

/Users/mlnick/Documents/Images/tsuku

Practical Development for IoT and Embedded Systems

Report 1 on the Individual Application Project

Mamanchuk Mykola, SID.202420671

June 6, 2024

1 Introduction

1.1 Application Description

The **AlcoTester** application is designed with a humorous touch, intended to be funny among friends. The idea stems from the observation that a person, especially when drunk, might not quickly understand the sequence of events in the test. Imagine the following scenario: you start the test and hand the phone to your friend. Confused and possibly intoxicated, your friend changes the position of the phone during the accelerometer test. Next, during the voice test, there is a high likelihood of your friend altering his voice due to inebriation. Although this situation is somewhat silly, it highlights the fun aspect of the application.

While the primary purpose of the application is to entertain, it also serves a functional role in assessing a person's sobriety. However, it is important to note that the current version of the application requires further development to confidently catch a person who is drunk. If you are alone, it is still not recommended to solely rely on this application to test your sobriety. In cases where you are unsure of your state, this application can provide an independent assessment to help you make an informed decision.

The complete listing source for the application can be accessed following the reference [1].

1.2 Development Environment

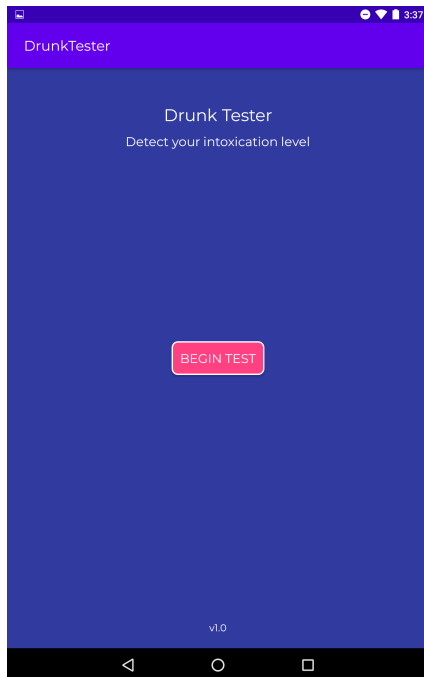
The **AlcoTester** application was developed using *Android Studio* on a Windows 11 Pro system with an Intel Core i7-1185G7 processor. The device used for testing the application was a Nexus 7.

2 How to Use

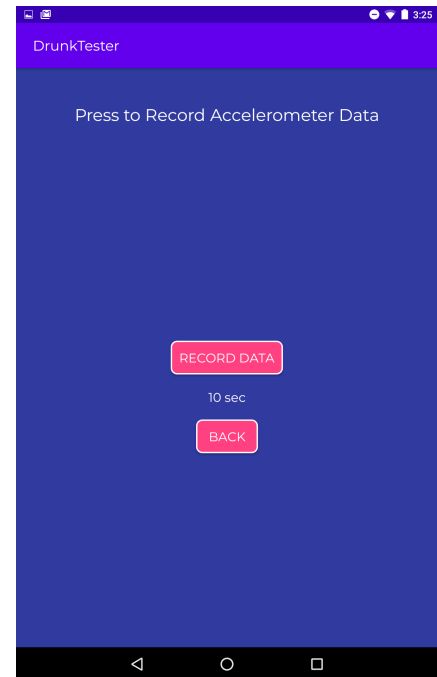
The UI is designed to be simple and intuitive to avoid user confusion. Once the user starts the test, the data recording begins with the accelerometer, followed by the voice data. The interface shows the progress of the recordings, and the results are displayed as a probability of drunkenness.

1. **Start Screen:** Launch the app and press the "Begin Test" button.
2. **Accelerometer Recording:**
 - Press "Record Data".
 - Follow on-screen instructions to hold the device steady for 10 seconds.
3. **Voice Recording:**
 - Press "Record Voice".
 - Speak a sentence into the microphone for 5 seconds.
4. **Analyze Results:**
 - Press "Analyze Results" to process the recorded data.
 - View the displayed percentage likelihood of being drunk.

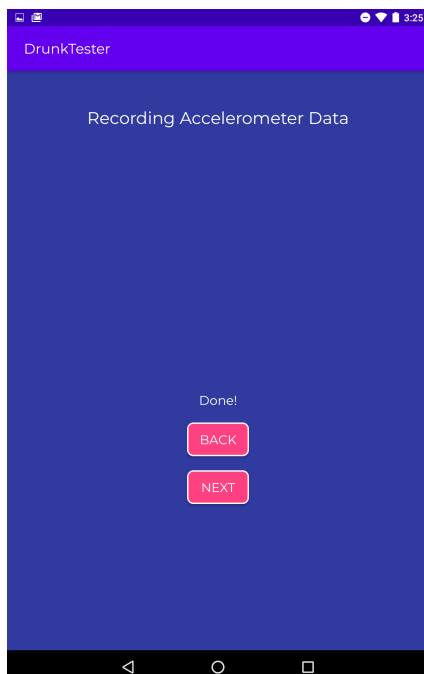
3 Screenshots



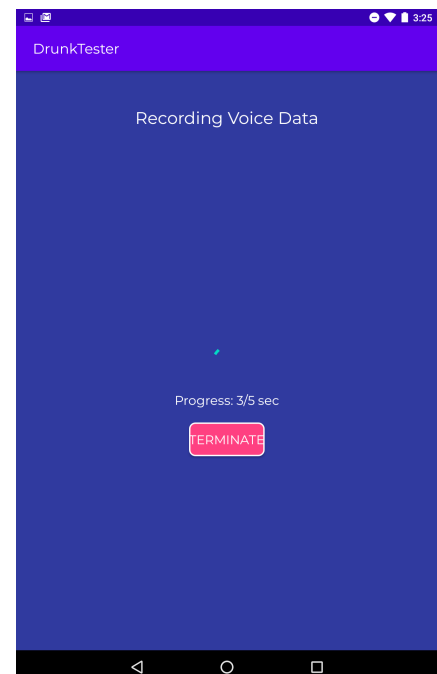
Start Screen: The initial screen of the Drunk Tester application.



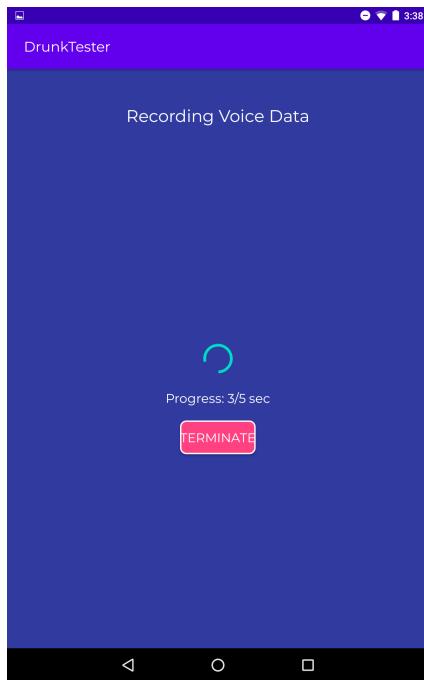
Press to Record Accelerometer Data: The screen instructing the user to record accelerometer data.



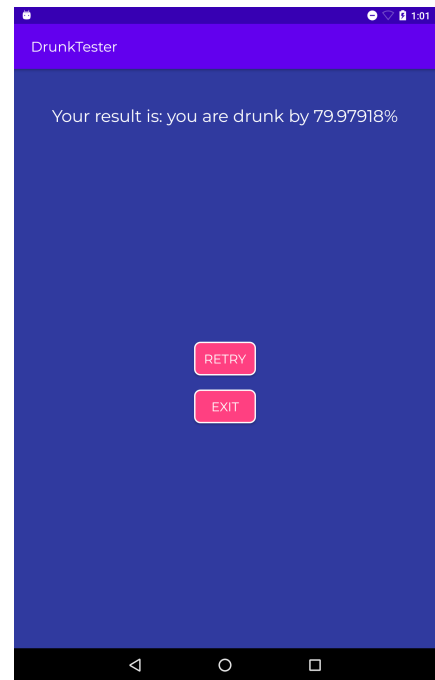
Recording Accelerometer Data: Progress screen showing the recording of accelerometer data.



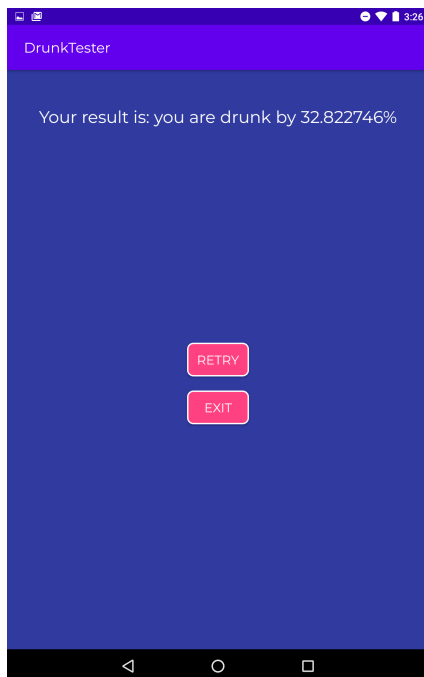
Press to Record Voice Data: The screen instructing the user to record voice data.



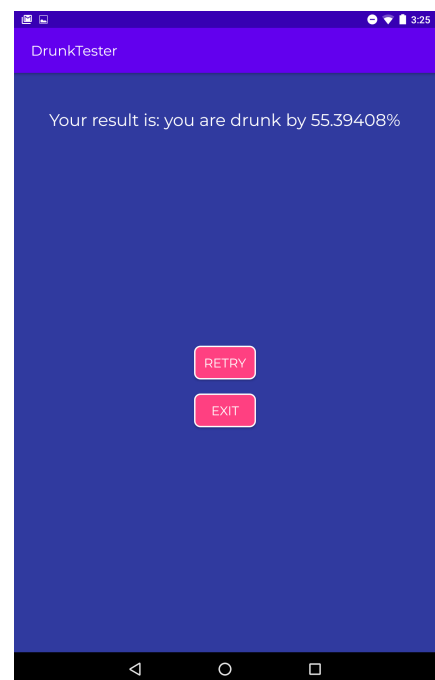
Recording Voice Data: Progress screen showing the recording of voice data.



Result Showcase: The screen displaying the result of the drunk test.



Result Showcase Low: The screen displaying a lower percentage of being drunk.



Result Showcase Undetermined: The screen displaying 50% result.

Screenshots from the Drunk Tester application.

4 Implementation Details

- **Technologies Used:**
 - **Android Studio:** Development environment.
 - **Java:** Programming language for Android app.
 - **JTransforms:** Library for performing FFT on voice data.
- **Data Collection:**
 - **Accelerometer:** Captures movement data.
 - **Microphone:** Records voice.
- **Data Analysis:**
 - **Accelerometer:** Computes average acceleration and changes to determine stability.
 - **Voice:** Applies FFT to analyze frequency components and compare power across frequency lanes.
- **Algorithm:**
 - Accelerometer data is analyzed using variance and rate of change.
 - Voice data is processed using FFT to compute power in specified frequency lanes.

5 Code Overview

5.1 Class Descriptions

- **AccelerationAnalyzer.java** - Contains the class **AccelerationAnalyzer** which computes the probability of being drunk based on accelerometer data.
- **AccelerometerActivity.java** - Manages the accelerometer recording activity.
- **AnalyzeActivity.java** - Handles the analysis and display of results.
- **DataManager.java** - Manages the storage and retrieval of recorded data.
- **MainActivity.java** - The main entry point of the application.
- **RecordingAccelerometerActivity.java** - Records accelerometer data.
- **RecordingVoiceActivity.java** - Records voice data.
- **VoiceAnalyzer.java** - Analyzes the recorded voice data.
- **VoiceRecordActivity.java** - Manages the voice recording activity.

5.2 Class and Method Descriptions

| Class | Description |
|--|--|
| AccelerationAnalyzer.java | Contains methods to compute essential metrics from accelerometer data. |
| Method | Description |
| analyzeAcceleration- (List<float[]>) | Computes essentials passing the recorded data from accelerometers. |

| Class | Description |
|-----------------------------------|--|
| AccelerometerActivity.java | Manages the activity for recording accelerometer data. |
| Method | Description |
| onCreate | Initializes the activity. |

| Class | Description |
|-----------------------------|---|
| AnalyzeActivity.java | Handles the analysis of the recorded data. |
| Method | Description |
| onCreate analyzeData | Initializes the activity. Analyzes the collected accelerometer and voice data. |

| Class | Description |
|---|--|
| DataManager.java | Manages data storage and retrieval for the application. |
| Method | Description |
| DataManager getInstance setAccelerometerData getAccelerometerData setVoiceFileName getVoiceFileName clearData | Constructor for initializing the DataManager. Returns the singleton instance of the DataManager. Sets the accelerometer data. Retrieves the accelerometer data. Sets the filename for the recorded voice data. Retrieves the filename for the recorded voice data. Clears all stored data. |

| Class | Description |
|--|---|
| MainActivity.java | Main activity that initializes the application and checks permissions. |
| Method | Description |
| onCreate checkPermissions onRequestPermissionsResult | Initializes the activity. Checks if the required permissions are granted. Handles the result of the permission request. |

| Class | Description |
|--|--|
| RecordingAccelerometerActivity.java | Manages the recording of accelerometer data. |

| Method | Description |
|-------------------|---|
| onCreate | Initializes the activity. |
| startRecording | Starts the recording of accelerometer data. |
| run | Runnable for recording data at regular intervals. |
| stopRecording | Stops the recording of accelerometer data. |
| onSensorChanged | Handles sensor data changes. |
| onAccuracyChanged | Handles changes in sensor accuracy. |

| Class | Description |
|-------------------------------------|---|
| RecordingVoice-Activity.java | Manages the recording of voice data. |
| Method | Description |
| onCreate | Initializes the activity. |
| startRecording | Starts the recording of voice data. |
| run | Runnable for recording data at regular intervals. |
| stopRecording | Stops the recording of voice data. |

| Class | Description |
|---------------------------|--|
| VoiceAnalyzer.java | Analyzes the recorded voice data using FFT. |
| Method | Description |
| analyzeVoice | Analyzes the voice data using FFT. |
| readAudioFile | Reads the recorded audio file. |
| convertToDoubleArray | Converts audio data to a double array. |
| calculatePowerSpectrum | Calculates the power spectrum of the audio data. |
| analyzeFrequencyLanes | Analyzes the power in specified frequency lanes. |

| Class | Description |
|---------------------------------|---------------------------------------|
| VoiceRecordActivity.java | Manages the voice recording activity. |
| Method | Description |
| onCreate | Initializes the activity. |

6 Data Analysis Details

6.1 Analyzing and Computing Accelerometer Data

The listing of the computations related to this section will be followed in the Appendix A. To visualize the overall computation, let's break it down into a flowchart or pseudocode:

1. **Compute Norm of Accelerometer Data:**

$$xyzi[i] = \sqrt{x_i^2 + y_i^2 + z_i^2}$$

2. **Compute Rate of Change:**

$$XYZ[0] = \frac{|xyzi[0] - xyzi[1]|}{\Delta}$$

$$XYZ[i] = \frac{|xyzi[i-1] - 2 \times xyzi[i] + xyzi[i+1]|}{2 \times \Delta}$$

$$XYZ[length-1] = \frac{|xyzi[length-2] - xyzi[length-1]|}{\Delta}$$

3. Compute Average Magnitude:

$$xyz_avg = \frac{1}{length} \sum_{i=0}^{length-1} xyzi[i]$$

4. Compute Final Result:

$$result = 100 \times \left(\frac{1}{1 + e^{-\tau \times (xyz_avg - b)}} \right)$$

The parameters b and τ are defined based on experimental data. The parameter b corresponds to the accumulated bias, while τ determines the slope of the sigmoid function for a more responsive result.

6.2 Analyzing and Computing Voice Data

Just as with the previous subsection, the listing related to this part is provided in Appendix B. The process involves applying Fourier transformation to analyze the voice in the frequency space.

1. Fourier Transform:

$$A[k] = \sum_{n=0}^{N-1} a[n] e^{-2\pi i k n / N}$$

2. Power Spectrum:

$$P[k] = (\text{Re}(A[k]))^2 + (\text{Im}(A[k]))^2$$

3. Frequency Lanes:

$$\text{Lanes} = \{[60, 90], [90, 120], \dots, [270, 300]\}$$

4. Power in Lanes:

$$P_{\text{lane}_i} = \sum_{k=f_{\text{start}}}^{f_{\text{end}}} P[k]$$

5. Result Calculation:

$$\text{Result} = \begin{cases} 1 & \text{if Count} \geq 5 \\ 0.8 & \text{if Count} = 4 \\ 0.75 & \text{if Count} = 3 \\ 0.5 & \text{if Count} = 2 \\ 0 & \text{otherwise} \end{cases}$$

7 Inferences and Future Improvements

The parameters for the acceleration activity, b and τ , are defined based on experiments. The parameter b corresponds to the accumulated bias due to the nature of accelerometers, which always have some motion and record an approximate bias of 9.8. The parameter τ changes the slope of the sigmoid function to make the result more responsive to changes in acceleration.

The current version of the application computes the result as a weighted average of the accelerometer and voice data results, with weights of 0.8 and 0.2, respectively. This weighting may need adjustment based on further testing.

Future improvements include dynamically adjusting the sound frequency bandwidth lanes for less discrete analysis and further tuning the parameters for more accurate results.

References

1. Mamanchuk N., University of Tsukuba, DrunkTester Project for Android - Github, June 6, 2024. Available online: <https://github.com/RIFLE/IoT-Projects>

Appendix A. Listing AccelerationAnalyzer.java

The following code is the implementation of the class which provides a method `analyzeAcceleration()` to compute a result for the correspondingly recorded data.

```
1 public class AccelerationAnalyzer {
2
3 public static float analyzeAcceleration(List<float[]> accelerometerData
4 ) {
5
6     double b = 9.805;
7     double tau = 22.67;
8
9     double[] xyzi = new double[accelerometerData.size()];
10    for (int i = 0; i < accelerometerData.size(); i++) {
11        float x = accelerometerData.get(i)[0];
12        float y = accelerometerData.get(i)[1];
13        float z = accelerometerData.get(i)[2];
14        xyzi[i] = Math.sqrt(x * x + y * y + z * z);
15    }
16
17    // Compute the rate of change
18    double[] XYZ = new double[xyzi.length];
19    double delta = 0.1;
20    XYZ[0] = Math.abs(xyzi[0] - xyzi[1]) / delta;
21    for (int i = 1; i < xyzi.length - 1; i++) {
22        XYZ[i] = Math.abs(xyzi[i - 1] - 2 * xyzi[i] + xyzi[i + 1])
23        / (2 * delta);
24    }
25    XYZ[xyzi.length - 1] = Math.abs(xyzi[xyzi.length - 2] - xyzi[
26    xyzi.length - 1]) / delta;
27
28    // Compute the average modulus of momentary acceleration
29    double xyz_avg = 0;
30    for (double xyz : xyzi) {
31        xyz_avg += xyz;
32    }
33    xyz_avg /= xyzi.length;
34
35    // Compute the result
36    return (float) (100 * (1 / (1 + Math.exp(- tau * (xyz_avg - b))
37    )));
38 }
```

Appendix B. Listing VoiceAnalyzer.java

Similarly to the previously defined we provide the listing for analyzing voice data.

```
1 public class VoiceAnalyzer {
2
3 public static float analyzeVoice(String voiceFileName) {
4     try {
5         // Read the audio file
6         byte[] audioBytes = readAudioFile(voiceFileName);
7
8         // Convert bytes to double for FFT
9         double[] audioData = convertToDoubleArray(audioBytes);
10
11        // Perform FFT
12        DoubleFFT_1D fft = new DoubleFFT_1D(audioData.length);
13        fft.realForward(audioData);
14
15        // Calculate power spectrum
16        double[] powerSpectrum = calculatePowerSpectrum(audioData);
17
18        // Analyze power in specified frequency lanes
19        return analyzeFrequencyLanes(powerSpectrum);
20
21    } catch (IOException e) {
22        e.printStackTrace();
23        return 0;
24    }
25 }
26
27 private static byte[] readAudioFile(String voiceFileName) throws
IOException {
28     File file = new File(voiceFileName);
29     FileInputStream fis = new FileInputStream(file);
30     byte[] data = new byte[(int) file.length()];
31     fis.read(data);
32     fis.close();
33     return data;
34 }
35
36 private static double[] convertToDoubleArray(byte[] audioBytes) {
37     double[] audioData = new double[audioBytes.length];
38     for (int i = 0; i < audioBytes.length; i++) {
39         audioData[i] = (double) audioBytes[i];
40     }
41     return audioData;
42 }
43
44 private static double[] calculatePowerSpectrum(double[] audioData)
{
45     int n = audioData.length / 2;
46     double[] powerSpectrum = new double[n];
47     for (int i = 0; i < n; i++) {
48         double real = audioData[2 * i];
49         double imag = audioData[2 * i + 1];
50         powerSpectrum[i] = Math.sqrt(real * real + imag * imag);
51     }
```

```

52     return powerSpectrum;
53 }
54
55 private static float analyzeFrequencyLanes(double[] powerSpectrum)
56 {
57     int sampleRate = 44100; // Change if your sample rate is
different
58     int n = powerSpectrum.length;
59
60     // Frequency resolution
61     double freqResolution = (double) sampleRate / n;
62
63     // Define lanes
64     int[] lanes = {60, 90, 120, 150, 180, 210, 240, 270, 300};
65     double[] lanePowers = new double[lanes.length - 1];
66
67     // Calculate power in each lane
68     for (int i = 0; i < lanes.length - 1; i++) {
69         int startFreq = lanes[i];
70         int endFreq = lanes[i + 1];
71         int startIndex = (int) (startFreq / freqResolution);
72         int endIndex = (int) (endFreq / freqResolution);
73
74         for (int j = startIndex; j < endIndex; j++) {
75             lanePowers[i] += powerSpectrum[j];
76         }
77     }
78
79     // Find the most powerful lane
80     double maxPower = 0;
81     for (double power : lanePowers) {
82         if (power > maxPower) {
83             maxPower = power;
84         }
85     }
86
87     double centralLanePower = 0;
88     int centralLaneIndex = 0;
89
90     for (int i = 0; i < lanePowers.length; i++) {
91         if (lanePowers[i] == maxPower) {
92             centralLanePower = lanePowers[i];
93             centralLaneIndex = i;
94             break;
95         }
96     }
97
98     // Compare power of other lanes with the central lane
99     int count = 0;
100     for (int i = 0; i < lanePowers.length; i++) {
101         if (i != centralLaneIndex && lanePowers[i] >= 0.75 *
centralLanePower) {
102             count++;
103         }
104     }
105
106     // Calculate the result
107     if (count >= 5) {

```

```
107         return 1;
108     } else if (count == 4) {
109         return 0.8f;
110     } else if (count == 3) {
111         return 0.75f;
112     } else if (count == 2) {
113         return 0.5f;
114     } else {
115         return 0;
116     }
117 }
118 }
```