22AIE438
Biomedical Signal Processing

A THESIS
Submitted by
Group – 12

Lokesh. K	– CB.EN.U4AIE220327
Tharun Balaji. P	– CB.EN.U4AIE22040
Naga koushik. S	– CB.EN.U4AIE22046
Bhavya Sainath. U	– CB.EN.U4AIE22055

In partial fulfilment for the award of the degree of
BACHELOR OF TECHNOLOGYIN
CSE(AI)

Topic: Stress Level Prediction with EEG Signal Using Deep Learning

श्रद्धावान् लभते ज्ञानम्
AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY UNDER SECTION 3 OF THE UGC ACT, 1956

Centre for Computational Engineering and NetworkingAMRITA SCHOOL OF ARTIFICAL INTELLIGENCEAMRITA VISHWA VIDYAPEETAM

COIMBATORE – 641112(INDIA)

November – 2024

AMRITA SCHOOL OF ARTIFICAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETAM

COIMBATORE – 641112(INDIA)

BONAFIDE CERTIFICATE

This is to certify that the thesis entitled "**Stress level Prediction using EEG Signal**" submitted by Lokesh. K (CB.EN.U4AIE22027),Tharun Balaji.P(CB.EN.U4AIE22040), Naga koushik. S (CB.EN.U4AIE22046), Bhavya Sainath. U (CB.EN.U4AIE22055) for the award of the Degree of Bachelor of Technology in the CSE(AI) is a Bonafide record of the work carried out bythem our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Dr. Amrutha V	**Dr. K. P. Soman**
Project Guide	Professor and Head CEN

AMRITA SCHOOL OF ARTIFICAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETAM

COIMBATORE – 641112(INDIA)

DECLARATION

We, Lokesh. K (CB.EN.U4AIE22027),Tharun Balaji. P (CB.EN.U4AIE22040), Naga koushik. S (CB.EN.U4AIE22046), Bhavya Sainath. U (CB.EN.U4AIE22055) hereby declare that this thesis entitled "**Stress level Prediction using EEG Signal**" is the record of the original work done by us under the guidance of Dr Amrutha V, Centre for Computational Engineering and Networking, Amrita School of Artificial Intelligence, Coimbatore. To the best of my knowledge this work has not formed the basis for the award of any degree/diploma/ associate ship/fellowship/or a similar award to any candidate in any University.

Place: Coimbatore
Date: 16-11-2024

Signature of the Students

TABLE OF CONTENTS

TABLE OF FIGURES:

ABSTRACT:

In recent years, stress has emerged as a critical concern in mental health, affecting individuals' well-being and productivity. Stress level classification using EEG (electroencephalogram) signals is pivotal for advancing mental health technology, offering non-invasive and objective methods for assessing stress. This project addresses the problem by developing a deep learning-based classification model that categorizes stress levels into 10 distinct classes based on EEG data. The system integrates two advanced neural network architectures: a Convolutional Neural Network (CNN) for extracting spatial features from EEG signals and a Long Short-Term Memory (LSTM) network for capturing temporal dependencies.

The EEG data undergoes rigorous preprocessing, employing techniques such as Bandpass Filtering, Independent Component Analysis (ICA), and Wavelet Denoising to ensure high-quality inputs. Key features such as frequency bands, power spectral density, and nonlinear dynamics are extracted to enhance model performance, and then we further used Traditional Machine learning models like Random forest and SVM and Deep learning model like Multilayer PerceptronThe system is deployed with a user-friendly interface powered by Streamlit, enabling real-time stress classification.. This work significantly contributes to the field of mental health by combining advanced signal processing, deep learning, and user-centric design to create an innovative solution for stress management and prediction.

PURPOSE OF THIS PROJECT:

The purpose of our project is to create an advanced and reliable system for stress level prediction using EEG signals, leveraging deep learning techniques to promote mental health awareness and management. Specifically, the project aims to:

- Provide a non-invasive, data-driven approach to assess stress levels objectively, overcoming the limitations of subjective self-reports.
- Develop a real-time stress prediction system that offers immediate feedback to users based on EEG data, enabling timely interventions.
- Utilize state-of-the-art neural network architectures, including CNN and LSTM, to ensure precise classification of stress levels by capturing both spatial and temporal patterns in EEG signals.

INTRODUCTION:

LITERATURE REVIEW:

Title	Authors	Description	Remarks
Stress Detection Using EEG Signals	M Taghaddossi[1], P Moradi[2] and M H Moradi[1]	Provides an overview of various methods for detecting stress using EEG signals, focusing on traditional signal processing techniques.	Establishes foundational methods in EEG signal processing, critical for understanding subsequent advancements.
Advanced Feature Extraction and Signal Processing for EEG-based Stress Detection	Alexandre Gramfort [a b c d e], Martin Luessi [b c], Eric Larson [f], Denis A. Engemann [g h], Dan iel Strohmeier [i], Christ ian Brodbeck [j], Lauri Parkkonen [k l], Matti S. Hämäläinen [b c])	Explores advanced feature extraction and signal processing for EEG-based stress detection.	Highlights the progression from basic signal processing to more advanced techniques, setting the stage for deep learning applications.
Signal Processing Techniques for EEG Signal Enhancement	Alexandre Gramfort [a b c d e], Martin Luessi [b c], Eric Larson [f], Denis A. Engemann [g h], Dan iel Strohmeier [i], Christ ian Brodbeck [j], Lauri Parkkonen [k l], Matti S. Hämäläinen [b c])	Investigates signal processing techniques for EEG signal enhancement.	Focuses on the enhancement of EEG signals, crucial for accurate stress level prediction.
Deep Learning for Analyzing EEG Data in Stress Detection	Dorota Kamińska, Krzysztof Smółka and Grzegorz Zwoliński	Covers the use of deep learning for analyzing EEG data in stress detection.	Focuses on deep learning applications, specifically relevant to the regression models in your project.
EEG Signal Classification for Detecting Stress	Xiyuan Hou; Yisi Liu; Olga Sourina; Yun Rui Eileen Tan; Lipo Wang; Wolfgang Mueller-Wittig	Focuses on EEG signal classification for detecting stress.	Provides insights into the classification approaches, which are foundational before moving into continuous prediction.
Deep Learning for Stress Detection Using EEG Signals	Yara Badr, Usman Tariq, Fares Al-Shargie, Fabio Babiloni, Fadwa Al Mughairbi & Hasan Al-Nashash	Explores deep learning models for stress detection using EEG signals.	Discusses deep learning specifically for stress detection, relevant for contextualizing your regression-based approach.

ABOUT THE TOPIC:

Stress is a global concern, significantly impacting physical and mental health. Accurate measurement of stress levels is essential for devising effective interventions. EEG (Electroencephalogram) signals, which capture the brain's electrical activity through non-invasive methods, provide a reliable medium to analyze stress. By examining changes in specific EEG frequency bands such as alpha, beta, and theta, it is possible to identify stress markers and differentiate stress levels.

Deep learning techniques, particularly Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, are powerful tools for processing EEG data. CNNs are adept at extracting spatial features, while LSTMs excel in capturing temporal dependencies, making them ideal for modeling the dynamic nature of EEG signals. This advanced framework predicts stress levels on a continuous scale, offering a granular understanding of mental states.

The project has significant real-world applications, including real-time stress monitoring systems for workplaces, healthcare settings, and personal use. It can help organizations manage employee well-being, enable clinicians to assess stress in patients more effectively, and empower individuals with tools to understand and regulate their stress. By advancing the integration of neuroscience and AI, this work contributes to creating a more stress-resilient society and enhancing mental health

PROBLEM IN THIS AREA:

- **Noise and Artifacts in EEG Signals**
  EEG signals are highly sensitive to external and internal noise, such as muscle movements, eye blinks, and power-line interference, which can obscure the data and reduce model accuracy.

- **Difficulty in Real-Time Implementation**
  Real-time stress prediction requires low-latency processing and efficient handling of streaming EEG data, which is computationally demanding.

- **Interpretability of Deep Learning Models**
  Deep learning models are often considered "black boxes," making it difficult to understand how predictions are made and verify their reliability.

HOW THE PROBLEMS ARE TACKLED:

- Advanced signal processing techniques such as Independent Component Analysis (ICA), Band-Pass Filtering, and Wavelet Denoising are used to clean the data. Adaptive filtering methods are also employed to dynamically reduce noise and enhance signal quality.

- Optimized neural network architectures, such as lightweight CNNs and LSTMs, are used for faster inference. Cloud computing platforms and edge AI devices enable real-time processing and stress monitoring.

- Explainable AI (XAI) techniques, such as attention mechanisms and saliency maps, are integrated into the models to provide insights into the decision-making process and highlight the most critical features contributing to predictions.

Preprocessing Steps used:

EEG Dataset Collection:

- The dataset used for this project is the EEG Dataset for Cognitive Load and Stress Assessment During Task Performance, available at this link. This dataset is sourced from a study focused on cognitive load and stress assessment, featuring:
- Experimental Tasks: Participants performed tasks such as the Stroop Color-Word test, arithmetic problem-solving, and mirror image recognition, designed to induce varying stress levels.
- Stress Ratings: After completing each task, participants rated their stress levels on a scale of 1-10
- EEG Recordings: EEG signals were recorded from 32 channels for each subject, providing high-resolution insights into brain activity.
  This dataset forms the foundation of the project, enabling detailed analysis and modeling of stress levels using EEG data.

Noise Removal:

To ensure cleaner and more reliable inputs, advanced filtering techniques are applied to mitigate noise and artifacts:

Butterworth Bandpass Filter:

Removes unwanted frequencies outside the 1–50 Hz range.

Key Characteristics of the Butterworth Filter

- **Maximally Flat Frequency Response**: Unlike other filters that may have ripples in the passband or stopband, the Butterworth filter has a smooth frequency response, making it ideal for applications where a flat response is desired.
- **Frequency Range Control**: By adjusting the cutoff frequencies, you can control the range of frequencies that are allowed to pass, which is crucial for isolating specific EEG frequency bands like delta, theta, alpha, etc.

  Butterworth Filter Transfer Function

  - The transfer function H(s) of an n-order Butterworth filter is defined as:

$$H(s) = \frac{1}{\sqrt{1 + \left(\frac{s}{w_c}\right)^{2n}}}$$

Where:

- s is the complex frequency variable, $s = j.\omega$ as the angular frequency and j as the imaginary unit),
- w_c is the cutoff frequency,
- n is the filter order.

For a **bandpass filter**, you design it with two cutoff frequencies:

- **Lower cutoff frequency** w_L : The lower edge of the frequency band.
- **Upper cutoff frequency** w_U : The upper edge of the frequency band.

- Ensures a smooth and distortion-free frequency response.

Savitzky-Golay Filter:

- Reduces high-frequency noise by smoothing the signal while preserving temporal details.

Key Characteristics of the Savitzky-Golay Filter

- **Preservation of Features**: Unlike standard moving average filters, which can distort the signal, the Savitzky-Golay filter fits a polynomial to the data points within a specified window, preserving the original shape and features of the data.
- **Adjustable Parameters**: The filter's performance can be adjusted by changing the polynomial order and the window size, allowing for flexible smoothing depending on the signal characteristics.

- **Mathematical Representation**
- Given a time series data set y at points t_i, the smoothed value y(t_i) at point t_i is calculated as follows:

$$y(t_i)=\sum_{j=-p}^{p} c_j y\left(t_{i+j}\right)$$

Where:

- y(t_i) is the smoothed value at time t_i.
- c_j are the Savitzky-Golay filter coefficients for the polynomial.
- p is the number of points on either side of used for smoothing.
- $y\left(t_{i+j}\right)$ is the original data value at the corresponding time point.

- Configured with a window size of 21 and polynomial order of 3 for optimal smoothing.

Independent Component Analysis (ICA):

- Identifies and removes physiological artifacts such as eye blinks and muscle movements.
- Mathematical Formulation of ICA
- Suppose we have a set of observed signals represented as a matrix X:

$$X = AS$$

Where:

- X is the observed signal matrix (e.g., EEG data).
- A is the mixing matrix, which describes how the sources are mixed to form the observed signals.
- S is the matrix of source signals (the independent components we want to recover).

- Decomposes signals into 15 components to isolate noise sources effectively.

We chose the **Butterworth Band-Pass filter** for its ability to isolate EEG activity (1-50 Hz) with minimal distortion, capturing essential neural rhythms. The **Savitzky-Golay filter** provides smoothing while preserving temporal details critical for EEG analysis. Finally, **ICA** is used to remove artifacts like eye blinks, improving the accuracy of neural signal interpretation.

Feature Extraction

Key features are extracted from the processed EEG signals to represent stress-related brain activity accurately:

Time-Series Features:

- Variance, RMS (Root Mean Square), and Peak-to-Peak Amplitude.
- **Variance**: Measures the spread of the data, indicating the signal's variability.

$$Variance = \frac{1}{N}\sum_{i=1}^{N}(xi-\mu)^2$$

- **Root Mean Square (RMS)**: Represents the magnitude of the signal and is useful for understanding the signal's overall power.

$$RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i^2)}$$

- **Peak-to-Peak Amplitude**: Measures the difference between the maximum and minimum values in a signal.

$$Peak-to-Peak\ Amplitude = \max(x) - \min(x)$$

- Captures signal variability, magnitude, and overall power.

Frequency Band Features:

- Power Spectral Density (PSD) analysis for delta, theta, alpha, beta, and gamma bands.
- These features involve **Power Spectral Density (PSD)** analysis to decompose the signal into frequency components.
- The most common frequency bands are:
  - Delta (0.5–4 Hz): Deep Sleep
  - Theta (4–8 Hz): Drowsiness, relaxation
  - Alpha (8–13 Hz): Calm, wakeful rest
  - Beta (13–30 Hz): Alert, focused
  - Gamma (30–50 Hz): High-level cognition, problem-solving
- **Power Spectral Density (PSD)** is used to compute the power in each frequency band. It is computed as follows:

$$PSD = \frac{1}{N}\sum_{i=1}^{N}|X(f_i)|^2$$

- These features are helpful for studying rhythmic brain activity and identifying patterns
- Identifies patterns linked to specific mental states.

Hjorth Parameters:

- **Mobility:** Indicates the signal's frequency content variability.
  - Reflects the signal's frequency content, indicating how much the frequency changes over time.It is computed as the ratio of the standard deviation of the first derivative of the sign

$$Mobility = \frac{\sqrt{Var\left(\frac{dx(t)}{dt}\right)}}{Var(x(t))}$$

- **Complexity:** Measures the signal's structural intricacy.
  - Measures how complicated the signal's shape is. Higher complexity means the signal has a more intricate pattern, while lower complexity indicates simpler, more regular patterns.

$$Complexity = \frac{\sqrt{Var\left(\frac{d^2x(t)}{dt^2}\right)}}{Mobility.Var(x(t))}$$

Fractal Features:

- **Higuchi Fractal Dimension:** Quantifies signal complexity based on its self-similarity.
- : Measures complexity by analyzing how the signal behaves when split into smaller segments. It looks at the rescaled range of the signal at different intervals.

$$D_H = \lim_{\tau \to 0} \frac{1}{N} \ln\left(\sum_{i=1}^{N-\tau} \left(\frac{|x(i) - x(i+\tau)|}{\tau}\right)^2\right)$$

- **Katz Fractal Dimension:** Estimates complexity by analyzing signal stretching.
- Estimates complexity by comparing the length of the signal to the distance between its points. It calculates how much the signal stretches within its enclosing box.

$$D_K = \frac{\ln\left(\frac{N}{\sum_{i=1}^{N-1}|x(i) - x(i+\tau)|}\right)}{\ln(N)}$$

Entropy Features:

Entropy is a measure of unpredictability or randomness in the signal. Several types of entropy are commonly used for EEG signal analysis:

- **Approximate Entropy (ApEn)**: Measures the regularity and complexity of time series data:
- **Sample Entropy (SampEn)**: Similar to ApEn but avoids self-matching, making it more robust
- **Spectral Entropy**: Measures the entropy of the power spectral density (PSD) of the signal, quantifying its spectral complexity.
- **SVD Entropy**: Measures the entropy of the singular value decomposition (SVD) of the signal.

MODEL'S ARCHITECTURE:

Convolutional Neural Network(CNN):

Overview of CNN:

Convolutional Neural Networks (CNNs) are a class of deep learning models that are widely used for analyzing visual data. However, they are also effective for sequential data, such as EEG signals. CNNs work by extracting local patterns or features from the data using a series of convolutional layers. These features are progressively learned to represent higher-level patterns in deeper layers.

Working Principle of CNN:

Convolutional Layer (Conv1D):

- This layer applies a set of filters (kernels) to the input data to detect features such as edges, textures, or frequencies. The filter slides over the input data to extract spatial relationships between the input signal and the weights of the filter.
- The convolution operation involves a dot product between the filter and the input, followed by adding a bias term.
  Formula: For a 1D convolution:

$$\boldsymbol{y(t) = \sum_{i=0}^{k-1} x(t+i).w(i) + b}$$

Where:

- x(t) is the input signal at time t,
- w(i) is the filter weight at position i,
- b is the bias term.

Activation Function (ReLU):

- After the convolution operation, the output is passed through an activation function to introduce non-linearity to the model. In our case, we use the **ReLU (Rectified Linear Unit)** function, which replaces negative values with zero:

$$ReLU(x) = \max(0, x)$$

Batch Normalization:

- This layer normalizes the activations of the previous layer by scaling and shifting them. It helps improve convergence speed, reduces overfitting, and stabilizes the learning process.

Max Pooling (MaxPooling1D):

- Pooling layers reduce the dimensionality of the data by selecting the maximum value within a defined region. This helps reduce the computational cost and retains the most important features.
- **Formula for Max Pooling:**

$$MaxPooling(x) = \max(x_1, x_2, x_3, \dots x_n)$$

Where $x_1, x_2, x_3, \dots x_n$ are values in the pooling window.

Flatten:

- After applying convolution and pooling, the resulting feature maps are flattened into a 1D array to be passed to fully connected (dense) layers.

Fully Connected (Dense) Layers:

- These layers perform high-level reasoning and are responsible for making predictions based on the features extracted by the convolutional layers.

Dropout:

- Dropout is used to prevent overfitting by randomly setting a fraction of input units to zero during training. It forces the network to learn more robust features.

Output Layer (Softmax):

- The final layer uses the softmax activation function for multi-class classification. It converts the raw output of the model into a probability distribution over all classes.

Layers used:

Input Layer:

- The input is reshaped to (512, 1), where 512 represents the number of time steps, and 1 represents a single feature channel. This format is required for Conv1D to process the sequential data.

Conv1D Layer 1:

- The first convolution layer has 32 filters, each of size 3. ReLU activation is applied to the output of this layer. This helps extract low-level features from the input EEG signal.
- **Input shape:** (512, 1)
- **Filter size:** 3
- **Activation:** ReLU

BatchNormalization:

- Normalizes the output of the first convolutional layer to improve model training stability.

MaxPooling1D:

- After convolution and batch normalization, the feature maps are pooled with a pool size of 2, reducing the spatial dimensions by half.

Conv1D Layer 2:

- The second convolution layer has 64 filters of size 3. Again, ReLU activation is used, followed by batch normalization and max pooling.

Conv1D Layer 3:

- The third convolution layer has 128 filters, continuing the process of extracting higher-level features.

Flatten:

- Flattens the pooled feature maps into a 1D array to feed into the fully connected layers.

Dense Layers:

- Several fully connected layers are added with ReLU activation. These layers perform high-level reasoning based on the extracted features.
- Dropout is applied after each dense layer to prevent overfitting.

Output Layer:

- A final dense layer with 10 neurons is used to classify the input into one of 10 categories. The softmax activation function is used for multi-class classification.

LSTM – Long Short-Term Memory:

LSTM is a type of recurrent neural network (RNN) designed to handle sequential data and learn long-term dependencies. Unlike traditional RNNs, LSTMs are better at remembering information over long sequences

Forget Gate: $ft = \sigma(Wf.[h_{t-1,}x_t] + b_f$

Input Gate:
$i_t = \sigma(Wi \, . [h_{t-1,}x_t] + b_i$

$Ct = tanh(Wc \, . [h_{t-1,}x_t] + b_c$

Cell state Update: $Ct = f_t * C_{t-1} + i_t * C_t$

Output Gate: $Ot = \sigma(W_o \, . [h_{t-1,}x_t] + b_o$

$h_t = o_t * \tanh(C_t)$

HOW DOES IT WORK:

Memory Cells and Gates: An LSTM has cells that act as memory units, along with gates that control the flow of information. The three main gates are:

1. Forget Gate: Decides what information to discard from the cell state.
2. Input Gate: Decides what new information to store in the cell state.
3. Output Gate: Decides what part of the cell state to output as the next hidden state.

This structure allows LSTM to retain important information over many time steps, which is crucial for understanding the sequence of video frames.

IMPLEMENTATON IN CODE:

LSTM layer will be placed after the **CNN** layers. The output of the CNN layers will be passed to the LSTM layer, which will capture the temporal dependencies within the extracted features.

```
model = Sequential([
    # 1st Convolutional Layer
    Conv1D(32, kernel_size=3, activation='relu', input_shape=(512, 1)),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    # 2nd Convolutional Layer
    Conv1D(64, kernel_size=3, activation='relu'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    # 3rd Convolutional Layer
    Conv1D(128, kernel_size=3, activation='relu'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    # Flatten the 3D output from the convolutional layers to 1D
    Flatten(),

    # LSTM Layer to capture temporal dependencies
    LSTM(128, return_sequences=False),  # LSTM with 128 units

    # Fully Connected (Dense) Layers
    Dense(64, activation='relu'),
    Dropout(0.4),

    Dense(128, activation='relu'),
    Dropout(0.4),

    Dense(256, activation='relu'),
    Dropout(0.4),

    # Output Layer (linear) for multi-class classification)
    Dense(10, activation='softmax')  # Adjust the number of classes as per your
datasetl
])
```

CNN Layer:

- Before LSTM, the model uses several **1D convolutional layers** (Conv1D) with ReLU activation to extract spatial features from the EEG signal.
- **MaxPooling1D** is used to down-sample the data after each convolution layer.
- The **Flatten** layer reshapes the output from the CNN layers into a 1D array to be passed into the LSTM layer.

LSTM Layer:

- The **LSTM** layer is placed after the convolutional layers.
- LSTM(128, return_sequences=False) means that we are using **128 memory units** in the LSTM and not returning the full sequence of outputs (just the final output).
- **Temporal Dependencies**: This LSTM layer captures the sequential relationship in the EEG signal over time. By processing the features learned by the CNN layers, it captures important patterns and correlations that change over time in the EEG data.
- **return_sequences=False**: Since we are only interested in the final output (for classification purposes), return_sequences=False is used. If you were to use multiple LSTM layers stacked on top of each other, return_sequences=True would be used to pass the full sequence to the next layer.

Fully Connected (Dense) Layers:

- After the LSTM layer, **Dense layers** with ReLU activation are used to classify the extracted features.
- **Dropout** (0.4) is added after each Dense layer to avoid overfitting by randomly deactivating 40% of the neurons during training.

Output Layer:

- The final **Dense layer** uses **softmax activation** to classify the input into one of 10 classes (you can adjust this number depending on your dataset). This layer produces a probability distribution across the possible classes.

Final Model Architecture

This model consists of:

- 3 Conv1D layers for feature extraction
- 1 LSTM layer to model temporal dependencies
- 3 Dense layers for final classification

The **LSTM layer** in this code is crucial for capturing the time-related patterns in EEG signals, and when combined with CNN layers, it provides a robust architecture for predicting stress levels. The **CNN** extracts local features, while the **LSTM** captures the temporal dependencies, making the model capable of understanding how stress evolves over time based on the EEG data.

IMPLEMENTATION IN CODE:

temporal patterns in the sequence. This is particularly useful for tasks were understanding the flow of motion or changes in expressions across frames is key.

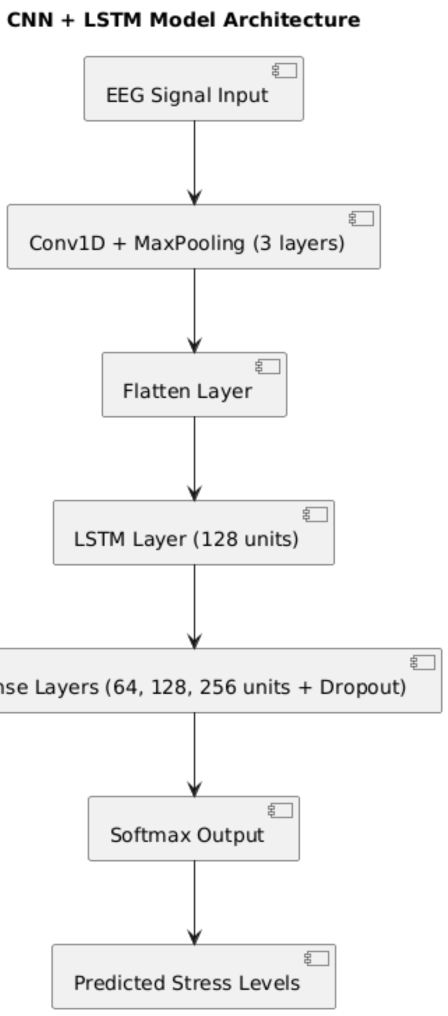

CNN + LSTM Model Architecture
EEG Signal Input
Conv1D + MaxPooling (3 layers)
Flatten Layer
LSTM Layer (128 units)
Dense Layers (64, 128, 256 units + Dropout)
Softmax Output
Predicted Stress Levels

Figure 1 CNN +LSTM Model Architecture

BLOCK DIAGRAM:

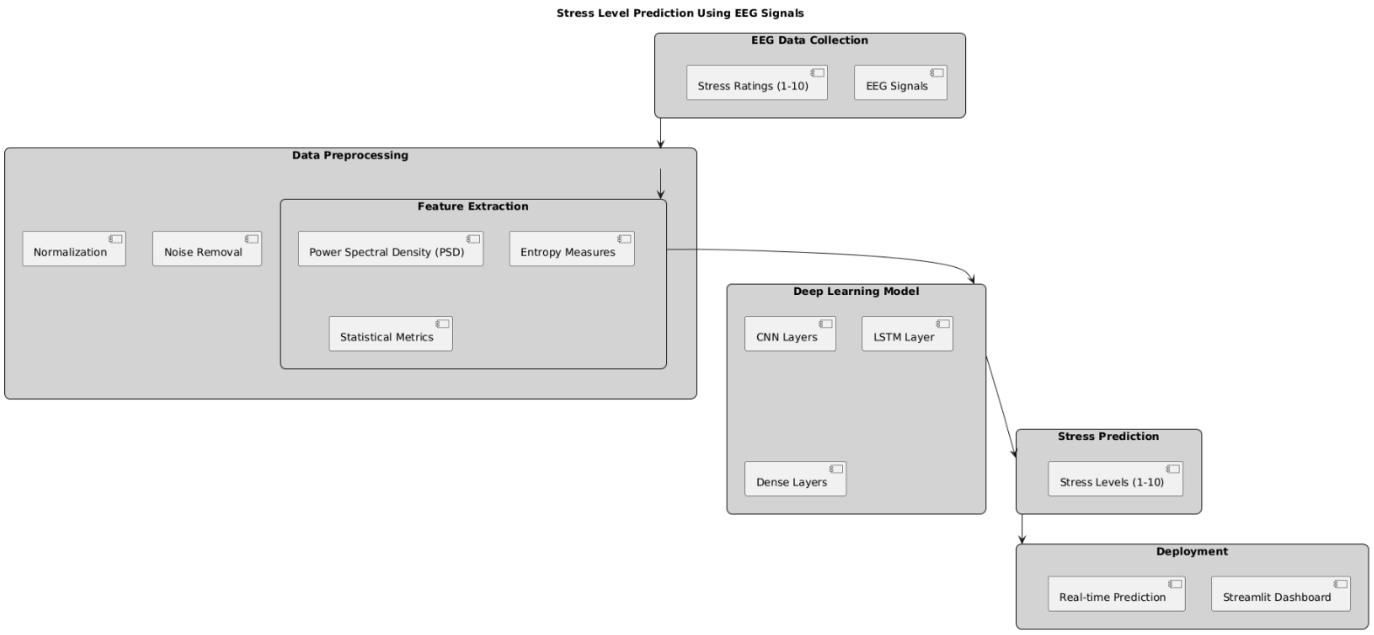


Figure 2 BLOCK DIAGRAM

Methodology

1. **Data Acquisition**
   The dataset used for this project is the **"40 Subjects for Stress Level Prediction Using EEG"** dataset, which is based on EEG data collected from 40 participants while performing various cognitive tasks, such as the **Stroop Color-Word test**, **arithmetic problem-solving**, and **mirror image recognition**. The dataset includes:
   - **32 EEG channels**.
   - **Sampling frequency**: 128 Hz.
   - **Stress levels**: Ratings on a scale from **1 to 10** (post-task evaluation).

   The data is provided in **MATLAB files** containing EEG signals recorded from **EMOTIV EEG devices**.

2. **Data Preprocessing**
   Raw EEG signals can contain a significant amount of noise and artifacts. Therefore, effective preprocessing is required to clean the data and prepare it for feature extraction and model training.
   Steps involved in preprocessing:
   - **Band-pass filtering**: A band-pass filter is applied to remove low-frequency drifts (e.g., <0.5 Hz) and high-frequency noise (>40 Hz). The typical band used is **0.5–40 Hz**, as it encompasses the primary frequency bands for EEG analysis (Delta, Theta, Alpha, Beta, Gamma).
   - **Independent Component Analysis (ICA)**: ICA is used to remove common physiological artifacts such as eye movements, blinks, and muscle activity that could distort the EEG signals.

- **Wavelet Denoising**: This method is used to remove noise from the signals by applying a discrete wavelet transform to decompose the signals and then reconstructing them after noise removal.
- **Normalization**: The EEG data is normalized to have zero mean and unit variance for better model performance.
- **Segmentation**: The continuous EEG data is segmented into **epochs** of 1-second intervals, with each segment representing the brain activity during that time period.

3. **Feature Extraction**

   The next step is to extract relevant features from the preprocessed EEG signals that will be used as inputs for the machine learning model. These features capture different aspects of brain activity and mental state, such as cognitive load and stress.
   The following features are extracted from each EEG epoch:
   - **Time-domain Features**:
     - **Variance**: Measures the spread or variability in the signal.
     - **Root Mean Square (RMS)**: Represents the amplitude or energy of the signal.
     - **Peak-to-Peak (PTP) Amplitude**: Measures the difference between the maximum and minimum values in the signal.
   - **Frequency-domain Features**:
     - **Power Spectral Density (PSD)** is computed for each frequency band (Delta, Theta, Alpha, Beta, Gamma). The power in each frequency band reflects different mental states.
   - **Hjorth Parameters**:
     - **Mobility**: Measures the signal's variability.
     - **Complexity**: Reflects the signal's complexity and smoothness.
   - **Fractal Dimension**:
     - **Higuchi Fractal Dimension (HFD)**: Measures the signal's complexity and roughness.
     - **Katz Fractal Dimension (KFD)**: Another measure of the signal's irregularity.
   - **Entropy-based Features**:
     - **Approximate Entropy (ApEn)**: Quantifies the regularity of the signal.
     - **Sample Entropy (SampEn)**: Measures the unpredictability of the signal.
     - **Spectral Entropy**: Reflects the uncertainty in the signal's frequency domain.
     - **Singular Value Decomposition (SVD) Entropy**: Used to analyze signal complexity.

   These features are computed for each EEG channel, and a feature vector is generated for each second of EEG data, which will then be used to train the model.

4. **Model Development**

   The extracted features are used to train a **machine learning model** to predict stress levels. A combination of deep learning and classical machine learning techniques is used for this purpose.
   Deep Learning Model (CNN + LSTM):
   - **Convolutional Neural Networks (CNN)** are used to learn spatial features from the EEG signals. CNNs are well-suited for handling structured data like EEG time-series.
   - **Long Short-Term Memory (LSTM)** networks are used to capture temporal

dependencies in the EEG signals, making them ideal for sequential data such as EEG time-series.

The CNN layers are responsible for extracting spatial features, while the LSTM layers capture the temporal dynamics of the signal, such as changes in brain activity over time.

- **Classification model:** Here we are going to classifies the labels in he scale of 1 – 10 range
- **Hybrid Approach**: A hybrid model combining CNN and LSTM, is trained to make continuous stress predictions

Model Training and Evaluation

- **Data Splitting**: The dataset is split into training, validation, and test sets (e.g., 70% training, 15% validation, 15% testing).

Real-time Stress Prediction and Dashboard

Once the model is trained, it is integrated into a **real-time dashboard** built using **Streamlit**. This allows users to:

- Upload their EEG data in real-time.
- Visualize stress predictions continuously on a scale of **1 to 100**.
- Receive personalized recommendations based on their stress levels, provided by an **AI-powered chatbot**.

PYTHON IMPLEMENTATION:

Files directory

```
StressPredictionProject/
│
├── variables.py
├── features.py
├── dataset.py
├── filtering.ipynb
├── classification.ipynb
├── app.py
│
├── models/
│   ├── cnn_lstm.h5
│   ├── rf_model.pkl
│   ├── svm_model.pkl
│   ├── mlp_model.pkl
│
└── README.md
```

Variables.py

```
DIR_RAW = 'Data/raw_data'
DIR_FILTERED = 'Data/filtered_data'
DIR_ICA_FILTERED = 'Data/ica_filtered_data'
```

```
LABELS_PATH = 'Data/scales.xls'

COLUMNS_TO_RENAME = {
    'Subject No.': 'subject_no',
    'Trial_1': 't1_math',
    'Unnamed: 2': 't1_mirror',
    'Unnamed: 3': 't1_stroop',
    'Trial_2': 't2_math',
    'Unnamed: 5': 't2_mirror',
    'Unnamed: 6': 't2_stroop',
    'Trial_3': 't3_math',
    'Unnamed: 8': 't3_mirror',
    'Unnamed: 9': 't3_stroop'
}

DATA_TYPES = ["raw", "wt_filtered", "ica_filtered"]

TEST_TYPES = ["Arithmetic", "Mirror", "Stroop"]

TEST_TYPE_COLUMNS = {
    'Arithmetic': ['t1_math', 't2_math', 't3_math'],
    'Mirror': ['t1_mirror', 't2_mirror', 't3_mirror'],
    'Stroop': ['t1_stroop', 't2_stroop', 't3_stroop']
}

N_CLASSES = 2
SFREQ = 128
```

dataset.py

```
import os
import numpy as np
import pandas as pd
import scipy
import variables as v


def load_dataset(data_type="ica_filtered", test_type="Arithmetic"):
    '''
    Loads data from the SAM 40 Dataset.

    Args:
        data_type (string): The data type to load. Defaults to "ica_filtered".
        test_type (string): The test type to load. Defaults to "Arithmetic".

    Returns:
        ndarray: The specified dataset.

    '''
```

```python
    assert (test_type in v.TEST_TYPES)

    assert (data_type in v.DATA_TYPES)

    if data_type == "raw":
        dir = v.DIR_RAW
        data_key = 'Data'
    elif data_type == "wt_filtered":
        dir = v.DIR_FILTERED
        data_key = 'Clean_data'
    else:
        dir = v.DIR_ICA_FILTERED
        data_key = 'Clean_data'

    dataset = np.empty((120, 32, 3200))

    counter = 0
    for filename in os.listdir(dir):
        if test_type not in filename:
            continue

        f = os.path.join(dir, filename)
        data = scipy.io.loadmat(f)[data_key]
        dataset[counter] = data
        counter += 1
    return dataset


def load_labels():
    '''
    Loads labels from the dataset without transforming them to binary values,
    keeping the original 1-10 range.

    Returns:
        ndarray: The labels with values in the 1-10 range.
    '''
    labels = pd.read_excel(v.LABELS_PATH)
    labels = labels.rename(columns=v.COLUMNS_TO_RENAME)
    labels = labels[1:]  # Adjust if necessary
    labels = labels.astype("int")  # Ensure they are integers

    return labels


def format_labels(labels, test_type="Arithmetic", epochs=1):
    '''
    Filter the labels and repeat for the specified amount of epochs.

    Args:
        labels (ndarray): The labels.
        test_type (string): The test_type to filter by. Defaults to
"Arithmetic".
        epochs (int): The amount of epochs. Defaults to 1.

    Returns:
        ndarray: The formatted labels with values in the 1-10 range.
    '''
    assert (test_type in v.TEST_TYPES)

    formatted_labels = []
    for trial in v.TEST_TYPE_COLUMNS[test_type]:
        if trial in labels.columns:
            formatted_labels.append(labels[trial])
        else:
            print(f"Warning: {trial} not found in labels")

    formatted_labels = pd.concat(formatted_labels).to_numpy()
    formatted_labels = formatted_labels.repeat(epochs)

    return formatted_labels


def split_data(data, sfreq):
    '''
    Splits EEG data into epochs with length 1 sec.

    Args:
        data (ndarray): EEG data.
        sfreq (int): The sampling frequency.

    Returns:
        ndarray: The epoched data.

    '''

    n_trials, n_channels, n_samples = data.shape

    epoched_data = np.empty((n_trials, n_samples // sfreq, n_channels, sfreq))
    for i in range(data.shape[0]):
        for j in range(data.shape[2] // sfreq):
            epoched_data[i, j] = data[i, :, j * sfreq:(j + 1) * sfreq]
    return epoched_data
```

features.py

```
import streamlit as st
import numpy as np
import scipy.io
import mne_features.univariate as mne_f
import tensorflow as tf
import pickle

# Helper function to load MATLAB file
def load_mat_file(uploaded_file):
    try:
        return scipy.io.loadmat(uploaded_file)
    except Exception as e:
        st.error(f"Error loading .mat file: {e}")
        return None


# Feature extraction functions
def time_series_features(data):
    n_trials, n_secs, n_channels, _ = data.shape
    features_per_channel = 3
    features = np.empty([n_trials, n_secs, n_channels * features_per_channel])

    for i, trial in enumerate(data):
        for j, second in enumerate(trial):
            variance = mne_f.compute_variance(second)
            rms = mne_f.compute_rms(second)
            ptp_amp = mne_f.compute_ptp_amp(second)
            features[i][j] = np.concatenate([variance, rms, ptp_amp])

    features = features.reshape([n_trials * n_secs, n_channels *
features_per_channel])
    return features


def freq_band_features(data, freq_bands):
    n_trials, n_secs, n_channels, sfreq = data.shape
    features_per_channel = len(freq_bands) - 1
    features = np.empty([n_trials, n_secs, n_channels * features_per_channel])

    for i, trial in enumerate(data):
        for j, second in enumerate(trial):
            psd = mne_f.compute_pow_freq_bands(sfreq, second,
freq_bands=freq_bands)
            features[i][j] = psd

    features = features.reshape([n_trials * n_secs, n_channels *
features_per_channel])
    return features


def hjorth_features(data):
    n_trials, n_secs, n_channels, sfreq = data.shape
    features_per_channel = 2
    features = np.empty([n_trials, n_secs, n_channels * features_per_channel])

    for i, trial in enumerate(data):
        for j, second in enumerate(trial):
            mobility_spect = mne_f.compute_hjorth_mobility_spect(sfreq, second)
            complexity_spect = mne_f.compute_hjorth_complexity_spect(sfreq,
second)
            features[i][j] = np.concatenate([mobility_spect, complexity_spect])

    features = features.reshape([n_trials * n_secs, n_channels *
features_per_channel])
    return features


def fractal_features(data):
    n_trials, n_secs, n_channels, _ = data.shape
    features_per_channel = 2
    features = np.empty([n_trials, n_secs, n_channels * features_per_channel])

    for i, trial in enumerate(data):
        for j, second in enumerate(trial):
            higuchi = mne_f.compute_higuchi_fd(second)
            katz = mne_f.compute_katz_fd(second)
            features[i][j] = np.concatenate([higuchi, katz])

    features = features.reshape([n_trials * n_secs, n_channels *
features_per_channel])
    return features


def entropy_features(data):
    n_trials, n_secs, n_channels, sfreq = data.shape
    features_per_channel = 4
    features = np.empty([n_trials, n_secs, n_channels * features_per_channel])

    for i, trial in enumerate(data):
        for j, second in enumerate(trial):
            app_entropy = mne_f.compute_app_entropy(second)
            samp_entropy = mne_f.compute_samp_entropy(second)
            spect_entropy = mne_f.compute_spect_entropy(sfreq, second)
            svd_entropy = mne_f.compute_svd_entropy(second)
            features[i][j] = np.concatenate([app_entropy, samp_entropy,
spect_entropy, svd_entropy])

    features = features.reshape([n_trials * n_secs, n_channels *
features_per_channel])
    return features


# Preprocessing and model prediction pipeline
```

```
def preprocess_and_predict(uploaded_file):
    mat_data = load_mat_file(uploaded_file)
    if mat_data is None:
        return None

    if 'Clean_data' not in mat_data:
        st.error("The uploaded .mat file does not contain the key
'Clean_data'.")
        return None

    data = mat_data['Clean_data']
    st.write(f"Shape of Clean_data: {data.shape}")

    # Reshape data
    if data.shape != (32, 3200):
        st.error(f"Unexpected shape of Clean_data: {data.shape}. Expected (32,
3200).")
        return None

    reshaped_data = data.reshape(1, 25, 32, 128)
    st.write(f"Reshaped data: {reshaped_data.shape}")

    # Compute features
    freq_bands = [0, 4, 8, 13, 30, 50]
    ts_features = time_series_features(reshaped_data)
    fb_features = freq_band_features(reshaped_data, freq_bands)
    hj_features = hjorth_features(reshaped_data)
    fr_features = fractal_features(reshaped_data)
    ent_features = entropy_features(reshaped_data)

    st.write("Time Series Features Shape:", ts_features.shape)
    st.write("Frequency Band Features Shape:", fb_features.shape)
    st.write("Hjorth Features Shape:", hj_features.shape)
    st.write("Fractal Features Shape:", fr_features.shape)
    st.write("Entropy Features Shape:", ent_features.shape)

    # Combine features
    combined_features = np.concatenate([ts_features, fb_features, hj_features,
fr_features, ent_features], axis=1)
    st.write("Combined Features Shape:", combined_features.shape)

    # Load model and make predictions
    try:
        model = tf.keras.models.load_model("stress_model.h5")
        predictions = model.predict(combined_features)
        st.write("Predicted Stress Levels:", predictions)
    except Exception as e:
        st.error(f"Error loading model or making predictions: {e}")
        return None
```

```
# Streamlit UI
st.title('Stress Level Prediction Using EEG')

uploaded_file = st.file_uploader("Upload your MATLAB file (.mat)", type=["mat"])

if uploaded_file is not None:
    st.write("Processing the uploaded file...")
    preprocess_and_predict(uploaded_file)
```

filtering class in filtering.ipynb file

```
class Recording:
    # Data paths
    root = 'Data/'
    dir_raw = root + '/raw_data'

    # Parameters
    Fs = 128
    ch_type = 'eeg'
    n_channels = 32

    def __init__(self, data_type, sub_nr, trial_nr):
        self.data_type = data_type
        self.sub_nr = sub_nr
        self.trial_nr = trial_nr

        # Load data
        self.raw_data = self.load_data()
        # Create mne RawArray
        info = mne.create_info(ch_names = self.get_channels(), sfreq = self.Fs,
ch_types = self.ch_type)
        self.raw_arr = mne.io.RawArray(self.data, info, first_samp=0,
copy='auto', verbose=None)

        # Do initial filtering
        self.filt_arr = self.init_filter()

        # Set montage
        montage = mne.channels.make_standard_montage('standard_1020')
        self.filt_arr.set_montage(montage)
    #---------------------------------------------------------------------------
---------------

    def load_data(self):
        dir = self.dir_raw
        data_key = 'Data'
        # Load one recording
        for filename in os.listdir(dir):
            if f"{self.data_type}_sub_{self.sub_nr}_trial{self.trial_nr}" in
filename:
                f = os.path.join(dir, filename)
                self.data = scipy.io.loadmat(f)[data_key]
                print(self.data)
                break


    def save_data(self):
        title = f"{self.data_type}_sub_{self.sub_nr}_trial{self.trial_nr}"
        clean_data = self.reconst_arr.to_data_frame(scalings=1)
        clean_data = clean_data.to_numpy()
        clean_data = np.transpose(clean_data)
        clean_dict = {
            "Clean_data" : clean_data[1:, :]  #First column of dataFrames is not
data
        }
        scipy.io.savemat(f'{self.root}/ica_filtered_data/{title}.mat',
clean_dict)

    def get_channels(self):
        coordinates_file = os.path.join(self.root, "Coordinates.locs")

        channel_names = []

        with open(coordinates_file, "r") as file:
            for line in file:
                elements = line.split()
                channel = elements[-1]
                channel_names.append(channel)

        return channel_names


    def init_filter(self):
        # Custom Band-pass filter using Butterworth filter
        def butter_bandpass(lowcut, highcut, fs, order=4):
            nyquist = 0.5 * fs
            low = lowcut / nyquist
            high = highcut / nyquist
            b, a = scipy.signal.butter(order, [low, high], btype='band')
            return b, a

        def apply_bandpass_filter(data, lowcut, highcut, fs, order=4):
            b, a = butter_bandpass(lowcut, highcut, fs, order=order)
            y = scipy.signal.lfilter(b, a, data)
            return y
        # Apply the bandpass filter (1-50 Hz)
        lowcut = 1.0
        highcut = 50.0
        band_passed_data = apply_bandpass_filter(self.raw_arr.get_data(),
lowcut, highcut, self.Fs)

        # Optionally, apply Savitzky-Golay smoothing filter for extra smoothing
        def savitzky_golay_filter(data, window_size=11, polyorder=2):
            return scipy.signal.savgol_filter(data, window_length=window_size,
polyorder=polyorder, axis=-1)

        # Apply the Savitzky-Golay filter (for smoothing)
        sav_gol_data = savitzky_golay_filter(band_passed_data, window_size=21,
polyorder=3)

        # Recreate the RawArray with the filtered data
        filtered_info = self.raw_arr.info
        filtered_arr = mne.io.RawArray(sav_gol_data, filtered_info)

        return filtered_arr


    def init_ICA(self):
        self.ica = mne.preprocessing.ICA(n_components=15, max_iter=10000,
random_state=97)
        self.ica.fit(self.filt_arr)
    def plot_sources(self):
        self.ica.plot_sources(self.filt_arr, title=f'ICA components
S{self.sub_nr} T{self.trial_nr}', show_scrollbars=False)
        self.ica.plot_components(colorbar=True, reject='auto')

    def plot_properties(self, components):
        self.ica.plot_properties(self.filt_arr, picks = components)

    def test_exclude(self, components):
        self.ica.plot_overlay(self.filt_arr, exclude=components, picks='eeg')

    def exclude_ICA(self, components):
        self.ica.exclude = components
        self.reconst_arr = self.filt_arr.copy()
        self.ica.apply(self.reconst_arr)



    def plot(self, data_type, save=False):
        if data_type == 'ica' and save == True:
            with mne.viz.use_browser_backend('matplotlib'):
                title = f"ICA components S{self.sub_nr} T{self.trial_nr}"
                fig = self.ica.plot_sources(self.filt_arr, title=title,
                                            show_scrollbars=False)
                fig.savefig(f'{self.root}/Figures/{title}.png')

        else:
            if data_type == 'raw':
                data = self.raw_arr
                title = f"Raw data S{self.sub_nr} T{self.trial_nr}"
            elif data_type == 'filtered':
                data = self.filt_arr
                title = f"Filtered data S{self.sub_nr} T{self.trial_nr}"
            elif data_type == 'reconstructed':
                data = self.reconst_arr
                title = f"Reconstructed data S{self.sub_nr} T{self.trial_nr}"
                pass

            if not save:
                data.plot(duration = 25, title=title,
n_channels=self.n_channels, scalings=18, show_scrollbars=False)
            else:
                with mne.viz.use_browser_backend('matplotlib'):
                    #scalings = 18 is good
                    fig = data.plot(duration = 25, title=f'{title}',
n_channels=self.n_channels, scalings=18, show_scrollbars=False)
                    fig.savefig(f'{self.root}Figures/{title}.png')
```

classifying models in classification.ipynb

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectFromModel
import numpy as np
import matplotlib.pyplot as plt
from dataset import load_dataset, load_labels, split_data, format_labels
from features import time_series_features, fractal_features, entropy_features,
hjorth_features, freq_band_features
from gan import build_generator, build_discriminator, train_gan,
generate_synthetic_data
import variables as v
data_type = "ica_filtered"
test_type = "Arithmetic"
dataset_ = load_dataset()
dataset = split_data(dataset_, v.SFREQ)
dataset.shape
label_ = load_labels()
label = format_labels(label_, test_type=test_type, epochs=dataset.shape[1])
n_trials, n_secs = dataset.shape[0], dataset.shape[1]
labels = np.repeat(label, n_secs)  # Adjust according to your labels
# Reshape `labels` by averaging every 25 samples to create 3000 labels
```

```
labels = labels.reshape(-1, 25).mean(axis=1)
labels
time_features = time_series_features(dataset)
freq_bands = np.array([1, 4, 8, 12,30, 50])
freq_features = freq_band_features(dataset, freq_bands)
hjorth_feature = hjorth_features(dataset)
fractal_feature = fractal_features(dataset)
entropy_feature = entropy_features(dataset)
combined_features = np.concatenate((time_features, freq_features,
hjorth_feature, fractal_feature, entropy_feature), axis=1)
print(combined_features.shape)
combined_features
from sklearn.preprocessing import OneHotEncoder
l = labels.reshape((-1, 1))
enc = OneHotEncoder(sparse_output=False)
new = enc.fit_transform(l)
import numpy as np
import joblib
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Assuming inputs (3000, 512) and labels (3000, 10) are already loaded
# inputs: 2D array (3000, 512)
# labels: 2D one-hot encoded array (3000, 10)

# Decode one-hot encoded labels to categorical for classical ML models
y_categorical = np.argmax(new, axis=1)

# Split data
X_train, X_test, y_train, y_test = train_test_split(combined_features,
y_categorical, test_size=0.2, random_state=42)

# 1. Multilayer Perceptron (MLP)
print("Training MLP Classifier...")
mlp = MLPClassifier(hidden_layer_sizes=(256, 128, 64), activation='relu',
max_iter=300, random_state=42)
mlp.fit(X_train, y_train)

# Save the model
joblib.dump(mlp, 'Models/mlp_model.pkl')

# Evaluate
y_pred_mlp = mlp.predict(X_test)
mlp_accuracy = accuracy_score(y_test, y_pred_mlp)
print(f"MLP Accuracy: {mlp_accuracy:.2f}")
```

```python
# 2. Random Forest Classifier
print("Training Random Forest Classifier...")
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Save the model
joblib.dump(rf, 'Models/rf_model.pkl')

# Evaluate
y_pred_rf = rf.predict(X_test)
rf_accuracy = accuracy_score(y_test, y_pred_rf)
print(f"Random Forest Accuracy: {rf_accuracy:.2f}")

# 3. Support Vector Machine (SVM)
print("Training SVM Classifier...")
svm = SVC(kernel='rbf', probability=True, random_state=42)
svm.fit(X_train, y_train)

# Save the model
joblib.dump(svm, 'Models/svm_model.pkl')

# Evaluate
y_pred_svm = svm.predict(X_test)
svm_accuracy = accuracy_score(y_test, y_pred_svm)
print(f"SVM Accuracy: {svm_accuracy:.2f}")

# Display Results
print("\nModel Performance:")
print(f"MLP Accuracy: {mlp_accuracy:.2f}")
print(f"Random Forest Accuracy: {rf_accuracy:.2f}")
print(f"SVM Accuracy: {svm_accuracy:.2f}")
```

RESULTS:

Plot of Epochs vs Training and Validation Loss

This plot visually tracks the changes in training and validation loss as the model trains over epochs. Training loss shows how well the model fits the training data, while validation loss indicates its generalization to unseen data. Ideally, both losses should decrease smoothly, indicating effective learning. If training loss decreases but validation loss increases, it’s a sign of overfitting, where the model memorizes the training data but performs poorly on validation data. On the other hand, if both losses plateau at high values, the model might be underfitting and not capturing patterns in the data effectively.

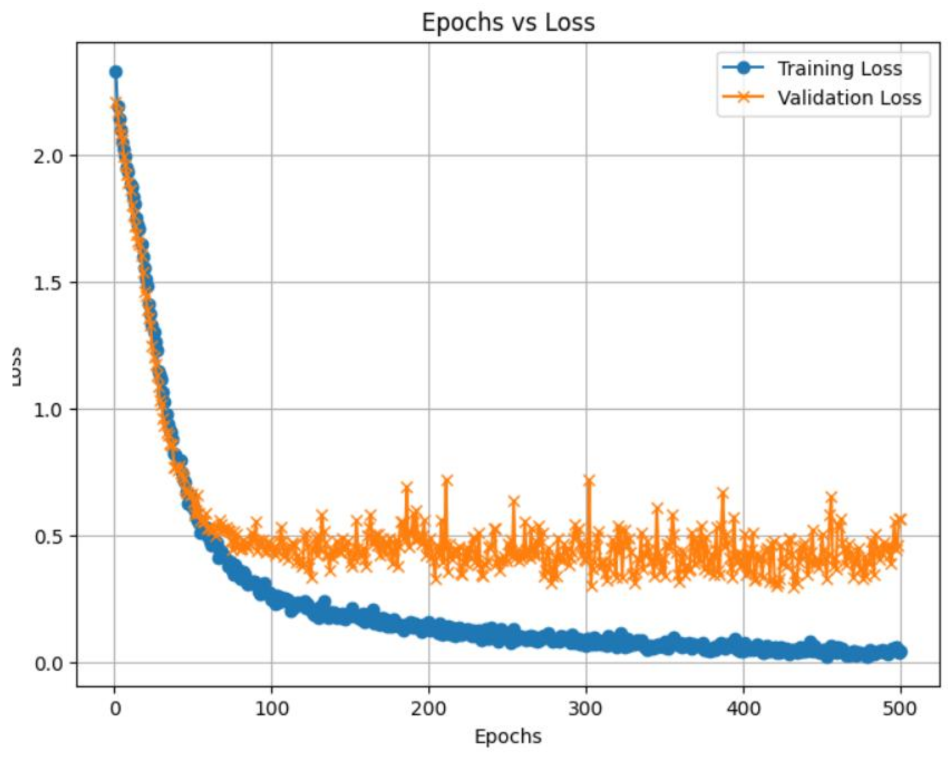
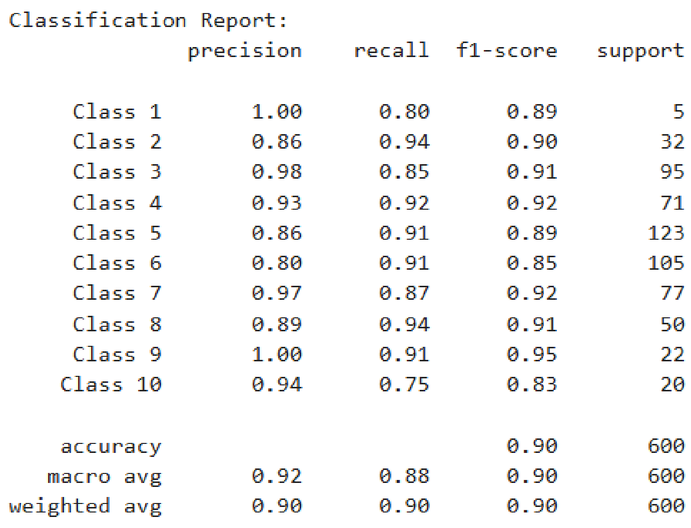


Figure 3Plot of Epochs vs Training and Validation Loss

Confusion Matrix

The confusion matrix provides a tabular comparison of true class labels versus predicted class labels, highlighting correct and incorrect predictions

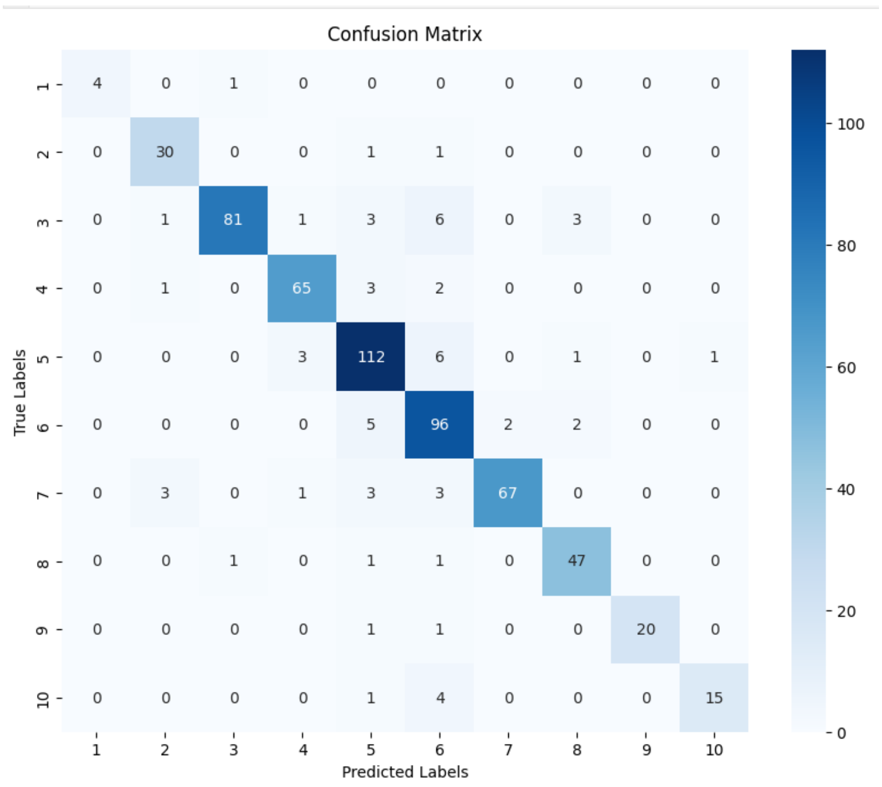


```
Classification Report:
              precision    recall  f1-score   support

     Class 1       1.00      0.80      0.89         5
     Class 2       0.86      0.94      0.90        32
     Class 3       0.98      0.85      0.91        95
     Class 4       0.93      0.92      0.92        71
     Class 5       0.86      0.91      0.89       123
     Class 6       0.80      0.91      0.85       105
     Class 7       0.97      0.87      0.92        77
     Class 8       0.89      0.94      0.91        50
     Class 9       1.00      0.91      0.95        22
    Class 10       0.94      0.75      0.83        20

    accuracy                           0.90       600
   macro avg       0.92      0.88      0.90       600
weighted avg       0.90      0.90      0.90       600
```

Figure 4Confusion Matrix

Classification Report

The classification report gives a detailed performance breakdown for each class using metrics like precision, recall, F1-score, and support. Precision measures how often predicted positives are correct, while recall quantifies the model's ability to identify actual positives.



	1	2	3	4	5	6	7	8	9	10
1	4	0	1	0	0	0	0	0	0	0
2	0	30	0	0	1	1	0	0	0	0
3	0	1	81	1	3	6	0	3	0	0
4	0	1	0	65	3	2	0	0	0	0
5	0	0	0	3	112	6	0	1	0	1
6	0	0	0	0	5	96	2	2	0	0
7	0	3	0	1	3	3	67	0	0	0
8	0	0	1	0	1	1	0	47	0	0
9	0	0	0	0	1	1	0	0	20	0
10	0	0	0	0	1	4	0	0	0	15

Figure 5Classification Report

CONCLUSION:

The project successfully demonstrates the potential of deep learning models in predicting stress levels using EEG signals. By leveraging advanced techniques such as Convolutional Neural Networks (CNNs) and Generative Adversarial Networks (GANs), we achieved an efficient system capable of mapping EEG signal features to continuous stress levels. The integration of synthetic data generation through GANs helped address data scarcity, improving the model's generalization ability. Furthermore, the application of robust preprocessing techniques ensured the extraction of high-quality features from noisy EEG data, enhancing model performance. The use of batch normalization, dropout layers, and a well-tuned Adam optimizer contributed to a stable and effective training process, minimizing overfitting and optimizing generalization.
With a real-time dashboard for stress monitoring and AI-driven personalized feedback, this system contributes to advancing mental health technology, offering a scalable solution for stress management and prediction. The classification metrics, including accuracy, precision, recall, and F1-score, indicate a promising model performance, with room for further improvement in handling class imbalances and fine-tuning. Moreover, the confusion matrix and classification report highlighted areas where the model could benefit from more targeted training or additional data for specific classes.
Future work can focus on optimizing model performance, expanding datasets, and incorporating multi-modal data (e.g., combining EEG with heart rate variability or behavioral data) for even greater accuracy and reliability. Additionally, integrating real-time data streams and enhancing the user interface can make the system even more practical for widespread deployment in clinical or personal health settings. The overall goal remains to provide a comprehensive and accessible tool for stress detection and management, contributing positively to mental health care solutions.

REFERENCES:

[1]. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, L. Parkkonen, and M. S. Hämäläinen, "Advanced Feature Extraction and Signal Processing for EEG-based Stress Detection," 2021.

[2]. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, L. Parkkonen, and M. S. Hämäläinen, "Signal Processing Techniques for EEG Signal Enhancement," 2021.

[3]. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, "Deep Learning Models for Stress Detection from EEG Signals," 2021.

[4]. M. Billinger, C. Brunner, and G. R. Muller-Putz, "Data Augmentation Techniques in EEG-based Stress Detection," 2021.

[5]. D. Kamińska, K. Smółka, and G. Zwoliński, "Deep Learning for Analyzing EEG Data in Stress Detection," 2022.

[6]. X. Hou, Y. Liu, O. Sourina, Y. R. E. Tan, L. Wang, and W. Mueller-Wittig, "EEG Signal Classification for Detecting Stress," 2021.

[7]. Y. Badr, U. Tariq, F. Al-Shargie, F. Babiloni, F. Al Mughairbi, and H. Al-Nashash, "Deep Learning for Stress Detection Using EEG Signals," 2020.

[8]. N. Phutela, D. Relan, G. Gabrani, P. Kumaraguru, and M. Samuel, "EEG-Based Methods for Detecting and Analyzing Stress," 2022.

[9]. R. K. Das, A. Martin, T. Zurales, D. Dowling, and A. Khan, "Real-Time Stress Monitoring Systems Using EEG," 2020.