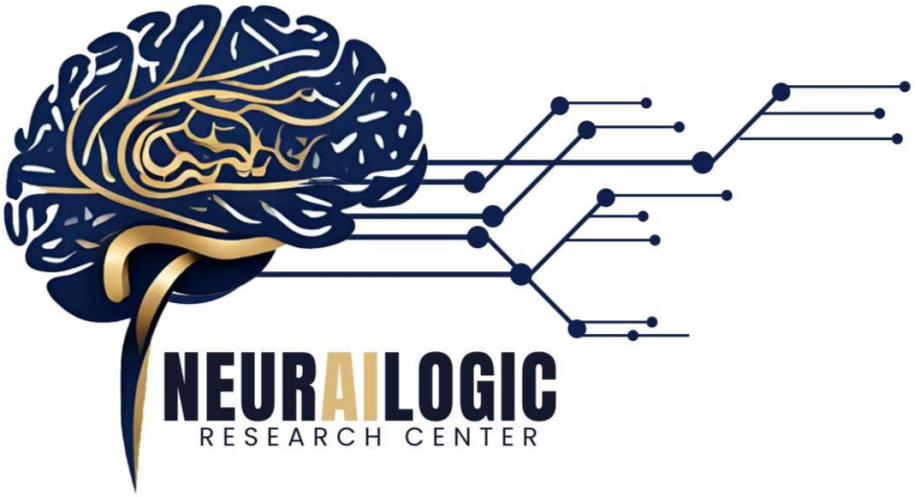

EEG Processing:

An Entry to the World of Brain Waves

Session 6 & 7:

Python – Machine learning & Scikit-learn

Author: Mohammadreza Shahsavari

Contact: mohamadrezashahsavary@gmail.com

What you are going to learn in this session?

This session will provide an overview of machine learning and its application in EEG data processing. We will explore key concepts such as supervised and unsupervised learning, feature extraction techniques, and model selection. You will learn how to extract relevant features and build machine learning models to classify or predict brain states. By the end of this session, you will have a solid understanding of the machine learning pipeline for EEG analysis and be able to apply these techniques to your own research or projects.

- *Mohammadreza Shahsavari*

Table of Contents

1. Introduction to Machine Learning 1

Types of Machine Learning 2

1. Supervised Learning 2
2. Unsupervised Learning 2
3. Reinforcement Learning 2

Two Examples of Machine Learning Applications for EEG Processing 3

1 - EEG-Based Motor Imagery Classification 3

2 - Parkinson's Disease Classification 5

Frequency Bands: 5

2. Feature Extraction from EEG Data 6

Decomposing EEG Signals 7

Bandpass Filtering 7

Spectral Feature Extraction 10

Power Spectral Density (PSD) 10

Band Power Features 12

Applying Band Power Extraction 14

Dimensionality Reduction with Principal Component Analysis (PCA) 14

Applying PCA 17

4. Applying Machine Learning to Extracted EEG Features....18

4.1 Train-Test Splitting 18

4.2 Applying SVM to EEG Data 19

1. Introduction to Machine Learning

What is Machine Learning?

Machine learning (ML) is a branch of artificial intelligence (AI) that focuses on building systems that can learn from data, identify patterns, and make decisions with minimal human intervention. Machine learning algorithms use statistical methods to enable computers to improve their performance on a specific task over time through experience. Fig. 1 illustrates relationships between Artificial Intelligence (AI), Machine Learning (ML), Deep Learning (DL),

In the context of EEG data processing, machine learning can be employed to classify different brain states, detect anomalies, and potentially predict neurological conditions. By learning from labeled EEG data, machine learning models can generalize patterns to new, unseen data, making them powerful tools for various applications in neuroscience and healthcare.

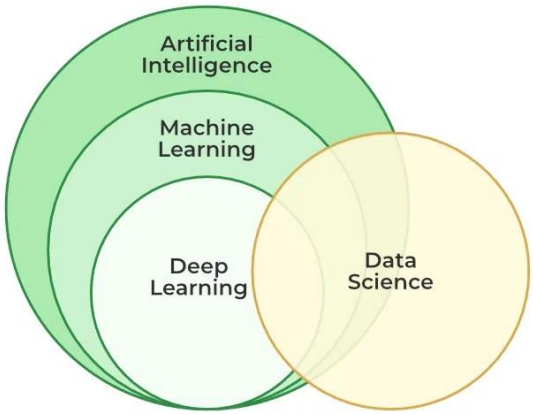

Fig 1. Venn diagram illustrating the relationships between Artificial Intelligence (AI), Machine Learning (ML), Deep Learning (DL), and Data Science (DS). AI encompasses ML, which in turn includes DL. Data Science overlaps with these fields, indicating its integration of techniques from AI, ML, and DL to analyze and interpret complex data.

Types of Machine Learning[1]

Machine learning can be broadly categorized into three main types:

1. **Supervised Learning**
   - **Description**: In supervised learning, the algorithm is trained on a labeled dataset, which means each training example is paired with an output label. The model learns to map input data to the desired output.
   - **Examples**: Classification (e.g., Support Vector Machine, K-Nearest Neighbor), Regression (e.g., Linear Regression).
   - **Applications**: EEG signal classification, predicting disease outcomes.
2. **Unsupervised Learning**
   - **Description**: In unsupervised learning, the algorithm is given data without explicit instructions on what to do with it. The model tries to find hidden patterns or intrinsic structures within the input data.
   - **Examples**: Clustering (e.g., K-Means), Dimensionality Reduction (e.g., Principal Component Analysis).
   - **Applications**: Discovering patterns in EEG data, reducing data dimensionality for visualization.
3. **Reinforcement Learning**
   - **Description**: In reinforcement learning, the model learns by interacting with its environment and receiving rewards or penalties based on its actions. The goal is to learn a policy that maximizes cumulative rewards.
   - **Examples**: Q-Learning, Deep Q-Networks.
   - **Applications**: Adaptive neurofeedback systems, optimizing EEG-based brain-computer interfaces.

[1] Supervised vs Unsupervised vs Reinforcement Learning | Data Science Certification Training | Edureka - YouTube

Some Basic Concepts in Machine Learning

1. **Features and Labels**
    - **Features**: Features are the input variables or attributes used to make predictions. In the context of EEG data, features could include various spectral power values, wavelet coefficients, or other derived metrics from the EEG signals.
    - **Labels**: Labels are the output or target variables that the model aims to predict. For EEG data, labels could represent different mental states, cognitive tasks, or the presence/absence of a neurological condition.
2. **Training and Testing**
    - **Training Set**: The training set is a portion of the dataset used to train the machine learning model. It includes both features and corresponding labels, allowing the model to learn the relationship between inputs and outputs.
    - **Testing Set**: The testing set is a separate portion of the dataset used to evaluate the performance of the trained model. It provides an unbiased assessment of how well the model generalizes to new, unseen data.

Understanding these foundational concepts is crucial for effectively applying machine learning techniques to EEG data processing. In the following sections, we will delve deeper into feature extraction and practical examples to illustrate how these concepts are implemented.

Two Examples of Machine Learning Applications for EEG Processing

1 - EEG-Based Motor Imagery Classification

Motor imagery classification is a fascinating application of EEG (Electroencephalography) technology, prominently used in brain-computer interface (BCI) systems. This process

involves recording brain signals when an individual imagines performing a motor task, such as moving a limb. The recorded EEG signals are then processed to decode the imagined movement and translate it into control commands for various devices.

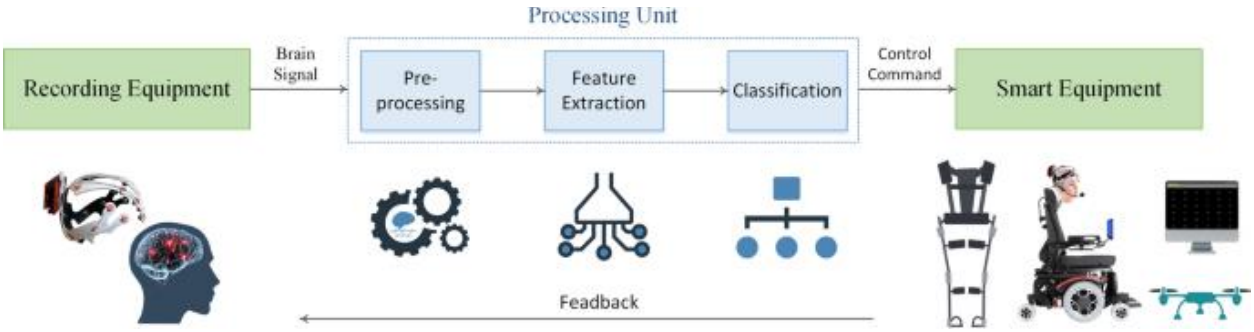

Fig 2.: **EEG-Based Motor Imagery Classification Workflow**
This diagram illustrates the process of classifying motor imagery tasks using EEG signals. It starts with the recording of brain signals, which are then pre-processed to remove noise and artifacts. Significant features are extracted from the cleaned data, which are subsequently classified to determine the imagined motor task. The classification results are used to control smart equipment such as robotic arms, wheelchairs, or computer interfaces, with feedback provided to enhance system accuracy and responsiveness.

The Fig. 2 above illustrates the typical workflow of an EEG-based motor imagery classification system. It begins with the *Recording Equipment* capturing the brain signals. These signals are then transmitted to a *Processing Unit*, where they undergo several stages:

1. **Pre-processing**: This stage involves filtering and cleaning the raw EEG data to remove noise and artifacts, ensuring that the signals are suitable for further analysis.
2. **Feature Extraction**: In this step, significant features that represent the essential characteristics of the EEG signals are extracted. These features could include frequency bands, power spectral densities, or entropies of these frequency bands.
3. **Classification**: The extracted features are then fed into a machine learning model that classifies the type of motor imagery being performed. Common classifiers

used in this context include Support Vector Machines (SVM), Linear Discriminant Analysis (LDA), and deep learning methods.

Finally, the classified commands are sent to the *Smart Equipment*, such as robotic arms, wheelchairs, or computer interfaces, allowing individuals to control these devices through their imagined movements. Feedback is often provided to the user to enhance the system's accuracy and responsiveness.

2 - Parkinson's Disease Classification

Parkinson's disease classification using machine learning is an emerging area of research aimed at improving early diagnosis and treatment planning. Parkinson's disease is a neurodegenerative disorder that affects movement, causing symptoms such as tremors, stiffness, and slowness of movement. Machine learning models can analyze various types of data, including clinical records, gait analysis, and voice recordings, to identify patterns that may indicate the presence of Parkinson's disease.

The process of classifying Parkinson's disease from EEG data follows a similar pipeline to motor imagery classification, involving pre-processing, feature extraction, and classification stages. However, the key distinction lies in the selection of optimal features which are specially affected by Parkinson disease. Previous research has identified several EEG signal characteristics influenced by Parkinson's, including:

Frequency Bands:

- **Beta Band (13-30 Hz):** Decreased power in the beta band is often observed in Parkinson's disease patients. Beta oscillations are associated with motor control, and their reduction may reflect impaired motor function.
- **Alpha Band (8-12 Hz):** Altered alpha band activity, particularly a decrease in alpha power, has been linked to Parkinson's disease. Alpha rhythm disturbances may be related to cognitive decline.

- **Theta Band (4-7 Hz):** An increase in theta band power is sometimes noted in Parkinson's disease. This might be associated with cognitive impairment and decreased alertness.
- **Delta Band (0.5-4 Hz):** Increased delta power may be observed in more advanced stages of Parkinson's disease, reflecting widespread cortical dysfunction.

Functional Connectivity:

Reduced functional connectivity, especially in motor-related regions, is observed in Parkinson's disease. This can be measured using techniques like phase synchronization and coherence analysis.

2. Feature Extraction from EEG Data

Feature extraction is a critical step in processing EEG data before applying machine learning algorithms. It involves transforming raw EEG signals into meaningful features that can effectively represent the data and enhance the performance of machine learning models. This section covers the decomposition of EEG signals, spectral feature extraction, and dimensionality reduction.

To give an example of the feature extraction process, we implemented several feature extraction techniques on EEG signals gathered from the UC San Diego Parkinson Disease EEG dataset. An example of an EEG signal from this dataset is depicted in Fig. 3 below.

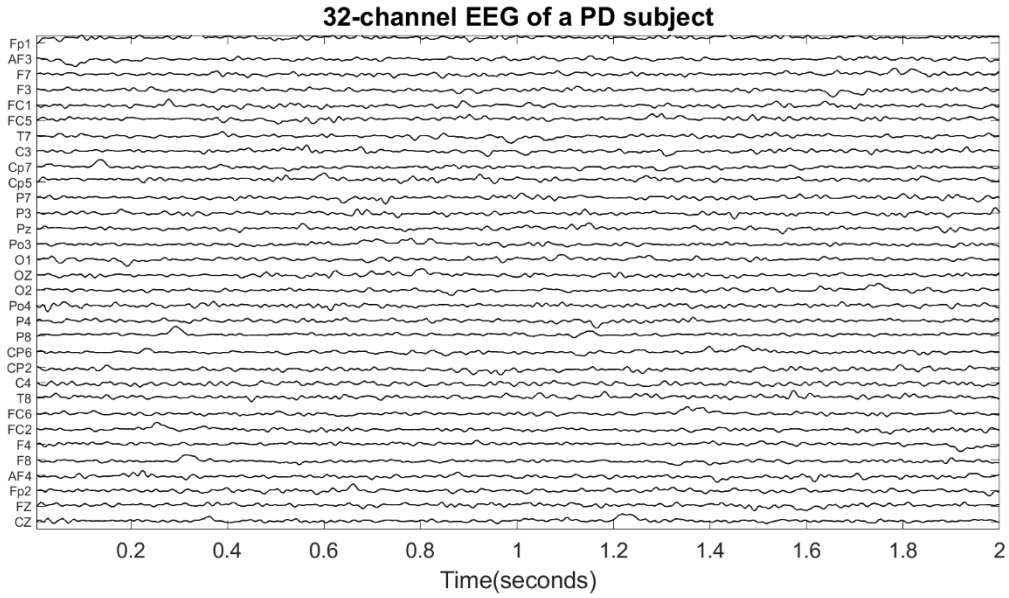

Fig 3. A 2 second segment of an EEG recorded from a Parkinson disease patient.

Decomposing EEG Signals

EEG signals can be decomposed into various frequency bands, each associated with different brain activities:

- **Gamma (30-100 Hz)**: Associated with high-level information processing and cognitive functioning.
- **Beta (12-30 Hz)**: Linked to active thinking, focus, and problem-solving.
- **Alpha (8-12 Hz)**: Associated with relaxation and calmness.
- **Theta (4-8 Hz)**: Linked to drowsiness, meditation, and early sleep stages.
- **Delta (0.5-4 Hz)**: Associated with deep sleep and restorative processes.

Bandpass Filtering

A simpler method for extracting features from EEG signals is bandpass filtering. Bandpass filters allow signals within a specific frequency range to pass through while

attenuating signals outside that range. This method can be used to isolate the desired frequency bands from the raw EEG signals.

Here's how to apply bandpass filtering to extract frequency bands using Python:

```
from scipy.signal import butter, filtfilt

def bandpass_filter(signal, lowcut, highcut, fs, order=4):
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='band')
    filtered_signal = filtfilt(b, a, signal)
    return filtered_signal

# Example usage
fs = 512  # Sampling frequency

# Path to the .mat file
file_path = 'PATH_TO_THE_EEG_FOLDER/s01.mat'
# Load the .mat file
eeg_signal = scipy.io.loadmat(file_path)

###### Here you need some extra lines of codes to access the real EEG data ######
                ### Just consider ‘eeg_signal’ as the real EEG ###

# Apply bandpass filters to extract different frequency bands
delta_signal = bandpass_filter(eeg_signal, 0.5, 4, fs)
theta_signal = bandpass_filter(eeg_signal, 4, 8, fs)
alpha_signal = bandpass_filter(eeg_signal, 8, 12, fs)
beta_signal = bandpass_filter(eeg_signal, 12, 30, fs)
gamma_signal = bandpass_filter(eeg_signal, 30, 100, fs)
```

The code demonstrates how to apply bandpass filtering to extract different frequency bands from an EEG signal. The new syntactic elements introduced are:

- `from scipy.signal import butter, filtfilt`: This imports the butter and `filtfilt` functions from the `scipy.signal` module, which are used to create and apply the bandpass filters.
- `def bandpass_filter(signal, lowcut, highcut, fs, order=4)`: This defines a function for applying a bandpass filter. The parameters are:
    - `signal`: The input EEG signal to be filtered.
    - `lowcut`: The lower cutoff frequency for the bandpass filter.
    - `highcut`: The upper cutoff frequency for the bandpass filter.
    - `fs`: The sampling frequency of the signal.
    - `order`: The order of the filter (default is 4).
- `nyquist = 0.5 * fs`: This calculates the Nyquist frequency, which is half of the sampling frequency.
- `low = lowcut / nyquist` and `high = highcut / nyquist`: These normalize the cutoff frequencies by dividing them by the Nyquist frequency.
- `b, a = butter(order, [low, high], btype='band')`: This creates the bandpass filter coefficients using the butter function. The filter is defined by its order and the normalized cutoff frequencies.
- `filtered_signal = filtfilt(b, a, signal)`: This applies the filter to the input signal using the `filtfilt` function, which performs forward and backward filtering to avoid phase distortion.

After running the above code, we can obtain our EEG's sub-frequency bands, as shown in Fig. 4 below.

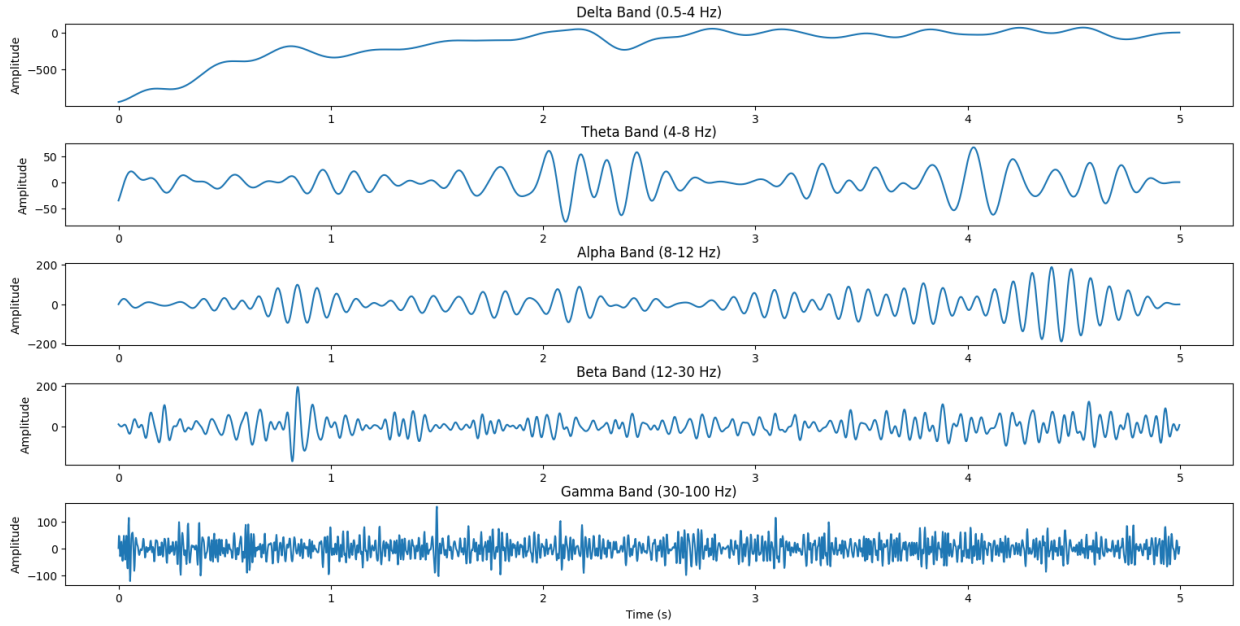

Fig 4.: Sub-frequency bands of an EEG signal. The figure includes the Delta (0.5-4 Hz), Theta (4-8 Hz), Alpha (8-12 Hz), Beta (12-30 Hz), and Gamma (30-100 Hz) bands. Each subplot illustrates the amplitude variations over the sample indices for a specific frequency range, highlighting different aspects of brain activity.

Each of these sub-frequency bands can hold different brain information, making them useful for EEG feature extraction. By analyzing the power of each sub-band, we can determine how active a particular frequency range is, providing insights into various mental and physiological states.

Spectral Feature Extraction

Power Spectral Density (PSD)

Power Spectral Density (PSD) represents the power distribution of a signal over different frequency components. It quantifies how the power of a signal is distributed across various frequencies, providing insights into the underlying frequency components. PSD is important as an EEG feature in machine learning because it helps in identifying and characterizing different brain states, which can be crucial for applications.

Here's how to calculate PSD using Welch's method:

```
from scipy.signal import welch

def calculate_psd(signal, fs=512, nperseg=256):
    freqs, psd = welch(signal, fs=fs, nperseg=nperseg)
    return freqs, psd

# Example usage
freqs, psd = calculate_psd(eeg_signal, fs=fs)
```

The code demonstrates how to calculate the Power Spectral Density (PSD) of an EEG signal using Welch's method. The new syntactic elements introduced are:

• `from scipy.signal import welch`: This imports the welch function from the `scipy.signal` module, which is used to compute the PSD.

• `def calculate_psd(signal, fs=256, nperseg=256)`: This defines a function for calculating the PSD. The parameters are:

- `signal`: The input EEG signal.
- `fs`: The sampling frequency of the signal (default is 256 Hz).
- `nperseg`: The length of each segment for Welch's method (default is 256).

• `freqs, psd = welch(signal, fs=fs, nperseg=nperseg)`: This calculates the PSD using Welch's method. The welch function returns two arrays:

- `freqs`: The array of sample frequencies.
- `psd`: The Power Spectral Density of the signal.

This PSD plot of the EEG segment which is showing in Fig. 5 reveals:

1. A dominant peak around 2-4 Hz, indicating significant delta wave activity.

2. A smaller but notable peak around 8-10 Hz, suggesting the presence of alpha waves.
3. Minimal power beyond 15 Hz, implying limited high-frequency activity in this EEG segment.

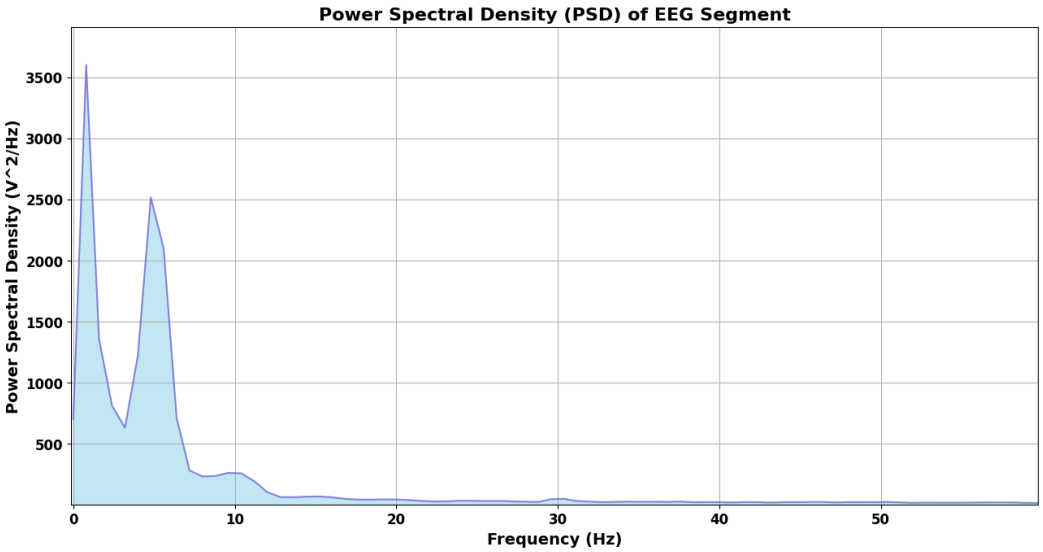

Fig 5 Power Spectral Density (PSD) plot of an EEG segment, highlighting dominant delta wave activity around 2-4 Hz and alpha wave presence around 8-10 Hz.

Band Power Features

Band power features represent the power within specific frequency bands (e.g., Delta, Theta, Alpha, Beta, Gamma). These features are useful for analyzing and characterizing EEG signals, as different brain activities are often associated with specific frequency bands.

Here's how to extract band power features from the PSD:

```
def extract_band_power(psd, freqs, band):
    band_power = np.sum(psd[(freqs >= band[0]) & (freqs < band[1])])
    return band_power

# Example usage
delta_band = (0.5, 4)
theta_band = (4, 8)
alpha_band = (8, 12)
beta_band = (12, 30)
gamma_band = (30, 100)

delta_power = extract_band_power(psd, freqs, delta_band)
theta_power = extract_band_power(psd, freqs, theta_band)
alpha_power = extract_band_power(psd, freqs, alpha_band)
beta_power = extract_band_power(psd, freqs, beta_band)
gamma_power = extract_band_power(psd, freqs, gamma_band)
```

The code calculates the power in the Delta, Theta, Alpha, Beta, and Gamma bands by summing the PSD values within the specified frequency ranges. The new syntactic elements introduced are:

• `def extract_band_power(psd, freqs, band)`: This defines a function for extracting band power features. The parameters are:

- `psd`: The Power Spectral Density array.
- `freqs`: The array of sample frequencies corresponding to the PSD values.
- `band`: A tuple specifying the frequency range for the band (e.g., `(0.5, 4)` for the Delta band).

• `band_power = np.sum(psd[(freqs >= band[0]) & (freqs < band[1])])`: This calculates the band power by summing the PSD values within the specified frequency range. The condition `(freqs >= band[0]) & (freqs < band[1])` is used to select the PSD values within the band.

Applying Band Power Extraction

The example code extracts the power for different EEG frequency bands:

- **Delta Band (0.5-4 Hz)**: `delta_power = extract_band_power(psd, freqs, delta_band)`
- **Theta Band (4-8 Hz)**: `theta_power = extract_band_power(psd, freqs, theta_band)`
- **Alpha Band (8-12 Hz)**: `alpha_power = extract_band_power(psd, freqs, alpha_band)`
- **Beta Band (12-30 Hz)**: `beta_power = extract_band_power(psd, freqs, beta_band)`
- **Gamma Band (30-100 Hz)**: `gamma_power = extract_band_power(psd, freqs, gamma_band)`

By calculating the power within these specific frequency bands, the band power features can be used in machine learning models to classify different brain states, detect anomalies, or identify patterns in the EEG data. These features provide valuable insights into the frequency-specific activity of the brain.

After extracting band power features for our EEG signal, the code will output:

- Delta band power: 2309.896240234375
- Theta band power: 3680.61328125
- Alpha band power: 739.017578125
- Beta band power: 746.1459350585938
- Gamma band power: 1282.8470458984375

Dimensionality Reduction with Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms data into a new coordinate system. It aims to maximize variance along the new axes (principal components) and make the features mutually uncorrelated.

Fig. 6 illustrates the concept of Principal Component Analysis (PCA), a statistical technique used to reduce the dimensionality of a dataset while retaining most of the variation present in it. Here's a detailed breakdown of the image:

1. **Scatter Plot of Data Points**:
    - The blue circles represent the original data points in a two-dimensional space.
2. **Principal Components**:
    - The green arrows labeled "PCA 1st Dimension" and "PCA 2nd Dimension" represent the principal components (PCs).
    - **PCA 1st Dimension**: The longest green arrow shows the direction of the greatest variance in the data, representing the first principal component.
    - **PCA 2nd Dimension**: The shorter green arrow is orthogonal (perpendicular) to the first principal component, representing the second greatest variance in the data.
3. **Dimensionality Reduction**:
    - PCA transforms the data to a new coordinate system where the greatest variances are aligned with the principal components.
    - By projecting the data onto the principal components, PCA reduces the dimensionality while preserving as much variability as possible.

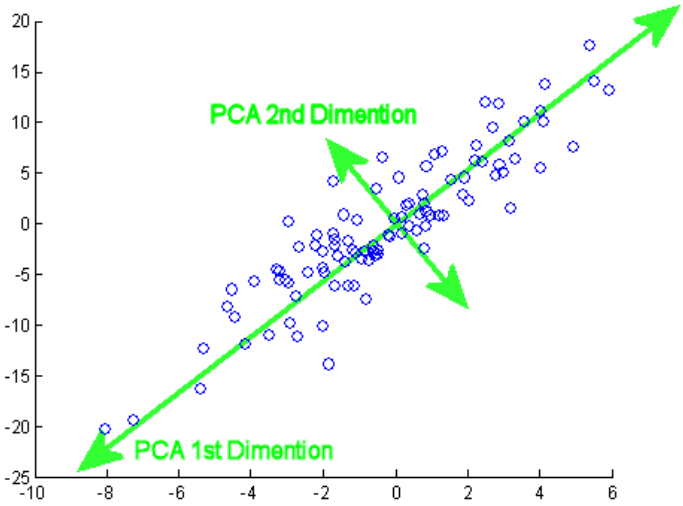

Fig 6. Scatter plot showing the application of Principal Component Analysis (PCA) on a two-dimensional dataset. The green arrows represent the principal components: the 1st principal component (PCA 1st Dimension) aligns with the direction of the greatest variance, and the 2nd principal component (PCA 2nd Dimension) is orthogonal to it.

Here's how to apply PCA using Scikit-Learn:

```
from sklearn.decomposition import PCA

def apply_pca(features, n_components=None):
    pca = PCA(n_components=n_components)
    transformed_features = pca.fit_transform(features)
    return transformed_features, pca

# Example usage
features = np.array([[delta_power, theta_power, alpha_power, beta_power,
gamma_power]])  # Replace with actual feature matrix
transformed_features, pca = apply_pca(features, n_components=5)
```

The code demonstrates how to apply PCA to a set of features. The new syntactic elements introduced are:

- `from sklearn.decomposition import PCA`: This imports the PCA class from the Scikit-Learn library, which is used to perform Principal Component Analysis.
- `def apply_pca(features, n_components=None)`: This defines a function for applying PCA. The parameters are:
    - `features`: The input feature matrix.

    - `n_components`: The number of principal components to keep (default is None, which keeps all components).
- `pca = PCA(n_components=n_components)`: This creates a PCA object with the specified number of components.
- `transformed_features = pca.fit_transform(features)`: This fits the PCA model to the data and transforms the features into principal components. The transformed features are returned along with the PCA object.

Applying PCA

The example code applies PCA to a set of features:

- **`features = np.array([[delta_power, theta_power, alpha_power, beta_power, gamma_power]])`:** This creates a feature matrix using the band power features. Replace this with the actual feature matrix from your dataset.
- **`transformed_features, pca = apply_pca(features, n_components=5)`:** This applies PCA to the feature matrix, keeping 5 principal components. The transformed features and the PCA object are returned.

Making Features Uncorrelated

PCA transforms the original features into new features (principal components) that are orthogonal, meaning they are uncorrelated. This transformation helps improve the performance of machine learning models by reducing redundancy and multicollinearity. The principal components capture the maximum variance in the data along new axes, making the data more suitable for various machine learning algorithms.

By applying PCA, you can effectively reduce the dimensionality of your feature space, making your machine learning models more efficient and potentially improving their

performance. This technique is especially useful when dealing with high-dimensional data, as it simplifies the dataset while retaining most of the important information.

4. Applying Machine Learning to Extracted EEG Features

4.1 Train-Test Splitting

Before applying machine learning algorithms like Support Vector Machines (SVM), it's crucial to split your dataset into two parts: a **training set** and a **testing set**. This step is important because it helps you evaluate how well your model will perform on new, unseen data.

Why Split the Data?

When you train a model, it learns from the data you provide. However, to truly understand how well the model will perform on new data, you need to test it on data it hasn't seen before. This is where the train-test split comes in:

- **Training Set**: This is the portion of your data that the model uses to learn patterns. Typically, 80% of the data is used for training.
- **Testing Set**: The remaining 20% of the data is used to test the model's performance. This set acts like new data to the model, helping you see how well it generalizes beyond what it has learned.

By doing this, you can get a better estimate of the model's accuracy and avoid overfitting, where the model performs well on the training data but poorly on new data.

Example of Train-Test Splitting

In your EEG data processing, you've already extracted features like band powers. These features are stored in the `features` variable, and the corresponding labels (e.g., different mental states) are in the `labels` variable.

Here's how you can split the data into training and testing sets using Python:

```
from sklearn.model_selection import train_test_split

# Perform the train-test split
X_train, X_test, y_train, y_test = train_test_split(features, labels,
test_size=0.2, random_state=42)
```

- `features`: This variable contains the extracted features from your EEG data, like band powers.
- `labels`: This variable contains the class labels corresponding to each set of features.
- `train_test_split`: This function from Scikit-learn automatically splits your data into training and testing sets.
- `test_size=0.2`: This parameter specifies that 20% of the data will be used for testing, while the remaining 80% will be used for training.
- `random_state=42`: This is a seed value that ensures the split is reproducible, meaning you'll get the same split each time you run the code.

By splitting your data this way, you create two sets: one for teaching the model (training) and one for evaluating it (testing), which helps ensure the model performs well on new data.

4.2 Applying SVM to EEG Data

After splitting your data into training and testing sets, the next step is to apply a machine learning algorithm to classify the EEG features. In this case, we'll use the Support Vector Machine (SVM), a popular and powerful tool for classification tasks.

What is SVM?

Support Vector Machine (SVM) is a supervised machine learning algorithm that is widely used for classification. SVM works by finding the best boundary, or *hyperplane*, that

separates different classes in the data. The goal is to choose a hyperplane that maximizes the margin between the classes, ensuring that the model can differentiate between them as clearly as possible.

Applying SVM to EEG Features

In your EEG analysis, you've already extracted features (like band powers) that capture important aspects of the brain signals. Now, you'll use these features to train the SVM model to classify the EEG data.

Here's how to do it using Python:

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Initialize the SVM model with a linear kernel
svm_model = SVC(kernel='linear')

# Train the model on the training data
svm_model.fit(X_train, y_train)

# Predict the labels for the test set
y_pred = svm_model.predict(X_test)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

Here is what this code dose:

1. **Model Initialization**:

   - `svm_model = SVC(kernel='linear')`: This line initializes an SVM model with a linear kernel, which is ideal for data that can be separated with a straight line.

2. **Model Training**:

   - `svm_model.fit(X_train, y_train)`: The fit method trains the SVM model on the training data (`X_train`), using the corresponding labels (`y_train`) to learn the classification boundaries.

3. **Prediction**:

   - `y_pred = svm_model.predict(X_test)`: The `predict` method uses the trained model to predict the labels for the test data (`X_test`), storing the results in `y_pred`.

4. **Evaluation**:

   - `accuracy = accuracy_score(y_test, y_pred)`: The `accuracy_score` function compares the predicted labels (`y_pred`) to the actual labels (`y_test`) and calculates the model's accuracy.
   - `print(f"Accuracy: {accuracy * 100:.2f}%")`: This line prints the accuracy as a percentage, providing a measure of how well the model performed.