FFR-based Listener Classification using Machine Learning

Yahya Massoud
Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
ymass049@uottawa.ca

***Abstract*—This study delves into the comparative analysis of machine learning models for listener classification based on auditory biometrics. A series of experiments involving Support Vector Machines (SVM), 2D Convolutional Neural Networks (CNNs), and 1D CNNs were conducted to evaluate their efficacy in classifying listeners from Frequency Following Response (FFR) signals. Our findings suggest that SVMs, particularly with the Radial Basis Function (RBF) kernel, offer stable performance when trained on spectrograms, with the ability to benefit from increased data resolution. However, 2D CNNs did not achieve the same level of success, possibly due to the non-localized nature of spectral features within spectrograms. In contrast, 1D CNNs demonstrated superior proficiency in feature extraction from raw signals, showing that direct analysis of time-domain data can be highly effective. Particularly, 1D CNNs trained on raw signals matched the performance of their SVM counterparts, suggesting an inherent advantage in processing unaltered signal data. The study's outcomes highlight the significant potential of 1D CNNs in developing sophisticated auditory biometric systems and the need for balanced model selection in real-world applications considering both accuracy and computational efficiency.**

***Index Terms*—Signal Processing, Biometrics, Auditory Evoked Potentials**

I. Introduction

Biometrics, the science of identifying individuals through unique physical characteristics, continues to advance rapidly. This field employs a variety of biometric features, each with distinct advantages and limitations. Common types include fingerprint scanning, facial recognition, iris recognition, and voice analysis. Fingerprints, for instance, offer high precision but can be duplicated, raising security concerns. Facial recognition, convenient and non-intrusive, struggles in varying light conditions and can be less reliable with facial changes. Iris recognition stands out for its accuracy, yet it requires proximity and specialized equipment, limiting its applicability. Voice recognition is notable for its ease of use but is vulnerable to variations in voice and background noise.

In the realm of auditory-evoked responses, two significant types are Auditory Evoked Potentials (AEP) and Frequency Following Responses (FFR). AEPs are time-locked neural responses triggered by the absence or presence of sound stimuli, providing valuable insights into auditory processing. However, their effectiveness can be influenced by the subject's state of attention and the surrounding environment. FFRs, on the other hand, are short-latency phase-locked neural responses that precisely track the processing of sound frequencies in the brain. These responses are remarkable for their ability to reflect individual auditory pathway functioning. While non-invasive and insightful, FFRs require sophisticated analysis and are sensitive to external noise, posing challenges in obtaining precise measurements.

The exploration of FFRs for listener classification has become a focal point in recent research [1]–[4]. Various machine learning approaches, such as Support Vector Machines (SVM), Convolutional Neural Networks (CNN), Random Forests, and XG-Boost, have been investigated to harness the potential of FFRs. These techniques differ in their capacity to process and interpret the intricate data derived from FFRs.

Our contribution to this area involves the use of SVM, alongside both 2D and 1D CNN models, for classifying listeners based on FFRs. We have conducted an in-depth comparison of these methods, assessing their performance across multiple metrics to determine their effectiveness in this specialized field. Our goal is to enhance the understanding and application of auditory biometrics, aiming for more precise and personalized listener identification. This report details our findings and perspectives on employing FFRs for biometric purposes, presenting a new angle in auditory biometric research.

II. Data

The dataset [5] that was provided in this course for the purpose of this study comprises Frequency Following Responses (FFRs) recorded using the Bio-logic BioMARK system. In this experimental setup, stimuli were delivered to the right ear of participants through an insert earphone, while the left ear remained open and unoccluded. The stimuli consisted of four synthetic vowels, each with a fundamental frequency of 100 Hz. These vowels were /a/ as in 'father,' '/ɔ/' as in 'call,' /U/ as in 'boot,' and /u/ as in 'who.'

The recording of FFR data was precisely timed at 106.6 milliseconds, equivalent to 1024 samples, with a sampling rate of 9606 Hz. This approach ensured high-resolution data capture, crucial for accurate analysis. The responses were recorded at a level of 85 dBA, a standard measure of sound pressure level. Our participant pool comprised 20 English-speaking adults, balanced in gender with 10 males and 10 females.

To enhance the reliability of the data, two recording sessions were conducted for each participant, labeled as 'Test' and 'Retest.' These sessions took place on different days to account for any day-to-day variability in the participants' auditory responses. This dual-session approach provides a robust framework for assessing the consistency and reliability of the FFR measurements across different temporal intervals.

A. Signal Preprocessing

Fig. 1: The complete signal preprocessing pipeline.

1) Initial Data Acquisition: The preprocessing pipeline begins with raw Frequency Following Response (FFR) signals. These signals are captured by applying synthetic vowel stimuli at a sampling rate of 9606 Hz. Each vowel signal spans 106.6 ms, constituting 1024 recorded samples. The FFR signals from all four vowels are concatenated, resulting in a combined signal duration of 426.4 ms (4096 samples).

- **Advantages**: This approach captures a comprehensive range of auditory responses, providing rich data for analysis.
- **Disadvantages**: The concatenation process might introduce discontinuities between different vowel sounds, potentially affecting the coherence of the signal.

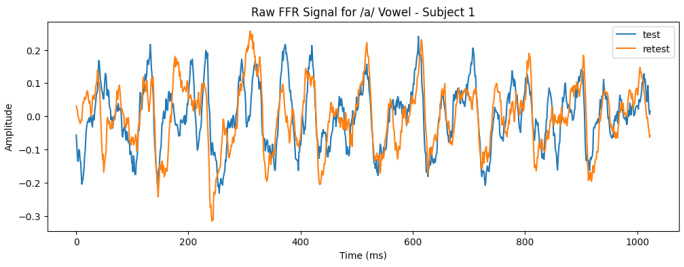

Fig. 2: Plotting the raw signal collected from Subject 1 on both Test and Retest sessions for the vowel /a/.

2) Signal Detrending: Signal detrending is employed to eliminate linear trends from the data, thereby enhancing signal stability and reducing potential artifacts. This step is crucial for maintaining the integrity of the signal for accurate analysis. The detrending process is performed using:

```
from scipy import signal
signal_detrended = signal.detrend(input_signal)
```

- **Advantages**: Enhances signal clarity and stability, making the data more suitable for accurate analysis.
- **Disadvantages**: Potential loss of low-frequency trends that could be relevant in some auditory studies.

3) Removing DC Offset: To center the signal around zero and improve the accuracy of spectral analysis, the DC offset is removed:

```
import numpy as np
signal_zero_mean = signal_detrended - np.mean(
    signal_detrended)
```

- **Advantages**: Centers the signal around zero, enhancing the reliability of spectral analysis.
- **Disadvantages**: Risk of altering original signal dynamics if not properly balanced.

4) Applying Hamming Window: The application of a Hamming window reduces spectral leakage by tapering the signal at both ends. This step minimizes edge discontinuities in each sampled window:

```
from scipy import signal
hamming_window = signal.windows.hamming(len(
    signal_zero_mean))
signal_windowed = signal_zero_mean * hamming_window
```

- **Advantages**: Reduces spectral leakage and minimizes edge effects, which is crucial for accurate frequency domain analysis.
- **Disadvantages**: May result in minor data loss at the signal edges, slightly reducing the effective data length.

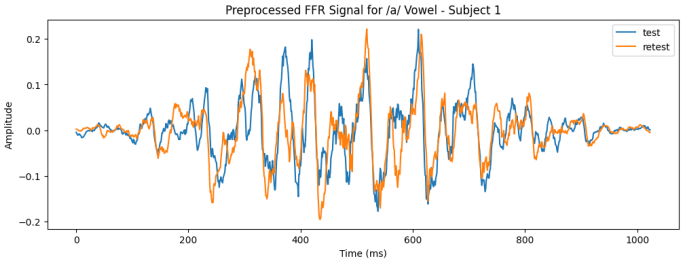

Fig. 3: Plotting the preprocessed signal collected from Subject 1 on both Test and Retest sessions for the vowel /a/.

5) Zero Padding: Zero padding increases the signal length, facilitating efficient computation and smoothing the peaks in the signal:

```
import numpy as np
signal_padded = np.pad(signal_windowed, (0,
    padding_length), "constant")
```

- **Advantages**: Improves Fourier transform efficiency and increases spectral resolution, making it easier to identify distinct frequency components.
- **Disadvantages**: Increases the total data size, which may lead to higher computational requirements, especially in large-scale datasets.

6) Generating Amplitude Spectrum: The amplitude spectrum is obtained by transforming the time-domain signal into the frequency domain using Fourier transform. This reveals the frequency components and their amplitudes, essential for identifying auditory evoked responses. We apply a frequency cutoff at 1300 Hz to focus on the most relevant frequency

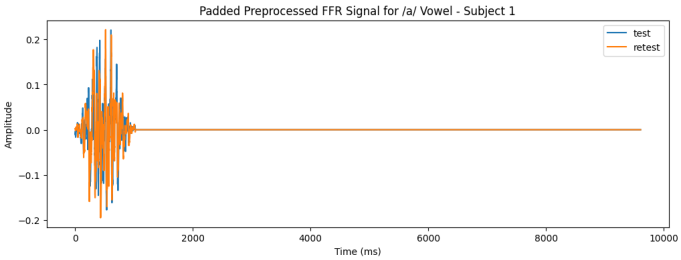

Fig. 4: Plotting the preprocessed and padded signal collected from Subject 1 on both Test and Retest sessions for the vowel /a/.

range for FFR, as higher frequencies typically contribute less to auditory evoked potentials.

- **Advantages**: Provides a clear depiction of frequency components, which is critical for understanding auditory responses.
- **Disadvantages**: Filtering out higher frequencies may omit some potentially informative components of the signal.

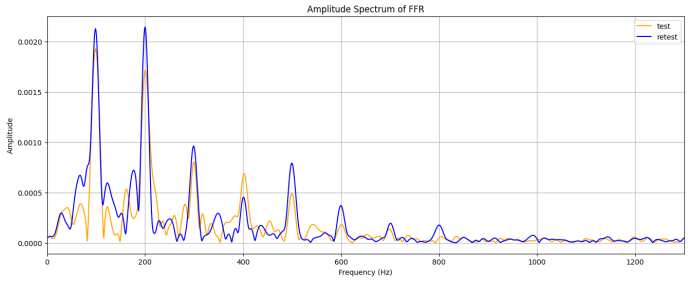

Fig. 5: Plotting the amplitude spectrum of the preprocessed signal collected from Subject 1 on both Test and Retest sessions for the vowel /a/. The cutoff frequency is at 1300 Hz.

7) Time-Frequency Representation: The time-frequency representation of the signal is generated to analyze how spectral content varies over time. We use varying window sizes and overlaps to achieve different resolutions:

- **Advantages**:
  - Smaller windows allow for analyzing rapid changes in the signal over time, capturing transient events effectively.
  - Larger windows provide higher frequency resolution, enabling more precise analysis of the frequency components.
- **Disadvantages**:
  - Smaller window sizes may lead to less detailed frequency information, potentially missing finer frequency nuances.
  - Larger window sizes result in reduced temporal resolution, possibly overlooking quick temporal variations in the signal.
  - Varying overlap sizes impact computational efficiency: lower overlap speeds up computation but may miss finer details, whereas higher overlap offers a more detailed representation at the cost of increased computational load.

We generate spectrograms with window sizes of 256 and overlaps of 8, 64, 128, 250, as well as window size 512 with overlaps of 0, 256, 511. These configurations are chosen to provide a balanced view of both time and frequency characteristics of the FFR signals, catering to the needs of our specific analytical goals.

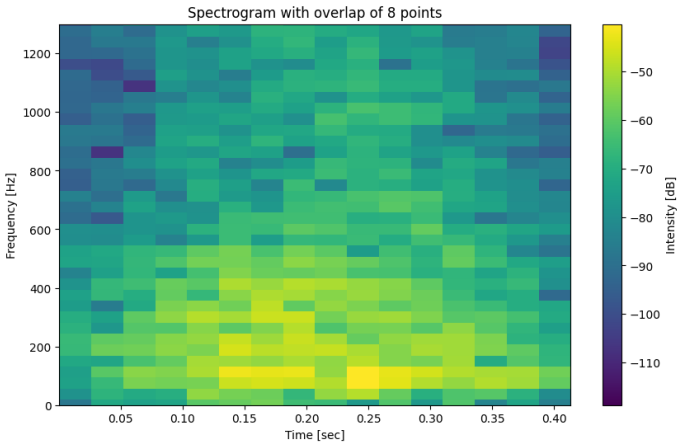

Fig. 6

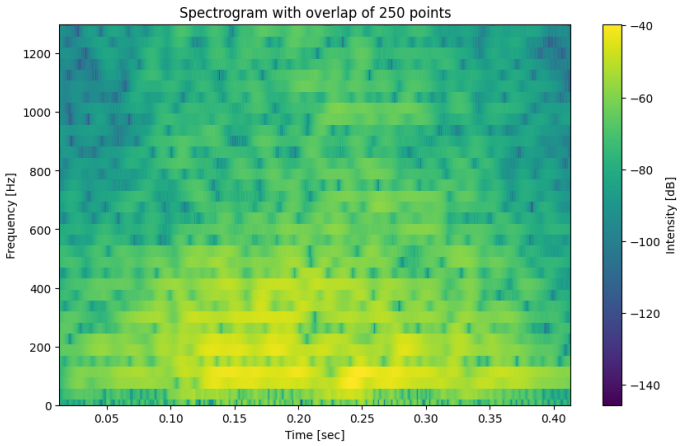

Fig. 7

B. Data Preparation

Data preparation is a critical step in ensuring the robustness and accuracy of machine learning models. Our study involves several stages of data preparation, each tailored to optimize the performance of the models we employ.

1) Data Splitting: The dataset is meticulously divided for training and testing to ensure robust model evaluation. Specifically, we allocate 40 samples for the test dataset and another 40 for the retest dataset, with each listener contributing

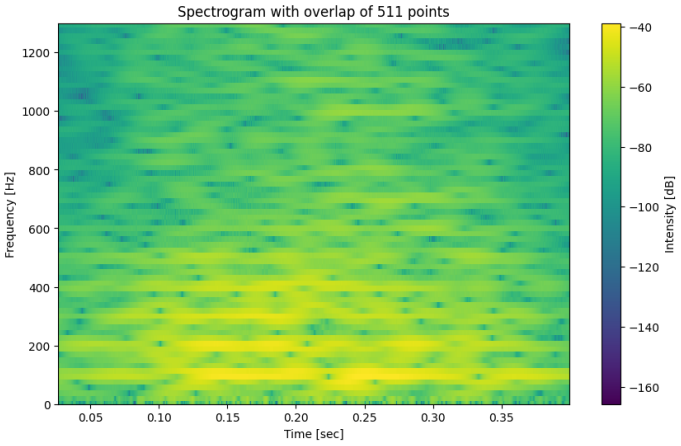

Fig. 8

2 samples. This balanced distribution across both datasets ensures comprehensive representation. Our training and testing strategy is articulated as:

1) Train on Test $\rightarrow$ Test on Retest
2) Train on Retest $\rightarrow$ Test on Test

This cross-validation approach allows for evaluating the model's performance across different data sets, enhancing the generalizability of the findings. We compute a range of metrics, with a particular focus on the F1 score, which harmonizes precision and recall, thus providing a holistic measure of model performance.

2) Spectrograms: Spectrograms serve as a pivotal component in our analysis, offering a visual representation of the sound spectrum over time. We prepare two versions:

- Version 1: Original spectrogram data encapsulated as 2D Numpy arrays. This format retains the original fidelity and detail of the sound data.
- Version 2: Spectrograms are processed into 32x32 grayscale images, optimizing them for input into CNN architectures. This transformation enables the use of advanced image processing techniques in sound analysis.

Example: A spectrogram of the vowel /a/ might display distinct patterns in both versions, which our models will use for classification.

3) Normalization: Normalization is a crucial step in data preprocessing, particularly for neural network models:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \tag{1}$$

By scaling input values between 0 and 1, min-max normalization ensures uniformity in data range across different features. This uniformity is vital for models that are sensitive to input scale, facilitating faster and more stable convergence during training.

4) One-hot Encoding: For categorical data like class labels, one-hot encoding is employed. This process converts categorical variables into a binary vector representation, which is more suitable for processing by neural networks. For instance, if we have four classes representing the vowels, each class will be represented by a unique binary vector.

Example. One-hot encoding transforms categorical class labels into a binary vector format. Consider a dataset with five class labels: A, B, C, D, and E. The one-hot encoding for each class is represented as a binary vector of length five, corresponding to the number of classes. Each vector has a '1' in the position of the class it represents and '0's in all other positions. The encoding is as follows:

- Class A: $[1, 0, 0, 0, 0]$
- Class B: $[0, 1, 0, 0, 0]$
- Class C: $[0, 0, 1, 0, 0]$
- Class D: $[0, 0, 0, 1, 0]$
- Class E: $[0, 0, 0, 0, 1]$

This method ensures that each class is uniquely identifiable in the model, facilitating effective classification.

5) Data Shuffling: Continuous shuffling of the data during training is implemented to prevent the model from memorizing the order of the training data, a common cause of overfitting. Shuffling ensures that each batch of data fed into the model during training is random, forcing the model to learn general patterns rather than dataset-specific anomalies.

C. Produced Datasets

Table I presents a detailed breakdown of the various datasets used in our study, categorized by their specific characteristics and dimensions. It includes spectrogram images and various spectrogram datasets processed with different window sizes and overlaps, ranging from 80 x 32 x 32 x 1 for the spectrogram images to 80 x 70 x 3585 for the largest spectrogram dataset. Additionally, it lists the raw signals and preprocessed signals both with and without padding, showing their respective dimensions. The amplitude spectrum data is also included, showcasing its dimensionality. This table serves as a comprehensive reference for understanding the structure and scale of the datasets employed in our analysis, highlighting the diverse nature of the data we have processed and analyzed. These datasets are going to be fed into the ML models to learn the ability to do listener classification.

TABLE I: Summary of dataset dimensions

Dataset Name	**Input Shape**
Spectrogram images	80 x 32 x 32 x 1
Spectrogram (window_size=256, overlap=8)	80 x 35 x 16
Spectrogram (window_size=256, overlap=64)	80 x 35 x 21
Spectrogram (window_size=256, overlap=128)	80 x 35 x 31
Spectrogram (window_size=256, overlap=250)	80 x 35 x 641
Spectrogram (window_size=512, overlap=0)	80 x 70 x 8
Spectrogram (window_size=512, overlap=256)	80 x 70 x 15
Spectrogram (window_size=512, overlap=511)	80 x 70 x 3585
Raw Signals	80 x 4096 x 1
Preprocessed Signals (w/o padding)	80 x 4096 x 1
Preprocessed Signals (w padding)	80 x 9606 x 1
Amplitude Spectrum	80 x 1300 x 1

III. METHOD

A. Support Vector Machine (SVM) for Listener Classification

1) Theoretical Framework: Support Vector Machines (SVM) are a set of supervised learning methods used for classification, regression, and outliers detection. The basic principle of SVM is to find a hyperplane in an N-dimensional space (N — the number of features) that distinctly classifies the data points. To separate the two classes of data points, many possible hyperplanes could be chosen. The goal is to find a plane that has the maximum margin, i.e., the maximum distance between data points of both classes. The hyperplane is defined by the equation:

$$\mathbf{w} \cdot \mathbf{x} - b = 0 \tag{2}$$

where $\mathbf{w}$ is the weight vector and b is the bias.

The decision function is given by:

$$f(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} - b) \tag{3}$$

where sgn is the sign function.

2) Application in Listener Classification: In our study, SVM is employed to classify listeners based on spectrogram data. The 2D spectrograms are first transformed into a 1D format for compatibility with the SVM algorithm. Unlike some machine learning applications, the class labels in our model are not one-hot encoded.

We explore three types of SVM kernels to evaluate their performance in our specific context: linear, polynomial (poly), and radial basis function (rbf). The kernel function is a critical component that determines the decision boundary in the transformed feature space. The implementation is conducted using Python's `scikit-learn` library:

```
from sklearn.svm import SVC

svm_linear = SVC(C=1.0, kernel="linear")
svm_poly = SVC(C=1.0, kernel="poly", degree=3)
svm_rbf = SVC(C=1.0, kernel="rbf", gamma="scale")
```

The polynomial kernel's degree is set to 3 as a hyperparameter. For the rbf kernel, the gamma parameter is scaled based on the statistical properties of the input data, calculated as:

$$\gamma = \frac{1}{n_{\text{features}} \times \text{X.var()}} \tag{4}$$

This scaling of gamma, especially when set to 'scale', ensures that its value is automatically adjusted to the variance of the input data, enhancing the kernel's ability to conform to the data distribution.

3) Training Configurations: The SVM models are trained using spectrograms with varying window sizes and overlap values to assess the impact of these parameters on classification performance. Specifically, we train with:

- Window size = 256, with overlap values: 8, 64, 128, 250.
- Window size = 512, with overlap values: 0, 256, 511.

These configurations provide a diverse range of feature resolutions, enabling a comprehensive evaluation of the SVM classifiers under different signal processing conditions.

B. 2D Convolutional Neural Network (CNN)

1) Overview of 2D CNN: Convolutional Neural Networks (CNNs) are a class of deep neural networks, most commonly applied to analyzing visual imagery. A typical CNN architecture consists of various layers, each performing distinct operations:

1) **Convolution Layer**: Applies a convolution operation to the input, passing the result to the next layer. This layer extracts features from the input image by sliding a convolution filter across the input to produce a feature map.
2) **Pooling Layer**: Performs down-sampling operations to reduce the dimensionality of the feature map, thus decreasing the computational power required and mitigating overfitting. Max pooling is a common approach used in CNNs.
3) **Fully Connected Layer**: After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer.
4) **Classification Layer**: The final layer uses a softmax activation function to output a probability distribution over the classes.

2) Implementation for Listener Classification: Our implementation of 2D CNN involves the following steps:

- Using 32x32 grayscale images as a representation of the spectrograms, where the input values are normalized between 0 and 1. One-hot encoding is applied to all class labels.
- The architecture comprises two convolution blocks. Each block contains a convolutional layer with a ReLU activation function, followed by a max-pooling layer that reduces the size by a factor of 2. The convolution layers use a kernel size of 5×5 with "same" padding to preserve the spatial dimensions of the input.
- The network then flattens the output and passes it through a dense layer with 512 units, followed by a ReLU activation and a dropout layer. The dropout layer, with a rate of 0.25, randomly sets input units to 0 at each step during training, which helps prevent overfitting.
- The final classification layer employs a softmax activation function to output the probability distribution over the classes. Additionally, L2 regularization with a factor of 0.0001 is used in the dense layer to further mitigate overfitting.
- The total number of learnable parameters in this architecture is approximately 1.3 million.

ReLU (Rectified Linear Unit) activation function is used for introducing non-linearity in the model, making it capable of learning more complex patterns. Softmax activation in the final layer is crucial for multi-class classification, as it outputs a probability distribution over the classes. L2 regularization penalizes large weights in the model, encouraging simpler models and thus reducing overfitting.

3) Training Process: The model is trained using the Adam optimizer and categorical cross-entropy loss function. We conduct training with a batch size of 40 for 500 epochs, ensuring the best model is saved and evaluated based on performance metrics. This fixed architecture is maintained across all multi-resolution spectrogram experiments to ensure consistency in the evaluation.

C. 1D Convolutional Neural Networks (CNN)

1) Overview of 1D CNN: 1D Convolutional Neural Networks (1D CNNs) are a variant of convolutional neural networks used for sequences or temporal data. Unlike 2D CNNs, which are typically used for image data and apply filters across two-dimensional space, 1D CNNs apply convolution operations along a single dimension. This makes them particularly well-suited for time-series data, audio signals, and any other type of sequential data.

In a 1D CNN, convolution layers slide along the temporal axis, extracting features from local segments of the data. This approach is efficient for capturing the temporal dynamics of the sequence.

2) Application in Signal Classification: Our application of 1D CNNs focuses on classifying various forms of signals, including:

- Raw signals
- Preprocessed signals with padding
- Preprocessed signals without padding
- Amplitude spectrum (frequency domain representation)

Each dataset undergoes one-hot encoding for class labels to facilitate classification. The 1D CNN architectures for these datasets consist of two convolution blocks. Each block comprises a convolutional layer followed by a ReLU activation function and a max-pooling layer, which reduces the dimensionality by half.

After the convolution blocks, the network structure includes:

- A flattening layer, transforming the output of the convolution blocks into a 1D feature vector.
- A dense layer with a ReLU activation function.
- A dropout layer to prevent overfitting by randomly setting input units to 0 at each step during training.
- A final classification layer with a softmax activation function, providing a probability distribution over the classes.

The kernel size for all convolutional layers is set to 32 with "same" padding, ensuring that the output size is equal to the input size. This padding strategy allows the network to learn features from the entire input sequence without losing information at the edges.

3) Training Process: For training the 1D CNN models, we use the RMSProp optimizer along with the categorical cross-entropy loss function. The training is conducted with a batch size of 40 for a total of 500 epochs. Throughout the training process, the best-performing model is saved and evaluated. This consistent approach across all datasets ensures a fair and comprehensive evaluation of the 1D CNN's ability to classify different types of signal data.

D. Evaluation Metrics

In our study, we employ several standard evaluation metrics to assess the performance of our classification models. These metrics include accuracy, precision, recall, F1-score, and the confusion matrix. Each metric provides unique insights into the effectiveness of our models.

1) Accuracy: Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observations to the total observations:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{5}$$

- **Advantages**: Provides a quick overview of model performance.
- **Disadvantages**: Can be misleading in imbalanced datasets.

2) Precision: Precision is the ratio of correctly predicted positive observations to the total predicted positive observations:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{6}$$

- **Advantages**: Highly useful in cases where False Positives are a larger concern than False Negatives.
- **Disadvantages**: Does not consider False Negatives.

3) Recall (Sensitivity): Recall is the ratio of correctly predicted positive observations to all observations in actual class:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{7}$$

- **Advantages**: Essential in cases where False Negatives are more significant than False Positives.
- **Disadvantages**: Does not consider False Positives.

4) F1-Score: The F1-score is the weighted average of Precision and Recall. Therefore, this score takes both False Positives and False Negatives into account:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{8}$$

The F1 score is a more balanced measure than accuracy, especially in imbalanced datasets.

5) Confusion Matrix: The confusion matrix is a table used to describe the performance of a classification model on a set of test data for which the true values are known:

- **True Positives (TP)**: Correctly predicted positive observations.
- **True Negatives (TN)**: Correctly predicted negative observations.
- **False Positives (FP)**: Incorrectly predicted positive observations.
- **False Negatives (FN)**: Incorrectly predicted negative observations.

While it can be more complex to interpret, especially with multiple classes, confusion matrices are advantageous as they provide a detailed analysis of the model's performance.

IV. Results

A. Support Vector Machine

We employ three SVM models: linear, polynomial with degree set to 3, and RBF. The C factor in all three SVM models is set to 1. Gamma is scaled based on the statistical properties of the input data. We train each SVM on the following window size and overlap values:

- Window size: 256 - Overlaps: 8, 64, 128, 250
- Window size: 512 - Overlaps: 0, 256, 511

We report accuracy, precision, recall, and F1 scores on Test/Retest and Retest/Test. Then, we combine both metrics using the F1 score and report it in our results. We believe this is a better way to compare the overall performance of the various methods. Figures 9 and 10 show the performance of the linear SVM using window sizes 256 and 512, respectively, with the best-performing model achieving a combined score of 86% accuracy when trained on window size 512 and overlap 511. Figures 11 and 12 show the performance of the polynomial SVM using window sizes 256 and 512, respectively, with the best-performing model achieving 81% accuracy with similar spectrogram resolution as the linear SVM. Finally, Figures 13 and 14 show the results obtained for the SVM with RBF kernel. The best-performing SVM with RBF kernel has an accuracy of 88%, outperforming both linear and polynomial SVMs. Furthermore, after inspecting the SVM experiments, we perceive an increase in classification accuracy proportional to the increases in overlap and spectrogram resolution.

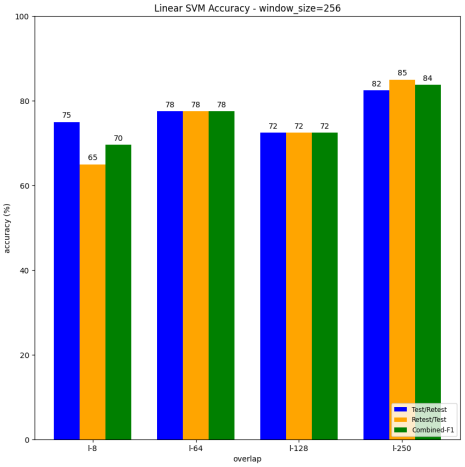

Fig. 9: The results of the linear SVM on four different overlap sizes for window size 256. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

Table II compares the performance of three best three SVM models on four metrics: accuracy, precision, recall, and F1-score, showing that the SVM with RBF kernel performs best in the listener classification task.

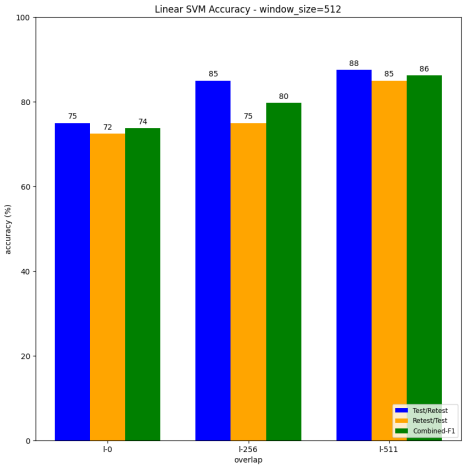

Fig. 10: The results of the linear SVM on four different overlap sizes for window size 512. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

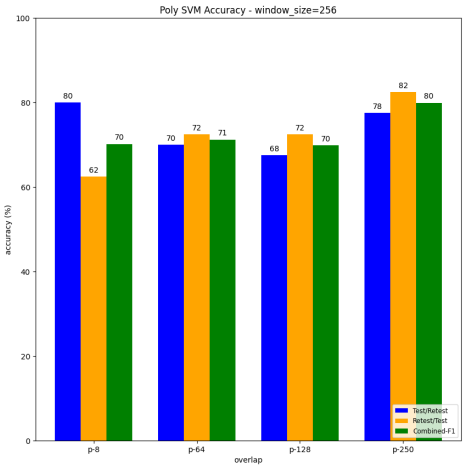

Fig. 11: The results of the polynomial SVM on four different overlap sizes for window size 256. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

B. 2D CNN

We train the aforementioned 2D CNN architecture on the 2D grayscale image representation of the spectrograms for 500 epochs and choose the best checkpoint to report our results. Figures 15 and 16 show the accuracy of the trained models on varying overlap values with window sizes 256 and 512, respectively. The best-performing model is achieving 74% combined accuracy and 77% on Test/Retest when trained on spectrograms generated with window size 512 and overlap

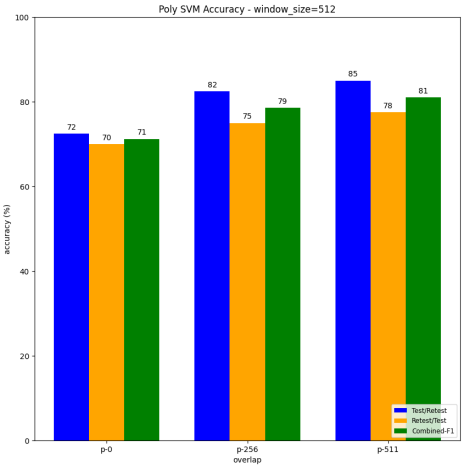

Fig. 12: The results of the polynomial SVM on four different overlap sizes for window size 512. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

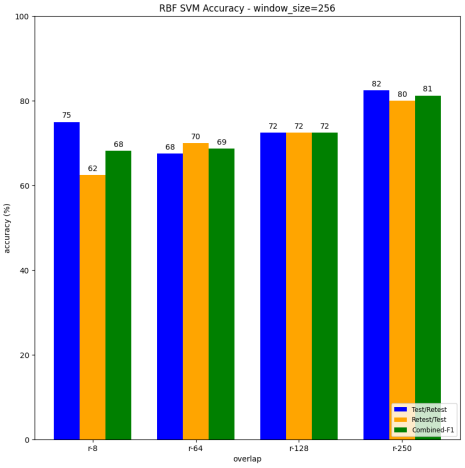

Fig. 13: The results of the RBF SVM on four different overlap sizes for window size 256. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

256. The overall performance of these models is lower than all three SVM models. Moreover, increasing the spectrogram's resolution does not have a direct impact on the classification accuracy. Table III compares the performance of the best SVM model (RBF) and the best 2D CNN model on four different metrics: accuracy, precision, recall, and F1 score. An SVM with RBF kernel trained on spectrograms with resolution as (w=512, o=511), consistently outperforms a 2D CNN trained on grayscale images of spectrograms with resolution (w=512,

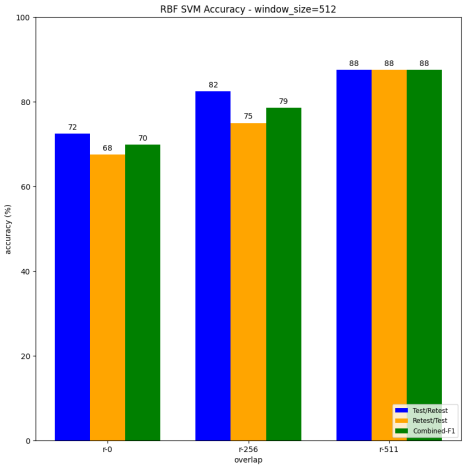

Fig. 14: The results of the RBF SVM on four different overlap sizes for window size 512. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

TABLE II: Performance Metrics for SVM with Different Kernels and Window Size = 512, Overlap = 511. The metric 'combined' is the F1 score calculated between the T/R and the R/T metrics.

Method	**Acc (%)**		**P (%)**		**R (%)**		**F1 (%)**	
	T/R	R/T	T/R	R/T	T/R	R/T	T/R	R/T
Linear	87.5	85.0	86.6	80.0	87.5	86.0	85.5	81.3
combined	86.2		83.2		86.8		83.3	
Poly	85.0	77.6	85.8	75.8	85.0	77.5	82.6	73.5
combined	81.1		80.5		81.1		77.8	
RBF	87.5	87.5	81.6	90.8	87.5	87.5	83.3	86.8
combined	87.5		86.0		87.5		85.0	

o=256).

TABLE III: Performance Metrics for SVM RBF (w=512, o=511) and 2D CNN (w=512, o=256). The metric 'combined' is the F1 score calculated between the T/R and the R/T metrics.

Method	**Acc (%)**		**P (%)**		**R (%)**		**F1 (%)**	
	T/R	R/T	T/R	R/T	T/R	R/T	T/R	R/T
SVM RBF	87.5	87.5	81.6	90.8	87.5	87.5	83.3	86.8
combined	87.5		86.0		87.5		85.0	
2D CNN	77.5	70.0	75.0	67.8	77.5	70.0	74.2	65.5
combined	73.6		71.2		73.6		69.6	

C. 1D CNN

We train the 1D CNN architectures on raw signals, pre-processed signals (with and without padding), and amplitude spectra (frequency domain). With this set of experiments, we aim to assess the efficacy of classifying listeners based on signals (time and frequency domain separately) compared to the spectrograms (time-frequency). Figure 17 shows the accuracy of the four aforementioned datasets. The best-performing model so far is the 1D CNN trained on raw signals, achieving

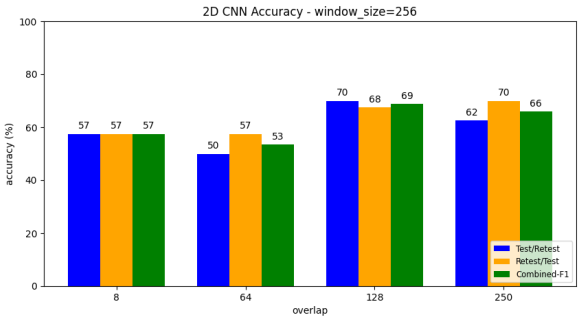

Fig. 15: The results of the 2D CNN on four different overlap sizes for window size 256. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

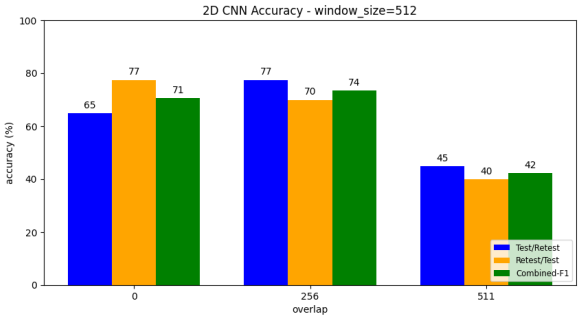

Fig. 16: The results of the 2D CNN on four different overlap sizes for window size 512. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

an 87% combined score, which is on par with the best SVM and outperforms the other 1D and 2D CNNs. Other 1D CNNs achieve better results compared to 2D CNNs trained on spectrograms, but fall short compared to the performance of SVMs.

Building on the premise of utilizing raw signals directly for listeners classification, we trained three SVMs specifically for this task. We trained an SVM with RBF kernel on raw data, which achieved a combined score of 75%, compared to 87% achieved by the 1D CNN, with a difference of more than 12% in classification accuracy.

D. Comparison

Table IV shows a quantitative comparison between the best models in all three categories: SVM, 2D, and 1D CNNs. Both SVM with RBF kernel and 1D CNN trained on raw signals are on par in all four metrics: accuracy, precision, recall, and F1-score.

V. Discussion

In the field of biometric classification, the effectiveness and flexibility of machine learning models are critical. Our

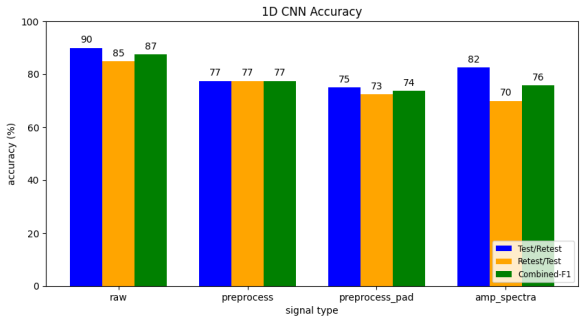

Fig. 17: The results of the 1D CNN on four different datasets: raw, preprocessing with and without padding, and amplitude spectrum. Blue bar: Test/Retest accuracy. Yellow bar: Retest/Test accuracy. Green bar: F1 score that combines the first two scores.

TABLE IV: Performance Metrics for SVM RBF (w=512, o=511), 2D CNN (w=512, o=256), and 1D CNN (raw signal). The metric 'combined' is the F1 score calculated between the T/R and R/T metrics.

Method	**Acc (%)**		**P (%)**		**R (%)**		**F1 (%)**	
	T/R	R/T	T/R	R/T	T/R	R/T	T/R	R/T
SVM RBF	87.5	87.5	81.6	90.8	87.5	87.5	83.3	86.8
combined	87.5		86.0		87.5		85.0	
2D CNN	77.5	70.0	75.0	67.8	77.5	70.0	74.2	65.5
combined	73.6		71.2		73.6		69.6	
1D CNN	90.0	85.0	88.3	85.0	90.0	85.0	87.7	82.3
combined	87.4		86.6		87.42		84.9	

research provides insights into how these models perform when used with auditory signals.

Support Vector Machines (SVM) have proven to be reliable when trained with spectrograms. A key finding is the positive relationship between the SVM's performance and an increase in the overlap size, which relates to better resolution in the spectrogram. This larger overlap introduces more redundant information, which appears to improve the SVM's ability to tell listeners apart by offering a broader range of features to learn from. In particular, the Radial Basis Function (RBF) kernel has performed better than linear and polynomial kernels. The RBF kernel's capability to handle complex, non-linear patterns makes it a strong choice for classifying spectrogram data.

The use of 2D Convolutional Neural Networks (2D CNNs) hasn't produced the high results we expected. This might be because images and spectrograms are fundamentally different. Frequencies in spectrograms often spread out over the entire plane, unlike features in images that tend to cluster together. This makes it hard for 2D CNNs to pick up on these patterns. Also, the detailed time-frequency patterns found in Frequency Following Response (FFR) spectrograms, which show subtle brain activity, can be hard for 2D CNNs to detect.

On the other hand, 1D Convolutional Neural Networks (1D CNNs) have done well with both time- and frequency-domain signals. Unlike SVMs, 1D CNNs don't need a preprocessing

step to create spectrograms and can work directly with raw signals. This keeps all the original information, like phase and amplitude, which are important for a full understanding of auditory responses. 1D CNNs are good at finding local patterns in time within signals and can build up layers of more complex feature representations. However, it's important to note that training CNNs takes longer than training SVMs due to their more detailed structure. The ability of 1D CNNs to work with the complete raw signal data, including phase information that's lost when making spectrograms, makes them a powerful tool for classifying auditory signals where keeping all the original details is important. Additionally, after conducting an experiment with SVM on raw data, it is evident that 1D CNNs are more robust and powerful when extracting features from raw signal information.

In sum, choosing the right model for listener classification depends on the specific data and available computing power. While SVMs are a consistent option for working with spectrograms, analyzing raw signals with 1D CNNs opens up possibilities for more accurate listener classification by using all the available auditory information.

VI. Conclusion

This study has thoroughly investigated the use of machine learning models for classifying listeners, carrying out a total of 21 experiments with Support Vector Machines (SVM), 7 with 2D Convolutional Neural Networks (CNNs), and 4 with 1D CNNs. Our results show that increasing the window size and overlap generally leads to better classification, which confirms the importance of higher resolution in analyzing auditory signals. Specifically, the SVM with the Radial Basis Function (RBF) kernel proved to be the most suitable for spectrogram data, likely due to its effectiveness in handling the complex patterns found in auditory signals. On the other hand, 2D CNNs did not perform as well, which could be due to the nature of spectrograms, where key features are not always localized like they are in images. In contrast, 1D CNN models were quite successful in classifying both time-domain and frequency-domain signals. Remarkably, a 1D CNN working with raw Frequency Following Response (FFR) signals matched the performance of the RBF kernel SVM, showcasing the benefits of analyzing signals directly in their time domain for auditory biometrics. Following an examination where SVMs were applied to raw data, it has become clear that 1D CNNs display greater strength and capability in extracting meaningful features directly from raw signal inputs. When applying these findings to real-world situations, it's important to weigh the balance between accuracy and the time it takes for the model to make its predictions, as quick responses are often crucial.

A. Future Work

Moving forward, there are several ways to build on the work of this study to make these models even better. The dataset we used was relatively small, which was enough for our initial tests but not for wider applications. Future studies should try to use bigger datasets to help the models work well in more general situations. Data augmentation could also help make the models less likely to overfit the training data, which would make their predictions more reliable. Further research should also look into refining the models with a broader range of architectural tweaks and tuning of model settings to find the right balance between accuracy and computational demands. Exploring methods like unsupervised and semi-supervised learning might also yield good results, particularly when there isn't much-labeled data available. In sum, there's a lot of promise for future work in machine learning for listener classification. Building on the findings of this study and considering the next steps we've outlined, we can look forward to the development of more advanced and trustworthy systems for auditory biometric identification.

References

[1] R. Sun, "Classification of Frequency Following Responses to English Vowels in a Biometric Application," Thesis, 2020.

[2] Borzou et al., "Machine Learning Based Listener Classification and Authentication Using Frequency Following Responses to English Vowels for Biometric Applications," PhD Thesis, 2023.

[3] Yang et al., "Convolutional Neural Network Based Side-Channel Attacks in Time-Frequency Representations," in *International Conference on Smart Card Research and Advanced Applications*, 2019.

[4] Khan et al., "CNN-XGBoost Fusion-Based Affective State Recognition Using EEG Spectrogram Image Analysis," 2022.

[5] B. Heffernan, "Characterization and Classification of the Frequency Following Response to Vowels at Different Sound Levels in Normal Hearing Adults," PhD Thesis, 2019.

[6] D. Rothmann, "What's Wrong with CNNs and Spectrograms for Audio Processing?" *Towards Data Science*, 2018. Accessed: Nov. 2023. [Online]. Available: https://towardsdatascience.com/whats-wrong-with-cnns-and-spectrograms-for-audio-processing-d3b8a1022a4e.

APPENDIX A
CODE

Here is the code used in our experiments:

Listing 1: Python code preprocessing - concatenating the vowels - creating spectrogram images - saving the NPY and PNG data

```
import itertools
import os

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from PIL import Image
from scipy import signal
from scipy.fft import fft
from scipy.ndimage import zoom


def concatenate_vowels(df):
    dataset = {} # Dictionary to hold the data for
        each subject

    # Iterate through each unique subject
    for subject in df["Subject"].unique():
        dataset[subject] = [] # List to hold the numpy
            arrays for this subject

        # Since we know there are 8 occurrences for
            each vowel, we loop 8 times
        for i in range(8):
            # Initialize a list to hold the
                concatenated arrays for this iteration
            concatenated_arrays = []

            # Iterate through each vowel and
                concatenate the ith occurrence
            for vowel in ["a␣vowel", "e␣vowel", "n␣
                vowel", "u␣vowel"]:
                # Filter the DataFrame for the current
                    subject and vowel, and get the ith
                    occurrence
                row_data = df[(df["Subject"] == subject)
                     & (df["Vowel"] == vowel)].iloc[
                    i, :1024
                ]

                # Append the row data to the
                    concatenated arrays list
                concatenated_arrays.append(row_data.
                    values)

            if i == 6 or i == 7:
                # Concatenate along the first axis to
                    get a single array for this subject
                    and iteration
                dataset[subject].append(
                    np.concatenate(concatenated_arrays).
                        astype(np.float32)
                )

    return dataset


# Function to preprocess a signal
def preprocess_signal(input_signal, sampling_rate
    =9606):
    # Detrend (data in PKL file was already detrended
        )
    signal_detrended = input_signal # signal.detrend(
        input_signal)

    # Remove DC offset
    signal_zero_mean = signal_detrended - np.mean(
        signal_detrended)

    # Apply Hamming window
    hamming_window = signal.windows.hamming(len(
        signal_zero_mean))
    signal_windowed = signal_zero_mean *
        hamming_window

    # # Normalize the signal to have a maximum value
        of 1
    # signal_normalized = signal_windowed / np.max(np
        .abs(signal_windowed))

    # Zero Padding
    original_length = len(input_signal)
    fft_length = int(sampling_rate / 1)

    # Calculate the padding length
    padding_length = fft_length - original_length
    if padding_length < 0:
        padding_length = 0 # No padding needed if
            fft_length is shorter than the signal

    signal_padded = np.pad(signal_windowed, (0,
        padding_length), "constant")

    return signal_windowed, signal_padded


# Define a function to calculate the amplitude
    spectrum
def calculate_amplitude_spectrum(s, sampling_rate,
    cutoff=-1):
    # Perform the Fourier Transform
    fft_result = fft(s)

    # Normalize the FFT result
    fft_normalized = fft_result / len(s)

    # Calculate the amplitude spectrum
    amplitude_spectrum = np.abs(fft_normalized)

    # Only take the first half of the spectrum (
        positive frequencies)
    half_spectrum = amplitude_spectrum[: len(
        amplitude_spectrum) // 2]

    # Create a frequency vector
    freq_vector = np.linspace(0, sampling_rate / 2,
        len(half_spectrum))

    return freq_vector, half_spectrum[:cutoff]


def compute_spectogram_img(input_signal, window_size
    , overlap):
    fs = len(input_signal) / 0.4264

    # Calculate the STFT and the spectrogram
    frequencies, times, Sxx = signal.spectrogram(
        input_signal, fs=fs, window="hamming", nperseg
            =window_size, noverlap=overlap
    )

    # Convert the spectrogram to dB
    Sxx_dB = 10 * np.log10(Sxx)

    return frequencies, times, Sxx_dB


def compute_spectogram_npy(input_signal, window_size
    , overlap):
```

```
    fs = len(input_signal) / 0.4264

    # Calculate the STFT and the spectrogram
    frequencies, times, Sxx = signal.spectrogram(
        input_signal, fs=fs, window="hamming", nperseg
            =window_size, noverlap=overlap
    )

    # Convert the spectrogram to dB
    Sxx_dB = 10 * np.log10(Sxx)

    # Find the index of the frequency that is just
        above 1300 Hz
    idx = np.where(frequencies <= 1300)[0][-1]

    # Slice the Sxx_dB array to include only the
        frequencies up to 1300 Hz
    Sxx_dB_limited = Sxx_dB[: idx + 1, :]

    return Sxx_dB_limited


def save_subject_arrays(efr_data, dataset_name):
    # Create a directory for the dataset if it doesn'
        t exist
    if not os.path.exists(dataset_name):
        os.makedirs(dataset_name)

    spectogram_map = {
        256: [8, 64, 128, 250],
        512: [0, 256, 511],
    }

    for subject_id, signals in efr_data.items():
        for i, input_signal in enumerate(signals):
            # Define the filename with the dataset name
                , subject, and iteration
            aenu_filename = f"{dataset_name}/{
                subject_id}_aenu_{i}.npy"
            # print(aenu_filename)
            print("input_signal.shape", input_signal.
                shape)
            np.save(aenu_filename, input_signal)

            preprocessed_filename = f"{dataset_name}/{
                subject_id}_preprocessed_{i}.npy"
            # print(aenu_filename)
            preprocessed_signal,
                preprocessed_signal_padded =
                preprocess_signal(
               input_signal, sampling_rate=9606
            )
            print("preprocessed_signal.shape",
                preprocessed_signal.shape)
            np.save(
               preprocessed_filename,
               preprocessed_signal,
            )

            preprocessed_padded_filename = (
               f"{dataset_name}/{subject_id}
                   _preprocessed_padded_{i}.npy"
            )
            # print(aenu_filename)
            print("preprocessed_signal_padded.shape",
                preprocessed_signal_padded.shape)
            np.save(
               preprocessed_padded_filename,
               preprocessed_signal_padded,
            )

            ampspectra_filename = f"{dataset_name}/{
                subject_id}_ampspectra_{i}.npy"
            # print(ampspectra_filename)
            ampspectra = calculate_amplitude_spectrum(
               preprocessed_signal_padded,
                   sampling_rate=9606, cutoff=1300
            )[1]
            print("ampspectra.shape", ampspectra.shape)
            np.save(ampspectra_filename, ampspectra)

            for window_size, overlaps in spectogram_map
                .items():
               for overlap in overlaps:
                  frequencies, times, Sxx_dB =
                      compute_spectogram_img(
                     preprocessed_signal, window_size,
                         overlap
                  )
                  spectogram_filename = f"{dataset_name
                      }/{subject_id}_spectogram_{i}_{
                      window_size}_{overlap}.png"
                  # print(spectogram_filename)
                  # np.save(spectogram_filename,
                      spectogram)

                  # Desired pixel size
                  pixel_size = 32
                  # Choose a DPI (could be any value,
                      but higher DPI means higher
                      resolution)
                  dpi = 512
                  # Calculate the figsize in inches
                  figsize_inch = pixel_size / dpi
                  fig, ax = plt.subplots(figsize=(
                      figsize_inch, figsize_inch))
                  plt.pcolormesh(
                     times, frequencies, Sxx_dB,
                         shading="nearest"
                  ) # Using 'nearest' for a discrete
                      look
                  plt.ylim(0, 1300)
                  plt.axis("off")

                  # Remove padding and margins around
                      the plot
                  plt.margins(0, 0)
                  ax.set_frame_on(False)

                  # Adjust the layout
                  plt.tight_layout(pad=0)

                  # To save the figure without white
                      space
                  fig.savefig(
                     spectogram_filename, bbox_inches="
                         tight", pad_inches=0, dpi=dpi
                  )

                  np.save(
                     f"{dataset_name}/{subject_id}
                         _spectogram_{i}_{window_size}_
                         {overlap}.npy",
                     compute_spectogram_npy(
                        preprocessed_signal,
                            window_size, overlap
                     ),
                  )


df = pd.read_pickle("study2DataFrame.pkl")

all_subjects = df["Subject"].unique()

# Remove rows where 'Avg_Type' is 'EFR'
# df_filtered = df[df['Avg_Type'] != 'EFR']
df_filtered = df[df["Avg_Type"] != "FFR"]
```

```
# Split the DataFrame based on the 'Condition'
    column
df_test = df_filtered[df_filtered["Condition"] == "
    test"]
df_retest = df_filtered[df_filtered["Condition"] ==
    "retest"]

test_dataset = concatenate_vowels(df_test)
retest_dataset = concatenate_vowels(df_retest)

# You would call the function like this:
save_subject_arrays(test_dataset, "test")
save_subject_arrays(retest_dataset, "retest")
```

Listing 2: Python code for training SVM on spectorgrams

```
import json

import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

from data_utils import flatten_data, prepare_data,
    shuffle_data


def train_svm(train_set, test_set):
   X_train, y_train = prepare_data(train_set)
   X_test, y_test = prepare_data(test_set)

   print("X_train.shape", X_train.shape)
   print("y_train.shape", y_train.shape)
   print("X_test.shape", X_test.shape)
   print("y_test.shape", y_test.shape)

   X_train_flat = flatten_data(X_train)
   X_test_flat = flatten_data(X_test)

   # Convert one-hot encoded labels to class indices
   y_train_idx = np.argmax(y_train, axis=1)
   y_test_idx = np.argmax(y_test, axis=1)

   X_train_flat, y_train_idx = shuffle_data(
       X_train_flat, y_train_idx)

   # Linear Kernel
   svm_linear = SVC(kernel="linear")

   # Polynomial Kernel
   svm_poly = SVC(
      kernel="poly", degree=3
   ) # degree is a hyperparameter for polynomial
       kernel

   # RBF Kernel
   svm_rbf = SVC(kernel="rbf", gamma="scale")

   svm_linear.fit(X_train_flat, y_train_idx)
   svm_poly.fit(X_train_flat, y_train_idx)
   svm_rbf.fit(X_train_flat, y_train_idx)

   # Making predictions and evaluating accuracy for
       each model
   y_pred_linear = svm_linear.predict(X_test_flat)
   y_pred_poly = svm_poly.predict(X_test_flat)
   y_pred_rbf = svm_rbf.predict(X_test_flat)

   accuracy_linear = accuracy_score(y_test_idx,
       y_pred_linear)
   accuracy_poly = accuracy_score(y_test_idx,
       y_pred_poly)
   accuracy_rbf = accuracy_score(y_test_idx,
       y_pred_rbf)

   print("-" * 50)
   print(f"Linear␣Kernel␣Accuracy:␣{accuracy_linear}
       ")
   print(f"Polynomial␣Kernel␣Accuracy:␣{
       accuracy_poly}")
   print(f"RBF␣Kernel␣Accuracy:␣{accuracy_rbf}")
   print("-" * 50)

   # save results in dict
   metrics = {
      "linear": accuracy_linear,
      "poly": accuracy_poly,
      "rbf": accuracy_rbf,
   }

   return metrics


NPY_DATA_DIR = "npy_datasets"
RESULTS_DIR = "svm_results"

spectogram_map = {
   256: [8, 64, 128, 250],
   512: [0, 256, 511],
}

results = {}

for window_size, overlaps in spectogram_map.items():
   for overlap in overlaps:
      print(f"window_size:␣{window_size},␣overlap:␣{
          overlap}")

      # Load the dataset
      test_dataset = np.load(
         f"{NPY_DATA_DIR}/test_{window_size}_{
             overlap}.npy", allow_pickle=True
      )
      retest_dataset = np.load(
         f"{NPY_DATA_DIR}/retest_{window_size}_{
             overlap}.npy", allow_pickle=True
      )

      test_retest_metrics = train_svm(test_dataset,
          retest_dataset)
      retest_test_metrics = train_svm(retest_dataset
          , test_dataset)

      results[f"{window_size}_{overlap}"] = {
         "test_retest": test_retest_metrics,
         "retest_test": retest_test_metrics,
      }

# save results as pretty printed json
with open(f"{RESULTS_DIR}/svm_results.json", "w") as
     f:
   json.dump(results, f, indent=4)
```

Listing 3: Python code for training 2D CNN on spectorgrams images

```
import json
import random

import numpy as np
import tensorflow as tf
from tensorflow.keras.callbacks import
    ModelCheckpoint, ReduceLROnPlateau
from tensorflow.keras.layers import (
   Activation,
   Conv2D,
   Dense,
   Dropout,
   Flatten,
```

```
   MaxPooling2D,
)
from tensorflow.keras.models import Sequential
from tensorflow.keras.regularizers import l2

from data_utils import normalize_min_max_v2,
    prepare_data, shuffle_data

RAND_SEED = 1
np.random.seed(RAND_SEED)
tf.random.set_seed(RAND_SEED)
random.seed(RAND_SEED)


def train_cnn(train_set, test_set, window_size,
    overlap, exp_name):
   X_train, y_train = prepare_data(train_set)
   X_test, y_test = prepare_data(test_set)

   X_train = normalize_min_max_v2(X_train, 0, 1)
   X_test = normalize_min_max_v2(X_test, 0, 1)

   # reshape only when grayscale
   X_train = X_train.reshape(X_train.shape[0],
       X_train.shape[1], X_train.shape[2], 1)
   X_test = X_test.reshape(X_test.shape[0], X_test.
       shape[1], X_test.shape[2], 1)

   DROPOUT_RATE = 0.15
   KERNEL_SIZE = (5, 5)
   L2_REGULARIZATION = 0.0001

   model = Sequential(
      [
         Conv2D(
            20,
            KERNEL_SIZE,
            padding="same",
            kernel_regularizer=l2(L2_REGULARIZATION)
                ,
            input_shape=(X_train.shape[1], X_train.
                shape[2], X_train.shape[3]),
         ),
         Activation("relu"),
         MaxPooling2D(
            (2, 2),
         ),
         # Dropout(DROPOUT_RATE),
         Conv2D(
            40,
            KERNEL_SIZE,
            padding="same",
            kernel_regularizer=l2(L2_REGULARIZATION)
                ,
         ),
         Activation("relu"),
         MaxPooling2D((2, 2)),
         # Dropout(DROPOUT_RATE),
         # Conv2D(
         # 80,
         # KERNEL_SIZE,
         # padding="same",
         # kernel_regularizer=l2(L2_REGULARIZATION),
         # ),
         # Activation("relu"),
         # MaxPooling2D((2, 2)),
         # # Dropout(DROPOUT_RATE),
         Flatten(),
         Dense(
            512,
            # kernel_regularizer=l2(
                L2_REGULARIZATION)
         ),
         Activation("relu"),
         Dropout(DROPOUT_RATE),
         Dense(20, activation="softmax"),
      ]
   )
   model.summary()

   model.compile(
      optimizer="adam", loss="
          categorical_crossentropy", metrics=["
          accuracy"]
   )

   # Define the path where you want to save the best
       checkpoint
   checkpoint_filepath = (
      f"results/cnn2d_models/cnn2d_{window_size}_{
          overlap}_{exp_name}.h5"
   )
   # Define the ModelCheckpoint callback
   model_checkpoint = ModelCheckpoint(
      checkpoint_filepath,
      monitor="val_accuracy", # Choose the metric to
           monitor (e.g., validation loss)
      mode="max", # 'min' for metrics like
          validation loss, 'max' for accuracy, etc.
      save_best_only=True, # Save only the best
          model checkpoint
      save_weights_only=False, # Save the entire
          model, not just weights
      verbose=1, # Print messages about checkpoint
          saving
   )

   # Learning rate scheduler
   lr_scheduler = ReduceLROnPlateau(
      monitor="loss", min_delta=0.01, factor=0.6,
          patience=1, min_lr=0.0000001
   )

   metrics = {
      "train_losses": [],
      "test_losses": [],
      "train_accuracies": [],
      "test_accuracies": [],
   }

   for epoch in range(500):
      X_train, y_train = shuffle_data(X_train,
          y_train)

      history = model.fit(
         X_train,
         y_train,
         epochs=1,
         batch_size=40,
         verbose=0,
         callbacks=[lr_scheduler, model_checkpoint],
         validation_data=(X_test, y_test),
      ) # Train for one epoch at a time

      # Store training metrics
      metrics["train_losses"].append(history.history
          ["loss"][0])
      metrics["train_accuracies"].append(history.
          history["accuracy"][0])

      # Store test/validation metrics
      metrics["test_losses"].append(history.history[
          "val_loss"][0])
      metrics["test_accuracies"].append(history.
          history["val_accuracy"][0])

   # save metrics in json file
   with open(
```

```
      f"results/cnn2d_models/cnn2d_{window_size}_{
          overlap}_{exp_name}.json", "w"
   ) as fp:
      json.dump(metrics, fp)


NPY_DATA_DIR = "npy_datasets"

spectogram_map = {
   256: [8, 64, 128, 250],
   512: [0, 256, 511],
}

window_size = 256
overlap = 8

for window_size, overlap_list in spectogram_map.
    items():
   for overlap in overlap_list:
      # Load the dataset
      test_dataset = np.load(
         f"{NPY_DATA_DIR}/test_{window_size}_{
             overlap}_png.npy", allow_pickle=True
      )
      retest_dataset = np.load(
         f"{NPY_DATA_DIR}/retest_{window_size}_{
             overlap}_png.npy", allow_pickle=True
      )

      test_retest_metrics = train_cnn(
         test_dataset, retest_dataset, window_size,
             overlap, exp_name="test_retest"
      )
      retest_test_metrics = train_cnn(
         retest_dataset, test_dataset, window_size,
             overlap, exp_name="retest_test"
      )
```

Listing 4: Python code for training 1D CNN on raw data

```
import random
import numpy as np
import tensorflow as tf
import json

from tensorflow.keras.callbacks import
    ReduceLROnPlateau, ModelCheckpoint
from tensorflow.keras.layers import (
   Dense,
   Flatten,
   Activation,
   Conv1D,
   MaxPooling1D,
   Dropout
)
from tensorflow.keras.models import Sequential
from tensorflow.keras.regularizers import l2
from data_utils import prepare_data, shuffle_data


RAND_SEED = 1
np.random.seed(RAND_SEED)
tf.random.set_seed(RAND_SEED)
random.seed(RAND_SEED)


def train_cnn(train_set, test_set, exp_name):
   X_train, y_train = prepare_data(train_set)
   X_test, y_test = prepare_data(test_set)

   X_train = X_train.reshape(X_train.shape[0],
       X_train.shape[1], 1)
   X_test = X_test.reshape(X_test.shape[0], X_test.
       shape[1], 1)


KERNEL_SIZE = 32
L2_REGULARIZATION = 0.0001

model = Sequential(
   [
      Conv1D(
         8,
         KERNEL_SIZE,
         padding="same",
         kernel_regularizer=l2(L2_REGULARIZATION)
             ,
         input_shape=(X_train.shape[1], 1), #
             Input signal size is 1300, and 1
             channel (1D)
      ),
      Activation('relu'),
      MaxPooling1D(
         2, # Pooling size
      ),

      Conv1D(
         16,
         KERNEL_SIZE,
         padding="same",
         kernel_regularizer=l2(L2_REGULARIZATION)
             ,
      ),
      Activation('relu'),
      MaxPooling1D(
         2, # Pooling size
      ),

      Flatten(),

      Dense(128),
      Activation('relu'),
      Dropout(0.25),

      Dense(20, activation="softmax"),
   ]
)
model.summary()

model.compile(
   optimizer='rmsprop', loss="
       categorical_crossentropy", metrics=["
       accuracy"]
)

# Define the path where you want to save the best
    checkpoint
checkpoint_filepath = f'results/cnn1d_models/
    cnn1d_raw_{exp_name}.h5'
# Define the ModelCheckpoint callback
model_checkpoint = ModelCheckpoint(
   checkpoint_filepath,
   monitor='val_accuracy', # Choose the metric to
        monitor (e.g., validation loss)
   mode='max', # 'min' for metrics like
       validation loss, 'max' for accuracy, etc.
   save_best_only=True, # Save only the best
       model checkpoint
   save_weights_only=False, # Save the entire
       model, not just weights
   verbose=1, # Print messages about checkpoint
       saving
)

# Learning rate scheduler
lr_scheduler = ReduceLROnPlateau(monitor='loss',
    min_delta=0.01, factor=0.6, patience=1,
    min_lr=0.0000001)
```

```
    metrics = {
        'train_losses': [],
        'test_losses': [],
        'train_accuracies': [],
        'test_accuracies': []
    }

    for epoch in range(500):
        X_train, y_train = shuffle_data(X_train,
            y_train)

        history = model.fit(
            X_train,
            y_train,
            epochs=1,
            batch_size=40,
            verbose=0,
            callbacks=[
                lr_scheduler,
                model_checkpoint
            ],
            validation_data=(X_test, y_test)
        )

        # Store training metrics
        metrics['train_losses'].append(history.history
            ['loss'][0])
        metrics['train_accuracies'].append(history.
            history['accuracy'][0])

        # Store test/validation metrics
        metrics['test_losses'].append(history.history[
            'val_loss'][0])
        metrics['test_accuracies'].append(history.
            history['val_accuracy'][0])

    # save metrics in json file
    with open(f'results/cnn1d_models/cnn1d_raw_{
        exp_name}.json', 'w') as fp:
        json.dump(metrics, fp)


NPY_DATA_DIR = "npy_datasets"

# Load the dataset
test_dataset = np.load(
    f"{NPY_DATA_DIR}/test_aenu.npy", allow_pickle=
        True
)
retest_dataset = np.load(
    f"{NPY_DATA_DIR}/retest_aenu.npy", allow_pickle=
        True
)

test_retest_metrics = train_cnn(test_dataset,
    retest_dataset, exp_name="test_retest")
retest_test_metrics = train_cnn(retest_dataset,
    test_dataset, exp_name="retest_test")
```

Listing 5: Example Python code for evaluating SVM - generating bar plots - generating confusion matrices

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import json
import time

from data_utils import prepare_data, flatten_data,
    shuffle_data
from sklearn.metrics import confusion_matrix,
    precision_score, recall_score, f1_score,
    accuracy_score
from sklearn.svm import SVC

def f1_metric(val_1, val2):
    if val_1 + val2 == 0:
        return 0 # To avoid division by zero
    return 2 * (val_1 * val2) / (val_1 + val2)

PATH = "results/svm"
NPY_DATA_DIR = "npy_datasets"

spectogram_map = {
    256: [8, 64, 128, 250],
    512: [0, 256, 511],
}

def train_svm(train_set, test_set, window_size,
    overlap, exp_name):
    training_times = {}

    X_train, y_train = prepare_data(train_set)
    X_test, y_test = prepare_data(test_set)

    print(window_size, overlap, 'X_train.shape',
        X_train.shape)
    print(window_size, overlap, 'X_test.shape',
        X_test.shape)

    X_train_flat = flatten_data(X_train)
    X_test_flat = flatten_data(X_test)

    # Convert one-hot encoded labels to class indices
    y_train_idx = np.argmax(y_train, axis=1)
    y_test_idx = np.argmax(y_test, axis=1)

    print("y_train_idx", y_train_idx)
    print("y_test_idx", y_test_idx)

    X_train_flat, y_train_idx = shuffle_data(
        X_train_flat, y_train_idx)

    # Linear Kernel
    svm_linear = SVC(kernel="linear", gamma="scale")

    # Polynomial Kernel
    svm_poly = SVC(
        kernel="poly", degree=3, gamma="scale"
    ) # degree is a hyperparameter for polynomial
        kernel

    # RBF Kernel
    svm_rbf = SVC(kernel="rbf", gamma="scale")

    start_time = time.time()
    svm_linear.fit(X_train_flat, y_train_idx)
    linear_training_time = time.time() - start_time
    print(f"Linear␣SVM␣Training␣Time␣(W:{window_size
        }-O:{overlap}):␣{linear_training_time}␣
        seconds")

    start_time = time.time()
    svm_poly.fit(X_train_flat, y_train_idx)
    poly_training_time = time.time() - start_time
    print(f"Polynomial␣SVM␣Training␣Time␣(W:{
        window_size}-O:{overlap}):␣{
        poly_training_time}␣seconds")

    start_time = time.time()
    svm_rbf.fit(X_train_flat, y_train_idx)
    rbf_training_time = time.time() - start_time
    print(f"RBF␣SVM␣Training␣Time␣(W:{window_size}-O
        :{overlap}):␣{rbf_training_time}␣seconds")

    training_times = {
```

```
    'linear': linear_training_time,
    'poly': poly_training_time,
    'rbf': rbf_training_time
}

# Making predictions and evaluating accuracy for
    each model
y_pred_linear = svm_linear.predict(X_test_flat)
y_pred_poly = svm_poly.predict(X_test_flat)
y_pred_rbf = svm_rbf.predict(X_test_flat)

accuracy_linear = accuracy_score(y_test_idx,
    y_pred_linear) * 100
accuracy_poly = accuracy_score(y_test_idx,
    y_pred_poly) * 100
accuracy_rbf = accuracy_score(y_test_idx,
    y_pred_rbf) * 100

# For the linear SVM model
precision_linear = precision_score(y_test_idx,
    y_pred_linear, average='weighted') * 100
recall_linear = recall_score(y_test_idx,
    y_pred_linear, average='weighted') * 100
f1_linear = f1_score(y_test_idx, y_pred_linear,
    average='weighted') * 100

# For the polynomial SVM model
precision_poly = precision_score(y_test_idx,
    y_pred_poly, average='weighted') * 100
recall_poly = recall_score(y_test_idx,
    y_pred_poly, average='weighted') * 100
f1_poly = f1_score(y_test_idx, y_pred_poly,
    average='weighted') * 100

# For the RBF SVM model
precision_rbf = precision_score(y_test_idx,
    y_pred_rbf, average='weighted') * 100
recall_rbf = recall_score(y_test_idx, y_pred_rbf,
     average='weighted') * 100
f1_rbf = f1_score(y_test_idx, y_pred_rbf, average
    ='weighted') * 100

print("Linear SVM Metrics:")
print(f"Precision: {precision_linear}, Recall: {
    recall_linear}, F1 Score: {f1_linear}, 
    Accuracy: {accuracy_linear}")

print("\nPolynomial SVM Metrics:")
print(f"Precision: {precision_poly}, Recall: {
    recall_poly}, F1 Score: {f1_poly}, Accuracy: 
    {accuracy_poly}")

print("\nRBF SVM Metrics:")
print(f"Precision: {precision_rbf}, Recall: {
    recall_rbf}, F1 Score: {f1_rbf}, Accuracy: {
    accuracy_rbf}")

cm_linear = confusion_matrix(y_test_idx,
    y_pred_linear)
cm_poly = confusion_matrix(y_test_idx,
    y_pred_poly)
cm_rbf = confusion_matrix(y_test_idx, y_pred_rbf)

# for cm, acc, type in zip(
# [cm_linear, cm_poly, cm_rbf],
# [accuracy_linear, accuracy_poly, accuracy_rbf],
# ['linear', 'poly', 'rbf']
# ):
# sns.heatmap(cm, annot=True, fmt='d')
# plt.title(f'{type} - {window_size}_{overlap} -
    {exp_name} - {acc:.2f}%')
# plt.xlabel('Predicted')
# plt.ylabel('True')
# plt.savefig(f'{PATH}/plots/svm_{type}_{
    window_size}_{overlap}_{exp_name}.png')
# plt.show()

print("-" * 50)
print(f"Linear Kernel Accuracy: {accuracy_linear}
    ")
print(f"Polynomial Kernel Accuracy: {
    accuracy_poly}")
print(f"RBF Kernel Accuracy: {accuracy_rbf}")
print("-" * 50)

# save results in dict
metrics = {
    'accuracy': {
        'linear': accuracy_linear,
        'poly': accuracy_poly,
        'rbf': accuracy_rbf
    },
    'training_time': training_times,
    'precision': {
        'linear': precision_linear,
        'poly': precision_poly,
        'rbf': precision_rbf
    },
    'recall': {
        'linear': recall_linear,
        'poly': recall_poly,
        'rbf': recall_rbf
    },
    'f1': {
        'linear': f1_linear,
        'poly': f1_poly,
        'rbf': f1_rbf
    },
}

return metrics

results = {}

for window_size, overlap_list in spectogram_map.
    items():
    linear_tr_metrics = []
    linear_rt_metrics = []
    linear_names = []

    poly_tr_metrics = []
    poly_rt_metrics = []
    poly_names = []

    rbf_tr_metrics = []
    rbf_rt_metrics = []
    rbf_names = []

    metric = 'accuracy'

    for overlap in overlap_list:
        # Load the dataset
        test_dataset = np.load(
            f"{NPY_DATA_DIR}/test_{window_size}_{
                overlap}.npy", allow_pickle=True
        )
        retest_dataset = np.load(
            f"{NPY_DATA_DIR}/retest_{window_size}_{
                overlap}.npy", allow_pickle=True
        )

        test_retest_metrics = train_svm(test_dataset,
            retest_dataset, window_size, overlap, '
            test_retest')
        retest_test_metrics = train_svm(retest_dataset
            , test_dataset, window_size, overlap, '
            retest_test')
```

```
    results[f"{window_size}_{overlap}"] = {
        "test_retest": test_retest_metrics,
        "retest_test": retest_test_metrics,
    }

    linear_tr_metrics.append(test_retest_metrics[
        metric]['linear'])
    linear_rt_metrics.append(retest_test_metrics[
        metric]['linear'])
    linear_names.append(f'l-{overlap}')

    poly_tr_metrics.append(test_retest_metrics[
        metric]['poly'])
    poly_rt_metrics.append(retest_test_metrics[
        metric]['poly'])
    poly_names.append(f'p-{overlap}')

    rbf_tr_metrics.append(test_retest_metrics[
        metric]['rbf'])
    rbf_rt_metrics.append(retest_test_metrics[
        metric]['rbf'])
    rbf_names.append(f'r-{overlap}')

combined_linear_f1 = []
for tr_f1, rt_f1 in zip(linear_tr_metrics,
    linear_rt_metrics):
    combined_linear_f1.append(f1_metric(tr_f1,
        rt_f1))

combined_poly_f1 = []
for tr_f1, rt_f1 in zip(poly_tr_metrics,
    poly_rt_metrics):
    combined_poly_f1.append(f1_metric(tr_f1, rt_f1
        ))

combined_rbf_f1 = []
for tr_f1, rt_f1 in zip(rbf_tr_metrics,
    rbf_rt_metrics):
    combined_rbf_f1.append(f1_metric(tr_f1, rt_f1)
        )

for tr_metrics, rt_metrics, combined, names,
    svm_type in zip(
    [linear_tr_metrics, poly_tr_metrics,
        rbf_tr_metrics],
    [linear_rt_metrics, poly_rt_metrics,
        rbf_rt_metrics],
    [combined_linear_f1, combined_poly_f1,
        combined_rbf_f1],
    [linear_names, poly_names, rbf_names],
    ['Linear', 'Poly', 'RBF']
):
    n = len(tr_metrics) # Number of data pairs
    ind = np.arange(n) # The x locations for the
        groups
    width = 0.25 # Width of the bars

    fig, ax = plt.subplots(figsize=(10, 10))
    x_bars = ax.bar(ind - width, tr_metrics, width
        , label='Test/Retest', color='blue')
    y_bars = ax.bar(ind, rt_metrics, width, label=
        'Retest/Test', color='orange')
    z_bars = ax.bar(ind + width, combined, width,
        label='Combined-F1', color='green')

    for bar in x_bars + y_bars + z_bars:
        height = bar.get_height()
        ax.annotate(f'{height:.0f}', # Adjust
            format as needed
                xy=(bar.get_x() + bar.get_width()
                    / 2, height),
                xytext=(0, 3), # Offset
                textcoords='offset points',
                ha='center', va='bottom')
        ax.set_xlabel('overlap')
        ax.set_ylabel(metric + ' (%)')
        ax.set_title(f'{svm_type} SVM Accuracy -
            window_size={window_size}')
        ax.set_xticks(ind)
        ax.set_xticklabels(names)
        ax.set_ylim([0, 100])
        ax.legend(fontsize='small', loc='lower right')

        plt.savefig(f'{PATH}/plots/svm_{svm_type}_{
            window_size}_{metric}.png')
        plt.show()

# save results as pretty printed json
with open(f"{PATH}/svm_results.json", "w") as f:
    json.dump(results, f, indent=4)
```

Listing 6: Example Python code for evaluating 2D CNN - generating bar plots - generating confusion matrices

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import json

from tensorflow.keras.models import load_model
from data_utils import prepare_data,
    normalize_min_max_v2
from sklearn.metrics import precision_score,
    recall_score, f1_score, confusion_matrix

def f1_metric(val_1, val2):
    if val_1 + val2 == 0:
        return 0 # To avoid division by zero
    return 2 * (val_1 * val2) / (val_1 + val2)

PATH = 'results/cnn2d_models'

spectogram_map = {
    256: [8, 64, 128, 250],
    512: [0, 256, 511],
}


for window_size, overlap_list in spectogram_map.
    items():
    tr_metrics = []
    rt_metrics = []
    tr_names = []
    for overlap in overlap_list:
        tr_exp = f'cnn2d_{window_size}_{overlap}
            _test_retest'
        rt_exp = f'cnn2d_{window_size}_{overlap}
            _retest_test'
        # Specify the path to the saved h5 model file
        test_retest_path = f'{PATH}/{tr_exp}.h5'
        retest_test_path = f'{PATH}/{rt_exp}.h5'

        # Load the model
        test_retest_model = load_model(
            test_retest_path)
        retest_test_model = load_model(
            retest_test_path)

        print(test_retest_model.summary())

        NPY_DATA_DIR = "npy_datasets"

        test_dataset = np.load(
            f"{NPY_DATA_DIR}/test_{window_size}_{
                overlap}_png.npy", allow_pickle=True
        )
```

```
retest_dataset = np.load(
   f"{NPY_DATA_DIR}/retest_{window_size}_{
       overlap}_png.npy", allow_pickle=True
)

X_test, y_test = prepare_data(test_dataset)
X_retest, y_retest = prepare_data(
    retest_dataset)

X_test = normalize_min_max_v2(X_test, 0, 1)
X_retest = normalize_min_max_v2(X_retest, 0,
    1)

# reshape only when grayscale
X_test = X_test.reshape(X_test.shape[0],
    X_test.shape[1], X_test.shape[2], 1)
X_retest = X_retest.reshape(X_retest.shape[0],
     X_retest.shape[1], X_retest.shape[2], 1)

_, test_acc = test_retest_model.evaluate(
    X_retest, y_retest, verbose=0)
_, retest_acc = retest_test_model.evaluate(
    X_test, y_test, verbose=0)

test_acc = test_acc * 100
retest_acc = retest_acc * 100

tr_metrics.append(test_acc)
rt_metrics.append(retest_acc)
tr_names.append(f'{overlap}')

print('test_acc', test_acc)
print('retest_acc', retest_acc)

test_pred = test_retest_model.predict(X_retest
    )
test_pred_classes = np.argmax(test_pred, axis
    =1)
y_test = np.argmax(y_test, axis=1)
test_confusion_mtx = confusion_matrix(y_test,
    test_pred_classes)

# Precision, recall, and F1 score for
    test_retest_model
precision_test = precision_score(y_test,
    test_pred_classes, average='weighted')
recall_test = recall_score(y_test,
    test_pred_classes, average='weighted')
f1_score_test = f1_score(y_test,
    test_pred_classes, average='weighted')

print(f"{window_size}_{overlap} Test-Retest
    Model Metrics:")
print(f"Accuracy: {test_acc}")
print(f"Precision: {precision_test}")
print(f"Recall: {recall_test}")
print(f"F1 Score: {f1_score_test}")

retest_pred = retest_test_model.predict(X_test
    )
retest_pred_classes = np.argmax(retest_pred,
    axis=1)
y_retest = np.argmax(y_retest, axis=1)
retest_confusion_mtx = confusion_matrix(
    y_retest, retest_pred_classes)

# Precision, recall, and F1 score for
    retest_test_model
precision_retest = precision_score(y_retest,
    retest_pred_classes, average='weighted')
recall_retest = recall_score(y_retest,
    retest_pred_classes, average='weighted')
f1_score_retest = f1_score(y_retest,
    retest_pred_classes, average='weighted')

   print(f"\n{window_size}_{overlap} Retest-Test
       Model Metrics:")
   print(f"Accuracy: {retest_acc}")
   print(f"Precision: {precision_retest}")
   print(f"Recall: {recall_retest}")
   print(f"F1 Score: {f1_score_retest}")

   # sns.heatmap(test_confusion_mtx, annot=True,
       fmt='d')
   # plt.title(f'{window_size}_{overlap} - Test/
       Retest - {test_acc:.3f}%')
   # plt.xlabel('Predicted')
   # plt.ylabel('True')
   # plt.savefig(f'{PATH}/plots/cnn2d_{
       window_size}_{overlap}_test_retest.png')
   # plt.show()

   # sns.heatmap(test_confusion_mtx, annot=True,
       fmt='d')
   # plt.title(f'{window_size}_{overlap} - Retest
       /Test - {retest_acc:.3f}%')
   # plt.xlabel('Predicted')
   # plt.ylabel('True')
   # plt.savefig(f'{PATH}/plots/cnn2d_{
       window_size}_{overlap}_retest_test.png')
   # plt.show()

combined_f1 = []
for i in range(len(tr_metrics)):
   combined_f1.append(f1_metric(tr_metrics[i],
       rt_metrics[i]))

print(tr_metrics)
print(rt_metrics)

n = len(tr_metrics) # Number of data pairs
ind = np.arange(n) # The x locations for the
    groups
width = 0.25 # Width of the bars

fig, ax = plt.subplots(figsize=(10, 5))
x_bars = ax.bar(ind - width, tr_metrics, width,
    label='Test/Retest', color='blue')
y_bars = ax.bar(ind, rt_metrics, width, label='
    Retest/Test', color='orange')
z_bars = ax.bar(ind + width, combined_f1, width,
    label='Combined-F1', color='green')

for bar in x_bars + y_bars + z_bars:
   height = bar.get_height()
   ax.annotate(f'{height:.0f}', # Adjust format
       as needed
            xy=(bar.get_x() + bar.get_width() /
                2, height),
            xytext=(0, 3), # Offset
            textcoords='offset points',
            ha='center', va='bottom')

ax.set_xlabel('overlap')
ax.set_ylabel('accuracy (%)')
ax.set_title(f'2D CNN Accuracy - window_size={
    window_size}')
ax.set_xticks(ind)
ax.set_xticklabels(tr_names)
ax.set_ylim([0, 100])
ax.legend(fontsize='small', loc='lower right')

plt.savefig(f'{PATH}/plots/cnn2d_{window_size}
    _accuracy.png')
plt.show()
```

Listing 7: Example Python code for evaluating 1D CNN - generating bar plots - generating confusion matrices

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import json

from tensorflow.keras.models import load_model
from data_utils import prepare_data,
    normalize_min_max_v2
from sklearn.metrics import precision_score,
    recall_score, f1_score, confusion_matrix

def f1_metric(val_1, val2):
   if val_1 + val2 == 0:
      return 0 # To avoid division by zero
   return 2 * (val_1 * val2) / (val_1 + val2)

PATH = 'results/cnn1d_models'

tr_metrics = []
rt_metrics = []
combined_metrics = []
tr_names = []


tr_exp = 'cnn1d_raw_test_retest'
rt_exp = 'cnn1d_raw_retest_test'
# Specify the path to the saved h5 model file
raw_test_retest_path = f'{PATH}/{tr_exp}.h5'
raw_retest_test_path = f'{PATH}/{rt_exp}.h5'

# Load the model
raw_test_retest_model = load_model(
    raw_test_retest_path)
raw_retest_test_model = load_model(
    raw_retest_test_path)

print(raw_retest_test_model.summary())

NPY_DATA_DIR = "npy_datasets"

# Load the dataset
test_dataset = np.load(
   f"{NPY_DATA_DIR}/test_aenu.npy", allow_pickle=
       True
)
retest_dataset = np.load(
   f"{NPY_DATA_DIR}/retest_aenu.npy", allow_pickle=
       True
)

X_test, y_test = prepare_data(test_dataset)
X_retest, y_retest = prepare_data(retest_dataset)

print(X_test.shape)

X_test = X_test.reshape(X_test.shape[0], X_test.
    shape[1], 1)
X_retest = X_retest.reshape(X_retest.shape[0],
    X_retest.shape[1], 1)

_, test_acc = raw_test_retest_model.evaluate(
    X_retest, y_retest, verbose=0)
_, retest_acc = raw_retest_test_model.evaluate(
    X_test, y_test, verbose=0)

test_acc = test_acc * 100
retest_acc = retest_acc * 100

tr_metrics.append(test_acc)
rt_metrics.append(retest_acc)
combined_metrics.append(f1_metric(test_acc,
    retest_acc))
tr_names.append('raw')

print('test_acc', test_acc)
print('retest_acc', retest_acc)

test_pred = raw_test_retest_model.predict(X_retest)
test_pred_classes = np.argmax(test_pred, axis=1)
y_test = np.argmax(y_test, axis=1)
test_confusion_mtx = confusion_matrix(y_test,
    test_pred_classes)

# Precision, recall, and F1 score for
    test_retest_model
precision_test = precision_score(y_test,
    test_pred_classes, average='weighted')
recall_test = recall_score(y_test, test_pred_classes
    , average='weighted')
f1_score_test = f1_score(y_test, test_pred_classes,
    average='weighted')

print("Test-Retest Model Metrics:")
print(f"Accuracy: {test_acc}%")
print(f"Precision: {precision_test}")
print(f"Recall: {recall_test}")
print(f"F1 Score: {f1_score_test}")

retest_pred = raw_retest_test_model.predict(X_test)
retest_pred_classes = np.argmax(retest_pred, axis=1)
y_retest = np.argmax(y_retest, axis=1)
retest_confusion_mtx = confusion_matrix(y_retest,
    retest_pred_classes)

# Precision, recall, and F1 score for
    retest_test_model
precision_retest = precision_score(y_retest,
    retest_pred_classes, average='weighted')
recall_retest = recall_score(y_retest,
    retest_pred_classes, average='weighted')
f1_score_retest = f1_score(y_retest,
    retest_pred_classes, average='weighted')

print("\nRetest-Test Model Metrics:")
print(f"Accuracy: {retest_acc}%")
print(f"Precision: {precision_retest}")
print(f"Recall: {recall_retest}")
print(f"F1 Score: {f1_score_retest}")

sns.heatmap(test_confusion_mtx, annot=True, fmt='d')
plt.title(f'Raw Signal - Test/Retest - {test_acc:.3f
    }%')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.savefig('results/cnn1d_models/plots/
    cnn1d_raw_test_retest.png')
plt.show()

sns.heatmap(test_confusion_mtx, annot=True, fmt='d')
plt.title(f'Raw Signal - Retest/Test - {retest_acc
    :.3f}%')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.savefig('results/cnn1d_models/plots/
    cnn1d_raw_retest_test.png')
plt.show()
```