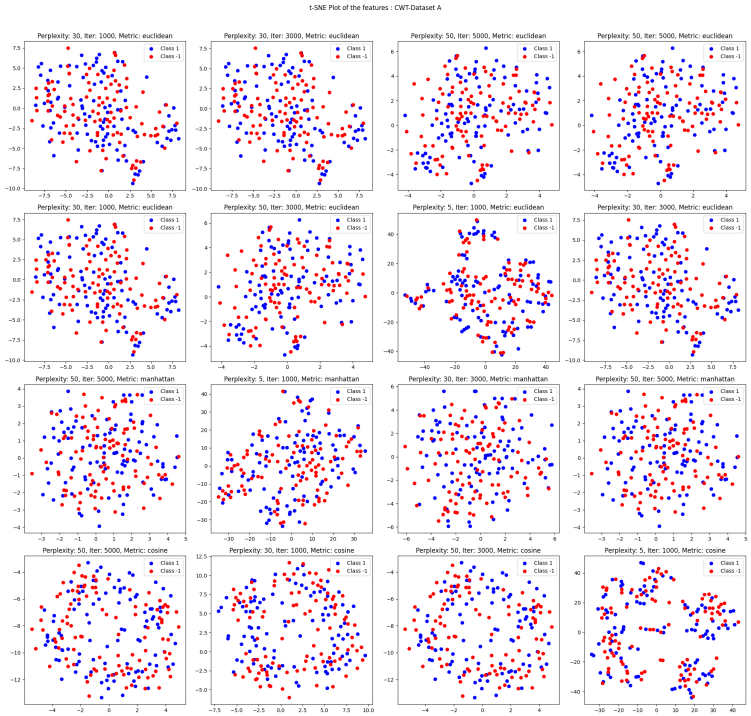
Code:

```
def continuous_wavelet_transform(data, wavelet='morl', scales=np.arange(1, 128)):
    """
    Apply continuous wavelet transfrom (CWT) to EEG data.

    Parameters
    ----------
    data : numpy.ndarray
        The EEG data of shape (n_epochs, n_times, n_channels).
    wavelet : str
        The type of wavelet to use (default = 'morl').
    scales : numpy.ndarray
        The scales to use for the CWT, deciding which frequencies
        to value.

    Returns
    -------
```

```
    results : numpy.ndarray
        The CWT applied on the data
    """

    # Transpose 2nd and 3rd component
    data = data.transpose(0, 2, 1)

    n_epochs, n_channels, n_times = data.shape

    # Store results
    result = []

    # Compute CWT for each epoch and channel combo
    for i in range(n_epochs):
        epoch_cwt = []
        for j in range(n_channels):
            cwt, _ = pywt.cwt(data[i, j, :], scales, wavelet)
            epoch_cwt.append(cwt)
        result.append(epoch_cwt)
        if i % 50 == 49:
            print(f"Epoch {i + 1} completed")

    print("Finished!")

    # Return as NumPy array
    return np.array(result)

# Apply CWT
cwt_results = continuous_wavelet_transform(seg_data, scales=np.array([32]), wavelet="morl")

# Flatten to feed to ML models
cwt_data = cwt_results.reshape(cwt_results.shape[0], -1)
```

Explanation by comments.

The t-SNE plot for dataset A is:



We can see that yet again, similar to FFT, the separation is not good. The classification accuracy is in the range of **56-62%**.

7.4 Comparison

Using the t-SNE visualization it is clear to see that the CSP leads to way better separation of the classes. This is better for both classification and clustering. The performance of the model support this observation. The models trained on the CSP data have a way higher accuracy than the ones trained on FFT or CWT data.

8 Classification

8.1 Data Used (Optimal Pre-Processed Data)

After much testing, we found the optimally pre-processed data for our models to be the data that have gone through these signal processing methods in order:

$$\text{Band-Pass Filter (8-30)} \Rightarrow \text{CAR} \Rightarrow \text{CSP} \Rightarrow \text{Standardiztion}$$

We will use the data that has this order of transformation applied to it for our classification task since it leads to the best performance in our trials.

8.2 Classification Code

Here is the code of the function used for training and evaluating the data described above in a 75-25 train/test split. It includes all of the five classifiers we have chosen for this task.

```python
def fit_classify(X_train, X_test, y_train, y_test, clf_num=3):
    """
    Train different classifiers and use them to perform classification
    using the same training and test sets.

    Parameters
    ----------
    X_train, X_test, y_train, y_test : numpy.ndarray
        Training and test sets and labels.

    clf_num : int
        Number of classifiers to be selected for the task. There is a
        priority set in place for which classifiers to select.

    Returns
    -------
    None
    """

    # List of classifiers to train
    clf_list = [SVC(kernel="linear", C=0.5, probability=True)
                , SVC(kernel="rbf", C=1, gamma='auto', probability=True)
                , LogisticRegression()
                , MLPClassifier(hidden_layer_sizes=(100, 50, 10), alpha=0.005)
                , KNeighborsClassifier(n_neighbors=3)]

    # And their names
    clf_names = [ "Support Vector Machine with Linear Kernel"
                , "Support Vector Machine with RBF Kernel"
                , "Logistic Regression"
                , "Multilayer Perceptron"
                , "K-Nearest Neighbors" ]

    clf_num = min(clf_num, len(clf_list))

    # Go through each classifier
    for i, (clf, clf_name) in enumerate(zip(clf_list[:clf_num], clf_names[:clf_num])):
        # Train on training set
        clf.fit(X_train, y_train)

        # Make prediction
        y_pred = clf.predict(X_test)

        # Evaluate model performance
        print()
        print(f"---------------------- {i + 1}. {clf_name} ----------------------")
        print()

        # Accuracy
        print(f'{clf_name} Accuracy: {accuracy_score(y_test, y_pred)*100}%')
        print(classification_report(y_test, y_pred))

        # Confusion Matrix
        cm = confusion_matrix(y_test, y_pred)
        cm_display = ConfusionMatrixDisplay(cm, display_labels=[1, -1])
```

```
        print(cm)
        cm_display.plot(cmap="Blues"); plt.show()

        # ROC curve
        y_probabilities = clf.predict_proba(X_test)[:, 1]
        fpr, tpr, thresholds = roc_curve(y_test, y_probabilities)
        roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr)
        roc_display.plot(); plt.show()
```

There is enough of an explanation in the code itself using the comments, so we will ommit further explanation.
Of course before using this code we have to call on the following function to first normalize (standardize) the data and then split the data into the 75-25 train/test ratio.

```
def train_test_normalize_split(X, y, normalize=True):
    """
    Split the data into 75-25 train/test data and standardize
    them if normalize flag is True.
    """
    # Standardize the data if normalize=True
    if normalize:
        # Initialize standardiztion transform
        scaler = StandardScaler()

        # Apply standardization
        X = scaler.fit_transform(X)

    # Split the data
    return train_test_split(X, y,\
        train_size=0.75, random_state=35, stratify=y)

# Normalize and split the data
X_train, X_test, y_train, y_test = train_test_normalize_split(csp_data, labels)
```

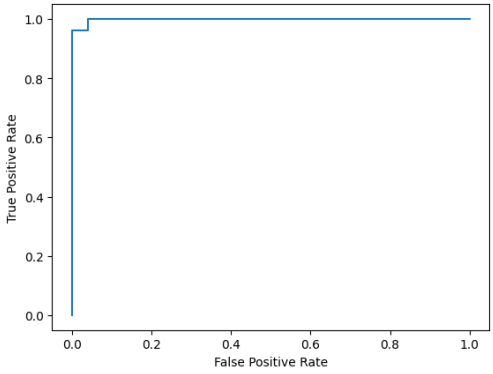
The stratify here makes sure that the ratio of the data between the train and test sets are consistent with the overall dataset.

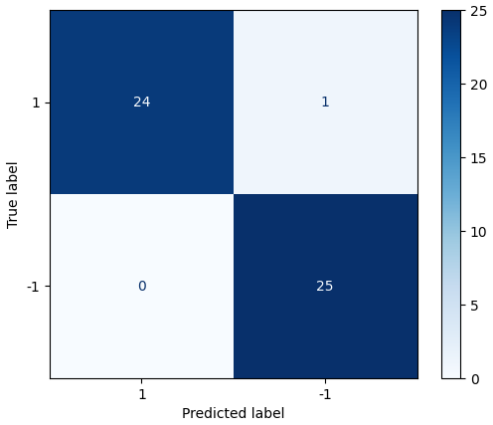
8.3 Support Vector Machine (SVM) with Linear Kernel

– **Model:** We chose to use the SVM with linear kernel due to its linear nature. By analyzing the features using the t-SNE visualization, it seemed plausible that a linear model could perform well on separating the classes. Thus, we chose to have two linear models, first the SVM with the linear kernel which is basically the maximum margine classifier, and second, the logistic regression model. Using two different linear models that create their linear decision boundary using two different methodology seemed like great choices. Specifically, SVM's have shown to be very flexible and powerful models and using it made a lot of sense.
– **Performance on Dataset A:**

```
---------------------- 1. Support Vector Machine with Linear Kernel ----------------------

Support Vector Machine with Linear Kernel Accuracy: 98.0%
```

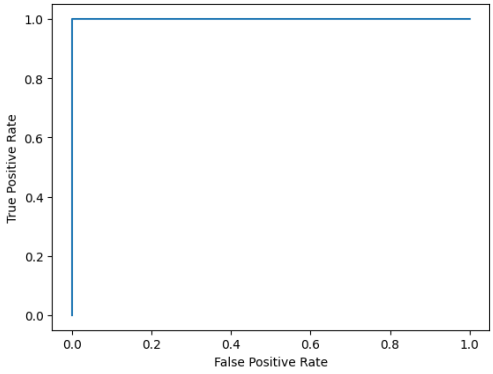
```
              precision    recall  f1-score   support

          -1       1.00      0.96      0.98        25
           1       0.96      1.00      0.98        25

    accuracy                           0.98        50
   macro avg       0.98      0.98      0.98        50
weighted avg       0.98      0.98      0.98        50
```

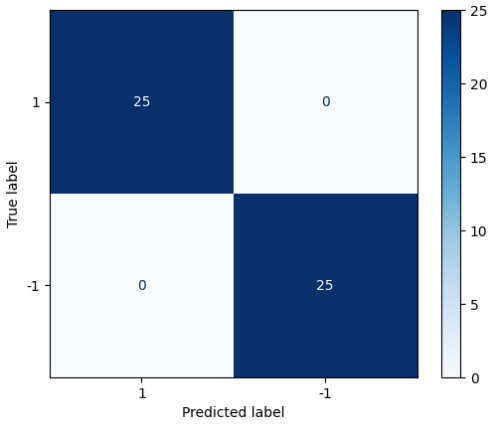




– Performance on Dataset C:

```
---------------------- 1. Support Vector Machine with Linear Kernel ----------------------

Support Vector Machine with Linear Kernel Accuracy: 100.0%
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00        25
           1       1.00      1.00      1.00        25

    accuracy                           1.00        50
   macro avg       1.00      1.00      1.00        50
weighted avg       1.00      1.00      1.00        50
```



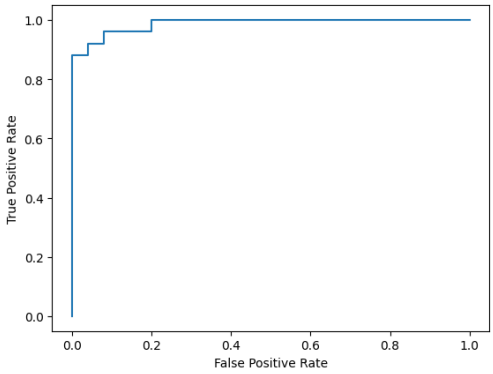


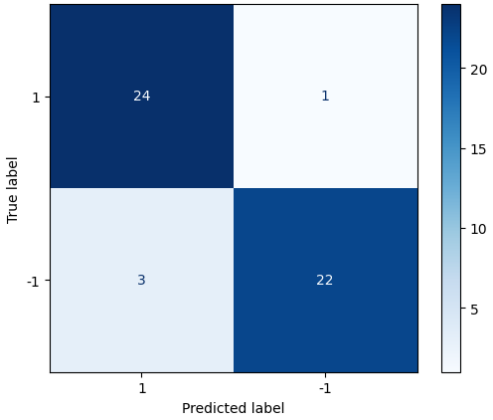
8.4 Support Vector Machine (SVM) with RBF Kernel

– **Model:** The use of this model was mostly done to have another SVM model with a different kernel than the linear to compare them. Looking at the performance, we can see that overall, the SVM with the linear kernel performs better than the one with the RBF kernel. Their difference is not too much but perhaps the slight difference in performance can be attributed to the assumption of Gaussian distribution that RBF kernel SVM has not being 100 percent true here. Overall though, the performance is still good.

– Performance on Dataset A:

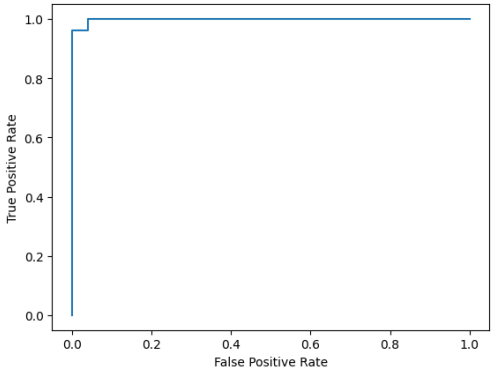
```
---------------------- 2. Support Vector Machine with RBF Kernel ----------------------

Support Vector Machine with RBF Kernel Accuracy: 92.0%
              precision    recall  f1-score   support

          -1       0.89      0.96      0.92        25
           1       0.96      0.88      0.92        25

    accuracy                           0.92        50
   macro avg       0.92      0.92      0.92        50
weighted avg       0.92      0.92      0.92        50
```

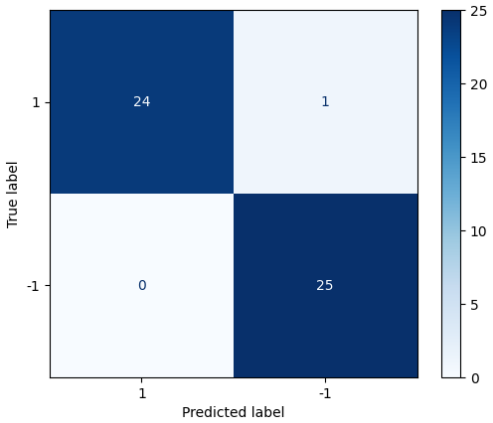




– Performance on Dataset C:

```
---------------------- 2. Support Vector Machine with RBF Kernel ----------------------

Support Vector Machine with RBF Kernel Accuracy: 98.0%
              precision    recall  f1-score   support

          -1       1.00      0.96      0.98        25
           1       0.96      1.00      0.98        25

    accuracy                           0.98        50
   macro avg       0.98      0.98      0.98        50
weighted avg       0.98      0.98      0.98        50
```



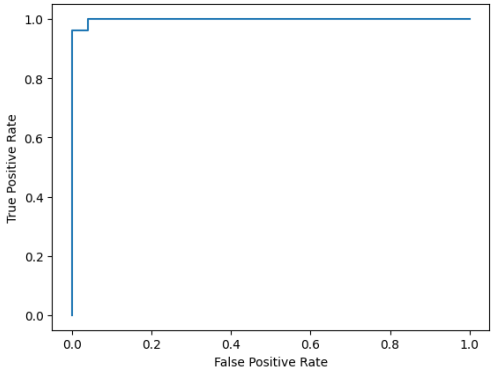


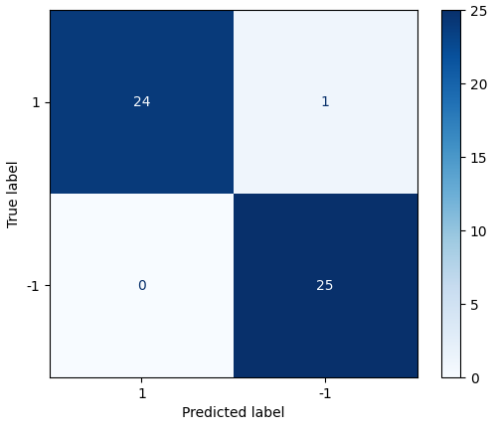
8.5 Logistic Regression

– **Model:** The logistic regression is the other linear model we used. It serves as another linear model counterpart to the SVM with linear kernel. Since the way it works is different from the SVM, it is an interesting model to have. The logistic regression while simple, can yield great performance given a dataset that is close to being linearly separable. Looking at the visualizations of the features extracted from the CSP, it is quite believable that linear decision boundary might obtain great performance. This is why logistic regression is used. The logistic regression along with the SVM with linear kernel actually turned out to be the best performing of the bunch.

– **Performance on Dataset A:**

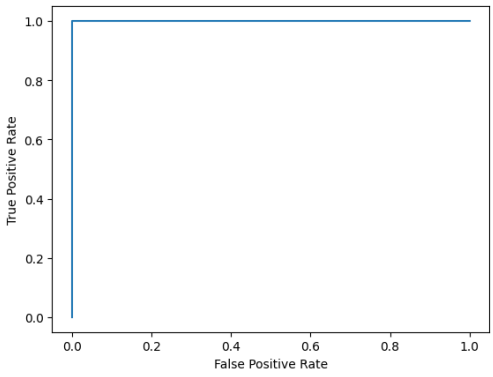
```
----------------------- 3. Logistic Regression -----------------------

Logistic Regression Accuracy: 98.0%
              precision    recall  f1-score   support

          -1       1.00      0.96      0.98        25
           1       0.96      1.00      0.98        25

    accuracy                           0.98        50
   macro avg       0.98      0.98      0.98        50
weighted avg       0.98      0.98      0.98        50
```

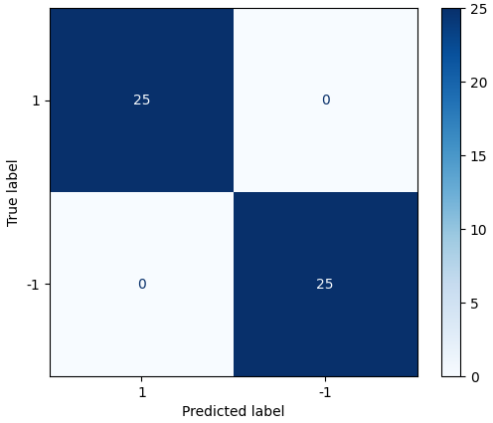




– Performance on Dataset C:

```
---------------------- 3. Logistic Regression ----------------------

Logistic Regression Accuracy: 100.0%
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00        25
           1       1.00      1.00      1.00        25

    accuracy                           1.00        50
   macro avg       1.00      1.00      1.00        50
weighted avg       1.00      1.00      1.00        50
```



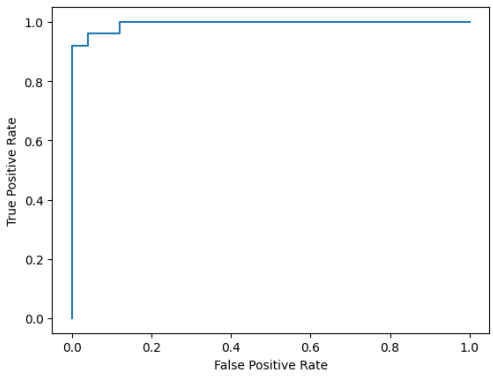


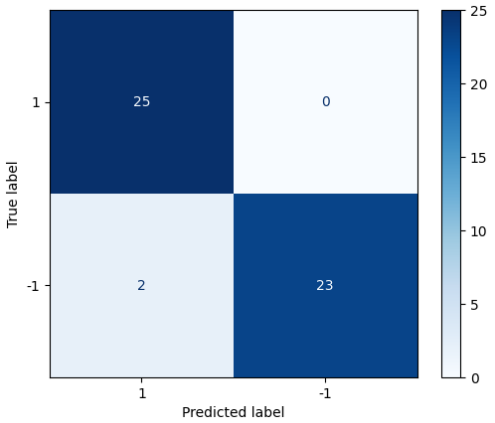
8.6 Multilayer Perceptron (MLP)

– Model: The MLP or the feedforward neural network, is a more sophisticated model that can fit to more complex patterns given enough data. That is why the use of the MLP in such a project as this one is definitely warranted. Neural networks have been recently shown to output excellent performance across a wide variety of applications. Hence, why we have used it here. It is performing well here as well. The MLP used here has three hidden layers, with 100, 50, 10 units in each of the first to third layers respectively. Given more fine-tuning the MLP could probably achieve even better performance.

– Performance on Dataset A:

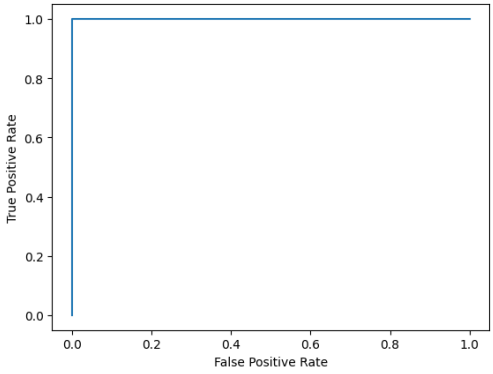
```
---------------------- 4. Multilayer Perceptron ----------------------

Multilayer Perceptron Accuracy: 96.0%
              precision    recall  f1-score   support

          -1       0.96      0.96      0.96        25
           1       0.96      0.96      0.96        25

    accuracy                           0.96        50
   macro avg       0.96      0.96      0.96        50
weighted avg       0.96      0.96      0.96        50
```

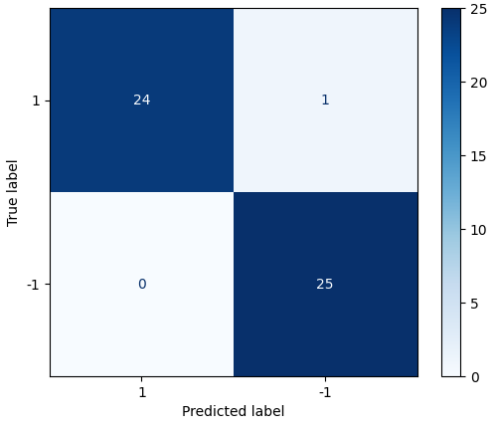




– Performance on Dataset C:

```
---------------------- 4. Multilayer Perceptron ----------------------

Multilayer Perceptron Accuracy: 98.0%
              precision    recall  f1-score   support

          -1       1.00      0.96      0.98        25
           1       0.96      1.00      0.98        25

    accuracy                           0.98        50
   macro avg       0.98      0.98      0.98        50
weighted avg       0.98      0.98      0.98        50
```



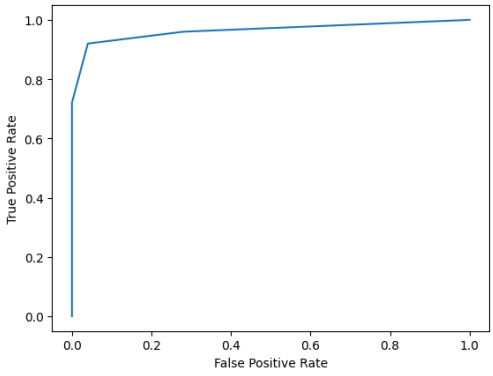


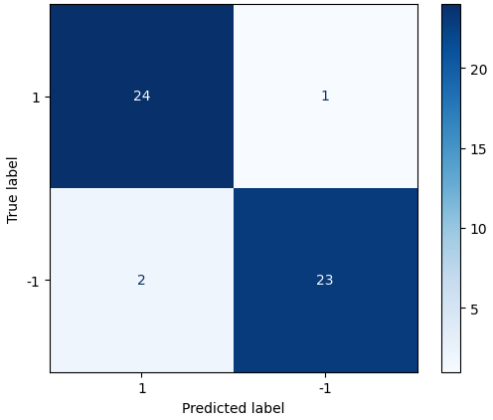
8.7 K-Nearest Neighbors (KNN)

– Model: Given that we might find the optimal hyperparameters for it, the KNN is also another powerful machine learning model. The main reason we have included it here, is to have yet another model to compare against the other that operates way differently compared to the others. Given the capabilities of a KNN model with a good hyperparameter, we expected it to perform well, but since we also knew that the data seemed to be quite dense after the pre-processing, it would achieve a worse performance when compared to MLP, logistic regression or linear-kernel SVM.

– Performance on Dataset A:

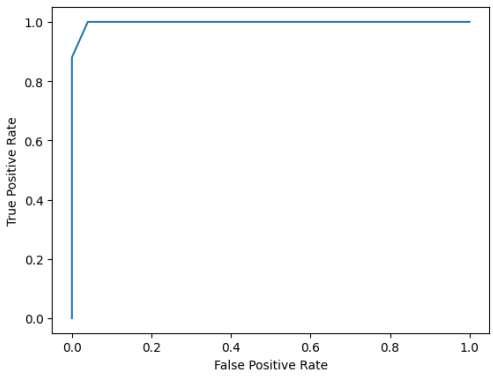
```
----------------------- 5. K-Nearest Neighbors -----------------------

K-Nearest Neighbors Accuracy: 94.0%
              precision    recall  f1-score   support

          -1       0.92      0.96      0.94        25
           1       0.96      0.92      0.94        25

    accuracy                           0.94        50
   macro avg       0.94      0.94      0.94        50
weighted avg       0.94      0.94      0.94        50
```

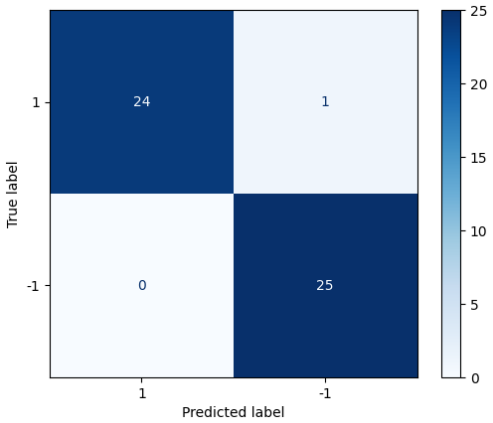




– Performance on Dataset C:

```
----------------------- 5. K-Nearest Neighbors -----------------------

K-Nearest Neighbors Accuracy: 98.0%
              precision    recall  f1-score   support

          -1       1.00      0.96      0.98        25
           1       0.96      1.00      0.98        25

    accuracy                           0.98        50
   macro avg       0.98      0.98      0.98        50
weighted avg       0.98      0.98      0.98        50
```





8.8 Comparison and Analysis

A large part of the comparison and analysis was explained in the sections of each of the classifiers. To add more details, all of these five models seem to achieve a great performance, all achieving an accuracy of at least 90%.
However, it is also clear to see that the linear models, logistic regression and linear-kernel SVM generally seem be peforming slightly better than the rest. It is also visible that the models trained on dataset C perform better than the ones performed on dataset A. It seems that classifying between the left and right hands is easier than classifying between the left hand and the feet using EEG data!
There are more performance evaluations using the other data processing and feature extraction methods in the notebook showing how feature extraction methods other than the CSP lead to way poorer performance. Moreover, in our code, the use of PCA or ICA (or them together) did not lead to better performance. CSP and CAR seem to be doing a great job preparing the data for the classifiers.

9 Clustering

In this project, we use `KMeans`, `DBScan` and `GMM` algorithms to cluster the datapoint of EEG signals.

9.1 Finding the best number of clusters

There are many ways to guess the best number of clusters in clustering algorithms. In `KMeans` algorithm we used `silhouette score` approach and `silhouette score` approach and `AIC/BIC` approach in `GMM`.

9.1.1 Silhouette score

The silhouette coefficient assesses how well each data point fits into its assigned cluster. For each data point we do the following steps:

- At the first we calculate the average distance to all other data points within the same cluster (cohesion).
- We calculate the average distance to all data points in the nearest neighboring cluster (separation).

- At the end we compute the silhouette coefficient using the following formula:

$$\text{silhouette coefficient} = \frac{\text{separation} - \text{cohesion}}{\max(\text{separation}, \text{cohesion})}$$

- Then we should average the silhouette coefficients across all data points to obtain the overall silhouette score for the clustering result.

The proper number of clusters in the graph of silhouette scores with considering different number of clusters, is typically at the peak of the plot! It signals to us highest average silhouette score, that indicating the case clusters are dense and well-separated.

9.1.2 AIC/BIC

These are two metrics that help us choose proper number of clusters in clustering algorithms like `GMM` with model selection approach.

`AIC` or Akaike Information Criterion is calculated as follows:

$$^{\circ}\, AIC = 2k - 2\ln(L)$$

where k is the number of parameters in the model and L is the maximum likelihood of the model.

`BIC` or Bayesian Information Criterion is calculated as:

$$^{\circ}\, BIC = k\ln(n) - 2\ln(L)$$

where k, L is the same and n is the number of data points.

To avoid redundant details, we ignore more mathematical aspects of these metrics. The only thing that important to know is: In the graph related to `AIC/BIC`, the "elbow point" or the point where the decrease in `AIC/BIC` values starts, is a hint for us for finding proper number of clusters.

9.2 Analyze the clustering method performance

9.2.1 Silhouette score

We defined the silhouette mathematically and express its meaning in the previous section! Here we just state the interpretation method ot that:

Silhouette values close to 1 indicate well-clustered data points. Silhouette values close to 0 suggest overlapping clusters. Silhouette values close to -1 indicate misclassified data points. So there is some kinds of overlapping in both of KMEANS clustering and DBSCAN clustering.

And that makes sense! Because if separation(inter-cluster distances) is higher than cohesion(intra-cluster distances) considerably, then our gained clusters by applying a specific clustering method, are too far from each other but are dense. So we are in a well-separated clustering problem.

Also in the cases that separation is near to cohesion, we have overlapped clusters. Because our gained clusters from an algorithm, are near to each other based on the definitions of separation and cohesion!

9.2.2 Homogeneity score

Homogeneity measures how well each cluster contains only members of a single class or category. For each cluster:

- we compute the proportion of data points that belong the majority class within that cluster.
- Then we sum up these proportions across all clusters.
- At the end we divide the sum by the total number of data points.

A higher homogeneity score means that clusters are more pure in terms of class membership!

9.3 Code section

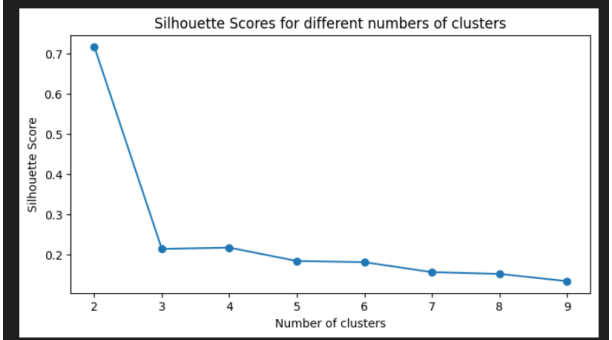
9.3.1 KMeans

At the first we applied the `KMeans` method. We picked the proper number of clusters by demonstrating silhouette score plot:

```
def k_means(data):
    # Initialize variables
    silhouette_scores = []
    range_n_clusters = range(2, 10)

    # Iterate over different numbers of clusters
    for n_clusters in range_n_clusters:
        kmeans = KMeans(n_clusters=n_clusters, random_state=42)
        pred = kmeans.fit_predict(data)
        silhouette_avg = silhouette_score(data, pred)
        silhouette_scores.append(silhouette_avg)
        print(f'Number of clusters: {n_clusters},\
            Silhouette Score: {silhouette_avg}')

    # Plot the silhouette scores
    plt.figure(figsize=(8, 4))
    plt.plot(range_n_clusters, silhouette_scores, marker='o')
    plt.title("Silhouette Scores for different numbers of clusters")
    plt.xlabel("Number of clusters")
    plt.ylabel("Silhouette Score")
    plt.show()

    # Determine the optimal number of clusters
    optimal_n_clusters = range_n_clusters[np.argmax(silhouette_scores)]
    print(f'Optimal number of clusters according to \
    silhouette score: {optimal_n_clusters}')
```

As we see in the following picture, two clusters may be better choice than any other options! Because we get highest `silhouette score` with it. Notice that we usually know that the number of clusters is higher than 1! So we ignore it in the plot!

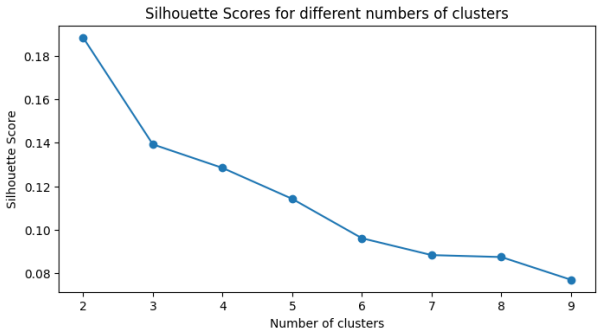
```
Number of clusters: 2, Silhouette Score: 0.7172523888216377
Number of clusters: 3, Silhouette Score: 0.2146360060520609 6
Number of clusters: 4, Silhouette Score: 0.21786386190860793
Number of clusters: 5, Silhouette Score: 0.18468241781876585
Number of clusters: 6, Silhouette Score: 0.181689230078823
Number of clusters: 7, Silhouette Score: 0.1568746280710572 3
Number of clusters: 8, Silhouette Score: 0.15255800780994414
Number of clusters: 9, Silhouette Score: 0.13455301788472893
```




```
Optimal number of clusters according to silhouette score: 2
```

The same code for dataset C produces:

```
Number of clusters: 2,          Silhouette Score: 0.188280846861299
Number of clusters: 3,          Silhouette Score: 0.13921078215576885
Number of clusters: 4,          Silhouette Score: 0.12837976498965264
Number of clusters: 5,          Silhouette Score: 0.11416317928447865
Number of clusters: 6,          Silhouette Score: 0.09607754291953167
Number of clusters: 7,          Silhouette Score: 0.08833152099038372
Number of clusters: 8,          Silhouette Score: 0.08744021148019625
Number of clusters: 9,          Silhouette Score: 0.07698350525776579
```



So we chose 2 as the number of clusters and run the `KMeans` algorithm:

```
def specific_kmeans(data, labels, number_clusters=2):
    '''Function for more detailed run of kmeans using a specific cluster number'''
    #Create a k-means model with specific number of clusters
    kmeans = KMeans(n_clusters=number_clusters, init='k-means++', max_iter=300, \
                    random_state=42)

    #Fit for kmeans clustering
    kmeans.fit(data)
```

```python

    #Get labels
    pred_labels_kmeans = kmeans.labels_

    #Get Centroids
    centroids_kmeans = kmeans.cluster_centers_

    print("Center(centroid) of clusters in KMEANS method  with \
        $n_clusters={}$ are as follows:\n\n".format(number_clusters),\
              centroids_kmeans)
    print("*"*90)
    print("Predicted labels in KMEANS method with $n_clusters={}$ \
        are as follows:\n".format(number_clusters))
    print(pred_labels_kmeans)
    print("*"*90)

    # Homogeneity
    homogeneity_kmeans = homogeneity_score(labels_true=labels,\
                                          labels_pred= pred_labels_kmeans)
    print("Homogeneity score of KMEANS method with {0} clusters \
        is: ".format(number_clusters),homogeneity_kmeans)

    # Silhouette
    silhouette_score_kmeans = silhouette_score(data,pred_labels_kmeans)
    print("Silhouette score of KMEANS method with {0} clusters\
        is:  ".format(number_clusters),silhouette_score_kmeans)
```

We got the following results: (Centroid of each cluster can be seen in the Jupyter Notebook file!)

```
Predicted labels in KMEANS method with $n_clusters=2$ are as follows:

[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0]
******************************************************************************************
Homogeneity score of KMEANS method with 2 clusters is:  0.0015284029975106007
Silhouette score of KMEANS method with 2 clusters is:   0.7172523888216377
```

For 5 clusters and 9 cluster we got:

```
Predicted labels in KMEANS method with $n_clusters=5$ are as follows:

[3 1 3 3 3 2 2 0 0 2 0 2 0 0 0 0 0 0 0 0 0 4 4 0 4 4 4 2 2 0 0 0 0 0 0 0 0
 2 2 2 0 2 2 0 2 2 2 3 2 2 2 0 0 0 2 2 0 2 0 0 0 0 0 0 2 4 2 0 2 0 4 4 1 1
 3 2 2 2 2 4 4 4 2 4 2 4 4 0 2 0 4 1 4 0 0 0 2 2 0 0 0 0 2 2 0 2 2 0 2 4 2
 4 2 4 4 0 0 0 2 2 4 4 4 2 4 4 2 2 4 4 0 0 2 4 2 4 2 3 4 4 4 2 4 2 2 3 4 0
 4 2 4 4 2 2 0 2 2 2 2 4 1 0 0 0 2 0 4 4 4 4 2 2 2 2 0 1 0 4 2 2 2 4 2 4 4
 4 2 4 2 0 1 0 1 3 2 2 0 0 3 2]
******************************************************************************************
Homogeneity score of KMEANS method with 5 clusters is:  0.2597801269819919
Silhouette score of KMEANS method with 5 clusters is:   0.18468241781876585
```

```
Predicted labels in KMEANS method with $n_clusters=9$ are as follows:

[8 1 3 3 8 8 8 6 6 8 6 8 6 6 6 2 2 2 6 2 4 4 5 6 4 5 5 0 8 6 6 6 6 6 6 6 6
 8 0 8 6 8 8 6 0 6 8 3 8 4 0 2 2 2 8 8 2 8 4 4 6 2 2 2 4 5 4 2 4 2 5 5 7 1
 3 8 8 8 0 5 5 5 0 4 0 5 0 2 4 2 5 1 4 2 2 2 4 4 2 4 6 2 4 0 2 0 4 2 8 5 4
 5 0 5 6 4 4 6 4 4 4 4 5 0 5 5 0 0 5 5 6 4 4 4 0 5 0 3 5 5 5 4 5 0 4 3 5 2
 4 0 5 5 0 0 2 4 0 0 4 5 1 2 4 2 4 2 4 0 5 5 0 0 0 8 2 1 6 5 8 0 0 5 0 5 5
 5 8 4 4 2 1 6 1 3 8 8 6 6 3 8]
******************************************************************************************
Homogeneity score of KMEANS method with 9 clusters is:  0.44684600467735075
Silhouette score of KMEANS method with 9 clusters is:   0.13455301788472893
```

For dataset C the results are in the same order:

```
Predicted labels in KMEANS method with $n_clusters=2$         are as follows:

[0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0
 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1 1 1 1
 1 0 0 1 1 0 1 1 1 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 1 1 1 1 1 0
 0 0 1 0 0 1 1 0 0 0 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0
 0 1 1 1 1 1 0 1 0 1 1 0 0 1 0]
*********************************************************************************
Homogeneity score of KMEANS method with 2 clusters          is:  0.003735975618982398
Silhouette score of KMEANS method with 2 clusters         is:   0.188280846861299

Predicted labels in KMEANS method with $n_clusters=5$         are as follows:

[4 2 1 1 1 4 4 1 1 0 1 4 4 1 2 1 4 1 1 4 4 2 4 4 2 4 2 1 2 4 1 2 4 0 4 2 2
 1 2 1 1 4 4 2 2 4 2 2 1 2 4 4 2 4 2 2 1 1 2 1 1 1 2 3 4 3 1 1 4 4 2 2 2 1
 2 3 1 1 2 2 1 4 4 2 2 3 1 2 4 4 1 4 1 1 1 2 2 4 1 1 0 1 1 4 1 1 3 0 1 3 0
 0 1 2 0 4 2 3 0 1 0 3 4 1 4 3 0 1 1 4 3 0 3 0 1 1 4 1 3 0 0 4 0 1 4 0 0 1
 2 4 3 1 1 3 0 1 1 2 4 0 3 4 4 3 3 0 0 1 0 0 3 3 3 1 0 0 3 1 1 4 3 4 4 4 4
 4 3 1 0 3 0 4 3 1 0 0 1 4 3 1]
*********************************************************************************
Homogeneity score of KMEANS method with 5 clusters          is:  0.7614209108546317
Silhouette score of KMEANS method with 5 clusters         is:   0.11416317928447865

Predicted labels in KMEANS method with $n_clusters=9$         are as follows:

[4 1 1 7 1 5 4 7 3 5 1 0 4 7 4 7 4 7 3 5 4 8 4 0 8 4 4 1 1 0 7 8 4 5 4 8 1
 7 1 3 3 0 0 4 1 4 8 4 1 4 5 4 8 4 8 4 7 7 8 1 7 3 4 3 4 2 3 1 5 0 1 4 8 1
 8 2 7 1 8 8 7 5 5 1 8 3 7 1 4 5 7 0 7 3 7 8 1 4 1 7 0 3 7 5 1 3 3 5 3 2 6
 6 3 8 5 5 4 2 6 3 6 3 5 1 4 3 6 1 7 0 2 6 3 6 7 3 4 3 2 5 6 0 6 3 0 6 6 7
 4 0 2 7 7 2 6 4 1 1 0 6 2 4 0 2 2 5 5 7 6 6 2 3 3 3 6 6 3 3 3 0 3 0 4 4 0
 0 3 3 6 2 6 4 2 1 6 6 3 4 3 7]
*********************************************************************************
Homogeneity score of KMEANS method with 9 clusters          is:  0.8973278486908582
Silhouette score of KMEANS method with 9 clusters         is:   0.07698350525776579
```

When we increase the number of clusters, homogeneity of clustering will increase! This makes sense! Because the `homogeneity score` measure the purity of clusters! When we increase the number of clusters, the probability of belonging of all data points to a single cluster will increases (we have more clusters and less deviation between members of each cluster!).

As we saw, when the number of clusters gets far away from 2, then the silhouette score will decreases!

9.3.2 DBScan

Now we apply a density based algorithm for clustering the data points:

```
def db_scan(data, labels):
    #Create DBSCAN model for clustering
    eps_ = 3
    min_samples_ = 6

```

```
    #dbscan = DBSCAN(eps=2, min_samples=6, metric='chebyshev')
    dbscan = DBSCAN(eps=eps_, min_samples=min_samples_, metric='euclidean')

    #Fit for dbscan clustering
    dbscan.fit(data)

    #Get labels and number of clusters
    pred_labels_dbscan = dbscan.labels_
    number_clusters_dbscan = len(np.unique(pred_labels_dbscan)) - 1

    print("Number of clusters in DBScan method with $eps={0},\
    min_samples={1}$ is:".format(eps_,min_samples_),number_clusters_dbscan)
    print("Predicted labels in DBScan method with $eps={0},\
    min_samples={1}$ are as follows:\n".format(eps_,min_samples_))
    print(pred_labels_dbscan)
    print("*"*90)

    # Homogeneity
    homogeneity_dbscan = homogeneity_score(labels_true=labels, \
                                           labels_pred= pred_labels_dbscan)
    print("Homogeneity score of DBScan method with with $eps={0},\
        min_samples={1}$ is: ".format(eps_,min_samples_),homogeneity_dbscan)

    # Silhouette
    silhouette_score_dbscan = silhouette_score(data,pred_labels_dbscan)
    print("Silhouette score of DBScan method with with $eps={0},\
        min_samples={1}$ is: ".format(eps_,min_samples_),silhouette_score_dbscan)
```

In this section, we choose many values for `eps` and `min_samples` that are our hyperparameters, but we did not get good results!

```
Number of clusters in DBScan method with $eps=3,min_samples=6$ is: 2
Predicted labels in DBScan method with $eps=3,min_samples=6$ are as follows:

[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 -1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  1  0
  1  0  0  0  0  0  0  0]
********************************************************************************
Homogeneity score of DBScan method with with $eps=3,min_samples=6$ is:  0.002526266704011455
Silhouette score of DBScan method with with $eps=3,min_samples=6$ is:  0.7084430275030855
```

We can conclude that the data points are so dense! That means this algorithm can't cluster data points properly. It means data points have considerable overlapping!

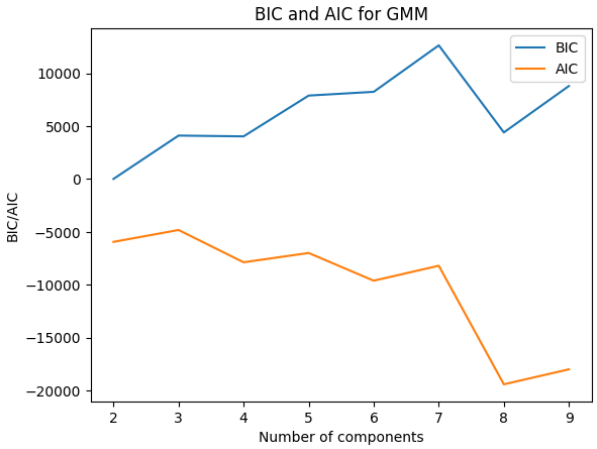
9.3.3 GMM

Now we apply `GMM` clustering method. At the first as we said before, we use `AIC/BIC` method to obtain proper number of clusters:

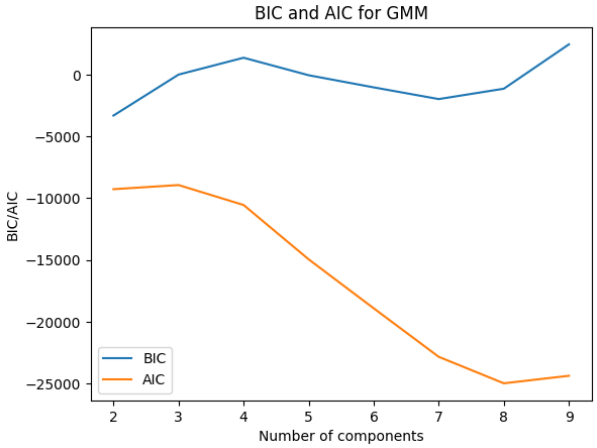
```
def best_gmm(data):
    # AIC and BIC analysis for finding the best number of components to cluster datapoints!
    n_components = np.arange(2, 10)
    models = [GaussianMixture(n, random_state=42).fit(data) for n in n_components]
    bics = [model.bic(data) for model in models]
```

```
    aics = [model.aic(data) for model in models]

    plt.plot(n_components, bics, label='BIC')
    plt.plot(n_components, aics, label='AIC')
    plt.legend(loc='best')
    plt.xlabel('Number of components')
    plt.ylabel('BIC/AIC')
    plt.title('BIC and AIC for GMM')
    plt.show()

    optimal_n_components = n_components[np.argmin(bics)]
    print("Optimal number of components according to BIC:", optimal_n_components)
```

The related plot that we got is as follows:



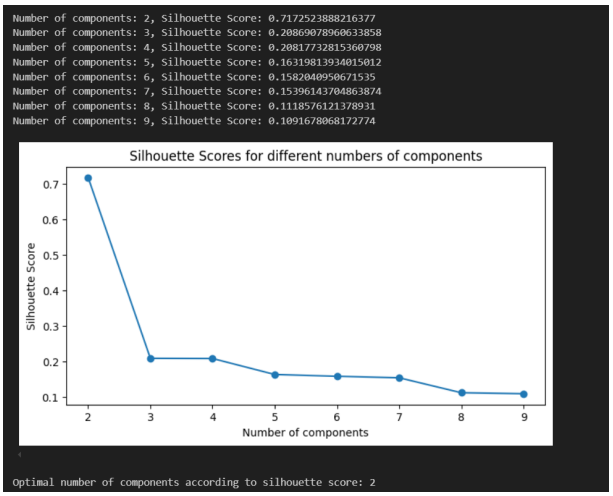
and for dataset C:



Based on the above plot and the policy about encountering it(we explained in the previous sections), we should choose 2 as the number of clusters in `GMM` clustering algorithm. Also we use `silhouette score`

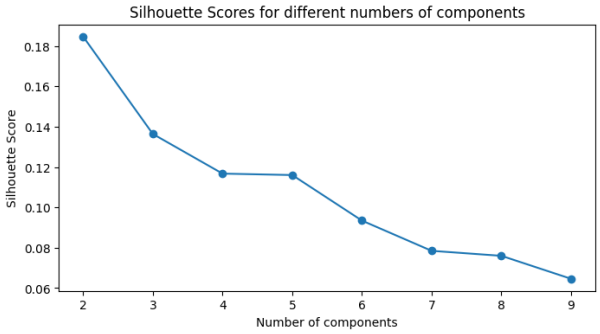
method for finding the proper number of clusters and the result was identical!

```
def optimal_gmm(data):
    # Initialize variables
    silhouette_scores = []
    range_n_components = range(2, 10)  # You can adjust this range based on your requirements

    # Iterate over different numbers of components
    for n_components in range_n_components:
        gmm = GaussianMixture(n_components=n_components, random_state=42)
        pred_labels = gmm.fit_predict(data)
        silhouette_avg = silhouette_score(data, pred_labels)
        silhouette_scores.append(silhouette_avg)
        print(f'Number of components: {n_components}, Silhouette Score: {silhouette_avg}')

    # Plot the silhouette scores
    plt.figure(figsize=(8, 4))
    plt.plot(range_n_components, silhouette_scores, marker='o')
    plt.title("Silhouette Scores for different numbers of components")
    plt.xlabel("Number of components")
    plt.ylabel("Silhouette Score")
    plt.show()

    # Determine the optimal number of components
    optimal_n_components = range_n_components[np.argmax(silhouette_scores)]
    print(f'Optimal number of components according to silhouette score: {optimal_n_components}')
```



and for dataset C:

```
Number of components: 2, Silhouette Score: 0.1847416790964504
Number of components: 3, Silhouette Score: 0.13642483442219125
Number of components: 4, Silhouette Score: 0.1167600813921243
Number of components: 5, Silhouette Score: 0.11604906063661904
Number of components: 6, Silhouette Score: 0.0934582048369348
Number of components: 7, Silhouette Score: 0.07845680293174936
Number of components: 8, Silhouette Score: 0.07599180765486122
Number of components: 9, Silhouette Score: 0.06458818560261614
```



We applied `GMM` for three different number of clusters and the results are here:(Means of Gaussian components and Covariances of Gaussian components are available in Jupyter Notebook file).

```
def apply_gmm(data, labels, n_comp=2):
    gmm = GaussianMixture(n_components=n_comp, random_state=42)
    pred_labels_gmm = gmm.fit_predict(data)

    silhouette_gmm = silhouette_score(data, pred_labels_gmm)
    homogeneity_gmm = homogeneity_score(labels_true=labels, labels_pred= pred_labels_gmm)

    print(f'Number of components: {n_comp}, Silhouette Score: {silhouette_gmm}')
    print(f'Number of components: {n_comp}, Homogeneity Score: {homogeneity_gmm}')
```

```
Number of components: 2, Silhouette Score: 0.7172523888216377
Number of components: 2, Homogeneity Score: 0.0015284029975106007
```

The code section for 5 clusters and 9 clusters uses this function as well. But the results are as follows:

```
Number of components: 2, Silhouette Score: 0.16319813934015012
Number of components: 2, Homogeneity Score: 0.29110627071035255
```

```
Number of components: 2, Silhouette Score: 0.1091678068172774
Number of components: 2, Homogeneity Score: 0.3696426590343457
```

for dataset C results check the notebook!
Same as `KMeans` algorithm, when we increase the number of clusters, `Homogeneity score` will increase and the silhouette score will decrease(By getting far away from 2).
Notice that `GMM` clustering method had near results in comparison with `KMeans` method and this can be checked by considering the `silhouette score` plots. But this is so important to say that `GMM` clustering method use covariance matrix despite of `KMeans` and this makes `GMM` more flexible with complex distributed data points that have stretched circle shape like ellipsoids!
Overall, it seems like that dataset C is more dense, leading to slightly more difficult clustering.

References

[1] Jean Decety and David H. Ingvar. Brain structures participating in mental simulation of motor behavior: A neuropsychological interpretation. *Acta Psychologica*, 73(1):13–34, February 1990.

[2] Philip L. Jackson, Martin F. Lafleur, Francine Malouin, Carol Richards, and Julien Doyon. Potential role of mental practice using motor imagery in neurologic rehabilitation. *Archives of Physical Medicine and Rehabilitation*, 82(8):1133–1141, August 2001.

[3] Cornelia Frank, William M. Land, Carmen Popp, and Thomas Schack. Mental representation and mental practice: Experimental investigation on the functional links between motor memory and motor imagery. *PLoS ONE*, 9(4):e95175, April 2014.

[4] Margaret Cocks, Carol-Anne Moulton, Shelly Luu, and Tulin Cil. What surgeons can learn from athletes: Mental practice in sports and surgery. *Journal of Surgical Education*, 71(2):262–269, March 2014.

[5] Omneya Attallah, Jaidaa Abougharbia, Mohamed Tamazin, and Abdelmonem A. Nasser. A bci system based on motor imagery for assisting people with motor deficiencies in the limbs. *Brain Sciences*, 10(11):864, November 2020.

[6] Mostafa Orban, Mahmoud Elsamanty, Kai Guo, Senhao Zhang, and Hongbo Yang. A review of brain activity and eeg-based brain–computer interfaces for rehabilitation application. *Bioengineering*, 9(12):768, December 2022.

[7] M. Lopez-Gordo, D. Sanchez-Morillo, and F. Valle. Dry eeg electrodes. *Sensors*, 14(7):12847–12870, July 2014.

[8] Jyoti Pillai and Michael R. Sperling. Interictal eeg and the diagnosis of epilepsy. *Epilepsia*, 47(s1):14–22, October 2006.

[9] Pawan and Rohtash Dhiman. Machine learning techniques for electroencephalogram based brain-computer interface: A systematic literature review. *Measurement: Sensors*, 28:100823, August 2023.

[10] Alex Suarez-Perez, Gemma Gabriel, Beatriz Rebollo, Xavi Illa, Anton Guimerà-Brunet, Javier Hernández-Ferrer, Maria Teresa Martínez, Rosa Villa, and Maria V. Sanchez-Vives. Quantification of signal-to-noise ratio in cerebral cortex recordings using flexible meas with co-localized platinum black, carbon nanotubes, and gold electrodes. *Frontiers in Neuroscience*, 12, November 2018.

[11] G. Pfurtscheller and C. Neuper. Motor imagery and direct brain-computer communication. *Proceedings of the IEEE*, 89(7):1123–1134, 2001.

[12] Amardeep Singh, Ali Abdul Hussain, Sunil Lal, and Hans W. Guesgen. A comprehensive review on critical issues and possible solutions of motor imagery based electroencephalography brain-computer interface. *Sensors*, 21(6):2173, March 2021.

[13] Dennis J. McFarland, Lynn M. McCane, Stephen V. David, and Jonathan R. Wolpaw. Spatial filter selection for eeg-based communication. *Electroencephalography and Clinical Neurophysiology*, 103(3):386–394, September 1997.

[14] Kip A. Ludwig, Rachel M. Miriani, Nicholas B. Langhals, Michael D. Joseph, David J. Anderson, and Daryl R. Kipke. Using a common average reference to improve cortical neuron recordings from microelectrode arrays. *Journal of Neurophysiology*, 101(3):1679–1689, March 2009.

[15] Harold Hotelling. Relations between two sets of variates. *Biometrika*, 28(3/4):321, December 1936.

[16] Nickolay Trendafilov and Michele Gallo. *Principal component analysis (PCA)*, page 89–139. Springer International Publishing, 2021.

[17] Aapo Hyvärinen, Juha Karhunen, and Erkki Oja. *Independent Component Analysis*. Wiley, May 2001.

[18] David B. Geselowitz. On bioelectric potentials in an inhomogeneous volume conductor. *Biophysical Journal*, 7(1):1–11, January 1967.

[19] J Sarvas. Basic mathematical and electromagnetic concepts of the biomagnetic inverse problem. *Physics in Medicine and Biology*, 32(1):11–22, January 1987.

[20] Olaf Hauk. Keep it simple: a case for using classical minimum norm estimation in the analysis of eeg and meg data. *NeuroImage*, 21(4):1612–1621, April 2004.

[21] Pritom Kumar Saha, Md. Asadur Rahman, and Md. Nurunnabi Mollah. Frequency domain approach in csp based feature extraction for eeg signal classification. In *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*. IEEE, February 2019.

[22] Jyoti Singh Kirar and R.K. Agrawal. Optimal spatio-spectral variable size subbands filter for motor imagery brain computer interface. *Procedia Computer Science*, 84:14–21, 2016.

[23] Quadrianto Novi, Cuntai Guan, Tran Huy Dat, and Ping Xue. Sub-band common spatial pattern (sbcsp) for brain-computer interface. In *2007 3rd International IEEE/EMBS Conference on Neural Engineering*. IEEE, May 2007.

[24] Kai Keng Ang, Zhang Yang Chin, Haihong Zhang, and Cuntai Guan. Filter bank common spatial pattern (fbcsp) in brain-computer interface. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. IEEE, June 2008.