POLITECNICO
MILANO 1863

Computer Music : Representations and Models

Homework n°1

Politecnico di Milano
Music and Acoustic Engineering
Master Degree 1^{st} year

Written by :
Sebastian Raul Gomez Martinez
Matéo Vitalone

academic year : 2024-2025
Supervisors : Alberto Bernardini,
Riccardo Giampiccolo,
Augusto Sarti

Contents

Introduction

The primary area covered in this report is the classification of electrocardiograms (ECG) using rhythmic features. The aim is to develop an algorithm able to classify ECG waveforms within the five classes of Arrhythmia based on exclusively rhythmic features.
In order to achieve this, we will first obtain the feature vector and then train a classifier with the MIT-BIH dataset. Moreover, using the Jupyter Notebook software, we will show the workflow and explain the methodology in the form of a report.

1 Question 1

1.1 Question 1.1.5

When working with machine learning and deep learning algorithms, the initial step is to load, analyze, and pre-process the dataset. Usually, these models are developed using Python scripts, and datasets are stored locally. Frequently, datasets are formatted as .csv files, which saves us from the hassle of downloading large .wav files. For our ECG classification task, we will use the MIT-BIH dataset. First, we need to install pandas, sklearn, and scikit-learn in our environment.

What does the dataset represent ?
References : [8.1]&[8.1] The MIT-BIH dataset consists of ECG signals designed to identify anomalies in cardiac rhythm, known as arrhythmias. It is frequently utilized for training and testing classification models that work with biomedical signals. To determine the number of items in the datasets, we can use the df.shape and df2.shape functions, which return the following:

$$\textit{Shape of train dataset:} \quad df.shape() \longrightarrow (87553, 188)$$

$$\textit{Shape of test dataset:} \quad df2.shape() \longrightarrow (21891, 188)$$

We can observe that these datasets contain arrays with 188 columns, 87,553 rows for the training set, and 21,891 rows for the testing set.The first element representing the row index, while the second indicates the column index.
Since there is a header for each column, there's no need to include the command function **header=none**, as pandas automatically recognizes the presence of a header, numbering each column from 0 to 187.
The final column of the dataset indicates the class of each ECG, ranging from 0 to 4 in this task. We can utilize **df.iloc[:, -1].unique()** to identify all the unique classes.

1.2 Question 1.1.6

We create a dictionary called label_names that maps the keys **[0, 1, 2, 3, 4]** to the corresponding values (labels) **['N', 'S', 'V', 'F', 'Q']**. These labels represent the five different arrhythmia conditions included in this dataset. Specifically, the labels correspond to **['Normal', 'Fusion of paced and normal', 'Premature ventricular contraction', 'Atrial premature', 'Fusion of ventricular and normal']**, in that order.

1.3 Question 1.1.7

We then want to analyze the distribution of the classes inside the train and test sets. We use the method **value_counts()** to print the number of items of each class (see figure 1.1).

```
Class distribution in train dataset:
187
0.0    72471
1.0     2223
2.0     5788
3.0      641
4.0     6431
Name: count, dtype: int64

Class distribution in test dataset:
187
0.0    18118
1.0      556
2.0     1448
3.0      162
4.0     1608
Name: count, dtype: int64
```

Figure 1.1: Class distribution of the two datasets.

Then, for both train and test datasets, we create a bar plot having on the x-axis the labels and on the y-axis the number of items with that label in the respective dataset (see figure 1.2a and figure 1.2b).

(a) Bar plot of the Train Dataset.

(b) Bar plot of the Test Dataset.

1.4 Question 1.1.8

We now define the sampling frequency of the curves Fs = 360 and plot some of them. Especially, we plot the waveforms in the range (first, last, 10000), where first and last are associated to the indices of the first and last items from the train dataset respectively. In the Jupyter notebook there are more plots from the training dataset, one from every 10,000 signals. The plots in the temporal domain are described in figure 1.3 and 1.4.

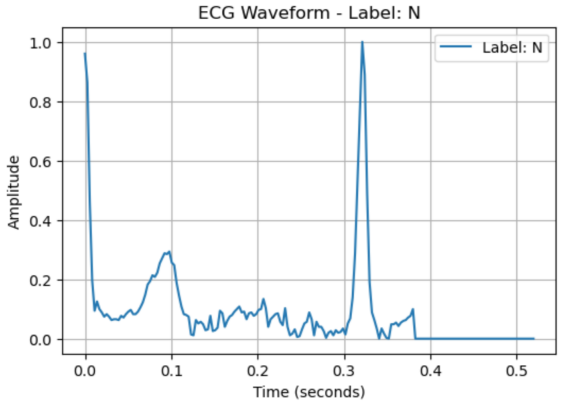

Figure 1.3: Temporal curve for the first item : **Train dataset**.

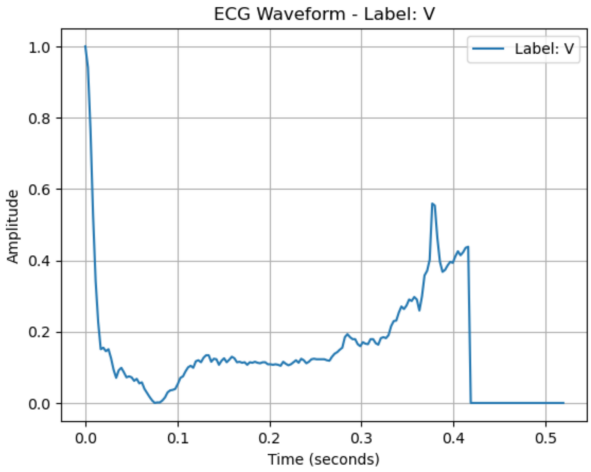

Figure 1.4: Temporal curve for the last item : **Train dataset**.

Typically, the 80 % of the dataset is used for training, while the 20% for testing the model. In our case, the data is already divided into two subsets as we have two distinct files. We will not use the whole dataset for training our classifier because this can easily occupy the whole RAM at disposal to Jupyter Notebook.
Due to this reason, the performance of our classifier will probably not be the best possible but still sufficient for the scope of this study.

In order to subsample the datasets, we import the function train_test split from sklearn.model_selection. Such a function returns two sets drawn from the one in input according to the values provided as argument. We are interested only into the first returned value, thus the second one is to be discarded.

We then apply the function to split the train DataFrame_df using the following values: train_size=0.1 and random_state=28. We also name the first returned value train_df (it is a DataFrame itself) and discard the second one.
We apply once again the function to split now the test DataFrame_df2 using the following values: train_size = 0.1 and random_state=5. We then name the first returned value test_df and discard the second one.
We print the shape of the new DataFrames by lecture we obtain (8755, 188) for train_df and (2189, 188) for test_df (see Jupyter Notebook and figure 1.5).

```
Shape of train_df: (8755, 188)
Shape of test_df: (2189, 188)
Class distribution in train_df:
0.00000000000000000e+00.88
0.0    7240
1.0     201
2.0     622
3.0      57
4.0     635
Name: count, dtype: int64

Class distribution in test_df:
0.00000000000000000e+00.65
0.0    1796
1.0      62
2.0     153
3.0      21
4.0     157
Name: count, dtype: int64
```

Figure 1.5: Display of new DataFrames shapes.

Now, we repeat the last two points of the previous list in order to print and plot the distributions of classes inside the newly defined DataFrames (see figures 1.6 and 1.7).

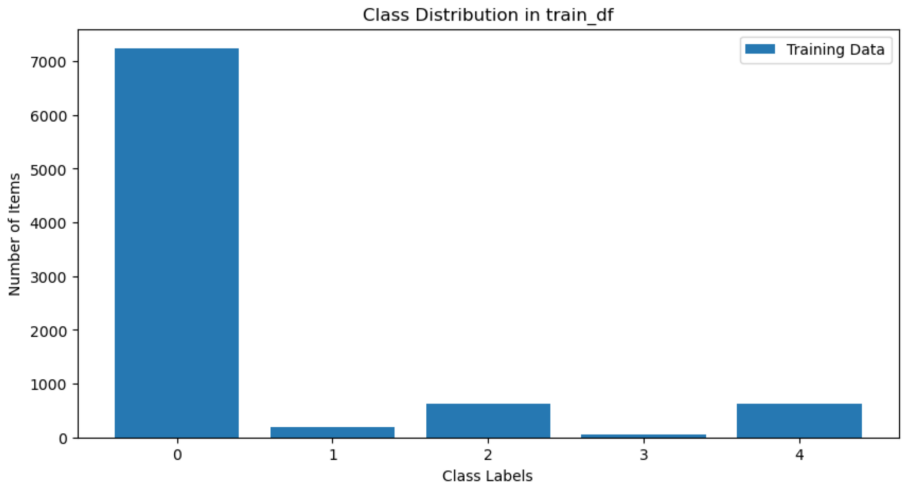

Figure 1.6: Class distributions in the new **train _df**.

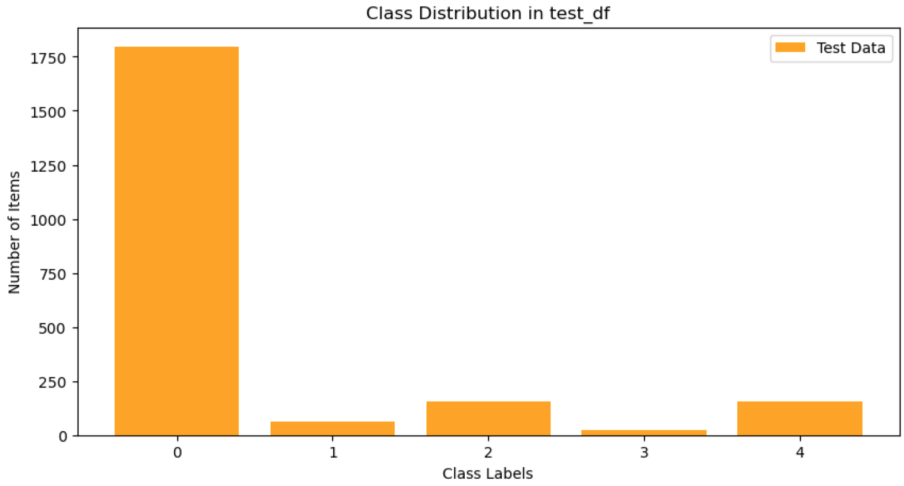

Figure 1.7: Class distributions in the new **test _df**.

2 Question 2

2.1 Question 2.1.5

Now that we have loaded the dataset, we can start processing it. Different curves will have varying dynamic ranges, and for this task, we want the curves to cross zero multiple times. To enhance the generalization capability of the algorithms, we usually normalize the data so that the model can effectively handle waveforms with the same co-domain.
We create the variables train_list and test_list by converting the values from train_df and test_df into lists using the .tolist() method.
Next, we apply preprocessing to both the train_list and test_list. Specifically, we set up a MinMaxScaler with feature_range=(-1, 1). This method is applied to both lists, and the results are stored in each respective set.

The **fit_transform** method calculates the minimum and maximum values from the training set and then applies the transformation. It's crucial to fit it only to the training set to prevent data contamination during evaluation.

The **transform** function uses the previously calculated minimum and maximum values from the training set to transform the test set. This ensures that the scale of the test set aligns with that of the training set (see the scale comparison between the original and scaled versions in Figure 2.1).

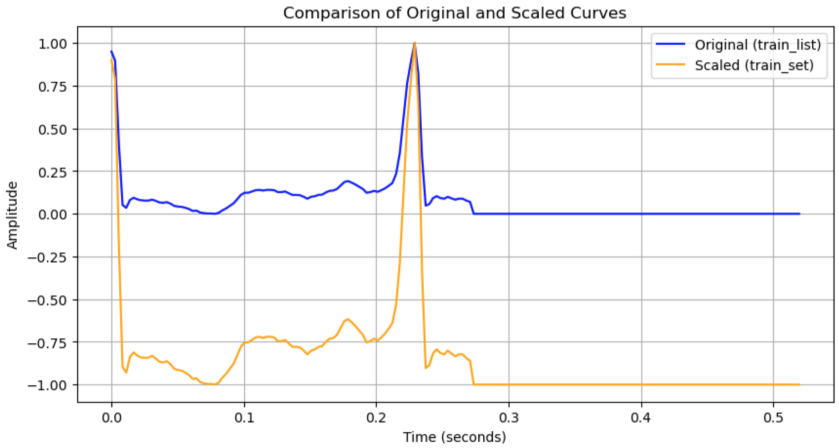

Figure 2.1: Display of the original and the Scaled curves.

In order to accomplish classification, we extract a feature vector containing different rhythmic features. Hence, we define the function compute_feature_vector which takes as inputs the ECG waveform x, the sampling frequency F_s, the length of the window N, the hop size H, and returns the feature vector.

We compute the standard deviation and the mean of the ECG waveform; and the zero-crossing rate using the librosa.feature.zero_crossing_rate function. The ZCR (Zeros Crossing Rate) measures how many times the signal (x_t for e.g) crosses the zero axis per unit of time, and it is useful for analyzing rhythmic properties, such as oscillation frequency (see equation 2.1).

$$ZCR = \frac{1}{T}\sum_{t=1}^{T} 1(x_t \quad x_{t-1} < 0) \tag{2.1}$$

After computing the standard deviation and the mean of this zero-crossing rate, we compute the STFT (Short Time Fourier Transform) of the waveform using the librosa.stft function with n_fft=win_length=N and hop_length=H. Then we compute the absolute value of the STFT and store it into the variable C.
Using the librosa.onset.onset_strength function with

$$S = librosa.amplitude\_to\_db(C, ref = np.max)$$

, we then compute the spectral flux (see equation 2.1)

$$SpectralFlux = \sum_{k=1}^{K}(|C_k(n)| - |C_k(n-1)|)^2. \tag{2.2}$$

The Spectral flux measures the change in spectrum magnitude between consecutive frames. Without librosa, it can be computed as :

$$spectral\_flux = np.sum(np.abs(C[:, 1:] - C[:, :-1]), axis = 0).$$

We compute the standard deviation and the mean of the spectral flux. Finally we define the feature vector f_vector by concatenating six of the the features obtained, in

order to have a vector of 30 elements: the standard deviation and the mean from the signal, the zero crossing rate along with its standard deviation and mean, and the spectral flux along with its standard deviation and mean. We compute and plot the f_vector with the first waveform of the training_set (see figures 2.2 and 2.3).

```
Feature Vector for the First Training Signal: [ 0.16735013  0.08256763  0.          0.          4.26949946  2.19501699
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          2.50307296
 14.4707602   7.62290491  1.6666894   0.07677642  0.          0.        ]
```

Figure 2.2: Display of the list of the feature vector.

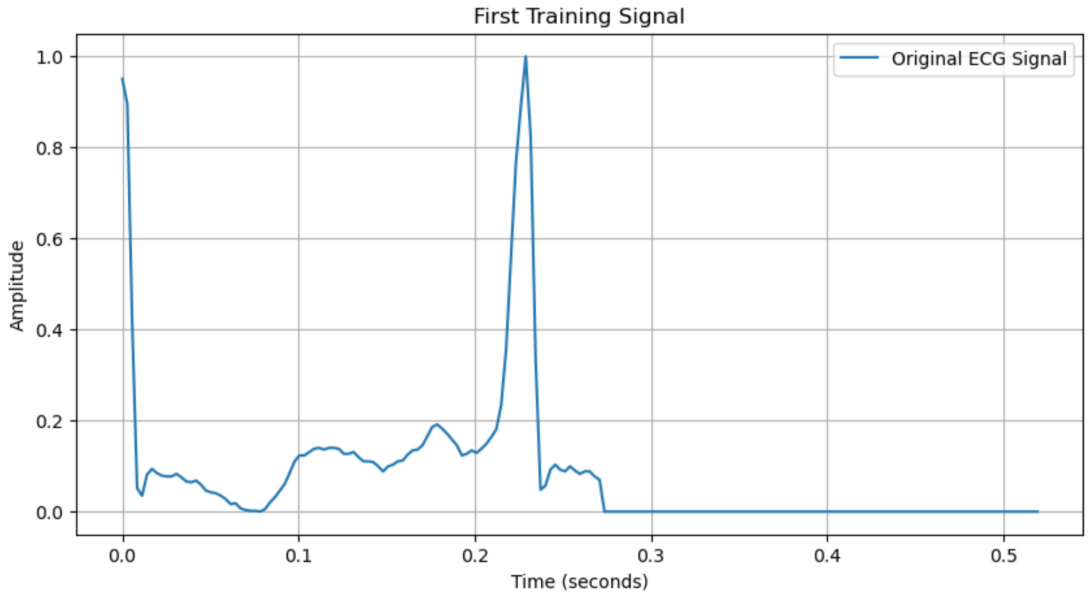

Figure 2.3: Plot of the feature vector.

We compute the f_vector of both the training set and the test set by collecting the results into two lists: train_f_vector and test_f_vector. In order to have a clue on the processing time, we instantiate a progress bar (see in Appendices 7.1) using tqdm, and the shape of the lists are the following:

$$\text{Shape of train\_}fvector : 8755 \times 30$$

$$\text{Shape of test\_}fvector : 2189 \times 30$$

A novelty function helps detecting significant changes in a signal over time. In an electrocardiogram, it can highlight transitions between different patterns of cardiac activity (e.g., changes in rhythm). The advantages of adding different rhythmic characteristics like ZCR and Spectral Flux, are that they complement basic metrics (such as mean and standard deviation), and they also hey help capturing different aspects of the signal, such as periodicity, changes in rhythm, and anomalies in energy flow.
As a trade-off, it has its computational complexity, as adding more features increases processing time. In terms of dimensionality: if the model receives too many redundant features, it may be more difficult to train an effective classifier. Also, some features may

not be useful for distinguishing certain classes and may be removed in future optimizations.

3 Question 3

In this section, we select a model for accomplishing classification. We train a Support Vector Machine (SVM). The main idea of SVMs is finding a frontier which separates observations into classes. In particular, the objective of an SVM classifier is to find the best P-1 dimensional hyperplane – also called the decision boundary which can separate a P-dimensional space into the classes of interest. Notably, a hyperplane is a subspace whose dimension is one less than that of its ambient space. SVM identifies the endpoints or end vectors that support this hyperplane while also maximizing the distance between them.

3.1 3.a) : How SVMs work.

References :[8.2]&[8.2] Support Vector Machines (SVMs) are supervised learning models primarily used for classification and regression tasks. They aim to find the optimal hyperplane that separates data points of different classes in a feature space. When the data is not linearly separable, SVMs leverage kernel functions to map data into a higher-dimensional space where a linear separation is possible.

A kernel is a mathematical function that allows SVMs to find a decision boundary in higher dimensional spaces. This helps to separate non-linearly separable data in the original space.
Common kernels:

- Linear: Useful for data that is linearly separable.
- RBF (Radial Basis Function): handles non-linear separations by transforming the data to a higher dimensional space.
- Polynomial: Generalizes the linear kernel, allowing more complex decisions,
- Sigmoid Kernel: Similar to neural networks.

We use them because they improve performance when data cannot be easily separated into the original space.
To find the Optimal Hyperplane, the SVM algorithm identifies a hyperplane that maximizes the margin, the distance between the hyperplane and the nearest data points (support vectors) of any class. The larger the margin, the better the generalization of the model. When data is not linearly separable, SVMs use kernel tricks to transform the input space into a higher-dimensional feature space where a linear hyperplane can separate the data.

A kernel function computes the similarity between two data points in the transformed space without explicitly performing the transformation. This reduces the computational

cost of working in high-dimensional spaces (see kernel equations below 3.1, 3.1, 3.1 and 3.1):

$$K_{linear}(x, y) = x \quad y, \tag{3.1}$$

$$K_{RBF}(x, y) = (x \quad y + c)^d, \tag{3.2}$$

where c is a constant, and d is the polynomial degree,

$$K_{polynomial}(x, y) = exp(-\gamma||x - y||^2), \tag{3.3}$$

where γ controls the influence of a single training sample,

$$K_{sigmod} = tanh(\alpha x \quad y + c), \tag{3.4}$$

where α and c are hyperparameters.
The choice of kernel depends on several parameters such as the nature of the data, computational constraints, regularization, tuning and scalability:

Nature of Data:

- For linear separability: Use the linear kernel.
- For non-linear patterns: Experiment with RBF, polynomial, or sigmoid kernels.

Computational Constraints:

- Simpler kernels (linear) are faster but may not capture complex relationships.
- Complex kernels (RBF, polynomial) are computationally expensive but may yield better performance.

Regularization and Tuning:

- Regularization parameter C balances the trade-off between maximizing the margin and minimizing classification errors.
- Hyperparameters like γ (RBF), d (polynomial), and others require tuning through grid search or cross-validation.

Scalability:

- For large datasets, linear kernels or approximations (e.g., using kernel approximations) are preferred for efficiency.

In order to improve the SVM performance, some techniques are used in order to get better and faster results. By leveraging appropriate kernels and fine-tuning their parameters, SVMs can achieve excellent performance across a wide range of applications:

Data Pre-processing: Normalize or standardize features to ensure consistent behavior of kernels.
Feature Engineering: Select or construct relevant features to enhance separability.
Kernel Selection and Tuning: Use techniques such as cross-validation to identify the best kernel and its parameters.
Regularization: Adjust C to control overfitting (high C) or underfitting (low C).
Dimensionality Reduction: Use techniques such as PCA to reduce noise and improve model efficiency.

3.2 3.b) : Training of a SVM.

We train a SVM using the sklearn.svm.SVC function, setting C=10 and kernel='rbf'. We save the model using jobling, following the next string format:
f'my_model/svc_kernel_C_C_N_N_H_H'. In this way, we can load it without the need of training from scratch.

3.3 3.c) : Hyperparameter, definition.

Hyperparameters are adjustable model settings that affect a training, as they control how the model learns. Tuning them correctly can improve performance without overfitting the data. In this context, our hyperparameters are the value of C, the kernel type, the window length (N) and the hop size (H).

3.4 3.d) : Accuracy and Over fitting definition.

Over fitting occurs when the model over fits the training data, capturing noise or specific patterns that do not generalize well to new data. To give an example in this context, getting a very high precision on the training set and a low precision on the test set, indicates overfitting.

3.5 3.e) : Classifier and classification accuracy.

We test the classifier in both the training and test sets, and compute their accuracy using the function accuracy_score functions from sklearn.metrics. Precision alone is not enough as it ignores unequal performance between classes. Here, we get in the Training Accuracy: 88.27% and in the Test Accuracy: 87.35%.

3.6 3.f) : Confusion Matrix .

The confusion matrix represents the performance of the classifier by showing the correct predictions (diagonal) and the errors between classes (off-diagonal). It also helps to identify classes where the model works well or fails. We compute it using the confusion_matrix function from sklearn. By plotting this matrix (3.1),we can see that the model has a clear bias towards class N, probably because this is the most represented class in the dataset (dominant class).

This is typical in unbalanced datasets. The model performs poorly on minority classes, such as S and F and Q. These classes are underrepresented in the dataset, and therefore the model cannot learn enough distinguishing features. This problem reinforces the need to balance the dataset (as we will see how to do it with augmentations) to improve the representation of the minority classes.
Some strategies can be used in order to improve the performance, such as data augmentation, penalization of dominant classes (for example, with adjustments in the SVM) or even hyperparameterization to improve the separation between minority classes.

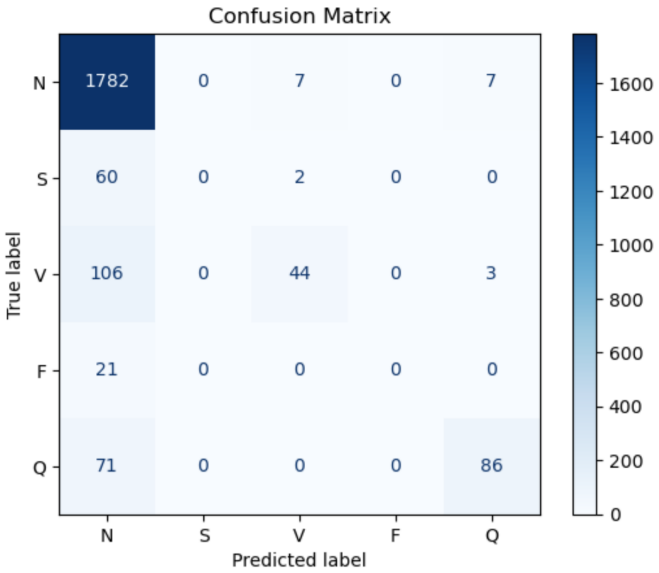

Figure 3.1: Plot of the confusion matrix.

3.7 3.g) : Recall and f1-score.

As the mere accuracy is not enough to verify the performance of a classifier, it is always better to take a look at other metrics. To this aim, we use the function classification_report of the library sklearn.metrics to compute recall and f1-score.
The recall is the proportion of true positives over all real examples in the class(3.7). Thus, we want to maximize recall to reduce false negatives.

$$Recall = \frac{TP}{TP + FN} \tag{3.5}$$

The function f1-score combines precision and recall into a single balanced metric: a high value indicates a good balance between precision and recall.

$$F1_{score} = 2 \quad \frac{Precision \quad Recall}{Precision + Recall} \tag{3.6}$$

3.8 3.h) : Normal - Ill classification.

The confusion matrix is built upon the information provided by the so-called false-positive, false-negative, true-positive, and true-negative. This is valid for two-class classification problems. In multi class classification problems, we cannot formally talk about these metrics, but we can improperly (for example, if we collapse everything into a "normal-ill" classification problem).

- True Positives (TP): Correct predictions for a class. How to compute: Count the diagonal value for the target class in the confusion matrix.

- False Positives (FP): Incorrect predictions of a class (predicted as positive, but is negative). How to compute: Sum the column of the target class (excluding the diagonal value).
- False Negatives (FN): Omitted predictions of a class (actually positive, but predicted as negative). How to compute: Sum the row of the target class (excluding the diagonal value).
- True Negatives (TN): Correct predictions for all other classes. How to compute: Sum all values in the confusion matrix excluding the row and column of the target class.

When we solve tasks in a data-driven fashion, we should always take into account the characteristics of the particular task that we are solving. We do not have just to look at numbers. In particular, our task is to detect an anomaly in the heart rate. In the light of this consideration, apart from the true positive, false negatives are critical, as an error can mean missing a dangerous condition (see the classification report figure 3.2).

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

           N       0.87      0.99      0.93      1796
           S       0.00      0.00      0.00        62
           V       0.83      0.29      0.43       153
           F       0.00      0.00      0.00        21
           Q       0.90      0.55      0.68       157

    accuracy                           0.87      2189
   macro avg       0.52      0.37      0.41      2189
weighted avg       0.84      0.87      0.84      2189

Class N -> TP: 1782, FP: 258, FN: 14, TN: 135
Class S -> TP: 0, FP: 0, FN: 62, TN: 2127
Class V -> TP: 44, FP: 9, FN: 109, TN: 2027
Class F -> TP: 0, FP: 0, FN: 21, TN: 2168
Class Q -> TP: 86, FP: 10, FN: 71, TN: 2022
```

Figure 3.2: Printing the test set classification report.

The difference between the performance on the test and training set can be due to a possible over fitting on the training set, different distribution of classes between the sets. But classes with few examples can also affect performance on the test set.
More represented classes are usually easier to classify, that's why certain classes have higher precision. The dominant class N has more examples, which explains its high performance.

4 Question 4

4.1 Data redefining.

When analyzing the items for each class (as seen in the bar plots), it becomes evident that the dataset is highly unbalanced. This imbalance can lead to biases in the model, as it will predominantly learn from the most represented classes. To address this issue,

the dataset is redefined to ensure that each class has the same number of items, allowing the model to learn from a balanced representation of the input space.

An empty DataFrame named train_df is initialized. Starting from df, a new dataset was extracted by selecting 641 items for each class using the sample (641, random_state=42) method. The loc and iloc functions are used to locate the items belonging to each class before applying the sampling method. The shape of train_df is then printed and verified to be (3205, 188).
The same procedure was applied to the test set. An empty DataFrame named test_df was initialized. Starting from df2, a new dataset was created by selecting 162 items per class using the sample(162, random_state=42) method. The shape of test_df is also printed and verified to be (810, 188). Finally, the values of train_df and test_df are converted into lists for further processing. However, this process is constrained by the number of items in the smallest class, for example, 641 for the training set and 162 for the test set.

4.2 Define augmentations.

We know that the performance of machine learning models is strictly dependent on data, and, typically, the more the data the better the performance. Thus, in order to remove the constraint imposed by the dataset, we can rely on data augmentation techniques. The purpose of such techniques is on the one hand to increase the number of tracks in the train dataset, on the other hand to improve the robustness and generalization of the model. The augmentations are typically applied with a certain probability in order to prevent biases in the datasets. The following steps are performed:

First, a class name augment is defined. The constructor, in this case, is not responsible for any particular setting.
A class method named stretch is defined, which has as arguments self and the signal x, and performs the following operations:

1. The new number of samples l is defined as seen in equation: 1 :

$$l = 87(1 + \frac{\beta - 0.5}{3}) \qquad , with \quad l \quad must \in \mathbb{N} \tag{4.1}$$

   where β is a random number in [0, 1] and can be computed with the random.random() function. l is casted as an int;

2. The signal x is resampled to have l samples using the scipy.signal.resample function, and the result is assigned to the variable y;

3. The following conditional structure is implemented: if $l < 187$ create an np.array y1 full of zeros of shape (187,) and assign the slice y1[:l] = y. Else, assign the slice y1[:187] = y;

4. y1 is returned.

A second class method named amplify is also defined, which has as arguments self and the signal x, and performs the operations:

1. Generate a random number $\alpha \in$ [-0.5, 0.5];
2. Return the value $x(\alpha + 1)$.

4.3 Method to apply augmentations.

Now, we need a method that applies such augmentations. A class method named perform is defined, which has as arguments self, x. As mentioned above, we want the augmentations to be randomly applied. In other words, we want the method to apply stretch, amplify, both, or neither of them in a random fashion. To this aim, these following operations are performed:

1. An empty variable named performed_augmentations is initialized. In this variable, it is necessary to store the augmentations that are performed in order to keep track of them (since we do not know which ones are applied as they are random);
2. The conditional structure is implemented: if the np.random.binomi.al(1, 0.5) == 1, apply the stretch to x and add the string 'stretch' to performed_augmentations;
3. The conditional structure is implemented: if the np.random.binomial(1, 0.5) == 1, apply the amplify to x and add the string 'amplify' to performed_augmentations;
4. The values x, performed_augmentations are returned.

At this point, the class augment is ready to test it. Thus:
• The class augment is instantiated.
• The augmentation is performed on the first item of train_list (remembering to remove the last column) and the list of augmentations is printed, with the help of a loop that helps to prove the randomness application of the methods.

We can then check by printing the list of the method applied three different times (see figure 4.1).

```
Run 1 - Performed Augmentations: ['amplify']
Run 2 - Performed Augmentations: ['stretch', 'amplify']
Run 3 - Performed Augmentations: ['stretch']
```

Figure 4.1: List of the method applied three different times.

Then we plot the augmented version on top of the original one, with labels and a legend to facilitate the comprehension (see Figure 4.2)

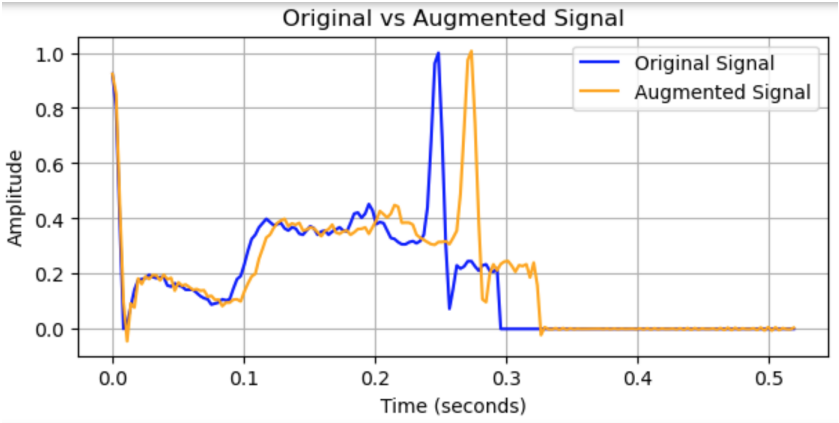

Figure 4.2: Comparison plot between original and augmented with the stretch method.

4.4 Apply augmentations.

Then it is time to apply such augmentations to our train dataset. In particular:

- The variable containing the number of waveforms per class that we want to add to our dataset is defined as n_aug=100.
- The class is instantiated. Then, for each class, starting directly from df and extracting n_aug items using sample(n_aug, random state=16). The methods loc and iloc are exploited to obtain the subset of df related to a certain class.
- For each of the n_aug items the augmentation is performed (removing the last column) and concatenating the augmented waveform to train_df.
- The counts for each class and the shape of train_df is printed in some of the previous points. The shape of the printing is (3705, 188) and an equal count of 741 per each class.
- The counts for each class and the shape of train_df is printed in some of the previous points. The shape of the printing is (3705, 188) and an equal count of 741 per each class.
- Both train_df and test_df are converted into lists with names train_list and test_list, respectively.
- A new MinMaxScaler is fitted and applied to normalize both train_df and test_df.

Other enhancements that could be implemented to serve the scope of this study and that were not implemented are:

- Random Noise: Add white Gaussian noise to signals.
- Phase Inversion: Invert signals to vary their behavior.
- Temporal Shifting: Move signals forward or backward.

5 Question 5

5.1 Train classifier in the new datasets.

It is time to train our new classifier on the augmented dataset and perform the classification in order to verify if the new dataset is actually able to provide us with a better classification. To do so:

- The feature vectors of train_set and test_set are computed, and the classifier is trained starting with a kernel rbf and C=10.
- The trained classifier is tested on the test set.

The accuracy of both the train and test sets is then printed:

Test Accuracy: 71.23%.
Training Accuracy: 74.79%

5.2 Analysis of the new statistics.

In the previous case, the model achieved 87.35% accuracy on the test set, but that high accuracy was due to a bias towards the dominant classes, since the dataset was unbalanced. Now, with the balanced and augmented dataset, the precision dropped to 71.23%, which indicates that the model is having difficulty learning from minority classes, although it now represents them better. This reflects a more realistic scenario, where the model no longer focuses only on the dominant class, but instead attempts to classify equally among all classes.
With a Training Accuracy of 74.79% and a Test Accuracy of 71.23%, the difference between the two is only 3.56%, indicating that there is no significant over fitting. The model generalizes reasonably well between training data and test data.
This happened because now the dataset is balanced: previously, the model prioritized the dominant class ("N"), which represented the majority of the data. Now, all classes have equal weight, forcing it to learn from everyone, including the minority ones. Although augmentation improves the representation of minority classes, it also introduces variability that can make initial learning more difficult. By this moment, the current model may not be fully optimized for this new dataset. Hyperparameters like C, the kernel, or even the window size N and displacement H may need adjustments.
The confusion matrix is plotted for our prediction (see figure 5.1) and using the function classification_report to obtain accuracy, recall, and f1-score values (see figure 5.2).

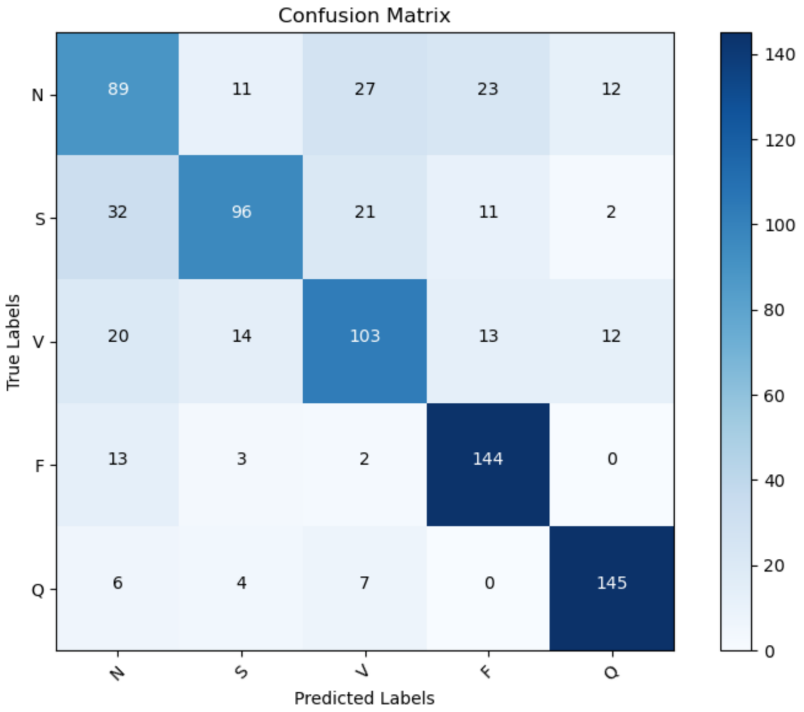

Figure 5.1: Plot of the confusion matrix.

Classification Report:

	precision	recall	f1-score	support
N	0.56	0.55	0.55	162
S	0.75	0.59	0.66	162
V	0.64	0.64	0.64	162
F	0.75	0.89	0.82	162
Q	0.85	0.90	0.87	162
accuracy			0.71	810
macro avg	0.71	0.71	0.71	810
weighted avg	0.71	0.71	0.71	810

Figure 5.2: classification_report display of accuracy, recall, and f1-score values

In general, the performance is more balanced with respect to the previous case, especially in the minority classes (F and Q). However, classes N, S, and V have greater difficulties due to frequent confusion with other classes. This reflects that these classes have more similar characteristics. Balancing and augmentation significantly improved the recognition of classes F and Q, but did not completely solve the problems with N, S, and V.

Regarding the recall:
Classes F and Q: The recall is excellent (89% and 90% respectively) and significantly better than in the previous case.
Classes N, S, and V: The recall is moderate (55%, 59%, and 64%), but slightly lower than the previous case due to the confusion added by the balancing.
In terms of f1-score:
F and Q: The f1-score values are high due to the combination of good recall and precision.

N, S, and V: Have a lower f1-score, reflecting that the model is not consistently classifying these classes.

Also the metrics of TP, FP, TN and FN are printed (see figure 5.3).

```
Class 0: {'TP': 89, 'FP': 71, 'FN': 73, 'TN': 577}
Class 1: {'TP': 96, 'FP': 32, 'FN': 66, 'TN': 616}
Class 2: {'TP': 103, 'FP': 57, 'FN': 59, 'TN': 591}
Class 3: {'TP': 144, 'FP': 47, 'FN': 18, 'TN': 601}
Class 4: {'TP': 145, 'FP': 26, 'FN': 17, 'TN': 622}
```

Figure 5.3: display of metrics : TP, FP, TN and FN for each class.

Chosen Metric: False Negative (FNs)
As discussed in Question 1.3, FNs are critical in this context because they represent real anomalous signals that the model fails to identify.

- Class N: 73 FN
- Class S: 66 FN
- Class V: 59 FN
- Class F: 18 FN
- Class Q: 17 FN

The reduced number of FNs for classes F and Q shows significant improvement in these categories. However, the high number of FNs for classes N, S, and V indicates that the model still has problems correctly capturing signals from these classes.

5.3 Varying hyperparameters.

We compared the performance by testing different parameter configurations, keeping the kernel fixed. Using all the values in the set C=0.1, 1, 100, 1000, 10000, 100000. The performance increases as C increases, but up until a specific limit (see figure 5.4)

```
C: 0.1, Test Accuracy: 61.98%
C: 1, Test Accuracy: 66.54%
C: 10, Test Accuracy: 71.23%
C: 100, Test Accuracy: 75.06%
C: 1000, Test Accuracy: 75.43%
C: 10000, Test Accuracy: 74.07%
C: 100000, Test Accuracy: 72.47%
```

Figure 5.4: Comparison of the different accuracy tests made on the test dataset with different values of C.

C is a regularization parameter in SVM. It controls the trade-off between achieving a low error on the training data and maintaining a margin that allows the model to generalize well. C determines how much the model penalizes mistakes in classification, it controls whether the model focuses on the broader structure of the data (low C) or individual training points (high C).

A low C emphasizes a larger margin at the cost of allowing "misclassifications" on the training data, and makes the model less sensitive to individual data points (less prone to over-fitting), but it might under-fit.

A high C penalizes "misclassifications" more heavily, leading to a smaller margin. Tries to fit the training data more closely, but may over fit and fail to generalize to unseen data.

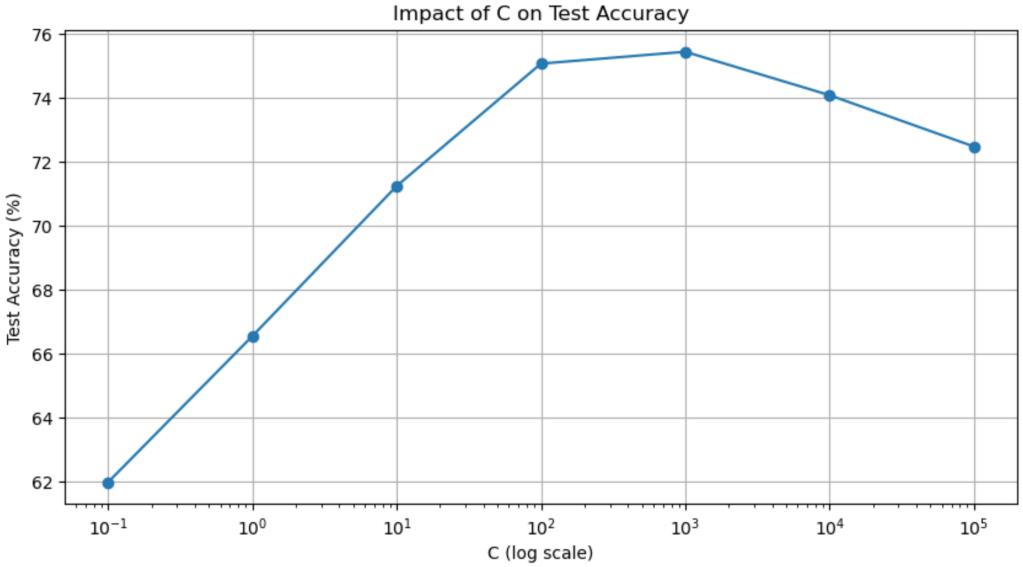

Figure 5.5: Graph showing the impact of C when it varies in the accuracy.

For C = 0.1, the test accuracy is 61.98%, which indicates under-fitting. The model prioritizes a larger margin and tolerates too many training errors, making it unable to capture the data's complexity.
As C increases, the test accuracy improves: when C=10, it achieves 71.23%, and when C=100, it improves to 75.06%. This suggests that the model is balancing the margin and the training errors effectively.
Beyond C=1000, the test accuracy stops improving and starts to decline slightly: C=10000 and C=100000 show 74.07% and 72.47%, respectively. This is a sign of over fitting: the model is over-penalizing training errors and losing its ability to generalize. The best value of C in this case is 1000, which achieves the highest test accuracy of 75.43%.

In this case, using a C=1000, the best kernel is the rbf, whit whom the test accuracy reaches 75.43%, but with the poly one following very close with a 74.57%. (see figure 5.6).

```
Kernel: linear, Test Accuracy: 69.26%
Kernel: poly, Test Accuracy: 74.57%
Kernel: rbf, Test Accuracy: 75.43%
Kernel: sigmoid, Test Accuracy: 30.00%
```

Figure 5.6: Comparison of the different accuracy tests made on the test dataset with different kernels.

The reasons why RBF kernel performs the best are the following:

- It handles non-linear separability efficiently.
- It adapts itself to the complex, augmented dataset better than linear or polynomial alternatives.
- It also provides a good balance between precision and recall across all classes.

Non-Linear Data Separation: The RBF kernel is particularly effective for datasets where the decision boundaries are not linearly separable. It maps the data into a higher-dimensional space, enabling the model to find more complex boundaries.

Flexibility:
Unlike the linear kernel, which assumes classes can be separated with a straight line, the RBF kernel adapts to the complexity of the dataset.

Balance Between Complexity and Overfitting:
The RBF kernel balances well between capturing intricate patterns in the data and avoiding overfitting, especially when paired with a well-chosen C.

A comparison with other Kernels can be done by plotting the impacts of Hyperparameters over their test accuracy (see Figure 5.7).

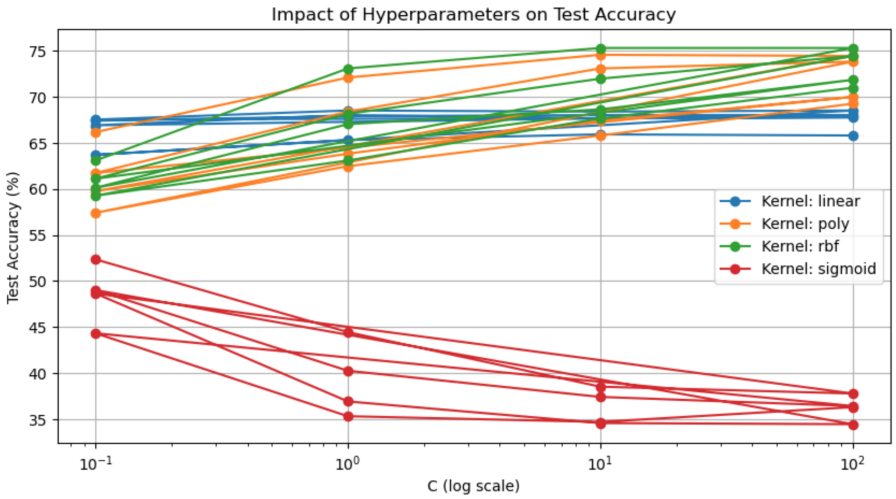

Figure 5.7: Comparison of the different accuracy tests made on the test dataset with different kernels.

The linear Kernel seems to work well for simple datasets but struggles with the augmented ECG data due to overlapping classes. Likely, it gets underperformed due to the non-linear separability of the classes.
On the other hand, the polynomial Kernel requires careful tuning of the polynomial degree. It tends to over-fit with higher degrees and under-fit with lower ones, making it less consistent.
In the case of the sigmoid Kernel, it often behaves like a poor approximation of the RBF kernel and struggles to capture the complexity of this dataset.

6 Conclusion

We have different performance with this new dataset with respect to the previous one thanks to the dataset augmentation and balancing: the new dataset provides equal representation for all classes, forcing the model to learn features from minority classes that were previously underrepresented. The balanced dataset prevents the model from focusing too heavily on the dominant class (e.g., "N"), which might explain the lower overall accuracy compared to the unbalanced dataset. Also because of the augmentation complexity: the augmented data introduces variability (via stretch and amplify operations), which can make the learning process more challenging for the model. However, this variability also improves the robustness of the model for real-world signals.

Comment on the confusion matrix:
As strengths, the classes F and Q seem well classified with high precision and recall, indicating that the augmentations have helped the model learn these minority classes effectively. With fewer False Negatives, the model successfully identifies a higher proportion of positive samples for F and Q compared to previous results.
As weaknesses, classes N, S, and V still exhibit confusion with one another, which is evident in the False Positives and False Negatives for these classes. The model struggles to separate these classes likely due to overlapping features in the input space.

The classifier still shows a bias toward F and Q, likely because these classes are more distinct compared to the others. For N, S, and V, the bias is less evident, as the model struggles to differentiate them consistently.

6.1 Improvements.

To further improve the performance, the following can be considered:

Model Enhancements:

1. Weighted SVM: Assign higher penalties to errors in minority classes using the class_weight parameter in SVM, which can help reduce False Negatives.

2. Use Advanced Models: Consider tree-based models (e.g., Random Forest, Gradient Boosting) or neural networks for potentially better performance.

Data Improvements:

1. Additional Augmentations: Apply more advanced augmentations, such as noise injection or time warping, to further enrich the dataset.

2. Feature Engineering: Extract additional features (e.g., frequency-domain features or morphological characteristics) to provide the model with more discriminative power.

Valuation Improvements:

1. Error Analysis: Analyze misclassified samples to identify specific patterns or features that might be misleading the model.

7 Appendices

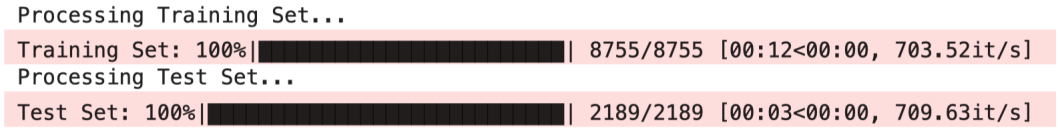

Figure 7.1: Display of the progress bar and processing to compute the feature vector..

8 Bibliography

8.1 MIT-BIH

[1] **MIT-BIH Arrhythmia Database v1.0.0**,
https://www.physionet.org/content/mitdb/1.0.0/, last consulted on 29/11/2024.

[2] **MIT-BIH Arrhythmia Database (Simple CSVs)**,
https://www.kaggle.com/datasets/protobioengineering/mit-bih-arrhythmia-database-modern-2023, last consulted on 29/11/2024.

8.2 Kernels

[3] **Kernel method**, Glossary,
https://www.engati.com/glossary/kernel-method, last consulted on 29/11/2024.

[4] **Kernel method**, Wikipedia Article,
https://en.wikipedia.org/wiki/Kernel_method, last consulted on 29/11/2024.