Introduction to Machine Learning(IML) Project

Identify hand motions from EEG recordings

This project focuses on the development of a system to identify hand motions based on Electroencephalogram (EEG) recordings. EEG signals represent the electrical activity of the brain and can be used to decode motor commands related to hand movements. The goal is to create a reliable and real-time system that can interpret EEG signals to recognize different hand motions accurately.

We extracted the contents of "test.zip" and "train.zip" files into our working directory.

Firstly we loaded the data and the events that are done in the grasping and lifting or hand motions.

	id	Fp1	Fp2	F7	F3	Fz	F4	F8	FC5	FC1	...	P7	P3	Pz	P4	P8	PO9	O1	Oz	O2	PO10
0	subj1_series1_0	-31	363	211	121	211	15	717	279	35	...	536	348	383	105	607	289	459	173	120	704
1	subj1_series1_1	-29	342	216	123	222	200	595	329	43	...	529	327	369	78	613	248	409	141	83	737
2	subj1_series1_2	-172	278	105	93	222	511	471	280	12	...	511	319	355	66	606	320	440	141	62	677
3	subj1_series1_3	-272	263	-52	99	208	511	428	261	27	...	521	336	356	71	568	339	437	139	58	592
4	subj1_series1_4	-265	213	-67	99	155	380	476	353	32	...	550	324	346	76	547	343	446	171	67	581

5 rows × 33 columns

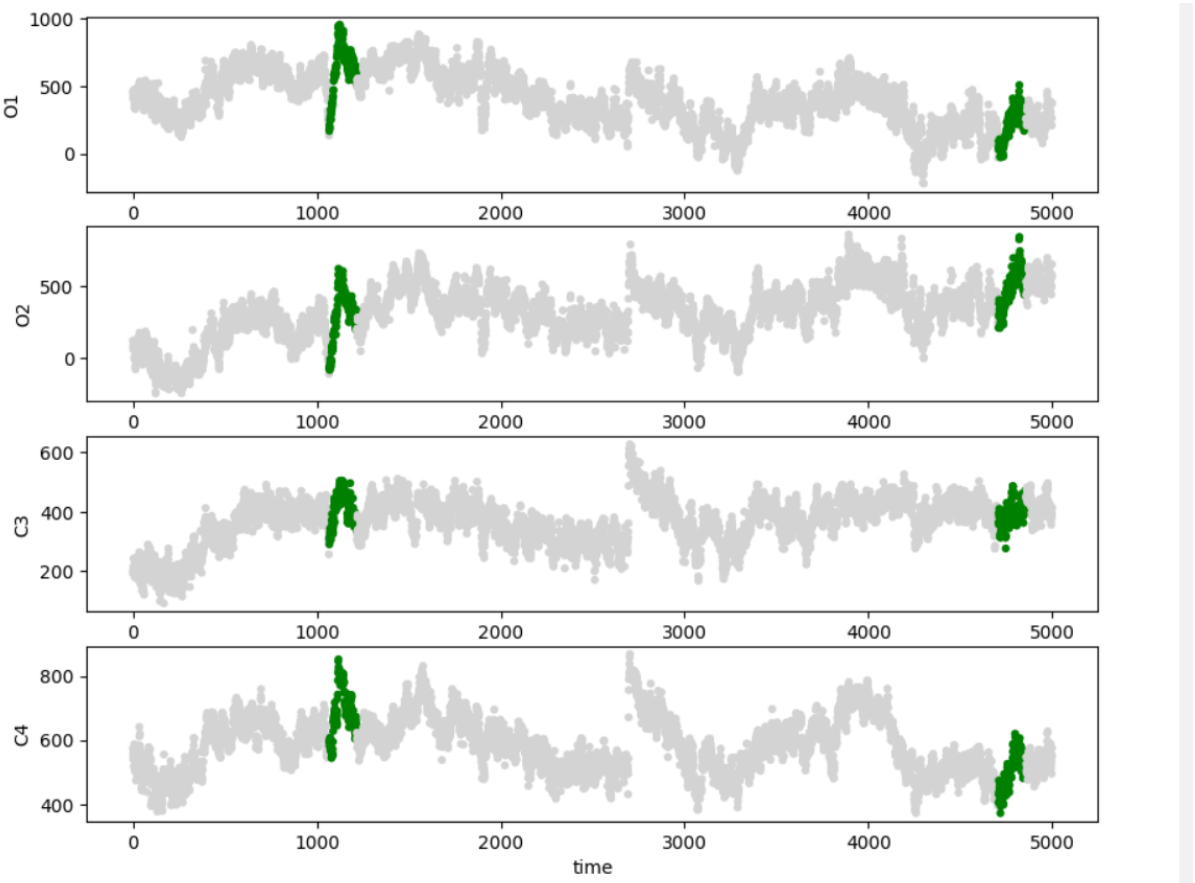
	id	HandStart	FirstDigitTouch	BothStartLoadPhase	LiftOff	Replace	BothReleased
0	subj1_series1_0	0	0	0	0	0	0
1	subj1_series1_1	0	0	0	0	0	0
2	subj1_series1_2	0	0	0	0	0	0
3	subj1_series1_3	0	0	0	0	0	0
4	subj1_series1_4	0	0	0	0	0	0

Then we read the signals and labels from CSV files, displayed the first few rows of the signals DataFrame, and then plotted the downsampled signals ('Fp1' and 'PO10') and labels for visualization using Matplotlib.

Then we combined the EEG signals and event labels into a single DataFrame, created a subset of the data for visualization, and then plotted EEG signals with highlighted events for a specified time duration. The colors of the highlights represent different event labels.

- HandStart = red
- FirstDigitTouch = purple
- BothStartLoadPhase = black
- LiftOff = green
- Replace = yellow
- BothReleased = blue

Then we visualized the EEG data by creating scatter plots for selected EEG channels against time, with points colored based on the occurrence of a specific event.
This one's for Handstart.



Wavelet Denoising

We have used Wavelet Denoising technique in our code for signal processing.

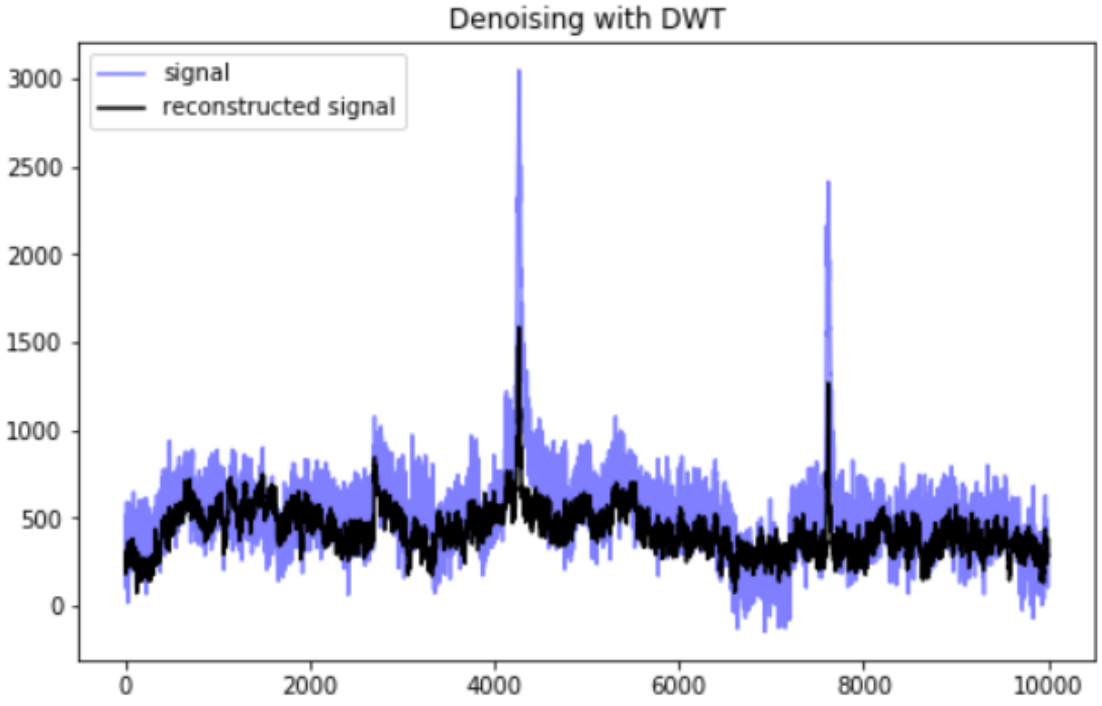
Wavelet denoising is a signal processing technique used to remove noise from signals, such as audio, image, or time-series data, by exploiting the properties of wavelet transforms. The basic idea is to decompose a signal into different frequency components using a wavelet transform, modify or threshold the coefficients associated with high-frequency components (considered as noise), and then reconstruct the denoised signals.

Our code is part of a larger pipeline for processing and classifying EEG signals using wavelet denoising and logistic regression. It loads data from multiple subjects, applies signal processing techniques, and prepares the

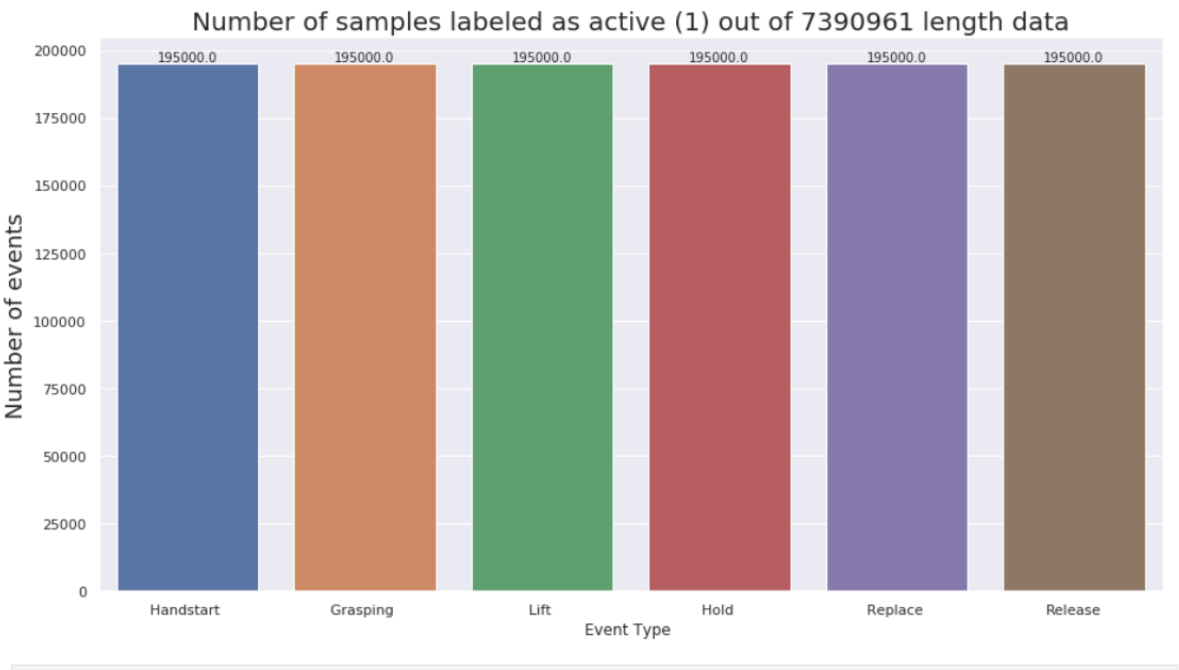
data for training and evaluation. The specific machine learning model and training steps are not included in this snippet.

Wavelet denoising has several advantages, including its ability to handle signals with non-stationary or time-varying characteristics. It is particularly useful when the noise in a signal has a distinct frequency content that can be separated from the signal's useful information

Then we coded to demonstrate the application of wavelet denoising to a signal using the Discrete Wavelet Transform and visualized the original and denoised signals for comparison.



Then we generated a bar plot using seaborn to visualize the distribution of different event types in a dataset. The plot provides a clear overview of the number of occurrences for each event type, with text labels on top of each bar indicating the exact count.



APPLYING CNN:-

This code defines a Convolutional Neural Network (CNN) model using the Keras library. Here's a brief summary:

1. Model Architecture:
   - The model consists of three convolutional layers with increasing kernel sizes and 64 filters each. Each convolutional layer is followed by Batch Normalization.
   - The input shape is set to (time_steps//subsample, 32, 1), indicating a time series input with 32 features and subsampled for efficiency.
   - The convolutional layers use the ReLU activation function.
   - The Flatten layer is used to transform the output into a one-dimensional array.
   - Two dense (fully connected) layers follow, with 32 and 6 neurons, respectively. The final layer uses sigmoid activation for multi-label classification.

2. Optimizer and Compilation:
   - The Adam optimizer with a learning rate of 0.0001 is used.

- The loss function is set to "binary_crossentropy" since it's a multi-label classification problem.
- Metrics include accuracy and mean squared error (mse).

3. Model Summary:
- The `model.summary()` function prints a summary of the model architecture, displaying layer types, output shapes, and the number of parameters in each layer.

This CNN is designed for a specific task i.e. multi-label classification, given the use of the sigmoid activation in the output layer and binary cross-entropy as the loss function.

This code defines a generator function, `valgenerator()`, for creating batches of validation data in an infinite loop. It selects random sequences of a specified length (`time_steps`) from validation data (`xval` and `yval`), subsamples them, and yields batches of 32 samples at a time. The generator is designed for use in training neural networks, ensuring efficient handling of large datasets by generating data on-the-fly.

This code is training a machine learning model using a generator for on-the-fly data generation during training.

1. Timing Start:
- `start = time.time()` records the starting time for measuring the total training duration.

2. Generator Function (`generator(batch_size)`):
- An infinite loop (`while 1`) generates training data on-the-fly during training.
- For each iteration, it creates a batch of training data (`x_time_data`) and corresponding labels (`yy`).
- Random sequences of length `time_steps` are selected from the training data (`xtrain` and `ytrain`), subsampled, and added to the batch.

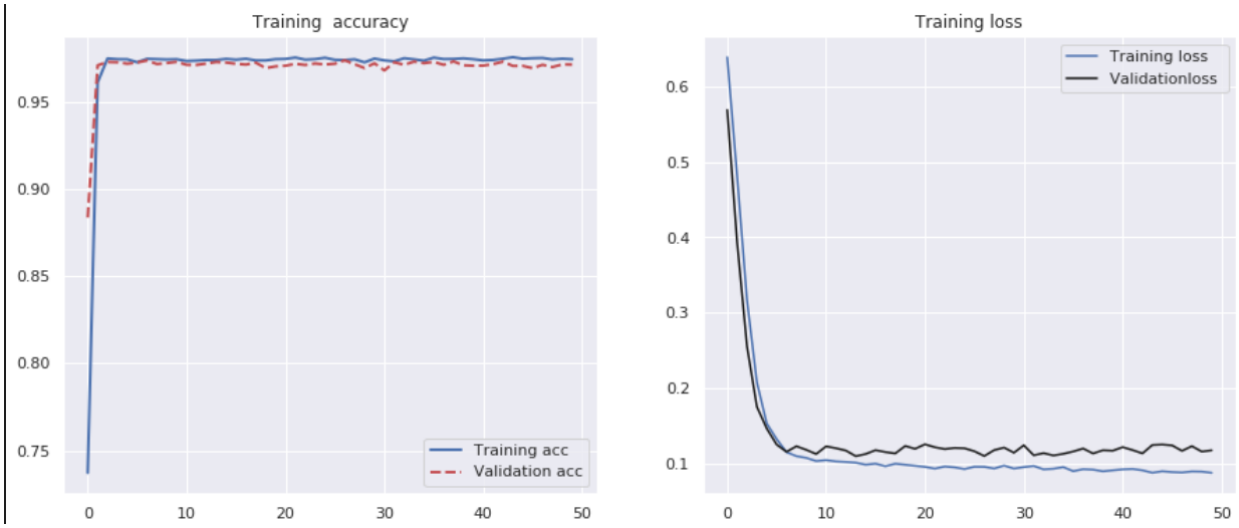
- The generator yields batches of 32 samples at a time, reshaped to match the expected input shape of the neural network.

3. Model Training (`model.fit_generator(...)))
- `fit_generator` is used to train the model.
- Training data is generated by the `generator(32)` function with a batch size of 32 samples.
- `steps_per_epoch` is set to 600, indicating the number of batches to process before declaring one epoch finished.
- The model is trained for 50 epochs (`epochs = 50`).
- Validation data is generated by the `valgenerator()` function during training, with `validation_steps` set to 200.
- The training progress and evaluation metrics are stored in the `history` variable.

4. Timing End:
- After training, the total time taken for training is calculated using `time.time() - start`.
- The result is printed, indicating the overall training time in seconds.

5.
- Generators are memory-efficient, generating and processing data on-the-fly, which is beneficial for handling large datasets during training.
- The training time provides insights into the efficiency and speed of the training process.

Here are the plots for training accuracy and training loss-



TESTING WITH UNSEEN DATA:-

This code defines a generator function `val_generator()` in Python, which is intended to be used for generating batches of validation data for a neural network.

1. `time_steps` and `subsample` are constants representing the length of the time sequence and the subsampling factor, respectively.

2. The function is defined using a `while 1` loop, creating an infinite loop. This is common in generator functions because they are often used to continuously generate batches of data during training.

3. Inside the loop, `batch_size` is set to 1, indicating that the generator will yield one batch of data at a time.

4. `x_time_data` is initialized as a 3D NumPy array with shape `(batch_size, time_steps//subsample, 32)`. This array is meant to hold the input data for the neural network. The shape implies that the input data is a

time series with 32 features, and the data is subsampled based on the `subsample` factor.

5. An empty list `yy` is initialized, which will be used to store the corresponding output labels.

6. The loop then iterates over the batch size (`for i in range(batch_size)`) to populate `x_time_data` and `yy`. For each iteration:

7. After the loop, `yy` is converted to a NumPy array using `np.asarray(yy)`.

8. The generator yields a tuple containing the input data and the corresponding labels, reshaped to be compatible with Convolutional Neural Network (CNN) input requirements:
`x_time_data.reshape((x_time_data.shape[0], x_time_data.shape[1], x_time_data.shape[2], 1)), yy`.

9. The `reshape` is done to add an extra dimension with size 1 at the end of the input data shape. This is often necessary when working with 1D convolutional layers in neural networks, which typically expect input shapes of the form `(batch_size, sequence_length, features, channels)`.

This generator is used with the Keras `fit_generator` function to train and validate a neural network that processes time series data with convolutional layers. The generator generates batches of input data and corresponding labels on-the-fly, making it memory-efficient for handling large datasets.

Now we evaluate the model using a generator (`val_generator()`) to produce batches of validation data. It iterates through a specified number of tests (`num_test`), retrieves a batch of data from the generator, and evaluates the model's performance on that batch. The evaluation scores (specifically, accuracy) for each test are recorded and then averaged to provide an overall assessment of the model's accuracy on the validation data.

Now , we calculate the roc curve and area for each of the classes and here are the plots for the same.