ECG BASED BIOMETRICS

Digital Signal Processing

مريم خالد محمد عبد الرحيم محمد	2022170413
مريم احمد صلاح احمد محمد	2022170409
مصطفى احمد محمد فهيم	2022170623

Introduction

Biometric systems play a crucial role in applications where security and authenticity are paramount. Among various biometric traits, the electrocardiogram (ECG) has emerged as a significant tool, leveraging its role as a diagnostic measure for cardiac activity over decades. ECG records the electrical activity of the heart over time, providing insights into the subject's underlying cardio-physiology.

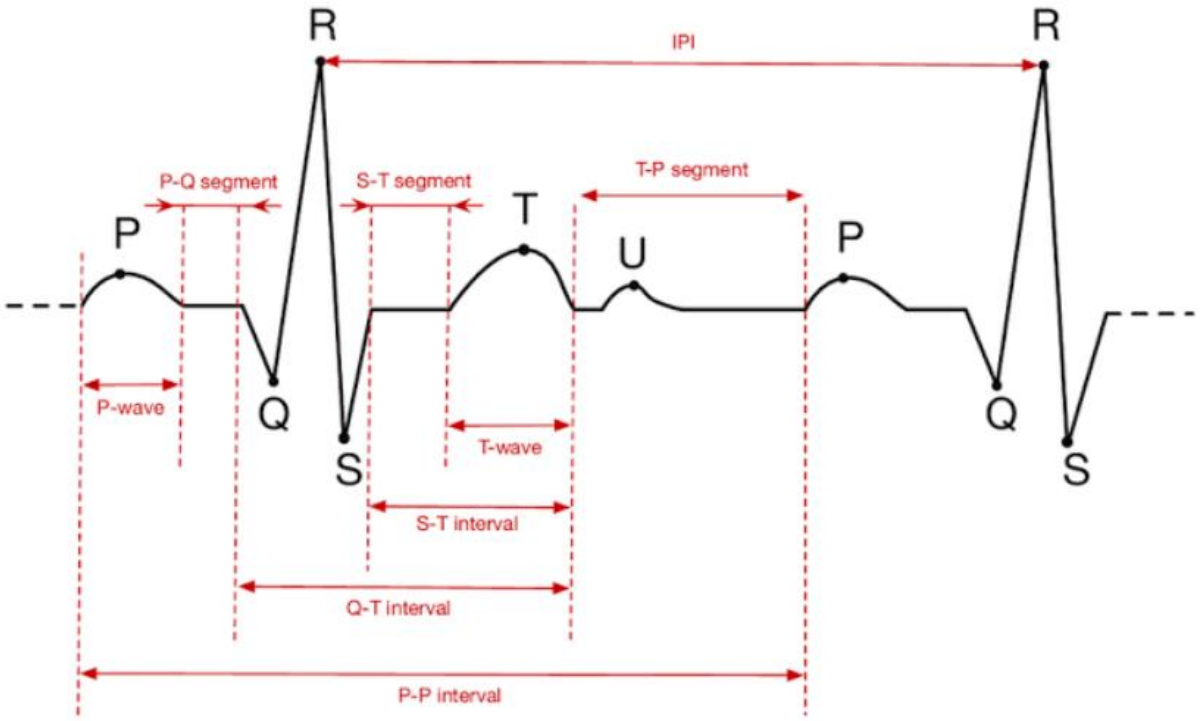
A detailed examination of each heartbeat in an ECG signal reveals distinct features, commonly referred to as waves—labeled P, QRS, and T—appearing in a specific temporal sequence. The physiological and structural variations in the heart among individuals result in unique ECG patterns, establishing its potential as a reliable biometric identifier.

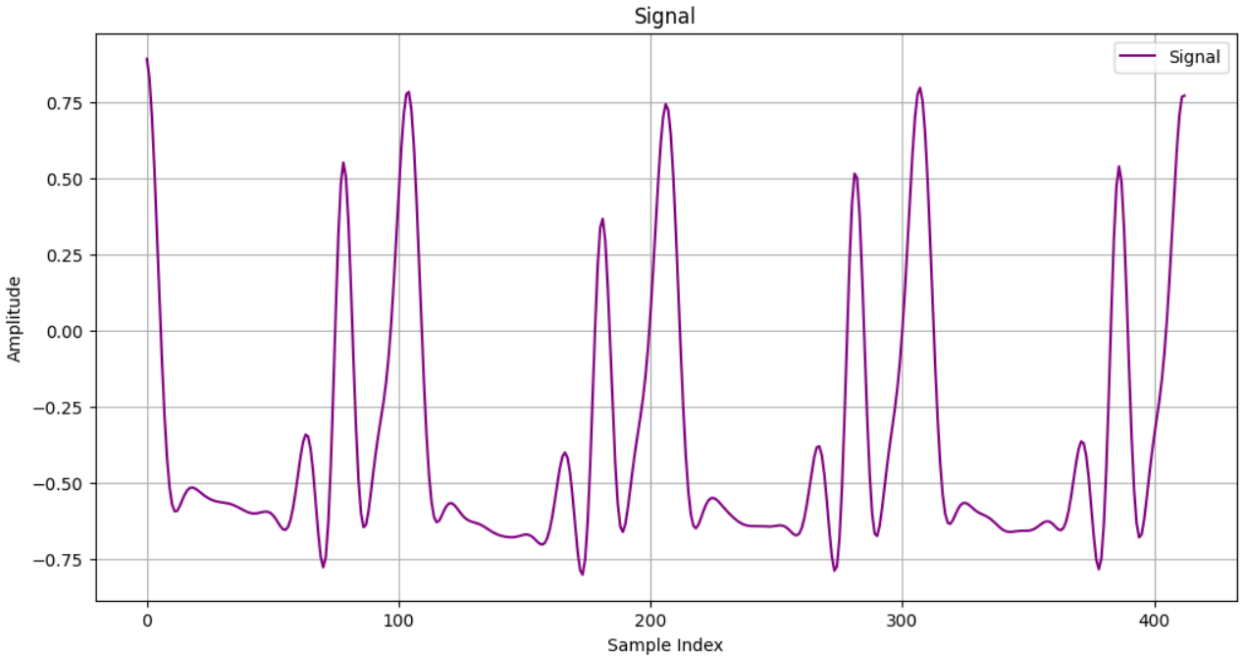
The distinctiveness of ECG signals lies not only in their individuality but also in their inherent properties. As a biological signal, ECG acts as a "life indicator," making it a robust tool for aliveness detection. Unlike other biometric traits, ECG signals are challenging to spoof or falsify, enhancing their security profile.

This project aims to leverage the unique characteristics of ECG signals to distinguish between three individuals. By analyzing their distinct cardiac patterns, this study explores ECG's viability as a novel biometric trait for secure and reliable identification systems.

Steps

Firstly, we set up the necessary imports and prepare the environment for preprocessing ECG signals, extracting features, and building a KNN classification model. We also load the training and testing datasets for the three individuals (S1, S2, S3). Secondly, we implemented functionalities needed for preprocessing to apply on training and testing data sets. After that, we extracted data frames from each signal file to concatenate and shuffle into one data frame to train and test the KNN model on.





Signal Segmentation: 4 heart beats/ segment

1.	Butterworth bandpass filter [1-40hz]	Essential for noise removal. For an **ECG signal** with a **Butterworth bandpass filter** having a passband of **1-40 Hz**, the typical order would range between **4 and 8**. A higher order will give us a sharper roll-off, but it may come with the risk of increasing computational cost or complexity. Accordingly, we chose 4 as order for the filter with 1000 sampling rate.
2.	Mean removal	To center the data around zero. The main goal of this step is to eliminate the DC component (the constant offset) from the signal. This step helps to ensure that the heart rate and other characteristics of the ECG waveform are not skewed by any constant offsets.
3.	Normalization [-1,1]	A **[-1,1]** normalization to preserve the bipolarity of the ECG signals (they have both +ve and –ve components). The resulting signal will be easier to process and compare, ensuring that the KNN model can treat all the data consistently, without being affected by differences in signal amplitude.
4.	Down sampling	Helps to reduce the size of the dataset while still retaining important features for subsequent analysis. Therefore, the algorithm can process a smaller dataset without significant loss of the heart signal's vital characteristics, improving both performance and speed, as a result, a **target sampling rate of 500** is chosen. Besides, the down sampling factor determines how much the signal will be compressed. `downsampling_factor= sampling_rate/target_sampling_sampling_rate`
5.	Segmentation	The goal of segmentation is to isolate specific portions of the ECG signal corresponding to individual heartbeats. To detect peaks, we plotted the down sampled signal to detect at which height we can consider as the signal's R-peak and the minimal horizontal distance in samples between neighboring peaks to calculate the number of peaks, by grouping each 4 beats into a segment after determining the start and end of each segment, we managed to extract all the segments for each signal separately.
6.	Auto-correlation & DCT	The autocorrelation is calculated to identify how much the segment resembles itself at different time lags. The DCT is applied to the autocorrelation to transform the signal from time domain to frequency domain. The non-zero DCT coefficients represent the features of the signal. We limited them to the **15 coefficients** to prevent overfitting.

7.	Dataset Construction (combining and shuffling for train and test)	• Converting the list of feature sets `(S_features)` into a pandas DataFrame with each feature as a separate column. Additionally, we add a target column to represent the label. Done for each individual (MK, Mostafa, Mariam). • Concatenating the dataframes of all individuals (MK, Mostafa, Mariam) into a single dataframe and shuffles the data to ensure that the model receives a mixed set of examples during training. • Same is done for the testing data set.

The data is now ready for training and testing the KNN model.

Model

This process demonstrates how to split data for training and testing, train a KNN model with hyperparameter tuning, evaluate its performance using a confusion matrix, and save the best model for future use. The final step makes predictions on new data using the optimized KNN model. The entire workflow is designed to build, evaluate, and deploy a robust machine learning model for classifying biometric data.

STEP 1: **Splitting Data into Features and Labels**

The dataset is split into features `(X_train and X_test)` and labels `(y_train and y_test)`. The features are the attributes used to predict the target class.

STEP 2: **Training the K-Nearest Neighbors (KNN) Model with Varying Number of Neighbors**

Using only the number of neighbors, we experimented with different values of `n_neighbors`. The loop iterates through a range of using values for `n_neighbors`, training a KNN model for each value. The `model.fit(X_train, y_train)` trains the model the training data, and `model.score(X_train, y_train)` and `model.score(X_test, y_test)` evaluate the model's accuracy on the training and testing datasets, respectively. The results for each `n_neighbors` value are printed, allowing us to assess how the number of neighbors affects model performance.

STEP 3: **Hyperparameter Tuning Using Grid Search**

As the previous step wasn't efficient enough, this led to performing hyperparameter tuning using grid search with cross-validation to test possible combinations of hyperparameters for the KNN model, such as: `n_neighbors`, `weights`, `metric`, and `p`.

`n_neighbors`: the number of nearest neighbors considered by the KNN classifier

`Weights`: determines how the neighbors' contributions are weighted when making a prediction

(uniform, distance)

`Metric`: defines the distance metric used to calculate the distance between points.

(minkowski, Euclidean, manhattan)

`P`: is used only when the **Minkowski** distance metric is chosen. It controls the power of the distance calculation and determines how the distance is computed. (1,2)

The grid search evaluates all combinations of these parameters, selecting the one that results in the best performance (using cross-validation). After fitting the grid search, the best combination of hyperparameters is extracted and used to evaluate the accuracy of the model on both the training and testing datasets.

STEP 4: **Generating and Displaying the Confusion Matrix**

A confusion matrix is generated to evaluate the classification performance of the KNN model. The `confusion_matrix(y_test, y_pred)` compares the predicted labels `(y_pred)` with the true labels `(y_test)`. The confusion matrix is then displayed using a heatmap, which visually represents how well the model performed across different classes. The heatmap provides insights into true positives, false positives, true negatives, and false negatives, helping to understand where the model might be misclassifying.

STEP 5**: Saving the Best Model and Reloading It for Future Use**

Once the model is trained and optimized, it is saved to a file using **Joblib** (`joblib.dump`). This allows the trained model to be reloaded later for predictions or further analysis and effective usage in our GUI, without needing to retrain it each time. The saved model is then reloaded using `joblib.load(joblib_file)` to verify that the model can be successfully reloaded.

Finally, the trained model is used to make predictions on new input data. The `predict_person` function loads the saved model and makes a prediction using the provided `input_data` (which is a subset of the test dataset without the target column). The predicted class label (which corresponds to an individual) is returned and displayed, providing a prediction based on the features of the test data.