



សាកលវិទ្យាល័យភូមិន្ទភ្នំពេញ

Royal University of Phnom Penh

មហាវិទ្យាល័យវិស្វកម្ម

Faculty of Engineering

ដេប៉ាតឺម៉ង់វិស្វកម្មទូរគមនាគមន៍ និងអេឡិចត្រូនិច

Department of Telecommunications and Electronic Engineering

របាយការណ៍៖

Deployment of Atrial Fibrillation Detection on ECG Records Using Convolutional Neural Networks (CNN)

និស្សិត : ហេង លីម៉េង (Heng Lymeng)

គ្រូជំងនាំ : យី រដ្ឋា (Yeu Ratha)

ថ្ងៃទី..០៧...ខែ.....កក្កដា...ឆ្នាំ...២០២៥..

Abstract

Atrial fibrillation (AFIB) is a common cardiac arrhythmia associated with increased risk of stroke and heart failure. Early detection is crucial for timely treatment. This project presents a method for detecting AFIB using rhythm-based features—specifically RR intervals—processed through a one-dimensional Convolutional Neural Network (1D CNN). ECG signals are extracted from WFDB records and stored in JSON format for analysis. A web application processes the raw ECG data, performs classification, and predicts the presence or absence of AFIB. The result is transmitted wirelessly to an ESP32 device, which indicates AFIB detection with a yellow light and normal rhythm with a green light.

Keywords: Atrial Fibrillation, Signal Processing, Deep Learning, ECG, AI Deployment, 1D CNN, ESP32.

Table of Contents

Abstract	i
I. Introduction.....	1
II. Machine Learning Model.....	1
2.1 Data collection and Data Cleaning	2
2.2 Data pre-processing	6
2.3 Convolutional Neural Network (CNN) Model.....	14
2.4 Model Training	18
2.4 Model Evaluation	19
III. Electronic Integration	23
3.1 Coding	23
IV. Application Deployment.....	28
4.1 Generated ECG Signal for application Testing	28
4.2 Web Application using Flask	29
4.3 Results and Discussion	33
V. Conclusion	37
5.1 Evaluation	37
5.2 Improvement	37
5.3 Future Work.....	37
References.....	38
Replication Basis.....	38
Appendix.....	38
Appendix A: Python – ECG Preprocessing, RRI Extraction and CNN Model Architecture (Code)	38
Appendix B: ESP32 Microcontroller Code (Arduino IDE)	47
Appendix C: Web Application – Flask and Frontend	49
Appendix D: Sample ECG JSON Files	51
Appendix E: Web Application Structure Application – HTML, CSS and Java Script	54

I. Introduction

Cardiac arrhythmias are caused by irregular electrical activity in the heart, leading to abnormal heartbeat rhythms as shown in fig 1. Among these, *Atrial Fibrillation (AFIB)* is one of the most common and clinically significant arrhythmias, especially prevalent in the aging population. AFIB is characterized by rapid and disorganized electrical signals in the atria, which can lead to inefficient blood pumping, increasing the risk of stroke, heart failure, and other complications.



Normal Heart Rhythm



Atrial Fibrillation Heart Rhythm

Fig 1: Atrial Fibrillation Heart Rhythm

Approximately 5% of people aged over 65 years and 10% of those over 80 years are affected by atrial fibrillation (AF) (Keech, Punekar, & Choy, 2012) [1]. The prevalence of AF in the general adult population is estimated to range between 0.4% and 2% [2]. While AF itself is not always immediately life-threatening, it is strongly associated with severe cardiovascular complications such as *stroke* and *heart failure* [3]. The *aging population* is a key factor contributing to the increasing prevalence of AF, and this trend is projected to pose a growing public health challenge in the coming decades [4]. This study is increasingly important for the detection of atrial fibrillation (AF), a type of irregular heartbeat.

II. Machine Learning Model

Here is Some essential Header for our Machine learning model

```
import wfdb
from wfdb import processing
import numpy as np
import scipy.signal as sp
import matplotlib.pyplot as plt
import neurokit2 as nk
```

- **WFDB:** Used to access and process ECG signal records from the PhysioNet waveform database (WFDB). It provides tools for reading, writing, and analyzing physiological signals.
- **NumPy (np):** Provides powerful numerical operations and array handling, which are essential for efficient computation and data manipulation.
- **SciPy.signal (sp):** Used for advanced signal processing tasks such as filtering, peak detection, and signal transformation.

- **Matplotlib.pyplot (plt):** A plotting library used to visualize ECG signals, R-peaks, and model outputs.
- **NeuroKit2 (nk):** A high-level library designed for neurophysiological signal analysis; used here to extract **R-peaks** from the ECG signals, which are essential for RR interval calculation.

2.1 Data collection and Data Cleaning

Data were taken from the *MIT-BIH Atrial Fibrillation Database (MIT-BIH AFDB)* [5]. The MIT-BIH AFDB includes 25 long-term ECG recordings, of which 23 recordings contain two ECG leads labeled 'ECG1' and 'ECG2', sampled at 250 Hz. Rhythm annotation files are also provided, manually labeled with rhythm types such as *AFIB* (atrial fibrillation), *AFL* (atrial flutter), *J* (junctional rhythm), and *N* (normal rhythm). For this study, AFIB segments from 11 ECG signals were selected based on availability, readability, and clear annotation data—signals that were unavailable, unreadable, or lacked a defined start time were excluded.

```
record_list = [
    '00735', '03665', '04015', '04043', '04048', '04126', '04746',
    '08378', '08405', '08434', '08455'
]
```

Download record from each record

```
for record in record_list:
    wfdb.dl_database('afdb', dl_dir='afdb', records=[record], overwrite=True)
```

Result:

<pre> v AF_Detection v dataset v afdb ≡ 00735.atr ≡ 00735.he ≡ 00735.qrs ≡ 03665.atr ≡ 03665.he ≡ 03665.qrs ≡ 04015.atr ≡ 04015.dat ≡ 04015.he ≡ 04015.qrs </pre>	<pre> ≡ 04043.atr ≡ 04043.dat ≡ 04043.he ≡ 04043.qrs ≡ 04048.atr ≡ 04048.dat ≡ 04048.he ≡ 04048.qrs ≡ 04126.atr ≡ 04126.dat ≡ 04126.he ≡ 04126.qrs ≡ 04746.atr </pre>
---	---

Function to Extract feature from each record.

```
def extract_windows(record_name, window_size_sec=10, lead=0):
    try:
        record = wfdb.rdrecord(f'afdb/{record_name}')
        ann = wfdb.rdann(f'afdb/{record_name}', 'atr')
    except Exception as e:
        print(f"[ERROR] Skipping record {record_name}: {e}")
        return []

    fs = record.fs
    window_size = int(fs * window_size_sec)

    if record.p_signal.shape[0] < window_size:
        print(f"[WARNING] Record {record_name} too short. Skipped.")
        return []

    signal = record.p_signal[:, lead:]
    segments = []

    for i in range(len(ann.aux_note)):
        if ann.aux_note[i].startswith('('):
            label = ann.aux_note[i][1:]
            start = ann.sample[i]
            end = ann.sample[i + 1] if i + 1 < len(ann.sample) else len(signal)

            for s in range(start, end - window_size, window_size):
                segment = signal[s:s+window_size]
                segments.append((segment, label))

    return segments
```

Extract feature from each record

```
all_segments = []
for rec in record_list:
    segmentslead = extract_windows(rec, window_size_sec=30)
    segmentsleadII = extract_windows(rec, window_size_sec=30, lead=1)
    segments = segmentslead + segmentsleadII
    all_segments.extend(segments)
```

Result:

[ERROR] Skipping record 00735: sampto must be greater than sampfrom

[ERROR] Skipping record 00735: sampto must be greater than sampfrom

[ERROR] Skipping record 03665: sampto must be greater than sampfrom

[ERROR] Skipping record 03665: sampto must be greater than sampfrom

Separate the list into each segment long with its label.

```
X = []
y = []

for segment, label in all_segments:
    if label in ['AFIB', 'N']:
        X.append(segment)
        y.append(1 if label == 'AFIB' else 0)

X = np.array(X)
y = np.array(y)

print("N Label: ", np.sum(y == 0))
print("AFIB Label: ", np.sum(y == 1))
```

Result:

N Label: 15818

AFIB Label: 5924

- The Normal Segment consist of: 15818 Segments.
- The AFIB Segment consist of: 5924 segments.

Checking the shape and sampling size of signal

```
fs =250
X.shape, y.shape, fs
```

Result:

((21742, 7500), (21742,), 250)

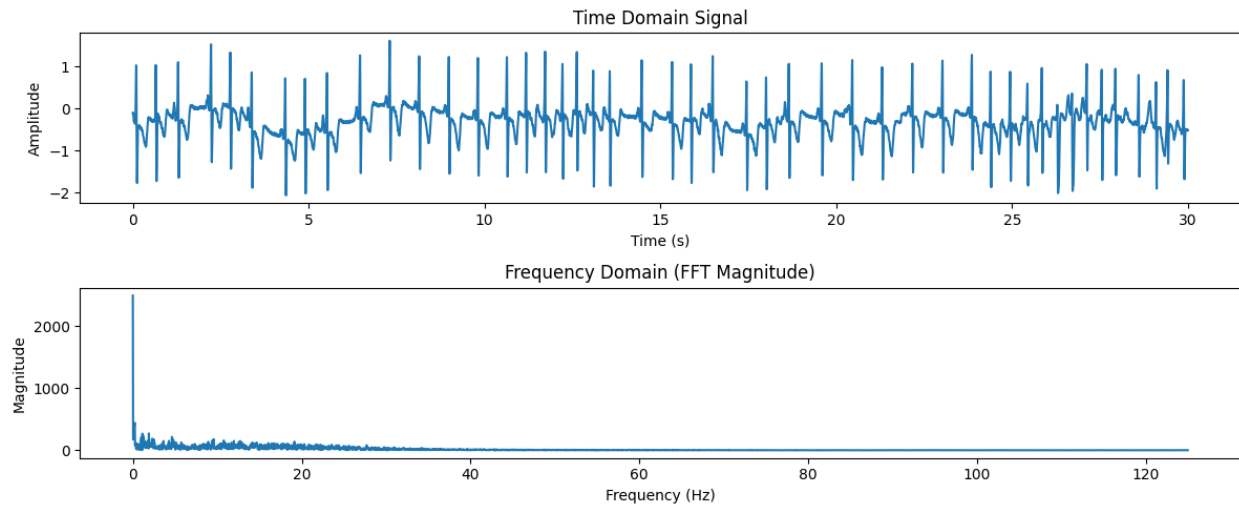
Function to View the function in both T-domain and f-domain

```
def plot_signal(X, fs=250):  
    """  
    Plots time-domain and frequency-domain views of an ECG signal segment.  
  
    Parameters:  
        X (np.ndarray): 1D ECG segment  
        fs (int): Sampling frequency in Hz  
    """  
    signal = X  
    n = len(signal)  
    freqs = np.fft.fftfreq(n, d=1/fs)  
    fft_magnitude = np.abs(np.fft.fft(signal))  
  
    # Plot time domain  
    plt.figure(figsize=(12, 5))  
  
    plt.subplot(2, 1, 1)  
    plt.plot(np.arange(n) / fs, signal)  
    plt.title("Time Domain Signal")  
    plt.xlabel("Time (s)")  
    plt.ylabel("Amplitude")  
  
    # Plot frequency domain  
    plt.subplot(2, 1, 2)  
    plt.plot(freqs[:n // 2], fft_magnitude[:n // 2])  
    plt.title("Frequency Domain (FFT Magnitude)")  
    plt.xlabel("Frequency (Hz)")  
    plt.ylabel("Magnitude")  
  
    plt.tight_layout()  
    plt.show()
```


Check out the first segment in both t-domain and f-domain

```
plot_signal(X[0], fs=fs)
```

Result:



2.2 Data pre-processing

All ECG signals were resampled to *250 Hz* for consistency. The resampled signals were segmented into *30-second durations*.

```
def bandpass_filter(x, lowcut=0.5, highcut=40, fs=250, order=4):
    nyq = fs/2
    # butter filter
    b, a = sp.butter(order, [lowcut/nyq, highcut/nyq], btype='band')
    return sp.filtfilt(b, a, x)

X_filt = bandpass_filter(X, fs=fs)
X_filt

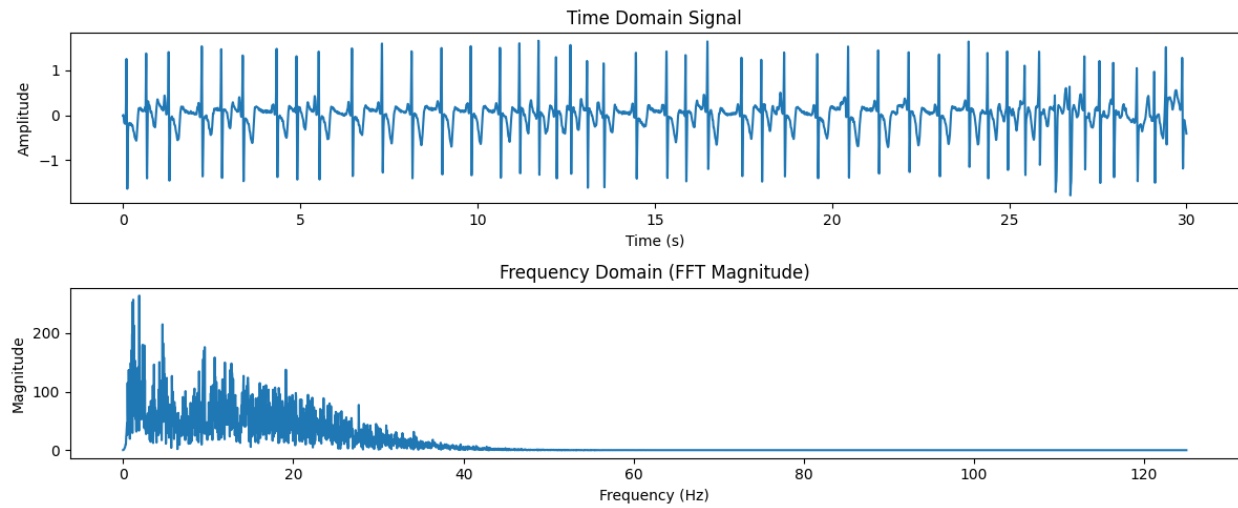
X_norm = processing.normalize_bound(X_filt, lb=-1, ub=1)
X_norm
```

- Bandpass filter: used to filter out the from ECG Signal band between 0.5Hz and 40Hz which is the usually the noise for ECG Signal.

Let's View both filtered and normalized of first segment

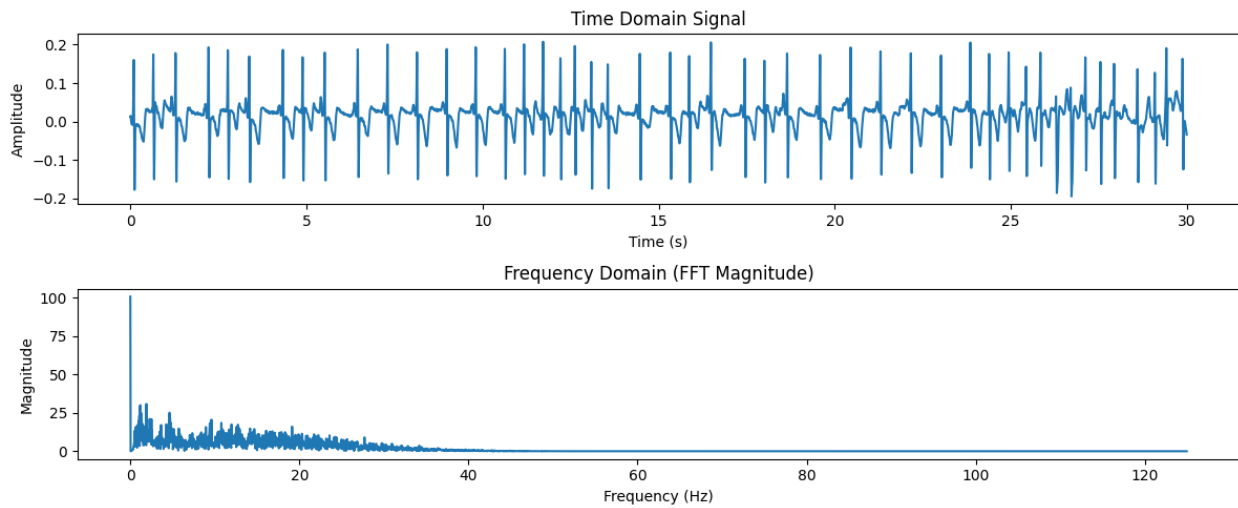
```
plot_signal(X_filt[0], fs=fs)
```

Result:



```
plot_signal(X_norm[0], fs=fs)
```

Result:



Let's what are second and third segment look like

```
time_axis = np.arange(len(X_norm[0])) / fs
afib_indices = np.where(y == 1)[0][:3]
normal_indices = np.where(y == 0)[0][:3]

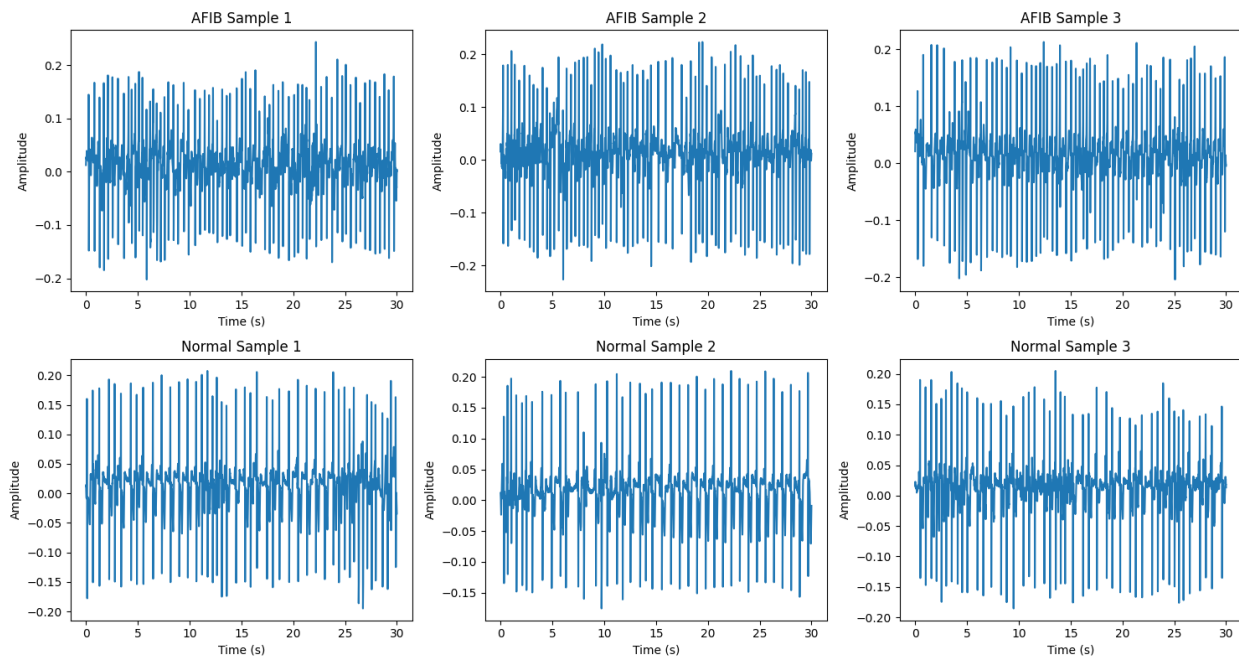
plt.figure(figsize=(15, 8))

for i, idx in enumerate(afib_indices):
    plt.subplot(2, 3, i + 1)
    plt.plot(time_axis, X_norm[idx])
    plt.title(f'AFIB Sample {i + 1}')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')

for i, idx in enumerate(normal_indices):
    plt.subplot(2, 3, i + 4)
    plt.plot(time_axis, X_norm[idx])
    plt.title(f'Normal Sample {i + 1}')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')

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

Result:



2.2.1 Extract R-Peak

```
from wfdb import processing

fs = 250 # Sampling frequency
r_peaks_all = [] # To store R-peak indices for all segments

for segment in X_norm:
    # segment is a 1D ECG array
    xqrs = processing.XQRS(sig=segment, fs=fs)
    xqrs.detect()
    r_peaks_all.append(xqrs.qrs_inds) # Save R-peak indices
```

Result:

```
Learning initial signal parameters...
Found 8 beats during learning. Initializing using learned parameters
Running QRS detection...
QRS detection complete.
Learning initial signal parameters...
Found 8 beats during learning. Initializing using learned parameters
Running QRS detection...
....
```

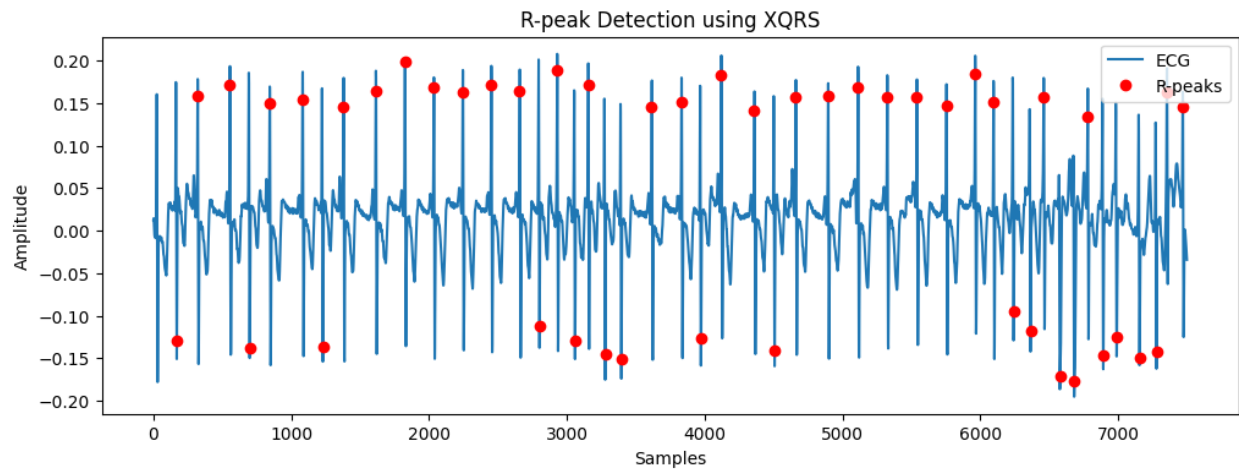
Let's see what R-peak of ECG Signal Look like

```
import matplotlib.pyplot as plt

# Select the first segment and its corresponding R-peaks
segment = X_norm[0]
r_peaks = r_peaks_all[0]

plt.figure(figsize=(12, 4))
plt.plot(segment, label='ECG')
plt.plot(r_peaks, segment[r_peaks], 'ro', label='R-peaks')
plt.legend()
plt.title("R-peak Detection using XQRS")
plt.xlabel("Samples")
plt.ylabel("Amplitude")
plt.show()
```

Result:



2.2.2 RR Interval Calculation

Equation (1) were used to determine the RR Interval (RRI) from each R-wave.

$$RR(i) = R(i + 1) - R(i) \quad (1)$$

Where $R(i)$ represents the position of the i^{th} R-wave in an ECG signal, and $RR(i)$ denoted the corresponding RR Interval (RRI), which is the time interval between consecutive R-waves.

Let's calculate all R-R Interval

```
rr_intervals_all = []

for r_peaks in r_peaks_all:
    # RR(i) = R(i+1) - R(i)
    rr_intervals = np.diff(r_peaks) / fs # Convert sample diff to seconds
    rr_intervals_all.append(rr_intervals)
```

Let's what R-R Interval looks like:

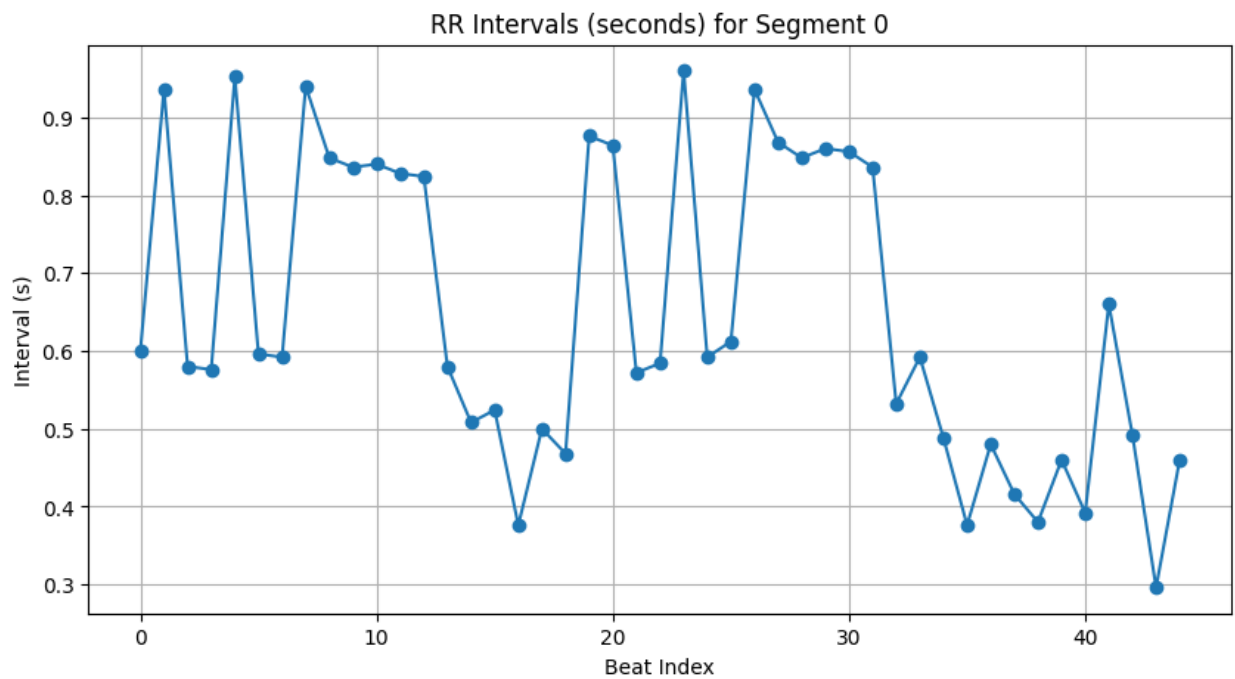
```
import matplotlib.pyplot as plt

# Plot RR intervals for the first segment (or any index you prefer)
rr_intervals = rr_intervals_all[0]

plt.figure(figsize=(10, 5))
```

```
plt.plot(rr_intervals, marker='o')
plt.title("RR Intervals (seconds) for Segment 0")
plt.xlabel("Beat Index")
plt.ylabel("Interval (s)")
plt.grid(True)
plt.show()
```

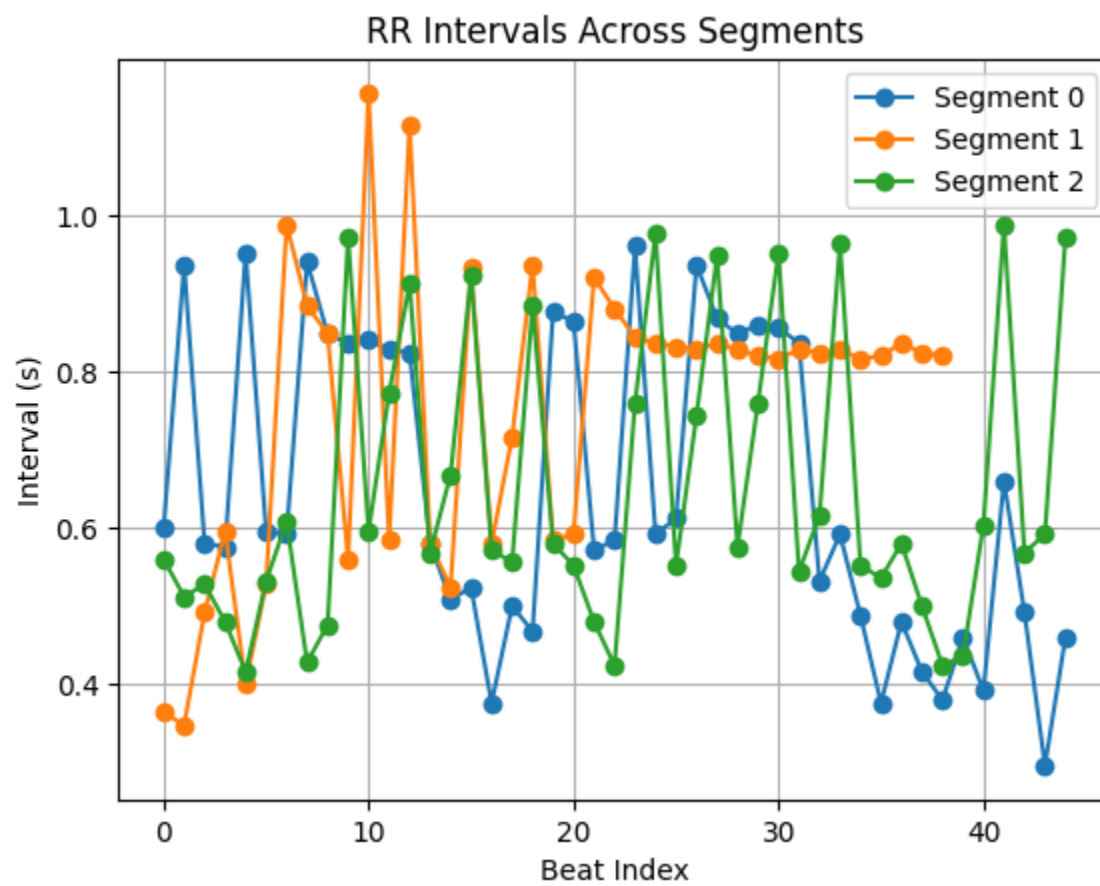
Result:



How about other Segment?

```
for i in range(3): # First 3 segments
    plt.plot(rr_intervals_all[i], marker='o', label=f'Segment {i}')
plt.title("RR Intervals Across Segments")
plt.xlabel("Beat Index")
plt.ylabel("Interval (s)")
plt.legend()
plt.grid(True)
plt.show()
```

Result:



Let's Select one universal length of R-R Interval for all Segment as it is the last step for data preprocess for our machine learning

```
rr_lengths = [len(rri) for rri in rr_intervals_all]
max_len = max(rr_lengths)
avg_len = int(np.mean(rr_lengths))
median_len = int(np.median(rr_lengths))

print(f"Max RR interval length: {max_len}")
print(f"Average RR interval length: {avg_len}")
print(f"median RR interval length: {median_len}")
```

Result:

Max RR interval length: 96

Average RR interval length: 39

median RR interval length: 36

As you can see, each segment of the ECG Signal has difference R-R Interval, so we set the average value as the base line for R-R interval in each Segment to valid in the training process.

```
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences

# rr_intervals_all is a list of np.arrays
X_rri = pad_sequences(rr_intervals_all, maxlen=avg_len, padding='post',
truncating='post', dtype='float32', value=-1)
```

Let's make R-R Interval usable for model training

```
X_rri.shape
```

The Shape of R-R Interval is: (21742, 39).

- 21742 Segment each with 39 Interval on each Signal


```
X_rri = np.expand_dims(X_rri, axis=-1)
X_rri.shape
```

Result:

(21742, 39, 1)

This created one additional channel for model output as 1D array onto other Layer.

Data Splitting for model training

```
# Split data
from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(
    X_rri, y, test_size=0.2, random_state=42, stratify=y)

X_train.shape, X_val.shape, y_train.shape, y_val.shape
```

Result:

((17393, 39, 1), (4349, 39, 1), (17393,), (4349,))

- Training data consist of 17393 segments with each label associated with.
- Validating data consist 4349 segments with each label associated with.

2.3 Convolutional Neural Network (CNN) Model

Convolutional Neural Networks (CNNs) are deep learning models capable of automatic feature extraction, particularly effective for time-series data like ECG signals. In this study, a 1D CNN is used to detect AFIB based on RR intervals (RRI) derived from ECG recordings. As the number of RRIs varies per sample, input sequences are zero-padded to match the longest sequence for uniform input dimensions.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense,
Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

model = Sequential([
```

```

    Conv1D(filters=32, kernel_size=5, activation='relu', padding='same',
input_shape=(X_rri.shape[1], 1)),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=64, kernel_size=3, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=64, kernel_size=5, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=128, kernel_size=3, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer=Adam(1e-4), loss='binary_crossentropy',
metrics=['accuracy'])
print(model.summary())

```

Result:

Layer (type)	Output Shape	Param #
conv1d_16 (Conv1D)	(None, 39, 32)	192
batch_normalization_16 (BatchNormalization)	(None, 39, 32)	128
max_pooling1d_16 (MaxPooling1D)	(None, 19, 32)	0
conv1d_17 (Conv1D)	(None, 19, 64)	6,208
batch_normalization_17 (BatchNormalization)	(None, 19, 64)	256
max_pooling1d_17 (MaxPooling1D)	(None, 9, 64)	0
conv1d_18 (Conv1D)	(None, 9, 64)	20,544
batch_normalization_18 (BatchNormalization)	(None, 9, 64)	256
max_pooling1d_18 (MaxPooling1D)	(None, 4, 64)	0
conv1d_19 (Conv1D)	(None, 4, 128)	24,704
batch_normalization_19 (BatchNormalization)	(None, 4, 128)	512
max_pooling1d_19 (MaxPooling1D)	(None, 2, 128)	0
flatten_4 (Flatten)	(None, 256)	0
dense_8 (Dense)	(None, 64)	16,448
dropout_4 (Dropout)	(None, 64)	0
dense_9 (Dense)	(None, 1)	65

Fig 2: Architecture of CNN

This architecture processes RR interval data to classify rhythms as either NSR or AFIB.

3.1.1 Input Layer

Receives a 1D RRI array. To standardize length, all sequences are **zero-padded** based on the longest sequence in the dataset.

3.1.2 Convolutional Feature Extraction Blocks

Each *convolutional block* in the network is composed of four key operations:

- *Zero Padding*: Adds zeros to the edges of the input to maintain the sequence length after convolution. This helps retain information at the boundaries of the input signal.
- *1D Convolution Layer*: Applies multiple filters (kernels) that slide across the temporal RR interval data to detect local patterns, such as abrupt changes in interval timing, which are common in AFIB. Each filter produces a feature map that captures a specific type of local variation or rhythm irregularity.
- *ReLU (Rectified Linear Unit) Activation*: Introduces non-linearity by outputting zero for negative values and the original value for positive ones. This allows the network to learn complex, non-linear relationships in the data and improves training performance by preventing vanishing gradients.
- *Batch Normalization*: Standardizes the outputs of each convolutional layer by adjusting the mean and variance. This stabilizes and accelerates training by reducing internal covariate shift, helping the model converge faster and generalize better.
- *Max Pooling*: Reduces the temporal resolution by downsampling the feature maps. This not only lowers computational cost but also helps the network focus on the most dominant features, increasing robustness to small variations in the input.

These blocks are stacked to capture both short- and long-term irregularities in heart rhythms.

3.1.3 Deeper Convolutional Layers

After initial feature extraction, additional *convolutional layers* are added (often without pooling) to further refine the learned features. These deeper layers capture *more global patterns* in the RRs, such as sustained irregularity over time—one of the hallmarks of AFIB.

3.1.4 Flattening Layer

Once feature extraction is complete, the resulting multidimensional feature maps are *flattened* into a single 1D vector. This transformation allows the model to transition from spatial feature learning to classification.

3.1.3 Fully Connected (Dense) Layers

The flattened vector is passed through one or more *dense layers*. Each neuron in a dense layer is connected to all neurons in the previous layer, allowing the network to combine features in complex ways. The dense layers also include *ReLU Activation* (As describes above) used to introduce non-linearity and enhance learning capacity.

3.1.6 Output Layer with Softmax

The final layer is a *dense layer with two neurons*, corresponding to the two classification classes: *NSR* and *AFIB*. A *Softmax activation function* is applied to convert the raw output scores (logits) into *probability distributions* across the two classes. This enables the model to output interpretable predictions, such as a 90% probability of AFIB.

2.4 Model Training

```
from tensorflow.keras.callbacks import EarlyStopping

Early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    callbacks=[Early_stopping]
)
```

Result:

Epoch 1/20

544/544 ————— 7s 7ms/step - accuracy: 0.8900 - loss: 0.2746 -
val_accuracy: 0.9616 - val_loss: 0.1137

Epoch 2/20

544/544 ————— 6s 10ms/step - accuracy: 0.9684 - loss: 0.0936 -
val_accuracy: 0.9742 - val_loss: 0.0757

Epoch 3/20

544/544 ————— 6s 10ms/step - accuracy: 0.9763 - loss: 0.0702 -
val_accuracy: 0.9770 - val_loss: 0.0665

Epoch 4/20

544/544 ————— 6s 11ms/step - accuracy: 0.9815 - loss: 0.0526 -
val_accuracy: 0.9795 - val_loss: 0.0654

Epoch 5/20

544/544 ————— 9s 16ms/step - accuracy: 0.9853 - loss: 0.0481 -
val_accuracy: 0.9837 - val_loss: 0.0579

Epoch 6/20

544/544 ————— 6s 10ms/step - accuracy: 0.9863 - loss: 0.0393 -
val_accuracy: 0.9839 - val_loss: 0.0528

Epoch 7/20

544/544 ————— 6s 10ms/step - accuracy: 0.9884 - loss: 0.0341 -
val_accuracy: 0.9830 - val_loss: 0.0640

Epoch 8/20

544/544 ————— 6s 11ms/step - accuracy: 0.9907 - loss: 0.0294 -
val_accuracy: 0.9857 - val_loss: 0.0573

Epoch 9/20

544/544 ————— 6s 10ms/step - accuracy: 0.9925 - loss: 0.0250 -
val_accuracy: 0.9867 - val_loss: 0.0599

2.4 Model Evaluation

```
print(X_train.shape, y_train.shape)  
print(X_val.shape, y_val.shape)
```

Result:

(17393, 39, 1) (17393,)

(4349, 39, 1) (4349,)

Evaluate on model

```
loss, accuracy = model.evaluate(X_val, y_val)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")
```

Result:

136/136 ————— 0s 3ms/step - accuracy: 0.9826 - loss: 0.0567
Test Loss: 0.0528
Test Accuracy: 0.9839

The result shows an accuracy of 98.38% and loss of 5.28%.

Learning Curve of Model

```
import matplotlib.pyplot as plt

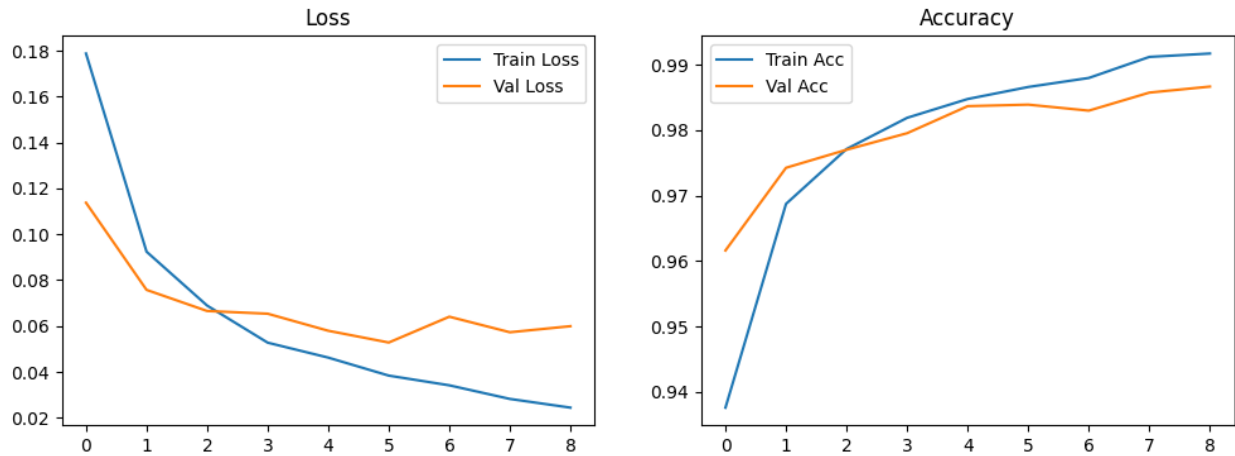
plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.legend()
plt.title('Loss')

plt.subplot(1,2,2)
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.legend()
plt.title('Accuracy')

plt.show()
```

Result:



The overall trend from learning curve seems fine across each epoch.

Checking on Classification Report and Confusion Matrix

```
from sklearn.metrics import classification_report, confusion_matrix

y_pred = (model.predict(X_val) > 0.9).astype("int32")
print(classification_report(y_val, y_pred))
```

Result of Classification report:

136/136 ————— 1s 4ms/step (from model predicting)

	precision	recall	f1-score	support
0	0.98	0.99	0.99	3164
1	0.98	0.94	0.96	1185
accuracy			0.98	4349
macro avg	0.98	0.97	0.97	4349
weighted avg	0.98	0.98	0.98	4349

Confusion Matric

```
print(confusion_matrix(y_val, y_pred))
```

Result of Confusion Matric;

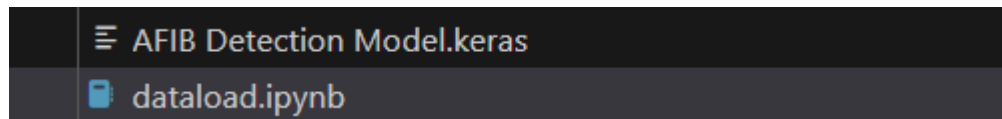
	N	AFIB
Actual N	3145	19
Actual AFIB	69	1116

- **True Negatives (TN)** = 3145 Normal predicted as normal
- **False Positives (FP)** = 19 Normal incorrectly predicted as AFIB
- **False Negatives (FN)** = 69 AFIB incorrectly predicted as normal
- **True Positives (TP)** = 1116 AFIB correctly predicted as AFIB

Let's Save our Model and use for AF Detection Application

```
model.save("AFIB Detection Model.keras")
```

Result:



We get our model in the directory. These Code can be access on *Appendix A: Python – ECG Preprocessing, RRI Extraction and CNN Model Architecture (Code)*.

III. Electronic Integration

For this study, we have use the ESP32 to control LED which receive the result from web server (flask).

3.1 Coding

Header Section

```
#include <WiFi.h>
#include <WebServer.h>
```

Essential Initialization

```
#define WIFI_STATUS 2 // Built-in LED
#define UNDETECTED_LED 0 // Green LED
#define DETECTED_LED 4 // RED or Yellow LED
#define BUZZER 5 // Buzzer
```

This Define some key Components of our detecting system.

Let's see what it looks like:

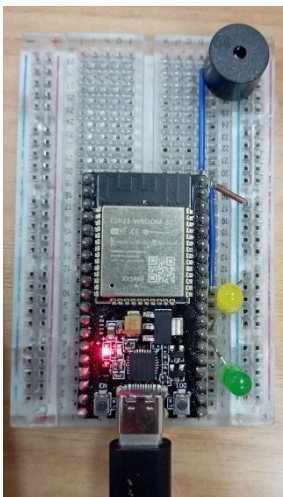


Figure 3: Actual Detecting Hardware

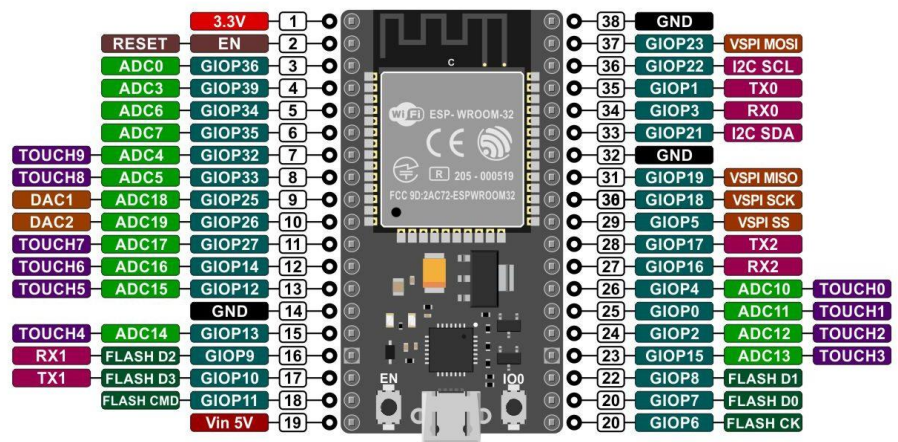


Figure 4: ESP32 Pin Layout

The fig 3 has shown our connection of responding hardware and match the ESP32 Pin Layout on the right fig. 4.

This code matches another one on the loop function to enable the time duration of buzzer

```
unsigned long buzzerStart = 0;
bool buzzerOn = false;
```

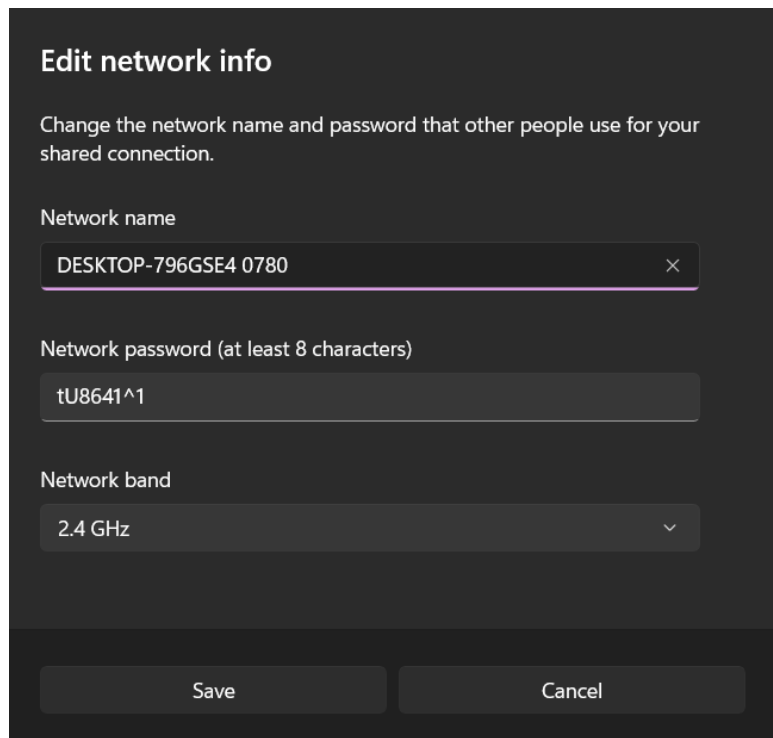
This condition ensures that after button is on when detected the AFIB, the buzzer turned on for 1s

```
if (buzzerOn && (millis() - buzzerStart > 1000)) {
    digitalWrite(BUZZER, LOW);
    buzzerOn = false;
}
}
```

Set up the hotspot for data transmission

```
const char* ssid = "DESKTOP-796GSE4 0780";
const char* password = "tU8641^1";
```

This was set up to match the hotspot from network.



Edit network info

Change the network name and password that other people use for your shared connection.

Network name

DESKTOP-796GSE4 0780

Network password (at least 8 characters)

tU8641^1

Network band

2.4 GHz

Save Cancel

Webserver Initialization

```
WebServer server(80);
```

HandleDetected Function for handling when the model predicted as present of AF.

```
void handleDetected() {  
    digitalWrite(DETECTED_LED, HIGH);  
    digitalWrite(UNDETECTED_LED, LOW);  
    digitalWrite(BUZZER, HIGH);  
    buzzerStart = millis();  
    buzzerOn = true;  
    server.send(200, "text/plain", "DETECTED - Devices ON");  
}
```

HandleUndetected Function for handling when the model predicted as absence of AF.

```
// UNDETECTED: turn OFF all devices  
void handleUndetected() {  
    digitalWrite(UNDETECTED_LED, HIGH);  
    digitalWrite(DETECTED_LED, LOW);  
    server.send(200, "text/plain", "UNDETECTED - Devices OFF");  
}
```

Set Up function for Initialization

```
void setup() {  
    Serial.begin(115200);  
  
    pinMode(WIFI_STATUS, OUTPUT);  
    pinMode(UNDETECTED_LED, OUTPUT);  
    pinMode(DETECTED_LED, OUTPUT);  
    pinMode(BUZZER, OUTPUT);  
  
    digitalWrite(WIFI_STATUS, LOW);  
}
```

```
digitalWrite(DETECTED_LED, LOW);
digitalWrite(UNDETECTED_LED, LOW);
digitalWrite(BUZZER, LOW);

WiFi.begin(ssid, password);
Serial.print("Connecting to WiFi");
while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
}
```

After a successful connection, the Build-In LED on ESP32 board turned on (BLUE LED: PIN 2)

```
digitalWrite(WIFI_STATUS, HIGH);
delay(10);
```

Result:

And this loop was used to indicate that The WiFi Connect successfully connected by buzz 3times each for 100ms or 0.1s.;

```
// Buzzer short beep after Wi-Fi connected
for (int i = 0; i < 3; i++) {
    digitalWrite(BUZZER, HIGH);
    delay(100);
    digitalWrite(BUZZER, LOW);
    delay(100);
}
```

Ensure Wifi Connection

```
Serial.println("\nConnected to WiFi");
Serial.print("IP address: ");
Serial.println(WiFi.localIP());
```

Enable the ESP32 to listen from Web Server

```
server.on("/DETECT", handleDetected);
server.on("/detect", handleDetected);
server.on("/UNDETECTED", handleUndetected);
server.on("/undetected", handleUndetected);

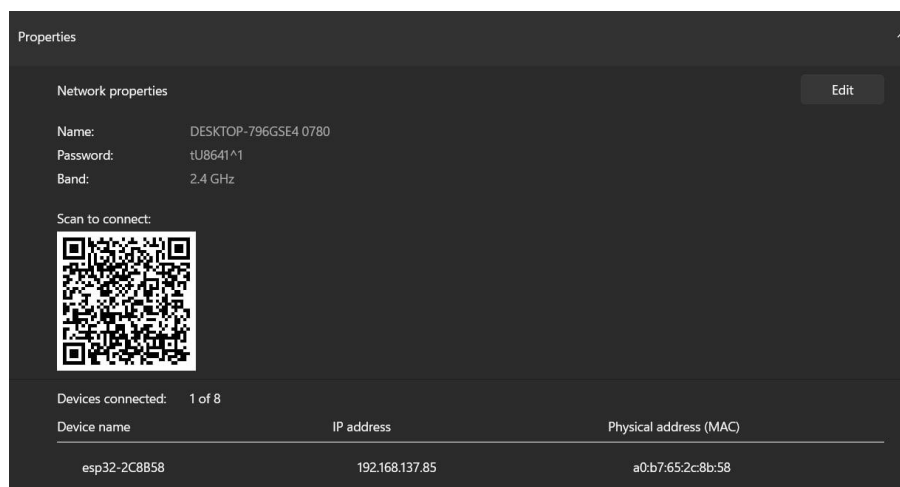
server.begin();
Serial.println("HTTP server started");
}
```

Result:

The Result shown inside Serial Monitoring to view the IP address of ESP32.

```
Connecting to WiFi...
Connected to WiFi
IP address: 192.168.137.85
HTTP server started
```

Or we can simply view it on the Network Properties of hotspot.



This Loop function enable the connection between ESP32 (Client) and Web Server (Server) to communication with each other.

```
void loop() {  
  server.handleClient();  
}
```

This Code was implemented on Arduino IDE as shown on *Appendix B: ESP32 Microcontroller Code (Arduino IDE)*.

IV. Application Deployment

4.1 Generated ECG Signal for application Testing

We are generating synthetic ECG signals for testing purposes, specifically targeting *Atrial Fibrillation (AFIB)* and *Normal rhythms*. Our Convolutional Neural Network (CNN) model is trained using *R-R intervals* as the primary input feature. This preprocessing step is essential and directly influences the model's performance.

While we initially experimented with image-based data inputs, the repeated conversion between signal and image formats resulted in significant loss of signal. Thus, we now rely on signal-based preprocessing.

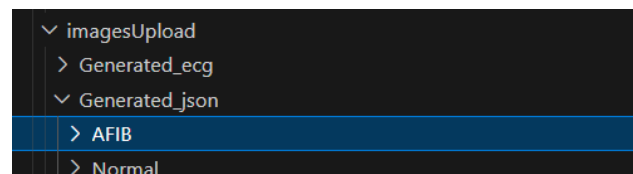
For testing, we selected a real ECG record from the **WFDB database**:

```
record_name = "04043"
```

This record is processed in a way that aligns with the model's requirements, particularly focusing on extracting **R-R intervals** from the ECG signals. The output is formatted as structured JSON files, following the schema outlined in *Appendix D: Sample ECG JSON Files*.

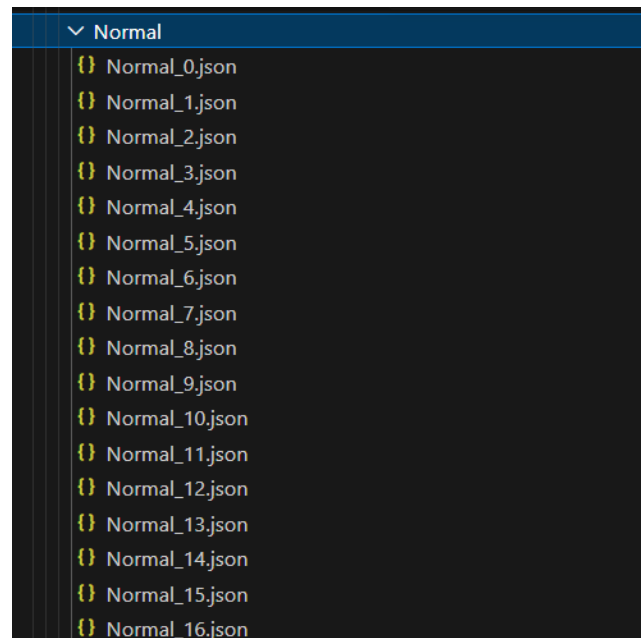
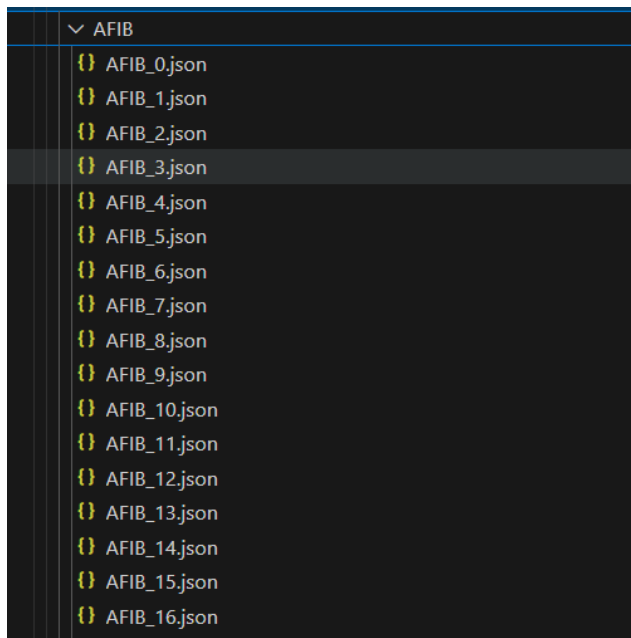
We generate and save two distinct JSON files:

- One representing **AFIB** rhythms



- One representing **Normal** rhythms

Result of each file:



4.2 Web Application using Flask

Now for the main web application

```
from flask import Flask, request, render_template, jsonify
from tensorflow.keras.models import load_model
import numpy as np
import os
import requests
import neurokit2 as nk
```

Initialize the App as file (app.py)

```
app = Flask(__name__)
ESP32_IP = "http://192.168.137.174" # Replace with actual ESP32 IP
```


Model Loading

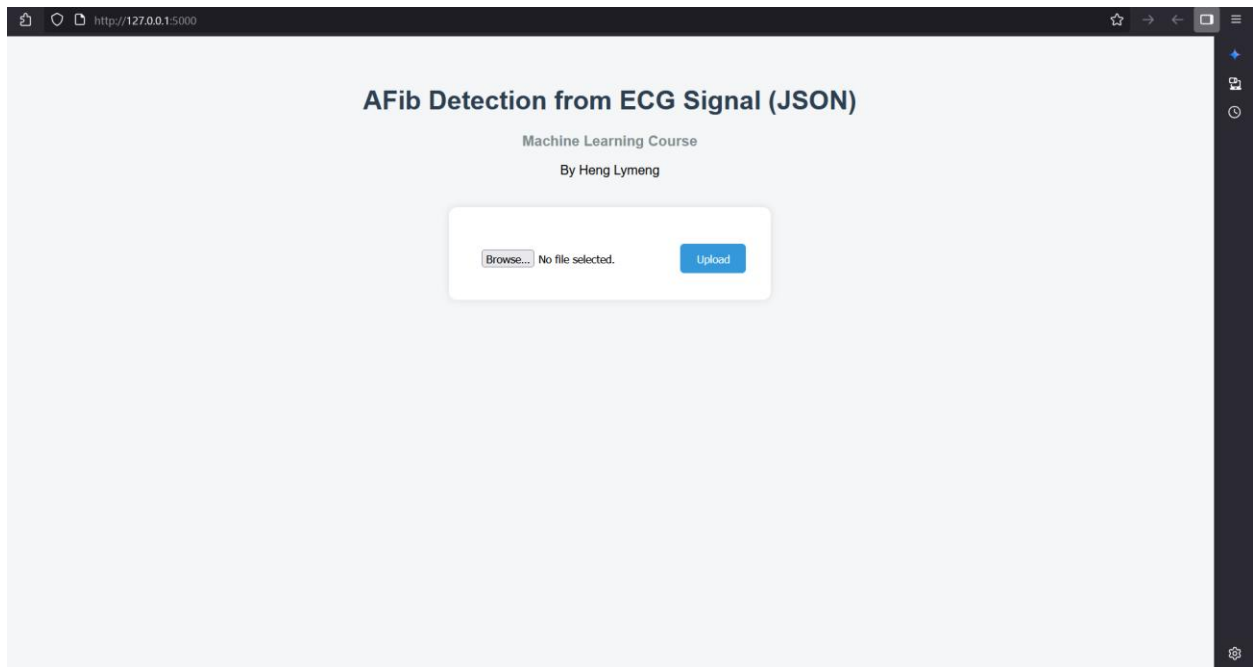
```
# Load model
model_path = "D:/RUPP/RUPP/RUPP Y3S1/Machine
Learning/FinalProject/AF_Detection/dataset/AFIB Detection Model.keras"
model = load_model(model_path)
```

This enable application to render HTML File as template which is our main Interface for Application.

```
@app.route('/')
def index():
    return render_template("index.html")
```

index.html allow user to interact and input JSON file as shown on *Appendix E: Web Application Structure Application – HTML, CSS and Java Script*.

Result:



Now for the main web application

```
@app.route('/predict', methods=['POST'])
```

- Post method allow user to post or send the data to Web server.

Predict Function to predict the result

```
def predict():
    data = request.get_json()

    if not data or 'signal' not in data:
        return jsonify({'error': 'No ECG signal provided'}), 400

    try:
        raw_ecg = np.array(data['signal'], dtype=np.float32)

        # Normalize raw ECG
        raw_ecg = (raw_ecg - np.min(raw_ecg)) / (np.max(raw_ecg) -
np.min(raw_ecg) + 1e-6)

        # RRI extraction using neurokit2
        signals, info = nk.ecg_process(raw_ecg, sampling_rate=250)
        rpeaks = info["ECG_R_Peaks"]

        # RR Intervals (in seconds)
        rri = np.diff(rpeaks) / 250.0

        if len(rri) < 39:
            return jsonify({'error': f'Not enough RR intervals: {len(rri)}'}),
400

        desired_len = model.input_shape[1] # 39
        if len(rri) > desired_len:
            rri = rri[:desired_len]
        else:
            rri = np.pad(rri, (0, desired_len - len(rri)), mode='constant')

        # Correct shape: (1, 39, 1)
        model_input = rri.reshape(1, desired_len, 1)

        # Predict
        score = model.predict(model_input)[0][0]
```

```

        result = "Atrial Fibrillation Detected" if score > 0.5 else "Normal
Rhythm"

        # ESP32 Command
        esp_status = "DETECT" if score > 0.9 else "UNDETECTED"
        send_to_esp32(esp_status)

        return jsonify({
            'prediction': result,
            'score': f"{score * 100:.2f}%", # formats score as percentage with 2
decimal places
            'esp32_status': esp_status
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 500

```

Define the function to send the result from web server to ESP32

```

def send_to_esp32(command):
    try:
        url = f"{ESP32_IP}/{command}"
        requests.get(url, timeout=2)
        print(f"[ESP32] Sent: {command}")
    except requests.exceptions.RequestException as e:
        print(f"[ERROR] ESP32 not reachable: {e}")

```

Update the Status of Application

```

@app.route('/status')
def status():
    return jsonify({"status": "running"})

```

Running Debug Command

```

if __name__ == '__main__':
    app.run(debug=True)

```

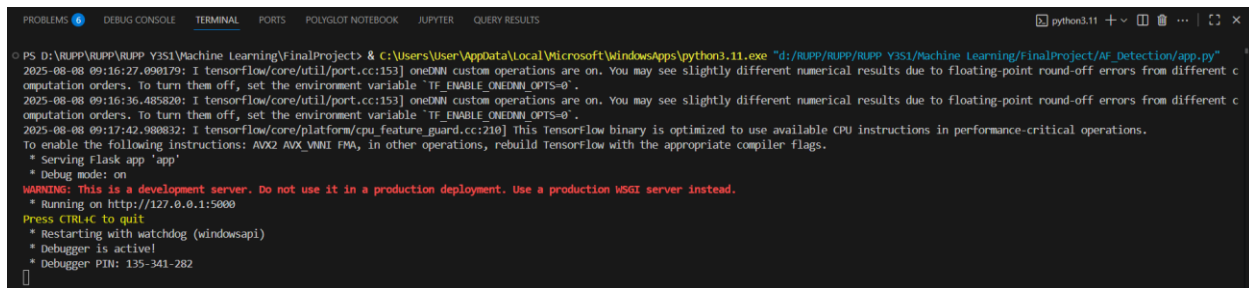
4.3 Results and Discussion

We can start the application by first run the app.py which is our Web Application Code from *Appendix C: Web Application – Flask and Frontend* and simply change the IP address of ESP32.

Replace with ESP32 IP Address

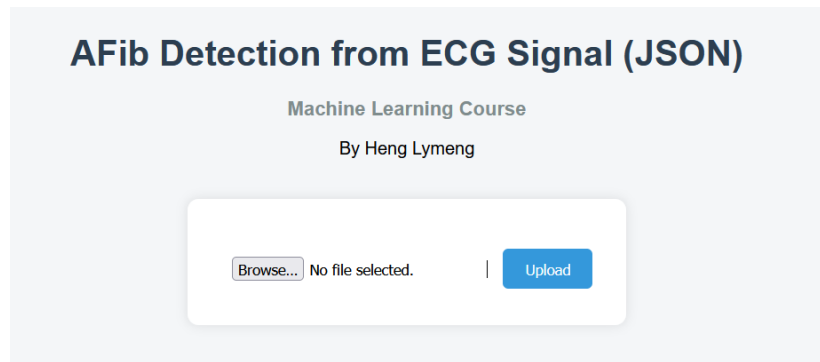
```
app = Flask(__name__)
ESP32_IP = "http://192.168.137.174" # Replace with actual ESP32 IP
```

The result after ran the code look something like this

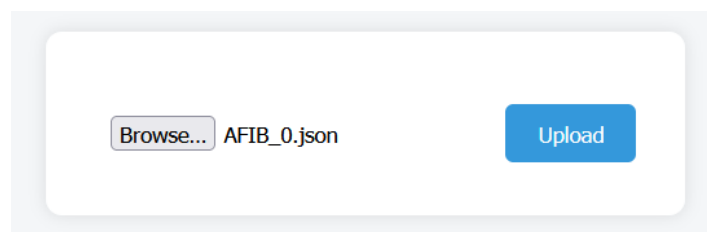
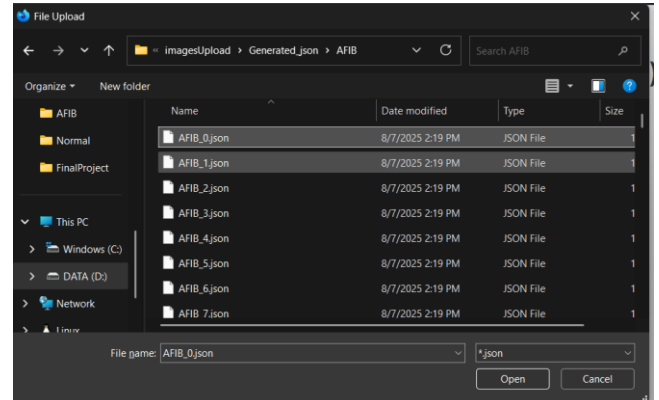
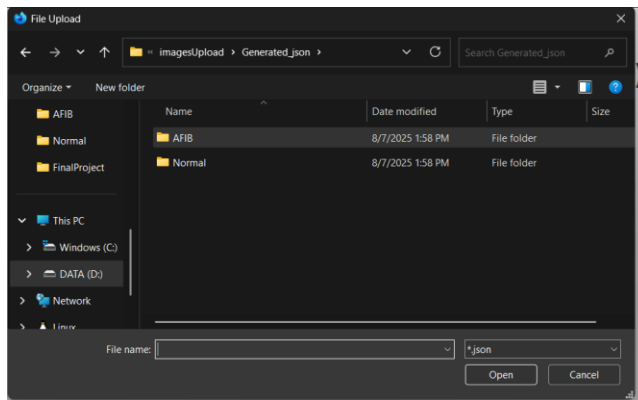


```
PS D:\RUPP\RUPP\Y3S1\Machine Learning\FinalProject> & C:\Users\User\AppData\Local\Microsoft\WindowsApps\python3.11.exe "d:/RUPP/RUPP/Y3S1/Machine Learning/FinalProject/AF_Detection/app.py"
2025-08-08 09:16:27.090179: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different c
computation orders. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
2025-08-08 09:16:36.485820: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different c
computation orders. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
2025-08-08 09:17:42.980832: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with watchdog (windowsapi)
* Debugger is active!
* Debugger PIN: 135-341-282
```

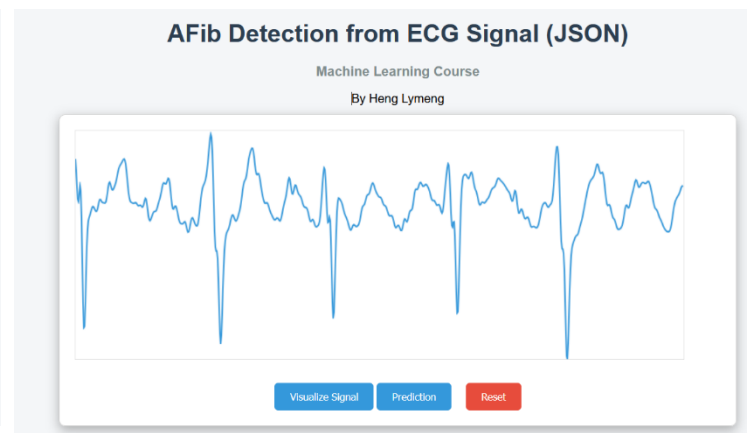
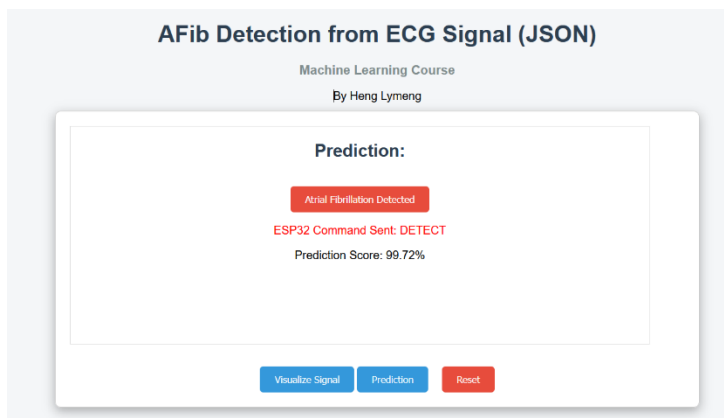
4.3.1 Testing on AFIB



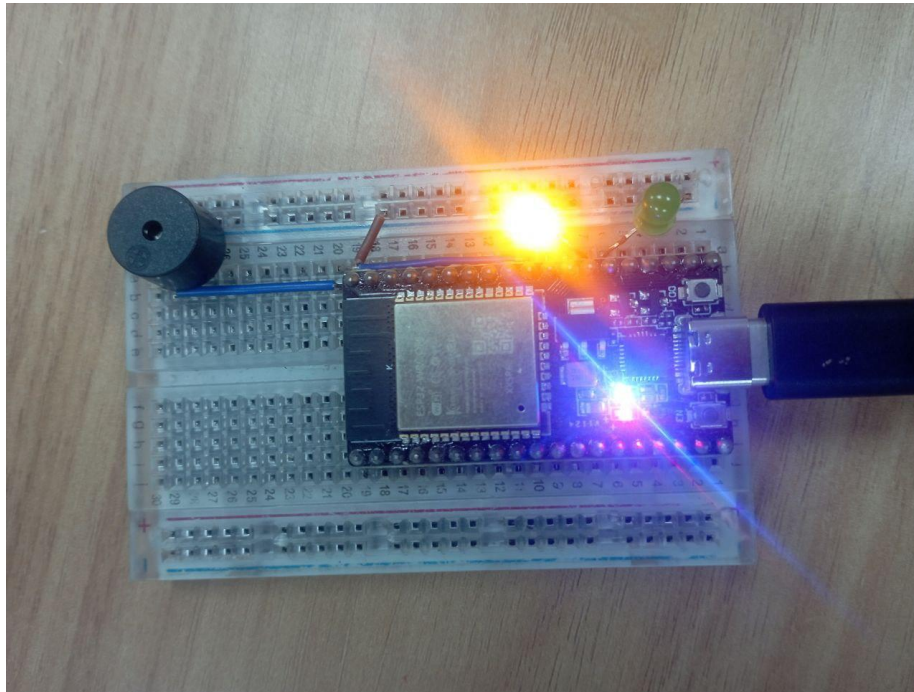
Browsing for File



Hit Upload, then we get our result along with Visualization of ECG Signal

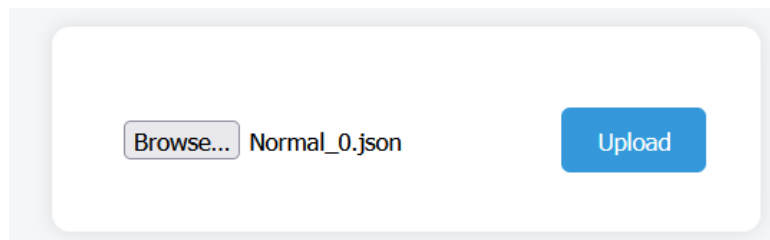


The Buzzer Turn on for 1s and Yellow LED turned on.

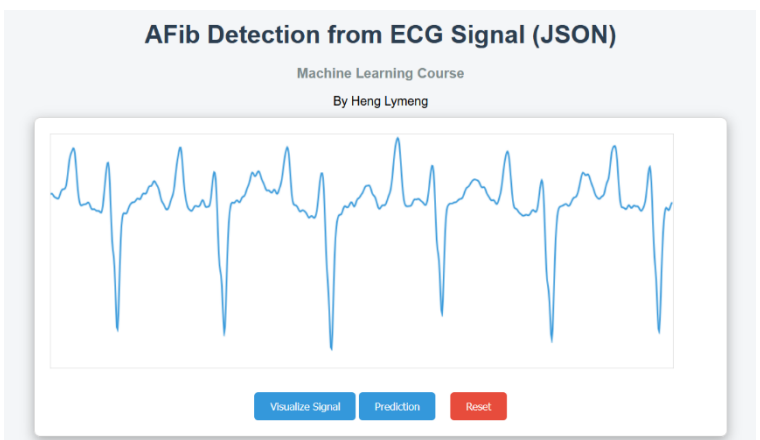
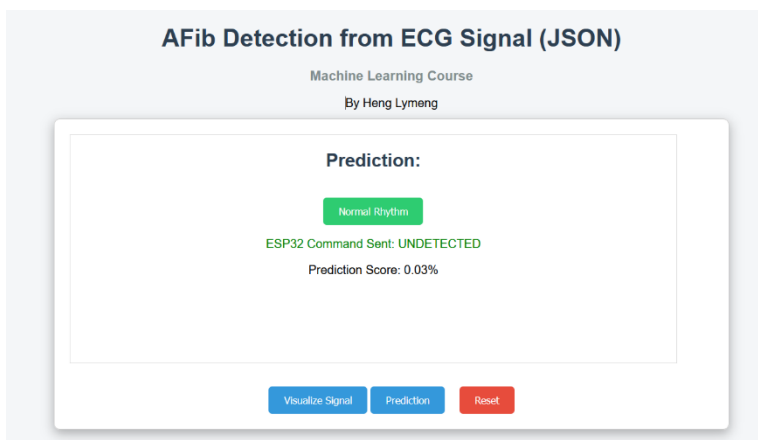


4.3.2 Testing on Normal

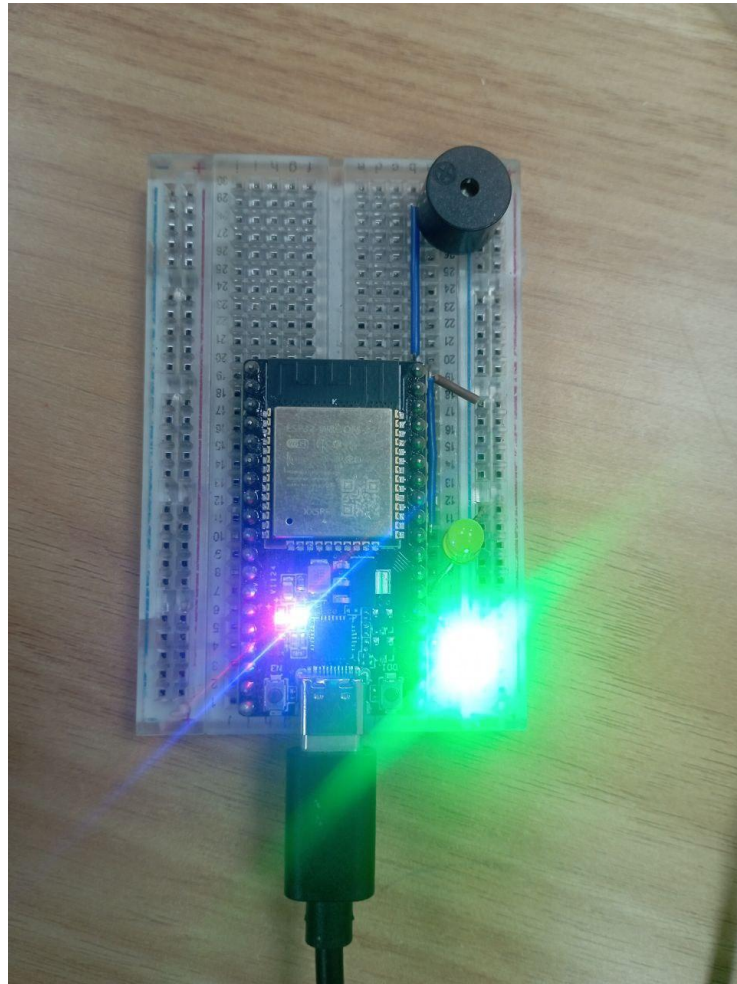
Let's Try on the normal one



The Result from prediction



As a Result from Prediction, Green LED turned on.



4.3.3 Debug Console

```
* Restarting with watchdog (windowsapi)
* Debugger is active!
* Debugger PIN: 135-341-282
1/1 1s 912ms/step
* Detected change in 'C:\\Program Files\\WindowsApps\\PythonSoftwareFoundation.Python.3.11.2544.0_x64_qbz5n2kfra8p0\\Lib\\netrc.py', reloading
[ESP32] Sent: DETECT
127.0.0.1 - - [08/Aug/2025 10:05:22] "POST /predict HTTP/1.1" 200 -
* Restarting with watchdog (windowsapi)
* Debugger is active!
* Debugger PIN: 135-341-282
1/1 0s 164ms/step
[ESP32] Sent: DETECT
127.0.0.1 - - [08/Aug/2025 10:10:46] "POST /predict HTTP/1.1" 200 -
1/1 0s 62ms/step
[ESP32] Sent: UNDETECTED
127.0.0.1 - - [08/Aug/2025 10:11:50] "POST /predict HTTP/1.1" 200 -
□
```

Every time, we do some wrong or there is error occur, we can view the Debug Console on the terminal.

V. Conclusion

This project demonstrates that *deep learning*, specifically *Convolutional Neural Networks (CNNs)*, can be effectively applied to detect *Atrial Fibrillation (AFIB)* using RR intervals derived from ECG signals.

5.1 Evaluation

The implemented CNN model exhibited excellent performance in detecting AFIB using RR intervals. With a final *validation accuracy of 98.39%* and *a loss of 5.28%*, it outperformed conventional rule-based and statistical methods. Both *the confusion matrix* and *classification report* highlight the model's strength in achieving high precision and recall, resulting in *low false-positive and false-negative rates*—an essential requirement for any healthcare-related application.

Furthermore, the model was successfully integrated into a real-time system with an ESP32 microcontroller, providing physical feedback (LEDs and buzzer) for immediate AFIB alerting. This real-time deployment confirms the model's viability in embedded health monitoring systems.

5.2 Improvement

- *Real-World Noise Handling*: Signals from wearable or ambulatory devices often include noise and motion artifacts. Incorporating noise augmentation techniques during training or applying signal denoising filters can improve model resilience and performance in real-life scenarios.
- *Variable-Length Sequence Support*: The model currently uses fixed-length, zero-padded RR interval sequences, which may reduce the ability to capture full rhythm dynamics. Replacing the CNN with architectures like RNNs, GRUs, or Transformers can allow the system to process variable-length sequences and better reflect temporal dependencies in heartbeat patterns.
- *Enhanced Feedback System*: The current hardware provides binary visual and auditory feedback (AFIB detected or not). Incorporating a small display (e.g., OLED) or haptic feedback could improve accessibility—particularly for elderly or hearing-impaired users—and provide more detailed insights such as heart rate trends.

5.3 Future Work

To expand the scope and applicability of this system, the following areas are identified for future work:

- *Integration with Real-Time ECG Sensors*: Implement direct interfacing with live ECG modules (e.g., AD8232 or MAX30003) for continuous heart monitoring and real-time analysis.

- *Cloud Connectivity and Remote Monitoring*: Integrate with platforms like *Firebase*, *AWS IoT*, or *Google Cloud* to allow remote access and monitoring by healthcare providers or family members.

References

- [1] M. Keech, Y. Punekar, and A.-M. Choy, “Trends in atrial fibrillation hospitalisation in Scotland: An increasing cost burden,” *Br. J. Cardiol.*, vol. 19, pp. 173–177, 2012.
- [2] F. Rahman, G. F. Kwan, and E. J. Benjamin, “Global epidemiology of atrial fibrillation,” *Nat. Rev. Cardiol.*, vol. 11, no. 11, pp. 639–654, 2014, doi: 10.1038/nrcardio.2014.118.
- [3] V. Fuster *et al.*, “ACC/AHA/ESC 2006 guidelines for the management of patients with atrial fibrillation,” *Europace*, vol. 8, no. 9, pp. 651–745, 2006, doi: 10.1093/europace/eul097.
- [4] G. V. Naccarelli, H. Varker, J. Lin, and K. L. Schulman, “Increasing prevalence of atrial fibrillation and flutter in the United States,” *Am. J. Cardiol.*, vol. 104, no. 11, pp. 1534–1539, 2009, doi: 10.1016/j.amjcard.2009.07.022.
- [5] A. L. Goldberger *et al.*, “PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals,” *Circulation*, vol. 101, no. 23, pp. e215–e220, 2003, doi: 10.1161/01.CIR.101.23.e215.
- [6] Z. Yao, Z. Zhu, and Y. Chen, “Atrial fibrillation detection by multi-scale convolutional neural networks,” in *Proc. 20th Int. Conf. Information Fusion*, 2017, pp. 1–6.

Replication Basis

This project is a replication of the work by Ratha and Nam (2019), titled *Detection of atrial fibrillation on ECG recordings using rhythm features and neural networks*, presented at the *3rd International Conference on ICT for Smart Health (ICT4sHealth)*.

Appendix

All code listings, scripts, and supplementary data are included in the following appendices to support the implementation described in the main report.

Appendix A: Python – ECG Preprocessing, RRI Extraction and CNN Model Architecture (Code)

```
# %% [markdown]
# ## Header Section

# %%
import wfdb
from wfdb import processing
import numpy as np
import scipy.signal as sp
import matplotlib.pyplot as plt
import neurokit2 as nk

# %% [markdown]
# ## Data Loading and Data Pre-processing
```

```

# %% [markdown]
# Data Description
#
# Of these, 23 records include the two ECG signals (in the .dat files); records
00735 and 03665 are represented only by the rhythm (.atr) and unaudited beat
(.qrs annotation files.

# %%
record_list = [
    '00735', '03665', '04015', '04043', '04048', '04126', '04746',
    '08378', '08405', '08434', '08455'
]

# for record in record_list:
#     wfdb.dl_database('afdb', dl_dir='afdb', records=[record], overwrite=True)

# %%
def extract_windows(record_name, window_size_sec=10, lead=0):
    try:
        record = wfdb.rdrecord(f'afdb/{record_name}')
        ann = wfdb.rdann(f'afdb/{record_name}', 'atr')
    except Exception as e:
        print(f"[ERROR] Skipping record {record_name}: {e}")
        return []

    fs = record.fs
    window_size = int(fs * window_size_sec)

    if record.p_signal.shape[0] < window_size:
        print(f"[WARNING] Record {record_name} too short. Skipped.")
        return []

    signal = record.p_signal[:, lead]
    segments = []

    for i in range(len(ann.aux_note)):
        if ann.aux_note[i].startswith('('):
            label = ann.aux_note[i][1:]
            start = ann.sample[i]
            end = ann.sample[i + 1] if i + 1 < len(ann.sample) else len(signal)

            for s in range(start, end - window_size, window_size):
                segment = signal[s:s+window_size]
                segments.append((segment, label))

```

```

        return segments

# %%
all_segments = []
for rec in record_list:
    segmentslead = extract_windows(rec, window_size_sec=30)
    segmentsleadII = extract_windows(rec, window_size_sec=30, lead=1)
    segments = segmentslead + segmentsleadII
    all_segments.extend(segments)

# %%
X = []
y = []

for segment, label in all_segments:
    if label in ['AFIB', 'N']:
        X.append(segment)
        y.append(1 if label == 'AFIB' else 0)

X = np.array(X)
y = np.array(y)

print("N Label: ", np.sum(y == 0))
print("AFIB Label: ", np.sum(y == 1))

# %%
fs = 250
X.shape, y.shape, fs

# %%
def plot_signal(X, fs=250):
    """
    Plots time-domain and frequency-domain views of an ECG signal segment.

    Parameters:
        X (np.ndarray): 1D ECG segment
        fs (int): Sampling frequency in Hz
    """
    signal = X
    n = len(signal)
    freqs = np.fft.fftfreq(n, d=1/fs)
    fft_magnitude = np.abs(np.fft.fft(signal))

    # Plot time domain

```

```

plt.figure(figsize=(12, 5))

plt.subplot(2, 1, 1)
plt.plot(np.arange(n) / fs, signal)
plt.title("Time Domain Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Plot frequency domain
plt.subplot(2, 1, 2)
plt.plot(freqs[:n // 2], fft_magnitude[:n // 2])
plt.title("Frequency Domain (FFT Magnitude)")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")

plt.tight_layout()
plt.show()

# %%
plot_signal(X[0], fs=fs)

# %% [markdown]
# ## Preprocessing

# %% [markdown]
# ##### Some essential function to view what is going on

# %%
fs = 250 # Sampling frequency in Hz
X.shape, fs

# %%
def bandpass_filter(x, lowcut=0.5, highcut=40, fs=250, order=4):
    #
    nyq = fs/2
    # butter filter
    b, a = sp.butter(order, [lowcut/nyq, highcut/nyq], btype='band')
    return sp.filtfilt(b, a, x)

X_filt = bandpass_filter(X, fs=fs)
X_filt

X_norm = processing.normalize_bound(X_filt, lb=-1, ub=1)
X_norm

```

```

# %%
plot_signal(X[0], fs=fs)

# %%
plot_signal(X_filt[0], fs=fs)

# %%
plot_signal(X_norm[0], fs=fs)

# %%
time_axis = np.arange(len(X_norm[0])) / 250
afib_indices = np.where(y == 1)[0][:3]
normal_indices = np.where(y == 0)[0][:3]

plt.figure(figsize=(15, 8))

for i, idx in enumerate(afib_indices):
    plt.subplot(2, 3, i + 1)
    plt.plot(time_axis, X_norm[idx])
    plt.title(f'AFIB Sample {i + 1}')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')

for i, idx in enumerate(normal_indices):
    plt.subplot(2, 3, i + 4)
    plt.plot(time_axis, X_norm[idx])
    plt.title(f'Normal Sample {i + 1}')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')

plt.tight_layout()
plt.show()

# %% [markdown]
# ## Extract R-Wave

# %%
from wfdb import processing

fs = 250 # Sampling frequency
r_peaks_all = [] # To store R-peak indices for all segments

for segment in X_norm:
    # segment is a 1D ECG array
    xqrs = processing.XQRS(sig=segment, fs=fs)

```

```

    xqrs.detect()
    r_peaks_all.append(xqrs.qrs_inds) # Save R-peak indices

# %%
r_peaks_all

# %%
import matplotlib.pyplot as plt

# Select the first segment and its corresponding R-peaks
segment = X_norm[0]
r_peaks = r_peaks_all[0]

plt.figure(figsize=(12, 4))
plt.plot(segment, label='ECG')
plt.plot(r_peaks, segment[r_peaks], 'ro', label='R-peaks')
plt.legend()
plt.title("R-peak Detection using XQRS")
plt.xlabel("Samples")
plt.ylabel("Amplitude")
plt.show()

# %%
rr_intervals_all = []

for r_peaks in r_peaks_all:
    #  $RR(i) = R(i+1) - R(i)$ 
    rr_intervals = np.diff(r_peaks) / fs # Convert sample diff to seconds
    rr_intervals_all.append(rr_intervals)

# %%
bpm = 60 / rr_intervals

# %%
plt.plot(X_norm[1])

# %%
import matplotlib.pyplot as plt

# Plot RR intervals for the first segment (or any index you prefer)
rr_intervals = rr_intervals_all[0]

plt.figure(figsize=(10, 5))
plt.plot(rr_intervals, marker='o')
plt.title("RR Intervals (seconds) for Segment 0")

```

```

plt.xlabel("Beat Index")
plt.ylabel("Interval (s)")
plt.grid(True)
plt.show()

# %%
for i in range(3): # First 3 segments
    plt.plot(rr_intervals_all[i], marker='o', label=f'Segment {i}')
plt.title("RR Intervals Across Segments")
plt.xlabel("Beat Index")
plt.ylabel("Interval (s)")
plt.legend()
plt.grid(True)
plt.show()

# %%
rr_lengths = [len(rri) for rri in rr_intervals_all]
max_len = max(rr_lengths)
avg_len = int(np.mean(rr_lengths))
median_len = int(np.median(rr_lengths))

print(f"Max RR interval length: {max_len}")
print(f"Average RR interval length: {avg_len}")
print(f"median RR interval length: {median_len}")

# %%
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences

# rr_intervals_all is a list of np.arrays
X_rri = pad_sequences(rr_intervals_all, maxlen=avg_len, padding='post',
truncating='post', dtype='float32', value=-1)

# %%
X_rri.shape

# %%
X_rri = np.expand_dims(X_rri, axis=-1)
X_rri.shape

# %%
# Split data
from sklearn.model_selection import train_test_split

```

```

X_train, X_val, y_train, y_val = train_test_split(
    X_rri, y, test_size=0.2, random_state=42, stratify=y)

# %%
X_train.shape, X_val.shape, y_train.shape, y_val.shape

# %%
X_rri[1].shape

# %%
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense,
Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

model = Sequential([
    Conv1D(filters=32, kernel_size=5, activation='relu', padding='same',
input_shape=(X_rri.shape[1], 1)),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=64, kernel_size=3, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=64, kernel_size=5, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=128, kernel_size=3, activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer=Adam(1e-4), loss='binary_crossentropy',
metrics=['accuracy'])
print(model.summary())

```



```

# %%
from tensorflow.keras.callbacks import EarlyStopping

Early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    callbacks=[Early_stopping]
)

# %%
print(X_train.shape, y_train.shape)
print(X_val.shape, y_val.shape)

# %%
loss, accuracy = model.evaluate(X_val, y_val)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# %%
import matplotlib.pyplot as plt

plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.legend()
plt.title('Loss')

plt.subplot(1,2,2)
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.legend()
plt.title('Accuracy')

plt.show()

```

```

# %%
from sklearn.metrics import classification_report, confusion_matrix

y_pred = (model.predict(X_val) > 0.9).astype("int32")
print(classification_report(y_val, y_pred))

# %%
print(confusion_matrix(y_val, y_pred))

# %%
model.save("AFIB Detection Model.keras")

```

Appendix B: ESP32 Microcontroller Code (Arduino IDE)

```

#include <WiFi.h>
#include <WebServer.h>

#define WIFI_STATUS 2    // Built-in LED
#define UNDETECTED_LED 0 // Green LED
#define DETECTED_LED 4   // RED or Yellow LED
#define BUZZER 5         // Buzzer

const char* ssid = "DESKTOP-796GSE4 0780";
const char* password = "tU8641^1";

WebServer server(80);

// DETECTED: turn ON LED, relay, buzzer
unsigned long buzzerStart = 0;
bool buzzerOn = false;

void handleDetected() {
    digitalWrite(DETECTED_LED, HIGH);
    digitalWrite(UNDETECTED_LED, LOW);
    digitalWrite(BUZZER, HIGH);
    buzzerStart = millis();
    buzzerOn = true;
    server.send(200, "text/plain", "DETECTED - Devices ON");
}

```

```

// UNDETECTED: turn OFF all devices
void handleUndetected() {
    digitalWrite(UNDETECTED_LED, HIGH);
    digitalWrite(DETECTED_LED, LOW);
    server.send(200, "text/plain", "UNDETECTED - Devices OFF");
}

void setup() {
    Serial.begin(115200);

    pinMode(WIFI_STATUS, OUTPUT);
    pinMode(UNDETECTED_LED, OUTPUT);
    pinMode(DETECTED_LED, OUTPUT);
    pinMode(BUZZER, OUTPUT);

    digitalWrite(WIFI_STATUS, LOW);
    digitalWrite(DETECTED_LED, LOW);
    digitalWrite(UNDETECTED_LED, LOW);
    digitalWrite(BUZZER, LOW);

    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500); Serial.print(".");
    }
    digitalWrite(WIFI_STATUS, HIGH);
    delay(10);

    // Buzzer short beep after Wi-Fi connected
    for (int i = 0; i < 3; i++) {
        digitalWrite(BUZZER, HIGH);
        delay(100);
        digitalWrite(BUZZER, LOW);
        delay(100);
    }
}

```

```

Serial.println("\nConnected to WiFi");
Serial.print("IP address: ");
Serial.println(WiFi.localIP());

server.on("/DETECT", handleDetected);
server.on("/detect", handleDetected);
server.on("/UNDETECTED", handleUndetected);
server.on("/undetected", handleUndetected);

server.begin();
Serial.println("HTTP server started");
}

void loop() {
    server.handleClient();

    if (buzzerOn && (millis() - buzzerStart > 1000)) {
        digitalWrite(BUZZER, LOW);
        buzzerOn = false;
    }
}

```

Appendix C: Web Application – Flask and Frontend

```

from flask import Flask, request, render_template, jsonify
from tensorflow.keras.models import load_model
import numpy as np
import os
import requests
import neurokit2 as nk

app = Flask(__name__)
ESP32_IP = "http://192.168.137.174" # Replace with actual ESP32 IP

# Load model
model_path = "D:/RUPP/RUPP/RUPP Y3S1/Machine
Learning/FinalProject/AF_Detection/dataset/AFIB Detection Model.keras"

```

```

model = load_model(model_path)

@app.route('/')
def index():
    return render_template("index.html")

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()

    if not data or 'signal' not in data:
        return jsonify({'error': 'No ECG signal provided'}), 400

    try:
        raw_ecg = np.array(data['signal'], dtype=np.float32)

        # Normalize raw ECG
        raw_ecg = (raw_ecg - np.min(raw_ecg)) / (np.max(raw_ecg) -
np.min(raw_ecg) + 1e-6)

        # RRI extraction using neurokit2
        signals, info = nk.ecg_process(raw_ecg, sampling_rate=250)
        rpeaks = info["ECG_R_Peaks"]

        # RR Intervals (in seconds)
        rri = np.diff(rpeaks) / 250.0

        if len(rri) < 39:
            return jsonify({'error': f'Not enough RR intervals: {len(rri)}'}),
400

        desired_len = model.input_shape[1] # 39
        if len(rri) > desired_len:
            rri = rri[:desired_len]
        else:
            rri = np.pad(rri, (0, desired_len - len(rri)), mode='constant')

        # Correct shape: (1, 39, 1)
        model_input = rri.reshape(1, desired_len, 1)

        # Predict
        score = model.predict(model_input)[0][0]
        result = "Atrial Fibrillation Detected" if score > 0.5 else "Normal
Rhythm"

```

```

# ESP32 Command
esp_status = "DETECT" if score > 0.9 else "UNDETECTED"
send_to_esp32(esp_status)

return jsonify({
    'prediction': result,
    'score': f"{score * 100:.2f}%", # formats score as percentage with 2
decimal places
    'esp32_status': esp_status
})

except Exception as e:
    return jsonify({'error': str(e)}), 500

def send_to_esp32(command):
    try:
        url = f"{ESP32_IP}/{command}"
        requests.get(url, timeout=2)
        print(f"[ESP32] Sent: {command}")
    except requests.exceptions.RequestException as e:
        print(f"[ERROR] ESP32 not reachable: {e}")

@app.route('/status')
def status():
    return jsonify({"status": "running"})

if __name__ == '__main__':
    app.run(debug=True)

```

Appendix D: Sample ECG JSON Files

```

import os
import wfdb
import numpy as np
import scipy.signal as sp
import matplotlib.pyplot as plt
from pathlib import Path

script_dir = Path(__file__).parent.resolve()

# Define the output folder relative to the script directory
output_folder = script_dir / "Generated_ecg"
output_folder.mkdir(exist_ok=True)

```

```

# Define paths and record info
base_dir = Path(r"D:/RUPP/RUPP/RUPP Y3S1/Machine
Learning/FinalProject/AF_Detection/dataset/afdb")
record_name = "04043"
record_path = base_dir / record_name

print("Exists?", record_path.with_suffix(".hea").exists()) # Should print True

# Load full record and annotation once
record = wfdb.rdrecord(str(record_path))
ann = wfdb.rdann(str(record_path), 'atr')

print("Record path: ", record_path)
print("Record loaded. Shape:", record.p_signal.shape)
print("Annotations sample:", ann.aux_note[:5])

def bandpass_filter(x, lowcut=0.5, highcut=40, fs=250, order=4):
    nyq = fs / 2
    b, a = sp.butter(order, [lowcut / nyq, highcut / nyq], btype='band')
    return sp.filtfilt(b, a, x)

def normalize_signal(sig):
    sig_min = np.min(sig)
    sig_max = np.max(sig)
    range_ = sig_max - sig_min
    if range_ == 0:
        return np.zeros_like(sig) # or just return sig unchanged
    return 2 * (sig - sig_min) / range_ - 1

def extract_real_ecg_segments(record, ann, label_filter, max_samples=100,
window_sec=30, lead=0):
    fs = record.fs
    window_size = int(fs * window_sec)

    signal = record.p_signal[:, lead] + record.p_signal[:, lead + 1]
    segments = []
    count = 0

    for i in range(len(ann.sample) - 1):
        label = ann.aux_note[i].strip "() "
        # Remove next_label check

        if label not in label_filter:
            continue

```

```

        start = ann.sample[i]
        end = ann.sample[i + 1]

        for s in range(start, end - window_size, window_size):
            seg = signal[s:s + window_size]
            if len(seg) < window_size:
                continue
            seg = bandpass_filter(seg, fs=fs)
            seg = normalize_signal(seg)
            segments.append((seg, label))
            count += 1
            if count >= max_samples:
                return segments

    return segments

import json

def save_ecg_json(segment, label, index, subfolder):
    folder = script_dir / subfolder
    folder.mkdir(parents=True, exist_ok=True)

    filename = f"{label}_{index}.json"
    path = folder / filename

    data = {
        "label": label,
        "signal": segment.tolist()
    }

    with open(path, 'w') as f:
        json.dump(data, f)

if __name__ == "__main__":
    afib_segments = extract_real_ecg_segments(record, ann, label_filter=["AFIB"],
max_samples=100)
    normal_segments = extract_real_ecg_segments(record, ann, label_filter=["N"],
max_samples=100)

    print(f"Extracted {len(afib_segments)} AFIB segments and
{len(normal_segments)} Normal segments.")

    for i, (seg, _) in enumerate(afib_segments):
        save_ecg_json(seg, "AFIB", i, "Generated_json/AFIB")

```



```

for i, (seg, _) in enumerate(normal_segments):
    save_ecg_json(seg, "Normal", i, "Generated_json/Normal")

print("JSON files saved in Generated_json/")
print(f"AFIB JSON saved: {len(afib_segments)}")
print(f"Normal JSON saved: {len(normal_segments)}")

```

Appendix E: Web Application Structure Application – HTML, CSS and Java Script

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />

    <style>
        body {
            font-family: Arial, sans-serif;
            text-align: center;
            margin-top: 60px;
            background-color: #f4f6f8;
        }

        h1 {
            color: #2c3e50;
        }
        h2 {
            color: #2c3e50;
        }

        form {
            margin: 20px auto;
            padding: 30px;
            border-radius: 10px;
            background: white;
            box-shadow: 0 0 10px rgba(0,0,0,0.1);
            display: inline-block;
        }

        input[type="file"] {
            padding: 10px;
        }

        button {
            background-color: #3498db;
            color: white;

```

```

        padding: 10px 20px;
        margin-top: 15px;
        border: none;
        border-radius: 5px;
        cursor: pointer;
    }

    .result {
        margin-top: 20px;
        font-size: 1.2em;
    }

    /* Popup styling */
    #popup {
        display: none;
        position: fixed;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
        background: white;
        border: 1px solid #ccc;
        border-radius: 8px;
        box-shadow: 0 5px 15px rgba(0,0,0,0.3);
        padding: 20px;
        width: 850px;
        max-width: 95%;
        z-index: 9999;
        text-align: center;
    }
</style>
</head>
<body>
    <h1>AFib Detection from ECG Signal (JSON)</h1>
    <h2 id="test-status" style="font-size: 1.1em; color: #7f8c8d;">Machine
Learning Course</h2>
    <p>By Heng Lymeng</p>

    <form id="json-form">
        <input type="file" id="jsonFile" accept=".json" required />
        <button type="submit">Upload</button>
    </form>

    <div class="result" id="result"></div>

    <!-- Popup container -->

```

```

<div id="popup">
  <div id="view-container">
    <canvas id="ecg-canvas" width="800" height="300" style="border:1px
solid #ddd; margin-bottom: 15px; display:none;"></canvas>
    <div id="prediction-result" style="display:none; width:800px;
height:300px; border:1px solid #ddd; margin-bottom: 15px;"></div>
  </div>

  <div style="margin-top: 10px;">
    <button id="btn-visualize">Visualize Signal</button>
    <button id="btn-predict">Prediction</button>
    <button id="btn-reset" style="background:#e74c3c; color:#fff; margin-
left: 15px;">Reset</button>
  </div>
</div>

<script>
  let ecgSignal = [];

  // Handle JSON file upload and prediction request
  document.getElementById("json-form").addEventListener("submit", function
(e) {
    e.preventDefault();

    const fileInput = document.getElementById('jsonFile');
    const file = fileInput.files[0];

    if (!file) {
      alert("Please select a JSON file.");
      return;
    }

    const reader = new FileReader();
    reader.onload = function (e) {
      try {
        const ecgData = JSON.parse(e.target.result);
        if (!ecgData.signal || !Array.isArray(ecgData.signal)) {
          throw new Error("JSON does not contain valid 'signal'
array.");
        }

        ecgSignal = ecgData.signal; // save globally

        fetch('/predict', {
          method: 'POST',

```

```

        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(ecgData)
    })
    .then(response => response.json())
    .then(data => {
        if (data.error) {
            document.getElementById('result').innerHTML = `
```

```

        } catch (error) {
            document.getElementById('result').innerHTML = `
```

```

        const maxVal = Math.max(...ecgSignal);
        const normVal = (val - minVal) / (maxVal - minVal);

        const y = height - normVal * height;

        if (i === 0) ctx.moveTo(i, y);
        else ctx.lineTo(i, y);
    }

    ctx.strokeStyle = "#3498db";
    ctx.lineWidth = 2;
    ctx.stroke();

    animationIndex += pointsPerFrame;

    if (animationIndex + width < ecgSignal.length) {
        animationFrameId = requestAnimationFrame(animateECG);
    }
}

// Show popup and default to visualize view
function showPopup() {
    popup.style.display = 'block';
    showPrediction();
}

// Hide popup and reset everything
function hidePopup() {
    popup.style.display = 'none';
    cancelAnimationFrame(animationFrameId);
    clearCanvas();
    predictionResultDiv.innerHTML = '';
    predictionResultDiv.style.display = 'none';
    canvas.style.display = 'none';
    animationIndex = 0;
    ecgSignal = [];

    // Also clear main results and file input
    document.getElementById('result').innerHTML = '';
    document.getElementById('jsonFile').value = '';
    document.getElementById('btn-visualize').style.display = 'none';
}

// Visualize ECG signal animation
function showVisualize() {

```

```

        cancelAnimationFrame(animationFrameId);
        predictionResultDiv.style.display = 'none';
        canvas.style.display = 'block';
        predictionResultDiv.innerHTML = '';
        animationIndex = 0;
        animateECG();
    }

    // Show prediction results inside popup
    function showPrediction() {
        cancelAnimationFrame(animationFrameId);
        clearCanvas();
        canvas.style.display = 'none';

        // Show prediction content copied from main result div
        predictionResultDiv.style.display = 'block';
        predictionResultDiv.innerHTML =
document.getElementById('result').innerHTML;
    }

    // Reset everything
    function resetAll() {
        hidePopup();
    }

    // Attach button event listeners
    btnVisualize.addEventListener('click', showVisualize);
    btnPredict.addEventListener('click', showPrediction);
    btnReset.addEventListener('click', resetAll);

    // Optional: close popup when clicking outside of it
    window.addEventListener('click', function(event) {
        if (event.target === popup) {
            resetAll();
        }
    });

    // Optional: close popup on ESC key press
    window.addEventListener('keydown', function(event) {
        if (event.key === "Escape" && popup.style.display === 'block') {
            resetAll();
        }
    });

    // Initially hide visualize button until prediction is done

```

```
        document.getElementById('btn-visualize').style.display = 'none';  
    </script>  
</body>  
</html>
```