Report HMW1

Francesco Moretti (270077) francesco.moretti14@mail.polimi.it

Filippo Paris (270270) filippo.paris@mail.polimi.it

November 2024

1 Introduction to the Dataset

The goal of this homework is to classify Electrocardiogram (ECG) signals based on rhythmic features. The dataset used for this task is the MIT-BIH Arrhythmia Dataset, which contains ECG data annotated with different types of arrhythmias. The dataset includes both training and testing sets. Each signal represents a heartbeat and is labeled with one of five classes representing different arrhythmic conditions. The five classes are:

- **N**: Normal (no arrhythmia)
- **S**: Fusion of paced and normal beats
- **V**: Premature ventricular contraction
- **F**: Atrial premature contraction
- **Q**: Fusion of ventricular and normal beats

The primary use of this MIT-BIH Arrhythmia Dataset dataset is to develop and test algorithms for detecting abnormal hearth rhythms

2 Dataset Exploration

The first step in the project was to load the training and testing datasets, which are provided in CSV format. In Python, this was done using the `pandas` library. The code below demonstrates how the data was loaded into `DataFrame` objects.

```
df = pd.read_csv('dataset/mitbih_train.csv')
df2 = pd.read_csv('dataset/mitbih_test.csv')
```

After loading the data, we inspected the contents to understand the structure of the dataset. The dataset consists of ECG signal data along with the corresponding arrhythmia labels. The last column of the dataset contains the labels, while the preceding 187 columns represent the ECG signal values. The

training dataset contains 87,553 samples, and the test dataset contains 21,891 samples.

Dataset characteristics:

- **Training set:** 87,553 samples
- **Test set:** 21,891 samples
- **Label classes:** 5 (N, S, V, F, Q)

3 Class Distribution

To understand the balance of the dataset, we analyzed the distribution of the class labels in both the training and testing datasets. We used the `value_counts()` method of pandas to count the occurrences of each class label.

The class distribution for both training and testing datasets is visualized below. We can observe that the class distribution is imbalanced, with the majority of the training samples belonging to the "Normal" class (label `N`).

```
train_class_counts = df.iloc[:, -1].map(label_names).value_counts()
test_class_counts = df2.iloc[:, -1].astype(int).value_counts()
```

Training dataset distribution:

- **N (Normal):** 72,470 samples
- **Q (Fusion of ventricular and normal):** 6,431 samples
- **V (Premature ventricular contraction):** 5,788 samples
- **S (Fusion of paced and normal):** 2,223 samples
- **F (Atrial premature):** 641 samples

Testing dataset distribution:

- **N (Normal):** 18,117 samples
- **Q (Fusion of ventricular and normal):** 1,608 samples
- **V (Premature ventricular contraction):** 1,448 samples
- **S (Fusion of paced and normal):** 556 samples
- **F (Atrial premature):** 162 samples

4 ECG Waveform Visualization

To better understand the nature of the ECG signals, we visualized some waveforms from the dataset. The sampling frequency used for the dataset is 360 Hz. To extract the ECG signals, we selected every 10,000th sample and plotted its corresponding waveform.

Each waveform was plotted with the time on the x-axis (in seconds) and the amplitude of the ECG signal on the y-axis. The title of each plot corresponds to the label of the sample.

The following code was used to generate the ECG waveform plots:

```
Fs = 360  # Sampling frequency
indices_to_plot = range(first, last, 10000)

for i, idx in enumerate(indices_to_plot, 1):
    waveform = df.iloc[idx, :-1]
    time = np.arange(0, len(waveform)) / Fs
    plt.plot(time, waveform, label=f'Label: {label_names[df.iloc[idx, -1]]}')
```

The resulting plots provide an insight into the characteristics of different arrhythmic conditions, with each label indicating the class of the respective ECG signal.

Then, we repeated this step also for the test set df2.

In this step, we successfully loaded and explored the MIT-BIH dataset. We visualized the class distribution, which showed a significant imbalance in the dataset, with the "Normal" class being dominant. We also visualized several ECG waveforms, which helped in understanding the nature of the signals associated with different arrhythmias.

5 Data Splitting and Subsampling

In machine learning tasks, datasets are often split into two parts: one for training and one for testing. In this case, the MIT-BIH dataset has already been divided into two distinct files: `mitbih_train.csv` and `mitbih_test.csv`. However, for practical reasons (such as memory constraints), we will only use 10% of each dataset for training and testing.

5.1 Splitting the Training and Testing Data

To perform this splitting, we utilized the `train_test_split` function from the `sklearn.model_selection` module. This function allows us to partition the data into random subsets. The splitting was done with the following parameters:

- **Training size**: 10% of the original dataset.

- **Random state**: A fixed seed for random number generation, ensuring that the split is reproducible.
- **Stratification**: The `stratify` argument ensures that the class distribution in the training and testing sets mirrors that of the original dataset.

The code for splitting the datasets is as follows:

```
from sklearn.model_selection import train_test_split

# Splitting the training data (df) into a subset for training
train_df, _ = train_test_split(df, train_size=0.1, random_state=28,
stratify=df.iloc[:, -1])

# Splitting the testing data (df2) into a subset for testing
test_df, _ = train_test_split(df2, train_size=0.1, random_state=5,
stratify=df2.iloc[:, -1])
```

After executing the code, the shapes of the new dataframes `train_df` and `test_df` were printed:

```
Shape of train_df: (8755, 188)
Shape of test_df: (2189, 188)
```

These results match the expected dimensions, confirming that the datasets were successfully subsampled to the appropriate size for training and testing.

5.2 Class Distribution in Subsampled Data

Next, we visualized the distribution of class labels in both the training and testing subsets. To ensure the splits were representative, we used the `value_counts()` function to calculate the distribution of labels in both the `train_df` and `test_df`.

Class Distribution in `train_df`:

```
N    7247
S     222
V     579
F      64
Q     643
```

Class Distribution in `test_df`:

```
N    1812
S      55
V     145
F      16
Q     161
```

We then plotted bar charts to visually compare the class distributions in both datasets:

```
fig, axes = plt.subplots(1, 2, figsize=(14, 6), sharey=True)

# Plotting Train_df Distribution
axes[0].bar(train_df_class_counts.index,
train_df_class_counts.values, color='blue', alpha=0.7)
axes[0].set_title("Train_df DataFrame Class Distribution")
axes[0].set_xlabel("Class Labels")
axes[0].set_ylabel("Number of Items")
axes[0].grid(axis='y', linestyle='--', alpha=0.7)

# Plotting Test_df Distribution
axes[1].bar(test_df_class_counts.index,
test_df_class_counts.values, color='green', alpha=0.7)
axes[1].set_title("Test Dataset Class Distribution")
axes[1].set_xlabel("Class Labels")
axes[1].set_ylabel("Number of Items")
axes[1].grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```

These visualizations confirm that the class distribution in both datasets is balanced, although some classes are underrepresented (e.g., 'F' and 'S' labels), which could affect model performance.

Then, we applied the waveform visualization also to these subsets of the train set and the test set.

5.3 Separating Features and Labels

To prepare the data for model training, we removed the label column (the last column) from both the training and testing DataFrames and stored the labels separately in `labels_train` and `labels_test`. This separation of features (ECG waveforms) and labels (classifications) is necessary for training the classifier.

The following code was used to perform this operation:

```
# Removing label columns from train_df and test_df
labels_train = train_df.iloc[:, -1].tolist()
train_df = train_df.drop(columns=train_df.columns[-1])

labels_test = test_df.iloc[:, -1].tolist()
test_df = test_df.drop(columns=test_df.columns[-1])

# Printing the shapes of the new DataFrames
print("Shape of train_df:", train_df.shape)
```

```
print("Shape of test_df:", test_df.shape)
print("Shape of labels_train:", np.shape(labels_train))
print("Shape of labels_test:", np.shape(labels_test))
```

The expected shapes after removing the label columns were:

```
Shape of train_df: (8755, 187)
Shape of test_df: (2189, 187)
Shape of labels_train: (8755,)
Shape of labels_test: (2189,)
```

This confirms that the data has been successfully prepared for training, with the features and labels now separated.

6 ECG Signal Preprocessing and Feature Extraction

In this task, we are processing ECG signals to extract rhythmic features, which will be used for classification. The preprocessing steps include normalization of the signals and the extraction of dynamic features like zero-crossing rate and spectral flux.

7 Data Preprocessing

The first step in preparing the ECG signals for classification is to normalize the data so that the model can better generalize across different signals with varying dynamic ranges. The normalization process is carried out using a `MinMaxScaler` from the `sklearn` library, with the feature range set between $(-1, 1)$.

We define two lists: `train_list` and `test_list`, which contain the train and test data respectively. These lists are derived by converting the values from `train_df` and `test_df` into lists using the `tolist()` method.

7.1 Normalization

After converting the data to lists, we apply the MinMaxScaler on the training set. It is important to use `fit_transform()` on the training data to calculate the scaling parameters, while for the test set we use `transform()` to apply the same scaling. This ensures that the test data is normalized based on the parameters derived from the training set. Using `fit_transform()` on the test set would result in data leakage and improper normalization.

Here is the code used to perform normalization:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(-1, 1))
```

```
# Apply normalization on training set
train_set = scaler.fit_transform(train_list)

# Apply normalization on test set using the same scaler
test_set = scaler.transform(test_list)
```

7.2 Verification

To verify the normalization, we plot the first ECG waveform from `train_list` and `train_set` on top of each other to visually inspect the normalization:

```
import matplotlib.pyplot as plt

# Plotting the first curve of train_list and train_set
plt.plot(train_list[0], label="Original Train List")
plt.plot(train_set[0], label="Normalized Train Set", linestyle="--")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.legend()
plt.title("Verification of Normalization")
plt.show()
```

8 Feature Extraction

The next step involves extracting a feature vector from each ECG signal in the dataset. The feature vector will be used for classification. We define a function `compute_feature_vector` which computes the following features for each ECG waveform:

- Mean and standard deviation of the ECG waveform.
- Zero-crossing rate: The number of times the waveform crosses zero per unit of time.
- Spectral flux: A measure of how quickly the spectral content of the signal changes over time.
- Short-Time Fourier Transform (STFT): Provides frequency-domain information about the signal.

8.1 Zero-Crossing Rate

The zero-crossing rate is a fundamental feature in the analysis of rhythmic signals, including audio and ECG signals. It is defined as the number of times the signal crosses the zero axis within a unit of time (usually expressed in Hertz).

This feature provides insight into the frequency of oscillations in the signal, which is particularly useful in rhythm analysis.

In our case, for ECG signals, the zero-crossing rate reflects the rate at which the signal changes direction, offering insight into rhythmic patterns, such as the heart's pulse. For instance, a higher zero-crossing rate might correlate with more rapid or frequent changes in the ECG waveform, potentially indicating a fast or irregular heartbeat.

By computing both the per-frame zero-crossing rate and its statistical properties (mean and standard deviation), we can capture both local and global information about the dynamics of the waveform. This approach enhances our ability to distinguish between different types of rhythms in the ECG signal.

Why is the zero-crossing rate useful for rhythmic analysis? The zero-crossing rate is a simple but effective measure of rhythmic behavior, quantifying how often the signal oscillates around zero. In classification tasks, it serves as a key distinguishing feature between signals with different periodicities or rhythms.

In this work, we utilized `librosa.feature.zero_crossing_rate` to calculate the zero-crossing rate over short overlapping frames. Notably, this function returns a 2D array, where each row corresponds to a specific channel of the signal, and each column represents the zero-crossing rate for a frame. For simplicity, we considered only the first array (row), assuming a single-channel signal, which is typical for ECG data. Additionally, we computed the mean and standard deviation of the zero-crossing rate across all frames to capture the overall trends and variability of this feature.

```
zero_crossing_rate_frames = librosa.feature.zero_crossing_rate(
    x, frame_length=N, hop_length=H
)[0]  # Extract the first row for single-channel signals
zero_crossing_rate_mean = np.mean(zero_crossing_rate_frames)
zero_crossing_rate_std = np.std(zero_crossing_rate_frames)
```

8.2 Spectral Flux

Spectral flux is another important rhythmic feature, representing the rate of change in the spectral content of a signal over time. It quantifies how much the amplitude of different frequency components changes from one frame to the next. This feature is particularly valuable when analyzing signals with rapid transitions or sudden changes in frequency content, such as ECG signals during heartbeats.

What is the aim of the spectral flux? The spectral flux aims to capture the dynamic changes in the signal's frequency domain, which is especially useful for detecting sudden shifts or variations in rhythm. For example, during a heartbeat, the frequency content of the ECG signal changes as the electrical activity of the heart propagates through the body. Spectral flux captures these transitions, making it a powerful tool for rhythm classification, particularly in distinguishing between normal and arrhythmic heart rhythms.

In this work, we computed the spectral flux using the `librosa.onset.onset_strength` function, which evaluates the intensity of onset events by measuring changes in the signal's spectral envelope. This method emphasizes transitions in the signal, such as the onset of a heartbeat or variations in its intensity. To summarize the spectral flux information, we calculated both its mean and standard deviation over the entire signal.

```
spectral_flux = librosa.onset.onset_strength(S=librosa.amplitude_to_db(
C, ref=np.max))
spectral_flux_mean = np.mean(spectral_flux)
spectral_flux_std = np.std(spectral_flux)
```

If we were not using `librosa`, spectral flux could be computed manually by summing the squared differences between successive frames of the Short-Time Fourier Transform (STFT). However, leveraging `librosa` ensures efficient computation and access to advanced signal processing methods.

8.3 Feature Vector Computation

We concatenate all the computed features (mean, standard deviation, zero-crossing rate, zero-crossing rate mean, zero-crossing rate standard deviation, spectral flux, spectral flux mean, spectral flux standard deviation) into a single feature vector. This vector serves as the input to the classification model. Here is the final code for computing the feature vector:

```
def compute_feature_vector(x, Fs, N=64, H=16):
    # 1. Mean and standard deviation of the ECG signal
    mean_x = np.mean(x)
    std_x = np.std(x)

    # 2. Zero-crossing rate using librosa's feature function
    zero_crossing_rate_frames = librosa.feature.zero_crossing_rate(x,
    frame_length=N, hop_length=H)[0]
    zero_crossing_rate_mean = np.mean(zero_crossing_rate_frames)
    zero_crossing_rate_std = np.std(zero_crossing_rate_frames)

    # 3. Short-Time Fourier Transform (STFT)
    C = np.abs(librosa.stft(x, n_fft=N, hop_length=H))
    # Magnitude of the STFT

    # 4. Spectral flux
    spectral_flux = librosa.onset.onset_strength(S=librosa.amplitude_to_db(C,
    ref=np.max))
    spectral_flux_mean = np.mean(spectral_flux)
    spectral_flux_std = np.std(spectral_flux)

    # 5. Concatenate all computed features into a single feature vector
```

```
    f_vector_stats = np.array([mean_x, std_x,
                               zero_crossing_rate_mean,
                               zero_crossing_rate_std,
                               spectral_flux_mean,
                               spectral_flux_std])

    # Include detailed per-frame features
    f_vector_details = np.concatenate([zero_crossing_rate_frames,
                               spectral_flux])

    # Combine statistics and detailed features
    f_vector = np.concatenate([f_vector_stats, f_vector_details])

    return f_vector
```

8.4 Testing the Function

We test the `compute_feature_vector` function on the first waveform from the training set and print and plot the resulting feature vector. We also plot the first waveform to visualize it.

```
x_first = train_set[0]  # Example: first waveform from the train set
Fs = 360  # Sampling frequency in Hz
f_vector_first = compute_feature_vector(x_first, Fs)

# Print the computed feature vector
print("Feature vector for the first waveform:", f_vector_first)

# Plot the first waveform from the train set
plt.figure(figsize=(12, 5))

plt.subplot(2, 1, 1)
plt.plot(f_vector_first, marker='o')
plt.title("Feature Vector for the First Waveform")
plt.xlabel("Feature Index")
plt.ylabel("Feature Value")
plt.grid(True)

# Plot the feature vector
plt.subplot(2, 1, 2)
plt.plot(x_first)
plt.title("First Waveform from the Train Set")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.grid(True)
```

```
# Show the plots
plt.tight_layout()
plt.show()
```

8.5 Computing Feature Vectors for the Training and Test Sets

We compute the feature vectors for both the training and test sets by iterating through each waveform and applying the `compute_feature_vector` function. The results are stored in two lists: `train_fvector` and `test_fvector`. We use the `tqdm` library to visualize the progress during computation.

```
train_fvector = []
test_fvector = []

# Compute feature vectors for the training set
for x in tqdm(train_set, desc="Train Set Processing"):
    f_vector = compute_feature_vector(x, Fs, N, H)
    train_fvector.append(f_vector)

# Compute feature vectors for the test set
for x in tqdm(test_set, desc="Test Set Processing"):
    f_vector = compute_feature_vector(x, Fs, N, H)
    test_fvector.append(f_vector)

# Convert lists to numpy arrays
train_fvector = np.array(train_fvector)
test_fvector = np.array(test_fvector)

# Print the shapes of the feature vectors
print(f"\nShape of train_fvector: {train_fvector.shape}")
print(f"Shape of test_fvector: {test_fvector.shape}")
```

8.6 Results

After computing the feature vectors for both the training and test sets, we verify that the shapes of the feature vectors are consistent with the number of samples in the original data sets. The shape of `train_fvector` is $(8755, 30)$, and the shape of `test_fvector` is $(2189, 30)$, confirming that the feature extraction process has been successfully applied.

The preprocessing steps and feature extraction process have been successfully implemented. The normalization of the signals ensures that the model can handle waveforms with varying dynamic ranges. The extracted feature vectors, containing rhythmic features such as zero-crossing rate and spectral flux, are

ready for classification. The next step will involve using these feature vectors to train a classification model.

9 Final Considerations

In this section, we will discuss the aim of a novelty function and the use of multiple rhythmic features and the trade-offs associated with adding such features.

9.1 Aim of a Novelty Function

What is the aim of a novelty function? The primary aim of a novelty function is to detect significant changes or transitions in a signal over time. These changes often correspond to meaningful events or structural boundaries in the signal. For example, in music analysis, a novelty function can help identify transitions between different sections (e.g., verse to chorus), while in biomedical signals, it can detect shifts in physiological patterns, such as arrhythmias in ECG data.

A novelty function typically works by analyzing short-term variations in certain features of the signal, such as energy, frequency content, or rhythm. Peaks in the novelty function indicate points of rapid change, which can then be used to segment the signal or identify key moments. This makes novelty functions crucial in tasks where understanding temporal dynamics or structure is essential, such as signal segmentation, event detection, or pattern recognition.

In summary, the novelty function provides a mechanism to quantify and highlight temporal changes, aiding in the interpretation and analysis of complex signals.

9.2 Adding Different Rhythmic Features: Value and Trade-Offs

Why is it worth adding different rhythmic features? Adding multiple rhythmic features, such as the zero-crossing rate and spectral flux, provides a more comprehensive description of the signal's rhythmic structure. Each feature captures different aspects of the signal's behavior:

- The zero-crossing rate captures the frequency of oscillations, which is useful for identifying periodic or cyclical patterns. - Spectral flux highlights changes in the frequency domain, helping to capture rapid shifts in rhythm or intensity.

By combining multiple rhythmic features, we increase the likelihood of distinguishing between different types of ECG rhythms that may share similar characteristics but differ in subtle ways. In classification tasks, a richer feature set generally leads to improved model performance, as the classifier has more information to base its decisions on.

Can you spot any trade-offs? While adding multiple features can improve classification accuracy, there are some trade-offs to consider:

1. Feature Dimensionality: Increasing the number of features can lead to higher dimensionality, which might require more data to train the model effectively. If the number of features is too high relative to the size of the dataset, the model may suffer from overfitting, where it learns to perform well on the training data but fails to generalize to unseen test data.

2. Computational Complexity: Extracting additional features increases the computational cost, both in terms of memory and processing time. For large datasets or real-time systems, this might be a concern, especially if the extraction of features like STFT or spectral flux is computationally expensive.

3. Irrelevant or Redundant Features: Not all features necessarily contribute to improved performance. Some features might be redundant or irrelevant to the task at hand, and adding such features could potentially harm the model's performance by introducing noise. For instance, if two features are highly correlated, they might convey the same information, leading to redundancy without adding new insights. Feature selection techniques or dimensionality reduction methods (e.g., PCA) can help mitigate this issue by removing unnecessary or highly correlated features.

In summary, adding multiple rhythmic features can provide valuable insights into the signal's characteristics, but it is important to balance the number of features with the available data and computational resources. Additionally, careful attention should be paid to feature selection to ensure that only the most relevant and distinct features are included in the final model.

10 SVM Classifier Training and Evaluation

In this section, we explore the use of a Support Vector Machine (SVM) for classifying ECG signals based on the feature vectors we computed earlier. We train the SVM, evaluate its performance on the test set, and analyze various metrics to understand the model's behavior.

10.1 SVM Overview and Kernel Selection

Support Vector Machines (SVM) are powerful classification models that work by finding the optimal hyperplane that separates different classes in a feature space. The choice of kernel is crucial for determining the type of decision boundary that the SVM uses. Common kernel types include:

- **Linear Kernel**: Assumes that classes are linearly separable. The decision boundary is a straight line (or hyperplane in higher dimensions).
- **Polynomial Kernel**: Introduces non-linear boundaries by considering polynomial decision boundaries.
- **Radial Basis Function (RBF) Kernel**: A non-linear kernel that transforms the data into a higher-dimensional space, where a linear decision boundary may be found. The RBF kernel is particularly effective when the classes are not linearly separable.

- **Sigmoid Kernel**: Mimics the behavior of neural networks by using the sigmoid function as the kernel. It is useful in certain non-linear scenarios, but its performance can be sensitive to the choice of parameters.

For our task, we selected the `rbf` kernel for the SVM, as it is well-suited for handling non-linearly separable data. The regularization parameter C was set to 10 to control the trade-off between model complexity and training data accuracy.

10.2 Model Training and Saving

The SVM was trained using the `train_fvector` feature vectors and their corresponding labels. We used the `sklearn.svm.SVC` function with the parameters $C = 10$ and the `rbf` kernel.

```
model = SVC(C=10, kernel='rbf')
model.fit(train_fvector, labels_train)
```

Once trained, the model was saved using `joblib`, which allows for easy loading of the model in the future without the need for retraining.

```
model_filename = f"my_model/svc_rbf_C_10_N_64_H_16"
joblib.dump(model, model_filename)
```

The trained model was successfully saved to the path: `my_model/svc_rbf_C_10_N_64_H_16`.

10.3 Hyperparameter Definition and Overfitting

A *hyperparameter* is a parameter that is set before the learning process begins and controls the model's training behavior. For example, in SVM, C and the choice of kernel are hyperparameters. Hyperparameter tuning often involves trying different combinations of these parameters to find the best configuration for a given task.

Overfitting occurs when the model performs well on the training data but poorly on new, unseen data. This typically happens when the model is too complex and learns to fit the noise or specific patterns in the training data rather than capturing the underlying distribution of the data. To detect overfitting, it is crucial to evaluate the model's performance on both the training and test sets.

10.4 Training Accuracy

The accuracy of the model on the training set was computed to be approximately 89.22%, and on the training set about 88.94%. In our case, having an high accuracy with imbalanced classes might not be a good indicator of the model's quality. The result might seem promising at first but it's a misleading result, in fact the model probably predicts the dominant class systematically,

achieving a high accuracy without actually doing a good job at distinguish the minority classes, that's why we will use a confusion matrix to better understand the results.

The classifier shows high accuracy on both the train set and the test set because they are both unbalanced with a majority of N values.

10.5 Confusion Matrix

The confusion matrix is a powerful tool for understanding how well the model classifies each class. It shows the number of correct and incorrect predictions for each class in a tabular form. The confusion matrix for the test set was generated using the following code:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(labels_test, test_predictions)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=label_names.values())
disp.plot(cmap='Blues')
```

The confusion matrix helps identify which classes are harder for the model to classify. The classifier does not perform in the same way for all the classes. In fact, for example, it is evident that the classifier performs poorly on the classes `S` and `F`, with a high number of misclassifications. Conversely, the `N` class (Normal) shows high accuracy, as it has many correct predictions, so with this class the classifier performs better. Now that we have these results, we can tell that the classifier mostly predicts the dominant class, which results in a high accuracy despite it's not a good classifier.

10.6 Classification Report: Precision, Recall, and F1-Score

To further evaluate the classifier, we computed the classification report using precision, recall, and F1-score for each class.

Precision measures the accuracy of positive predictions, i.e., the proportion of true positives among all predicted positives:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall measures the ability of the classifier to find all positive instances, i.e., the proportion of true positives among all actual positives:

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1-score is the harmonic mean of precision and recall, providing a single metric that balances the two:

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The classification report for the test set is shown below:

```
report = classification_report(labels_test, test_predictions,
target_names=[label_names[i] for i in range(5)], zero_division=0)
print(report)
```

We want to maximize the recall and the f1-score for each class.

10.7 False Positives, False Negatives, True Positives and True Negatives

We also computed the total counts of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN) across all classes using the confusion matrix. These metrics are crucial for understanding the model's behavior:

The confusion matrix is built upon four key metrics: True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN). These metrics are commonly used for evaluating classification models, particularly in two-class problems. For multi-class problems, these metrics can be extended or aggregated. Below, we define each metric and provide the corresponding code for computation.

- **True Positives (TP)**:
  - *Definition*: True Positives represent the cases where the model correctly predicts the presence of a class.
  - *How to Obtain*: These correspond to the diagonal elements of the confusion matrix, where the predicted class matches the true class.

    ```
    true_positives = np.diag(conf_matrix)
    ```

- **False Positives (FP)**:
  - *Definition*: False Positives represent the cases where the model incorrectly predicts the presence of a class.
  - *How to Obtain*: These are calculated by summing the column corresponding to the predicted class and subtracting the diagonal element (True Positive) for that class.

    ```
    false_positives = conf_matrix.sum(axis=0) - np.diag(conf_matrix)
    ```

- **False Negatives (FN)**:

- *Definition*: False Negatives represent the cases where the model incorrectly predicts the absence of a class.
- *How to Obtain*: These are calculated by summing the row corresponding to the true class and subtracting the diagonal element (True Positive) for that class.

```
false_negatives = conf_matrix.sum(axis=1) - np.diag(conf_matrix)
```

- **True Negatives (TN)**:
  - *Definition*: True Negatives represent the cases where the model correctly predicts the absence of a class.
  - *How to Obtain*: These are computed by subtracting the sum of False Positives, False Negatives, and True Positives for the class from the total sum of the confusion matrix.

```
true_negatives = conf_matrix.sum() - (false_positives +
false_negatives + true_positives)
```

10.7.1 Summing Metrics Across All Classes

For multi-class classification problems, we can calculate the total metrics across all classes as follows:

```
total_true_positives = true_positives.sum()
total_true_negatives = true_negatives.sum()
total_false_positives = false_positives.sum()
total_false_negatives = false_negatives.sum()
```

The values for TP, FP, FN, and TN for the test set are:

```
Total True Positives: 1947
Total True Negatives: 8514
Total False Positives: 242
Total False Negatives: 242
```

In our task of detecting heart rate anomalies, **False Negatives** (FN) are particularly important. A false negative occurs when an abnormality (e.g., arrhythmia) is missed, which is more critical than a false positive, where a normal case is incorrectly classified as abnormal.

11 Data Balancing

In this section, we balance the training and test datasets to ensure that each class has an equal number of samples. Specifically, we randomly sample 641 instances from each class in the training dataset, `df`, and 162 instances from each class in the test dataset, `df2`, to form the balanced datasets, `train_df` and `test_df`, respectively.

11.1 Training Dataset Balancing

We start by initializing an empty DataFrame `train_df`. For each unique class label in the training dataset (`df`), we use the `sample` method to randomly sample 641 instances, with a fixed random seed (`random_state=42`) to ensure reproducibility. The resulting samples for each class are concatenated to form the balanced `train_df`.

The Python code for this process is as follows:

```
# Train Dataset Balancing
train_df = pd.DataFrame()  # Initialize empty DataFrame
# for training dataset
labels_train = df.iloc[:, -1]  # Extract class labels
# from the training dataset

# Sample 641 instances for each unique class label
for class_label in labels_train.unique():
    class_df = df[labels_train == class_label]
    sample_class_df = class_df.sample(n=641, random_state=42)
    train_df = pd.concat([train_df, sample_class_df])

train_df = train_df.reset_index(drop=True)
print(f"Train DataFrame shape (641 instances * 5 classes):
{train_df.shape}")
print(train_df.iloc[:, -1].map(label_names).value_counts())
```

After balancing the training dataset, the shape of `train_df` is expected to be 641×5 samples, one for each of the five classes.

11.2 Test Dataset Balancing

Similarly, we balance the test dataset by sampling 162 instances per class. We initialize an empty DataFrame `test_df` and repeat the sampling process for each class label in the test dataset (`df2`).

The Python code for balancing the test dataset is as follows:

```
# Test Dataset Balancing
test_df = pd.DataFrame()
labels_test = df2.iloc[:, -1]

# Sample 162 instances for each unique class label
for class_label in labels_test.unique():
    class_df = df2[labels_test == class_label]
    sample_class_df = class_df.sample(n=162, random_state=42)
    test_df = pd.concat([test_df, sample_class_df])

test_df = test_df.reset_index(drop=True)
print(f"Test DataFrame shape (162 instances * 5 classes): {test_df.shape}")
print(test_df.iloc[:, -1].map(label_names).value_counts())
```

After balancing the test dataset, the shape of `test_df` is expected to be 162×5 samples.

11.3 Balanced Dataset Summary

After the balancing process, the shapes of the `train_df` and `test_df` are as follows:

- `train_df` has 641 samples per class, totaling $641 \times 5 = 3205$ samples.
- `test_df` has 162 samples per class, totaling $162 \times 5 = 810$ samples.

The final balanced datasets are now ready for further processing and training.

12 Data Augmentation

In order to improve the model's robustness and generalizability, we apply data augmentation techniques. These techniques are intended to artificially increase the size of the training dataset by introducing variations in the signal data. The augmentation methods used here include stretching the waveform and modifying the amplitude.

We define a class `augment` with two class methods: `stretch` and `amplify`.

12.1 Stretch Method

The `stretch` method randomly adjusts the length of the input signal by resampling it to a new length based on a random factor. The resampled signal is then truncated or padded to a fixed length of 187 samples.

The steps for the `stretch` method are as follows:

- A random factor β is generated in the range [0, 1].

- The new length l of the signal is calculated as:

$$l = 187 \cdot (1 + \beta - 0.5)^3$$

- The signal is resampled to length l using `scipy.signal.resample`.
- If $l < 187$, the resampled signal is padded with zeros to reach length 187. If $l > 187$, it is truncated to 187 samples.

The code for the `stretch` method is as follows:

```
[language=Python]
from scipy.signal import resample
import random
import numpy as np

class augment:
    def __init__(self):
        pass

    def stretch(self, x):
        beta = random.random()
        l = int(187 * (1 + beta - 0.5)**3)
        y = resample(x, l)
        y1 = np.zeros(187)

        if l < 187:
            y1[:l] = y
        else:
            y1[:187] = y[:187]

        return y1
```

12.2 Amplify Method

The `amplify` method adjusts the amplitude of the input signal by multiplying it with a random factor. The random factor α is generated in the range $[-0.5, 0.5]$, and the amplified signal is returned as:

$$x_{\text{amplified}} = x \cdot (\alpha + 1)$$

The code for the `amplify` method is as follows:

```
    def amplify(self, x):
        x = np.array(x)
        alpha = random.uniform(-0.5, 0.5)
        return x * (alpha + 1)
```

12.3 Perform Method: Randomly Applying Augmentations

We define a class method named `perform`, which randomly applies either or both augmentations (stretch and amplify) to the input signal `x`. The steps followed in this method are:

- Initialize an empty list `performed_augmentations` to keep track of the applied augmentations.
- Apply the stretch augmentation with a 50% probability using `np.random.binomial(1, 0.5)`. If the result is 1, the stretch augmentation is applied, and the string `'stretch'` is added to the list of performed augmentations.
- Apply the amplify augmentation with a 50% probability using `np.random.binomial(1, 0.5)`. If the result is 1, the amplify augmentation is applied, and the string `'amplify'` is added to the list of performed augmentations.
- Return the augmented signal and the list of applied augmentations.

The Python code for the `perform` method is as follows:

```
[language=Python]
def perform(self, x):
    """
    Randomly applies the stretch and/or amplify augmentations to the signal `x`.
    """
    performed_augmentations = []

    # Apply the stretch augmentation with a 50% probability
    if np.random.binomial(1, 0.5) == 1:
        x = self.stretch(x)  # Apply stretch
        performed_augmentations.append('stretch')

    # Apply the amplify augmentation with a 50% probability
    if np.random.binomial(1, 0.5) == 1:
        x = self.amplify(x)  # Apply amplify
        performed_augmentations.append('amplify')

    # Return the modified signal and the list of augmentations performed
    return x, performed_augmentations
```

In this section, we introduced a method to apply random augmentations to the signals in the dataset. The goal is to increase the diversity of the training data and improve the generalizability of the model. The augmentations include stretching the waveform and amplifying its amplitude, and they are applied randomly. We also scale the dataset using a `MinMaxScaler` to normalize the feature values between 0 and 1.

After defining the `perform` method, we instantiate the `augment` class and apply the augmentations to a sample signal from the training dataset.

```
augmenter = augment()

# Train
x = train_list[0][:-1]

# Apply the augmentations to the signal ‘x‘ using the
# ’perform’ method from the ’augment’ class
augmented_signal, augmentations = augmenter.perform(x)

# Print which augmentations were applied to the signal
print(f"Applied augmentations: {augmentations}")

# Plot the original vs augmented signal
plt.figure(figsize=(10, 6))
plt.plot(x, label='Original Signal', color='blue')
plt.plot(augmented_signal, label='Augmented Signal', color='red')

# Add Title, Labels, and Legend
plt.title('Original vs Augmented Signal')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```

This code randomly applies either the stretch or amplify augmentation, or both. The output is a modified signal, and the plot visualizes the difference between the original and augmented signals. The augmentations applied will vary each time the method is called, as demonstrated in the plot and the printed augmentation details.

12.4 Augmenting the Training Dataset

Now, we extend the training dataset by adding augmented samples. We define a variable `n_aug=100`, meaning that for each class in the training dataset, we generate 100 augmented samples.

The following steps are performed:

- For each class in the dataset, sample 100 signals using `sample(n_aug, random_state=16)`.
- For each of these sampled signals, apply the augmentation using the `perform` method.
- Store the augmented signals and their corresponding class labels.

- Concatenate the augmented signals with the original training dataset to increase the size of the dataset.

The code for augmenting the training dataset is as follows:

```
augmented_signals = []
train_labels = train_df.iloc[:, -1].values  # Last column (label)

# Looping through each unique class in the 'class' column
for label in np.unique(train_labels):
    # Check for initial possible NaN values
    if pd.isna(label):
        print(f"Warning: Missing class label for signal!")

    # We exctract the signals belonging to the current class
    class_subset = train_df[train_labels == label] # Filtering the data

    # Sampling `n_aug` signals from the current class
    sampled_signals = class_subset.sample(n=n_aug, random_state=16,
    replace=False)

    # Now, we can apply augmentation to each sampled signal
    for _, row in sampled_signals.iterrows():
        signal = row.iloc[:-1]

        # Apply the augmentation to the signal (using the perform method)
        augmented_signal, applied_augmentations = augmenter.perform(signal)

        # Checking if the augmentation is working good
        if np.isnan(augmented_signal).any():
            print(f"Warning: NaN detected in augmented signal
            for class {label}")


        augmented_signals.append({
            'signal': augmented_signal,
            'class': label
        })

# Converting the list of augmented signals into a DataFrame...
augmented_df = pd.DataFrame(augmented_signals)

# Now concatenate this augmented DataFrame with the original train_df
train_df = pd.concat([train_df, augmented_df], ignore_index=True)
train_labels = np.concatenate([train_labels, augmented_df['class'].values])
```

```
# Then, we ensure an equal counts per class
class_counts = train_df['class'].map(label_names).value_counts()

# We need to remove the label column from train_df
train_df = train_df.drop(columns=['signal','class'])  # Remove the last column

# Print the counts for each class after augmentation
print("Number of augmented signals for each class:")
print(class_counts)

# Print the shape of the Train DataFrame
print("\nShape of train_df after Augmentation:")
print(train_df.shape) # Expected (3705, 188)
# print(test_df.shape)

total_samples = train_df.shape[0]
feature_count = train_df.shape[1]
print(f"\nTotal Samples: {total_samples}, Feature Count: {feature_count}")
```

After augmenting the dataset, the training DataFrame `train_df` now contains 3705 samples, with 188 features per sample, ensuring that each class has the same number of samples. The dataset is now ready for training.

12.5 Data Normalization with MinMaxScaler

To prepare the dataset for training a machine learning model, we normalize both the training and test datasets using `MinMaxScaler`. This scaler transforms the features into the range [0, 1].

The following steps are performed:

- Separate the features and labels for both the training and test datasets.
- Apply the `MinMaxScaler` to the training features.
- Use the same scaling parameters to transform the test features.
- Convert the scaled features back into DataFrames and ensure the labels are included.

The code for applying the `MinMaxScaler` is as follows:

```
[language=Python]
from sklearn.preprocessing import MinMaxScaler

# Separate features and labels for train_df
train_features = train_df.iloc[:, :-1].values

# Apply MinMaxScaler to normalize the data
```

```
scaler = MinMaxScaler()
train_features_scaled = scaler.fit_transform(train_features)

# Transform the test dataset using the same scaler
test_features_scaled = scaler.transform(test_features)

# Convert the normalized data back into DataFrames
train_scaled_df = pd.DataFrame(train_features_scaled,
columns=train_df.columns[:-1])
train_scaled_df['label'] = train_labels

test_scaled_df = pd.DataFrame(test_features_scaled,
columns=test_df.columns[:-1])
test_scaled_df['label'] = test_labels

# Convert train_scaled_df and test_scaled_df into lists
train_list = list(zip(train_scaled_df.iloc[:,
:-1].values.tolist(), train_scaled_df['label'].tolist()))
test_list = list(zip(test_scaled_df.iloc[:,
:-1].values.tolist(), test_scaled_df['label'].tolist()))

# Print the length of the train and test lists
print(f"Train list length: {len(train_list)}")
print(f"Test list length: {len(test_list)}")
```

After applying the `MinMaxScaler`, the features in both the training and test sets are normalized, and the datasets are ready for model training.

12.6 Summary and Future Augmentations

In this section, we demonstrated how to apply random augmentations (stretching and amplifying) to the training dataset, followed by normalizing the data. This process helps increase the variability of the training data and prepares it for machine learning models.

Possible additional augmentations that could be implemented include:

- Noise Addition: Adding small amounts of random noise to the signal to simulate real-world variations.
- Time Shifting: Shifting the signal in time, which can help the model become invariant to small time shifts.
- Frequency Modulation: Modifying the frequency components of the signal to simulate different acoustic conditions.
- Time Stretching: Stretching or compressing the signal in time without altering its frequency content.

These augmentations can further enhance the robustness and generalizability of the model.

13 Training and Evaluation of the SVM Classifier

In this section, we describe the process of training an SVM classifier on the augmented dataset and performing classification to verify whether the new dataset improves performance. We also investigate potential overfitting by comparing the classifier's performance on the training and test sets.

13.1 Feature Vector Computation

To prepare the dataset for classification, feature vectors were computed for both the train and test sets. The following parameters were used for feature extraction:

- Window length (N): 64 samples
- Hop size (H): 16 samples
- Sampling rate (F_s): 360 Hz

The feature extraction process included the following steps:

1. **Signal Preprocessing**: Each signal was converted to a NumPy array and cleaned to ensure that all values were finite. Non-finite values were replaced using the `np.nan_to_num()` function.
2. **Feature Vector Computation**: Using the defined function `compute_feature_vector()`, feature vectors were extracted for each signal.
3. **Conversion to Arrays**: Feature vectors for the train and test sets were converted to NumPy arrays to ensure compatibility with the SVM classifier.
4. **Verification**: The shapes of the feature vectors and labels were checked, and it was confirmed that no NaN values were present in either the features or the labels.

13.2 Training the SVM Classifier

An SVM classifier with an RBF kernel and $C = 10$ was trained on the feature vectors of the training set. The training process involved the following steps:

1. Initializing the SVM classifier with the `SVC()` function from `scikit-learn`, specifying the RBF kernel and $C = 10$.
2. Fitting the classifier to the training set feature vectors and their corresponding labels.

13.3 Model Evaluation

The trained classifier was tested on the test set, and its performance was evaluated using the accuracy metric. The following results were obtained:

- **Accuracy on the Test Set**: 67.53%
- **Accuracy on the Training Set**: 65.75%

The comparison of the training and test accuracies indicated that the model does not overfit, as the test accuracy is higher than the training accuracy.

13.4 Code Implementation

Below is the core Python code used for training and evaluating the SVM classifier:

```
# Train the SVM classifier with RBF kernel and C=10
svm_classifier = SVC(kernel='rbf', C=10)
svm_classifier.fit(train_fvector, train_labels)

# Test the classifier on the test set
test_predictions = svm_classifier.predict(test_fvector)

# Compute accuracy on the test set
accuracy = accuracy_score(test_labels, test_predictions)
print(f"Accuracy on the test set: {accuracy * 100:.2f}%")

# Evaluate the classifier on the training set
train_predictions = svm_classifier.predict(train_fvector)
train_accuracy = accuracy_score(train_labels, train_predictions)
print(f"Accuracy on the training set: {train_accuracy * 100:.2f}%")

# Define a threshold for overfitting (e.g., 10%)
overfit_threshold = 0.10

# Check for overfitting with the threshold
if (train_accuracy - accuracy) > overfit_threshold:
    print("The model is overfitting, as the training accuracy
    is significantly higher than the test accuracy.")
else:
    print("The model is not overfitting.")
```

13.5 Discussion and Conclusion

The results indicate that the augmented dataset improves classification performance on the test set compared to the training set. The absence of overfitting

suggests that the model generalizes reasonably well to unseen data. However, the relatively low accuracy values (67.53% on the test set) indicate that there is room for improvement in either feature extraction or model tuning.

With respect to the previous case of the unbalanced sets, the accuracy is lower but it shows a more realistic measure of model performance, indicating the model is treating all the classes equally.

The model does not overfit, and the small difference in accuracy between the trainig and test sets suggests that the model is genralizing well. A model that isn't overfitting tends to have similar performance on both sets

14 Confusion Matrix and Classification Report

In this section, we analyze the performance of the SVM classifier after training on the augmented dataset. We will compute and visualize the confusion matrix, as well as evaluate key performance metrics such as accuracy, recall, precision, and F1-score.

14.1 Confusion Matrix and Classification Report

The confusion matrix provides insights into how well the classifier is able to distinguish between the different classes in the dataset. It shows the number of true positives, true negatives, false positives, and false negatives for each class. The classification report provides more detailed metrics such as precision, recall, and F1-score.

The classification report for the trained model is shown below:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
N	0.59	0.69	0.64	162
S	0.84	0.67	0.75	162
V	0.80	0.71	0.75	162
F	0.79	0.89	0.84	162
Q	0.89	0.91	0.90	162
Accuracy		0.78		810
Macro avg	0.78	0.78	0.78	810
Weighted avg	0.78	0.78	0.78	810

Table 1: Classification report for the trained model.

14.2 Comparison with Previous Results

In the previous case, where the dataset was highly imbalanced (with a strong prevalence of class `N`), the classifier showed high performance for the majority

class but struggled with the minority classes. The performance metrics for the previous classifier are shown below:

Class	Precision	Recall	F1-Score	Support
N	0.90	0.99	0.94	1812
S	0.82	0.25	0.39	55
V	0.74	0.14	0.23	145
F	0.00	0.00	0.00	16
Q	0.84	0.74	0.79	161
Accuracy		0.89		2189
Macro avg	0.66	0.42	0.47	2189
Weighted avg	0.87	0.89	0.86	2189

Table 2: Classification report for the previous model trained on the imbalanced dataset.

From the previous classification report, we observe the following:

- The classifier heavily favored the majority class (`N`), achieving a high recall of 0.99 for this class. However, recall values for the minority classes (`S`, `V`, `F`, `Q`) were very low, with class `F` having a recall of 0.00 and class `S` only achieving 0.25.
- The overall accuracy was high (89%), but this was misleading due to the imbalanced nature of the dataset.

Comparing the two reports, we observe the following improvements:

- The recall values for the minority classes (`S`, `V`, `F`, `Q`) have significantly improved. For instance, class `F` now has a recall of 0.89, compared to 0.00 previously. Similarly, class `Q` improved from a recall of 0.74 to 0.91.
- The F1-Score values for the minority classes (`S`, `V`, `F`, `Q`) have also improved substantially. For example, class `F`, which previously had an F1-Score of 0.00, now achieves 0.84. Likewise, class `Q` increased its F1-Score from 0.79 to 0.90.

These results demonstrate the effectiveness of addressing class imbalance in improving the classifier's performance across all classes, ensuring more equitable treatment of both majority and minority classes.

15 Test with Different Parameter Configurations

In this section, we experiment with different values of the regularization parameter C for the SVM classifier, while keeping the kernel fixed (RBF). The parameter C controls the trade-off between achieving a low training error and maintaining a simple model. Higher values of C prioritize minimizing the training error, potentially leading to overfitting, while smaller values of C allow more

flexibility in the decision boundary, which could help generalize better but might increase training error.

We evaluate the classifier using the following values of C: $C = \{0.1, 1, 100, 1000, 10000, 100000\}$. For each configuration, we compute and print the accuracy, F1-score (macro average), recall (macro average), and the total false negatives.

15.1 Results for Different Values of C

C Value	Accuracy	F1-Score (avg)	Recall (avg)	False Negatives
0.1	67.53	0.6666	0.6753	263
1	72.72	0.7252	0.7272	221
100	79.26	0.7926	0.7926	168
1000	80.00	0.8003	0.8000	162
10000	77.28	0.7732	0.7728	184
100000	77.04	0.7722	0.7704	186

Table 3: Performance of the SVM classifier with different values of C.

15.2 Analysis of Results

The performance of the classifier is evaluated for different values of C as shown in the table above.

- **Accuracy:** The accuracy increases as C increases from 0.1 to 1000, reaching its highest value of 80.00% for $C = 1000$. However, beyond $C = 1000$, the accuracy decreases slightly (to 77.28% for $C = 10000$ and 77.04% for $C = 100000$).
- **F1-Score (macro avg):** The F1-score follows a similar trend, increasing with C and peaking at 0.8003 for $C = 1000$. For higher values of C, the F1-score decreases slightly, reaching 0.7732 for $C = 10000$ and 0.7722 for $C = 100000$.
- **Recall (macro avg):** Recall improves with C, peaking at 0.8000 for $C = 1000$. Similar to the F1-score, it slightly decreases for larger values of C, reaching 0.7728 for $C = 10000$ and 0.7704 for $C = 100000$.
- **Total False Negatives:** The total number of false negatives decreases as C increases, reaching the lowest value of 162 for $C = 1000$. For higher values of C, the number of false negatives increases slightly to 184 for $C = 10000$ and 186 for $C = 100000$.

15.3 Does the Performance Improve if We Increase C?

From the results, we observe that the performance improves as C increases from 0.1 to 1000, with a noticeable peak at $C = 1000$. Beyond this value, the performance slightly deteriorates for higher values of C, indicating a potential overfitting effect.

This suggests that the regularization parameter C controls the complexity of the decision boundary. When C is too small, the model underfits by allowing too much error. As C increases, the model starts fitting the data better, but if C becomes too large, the model may overfit by memorizing the training data and failing to generalize well to the test set.

15.4 What Does C Control in the Training of the SVM?

The parameter C in the SVM algorithm controls the trade-off between achieving a low training error and ensuring a simpler decision boundary that can generalize well. A small value of C allows more slack for the model, meaning it accepts a higher margin of error to avoid overfitting. In contrast, a large value of C penalizes misclassifications more heavily, forcing the model to fit the training data as closely as possible, which can lead to overfitting.
Based on the results, the best configuration is $C = 1000$, which achieves the highest accuracy (80.00%), F1-score (0.8003), and recall (0.8000) with the lowest number of false negatives (162).

15.5 Which is the best-performing kernel? Can you explain why?

The RBF kernel generally performs the best in most practical scenarios. It handles non-linear relationships in the data by mapping it to a higher-dimensional space where a linear decision boundary can be found. This kernel is more flexible compared to the linear and polynomial kernels and is less likely to underfit when there are complex patterns in the data. The polynomial and sigmoid kernels can also be useful, but they may require more tuning and are often less reliable than RBF.

The kernel trick allows SVMs to implicitly compute the mapping into the higher-dimensional space without actually computing the coordinates of the data in that space. This makes the RBF kernel computationally efficient while still being able to capture complex data structures.

15.6 Hyperparameter Tuning and Final Model Selection

In this section, we performed a hyperparameter tuning process to find the best combination of parameters for our Support Vector Machine (SVM) classifier. The hyperparameters explored include the kernel type, regularization parameter C, kernel coefficient γ, and the values for N and H, which control the STFT window size and hop length, respectively.

Although we already knew that the Radial Basis Function (RBF) kernel typically yields the best results for non-linear classification problems, we tested several kernels (`rbf`, `poly`, and `sigmoid`) to gain a more comprehensive understanding of how different kernel functions affect model performance. This approach allowed us to confirm our hypothesis that RBF is indeed the optimal kernel, while also exploring alternatives to ensure a thorough evaluation of the model's behavior under various settings.

The hyperparameter grid we explored was as follows:

- **Kernel Types:** `rbf`, `poly`, `sigmoid`.
- **C Values:** Various values of C from 0.1 to 100000.
- **Gamma Values:** `scale`, `auto`.
- **N Values:** 32, 64, 128.
- **H Values:** 8, 16, 32.

After completing the hyperparameter search, the best configuration found was:

- **Kernel:** `rbf`.
- **C:** 100.
- **Gamma:** `auto`.
- **N:** 64.
- **H:** 8.
- **Accuracy:** 80.49%.

This configuration yielded the highest accuracy of 80.49%, demonstrating the effectiveness of the RBF kernel with these hyperparameters. Although other kernels and parameter combinations were tested, the RBF kernel consistently outperformed the alternatives, confirming its suitability for this classification task.