Project Report

Title:

Gesture Recognition and Anomaly Detection using Machine Learning Algorithms in an Embedded Platform

Submitted by:
(1). **Frank Efe Erukainure**
I.D: 032020008
(2). **Jacob Herman**
I.D: 022020003

Submitted to
Course Instructor: Dr. Rami Zewail

Course Title: CSE 521: Embedded Machine Learning

February 22, 2022

Table of Content

Gesture Recognition and Anomaly Detection using Machine Learning Algorithms in an Embedded Platform

1. Introduction

Gesture recognition refers to the technology that uses sensors to read and interpret hand movements as commands. Gesture recognition is a computing process that attempts to recognize and interpret human gestures through the use of mathematical algorithms. Gesture recognition is not limited to just human hand gestures, but rather can be used to recognize everything from head nods to different walking gaits.

Gesture recognition is a growing field of computer science, with an international conference devoted to gesture and facial recognition. As the field continues to grow, so will the ways that it can be utilized. Gesture recognition computer processes are designed to enhance human-computer interaction, and can occur in multiple ways, such as through the use of touch screens, a camera, or peripheral devices (Virtusa Corp., 2022).

On the other hand, anomaly detection identifies data points, events, and/or observations that deviate from a dataset's normal behavior. Anomalous data can indicate critical incidents, such as a technical glitch, or potential opportunities, for instance a change in consumer behavior. Machine Learning (ML) is progressively being used to automate anomaly detection (Cohen, 2022).

In this project, we demonstrate the use of machine learning algorithms for both gesture recognition and anomaly detection in an embedded platform based on a publicly available resource in the Eloquent Arduino blog of GitHub (Simone, 2021). The embedded platform used in this project is the Arduino Mega 2560 board, and the sensors utilized are the MPU9250 and the Sound sensors.

2. Gesture Recognition

Performing gesture recognition on Arduino boards can be very straightforward; it comprises of 2 steps:

- capture data to train a Machine Learning model on your PC, and
- export that model to plain C++ back into your board to predict the gestures.

It requires a great deal of efforts with the Eloquent Arduino machine learning library as prior knowledge of coding with C++ and Python scripts is required.

The steps employed in achieving the gesture recognition project are as discussed in the following subsections.

2.1 Capturing the training data

When developing a ML project (be it with Arduino or not), the first thing that is needed is data. The kind and complexity of the data required will depend on the specific project and can vary broadly: it can be sounds, images, sensor data, and so on.

In our case, we used a sensor data produced from an accelerometer. The name of the sensor was the MPU9250 which was purchased from Amazon online shop. Fig. 1 shows the sensor and its connection to the Arduino Mega 2560 board.

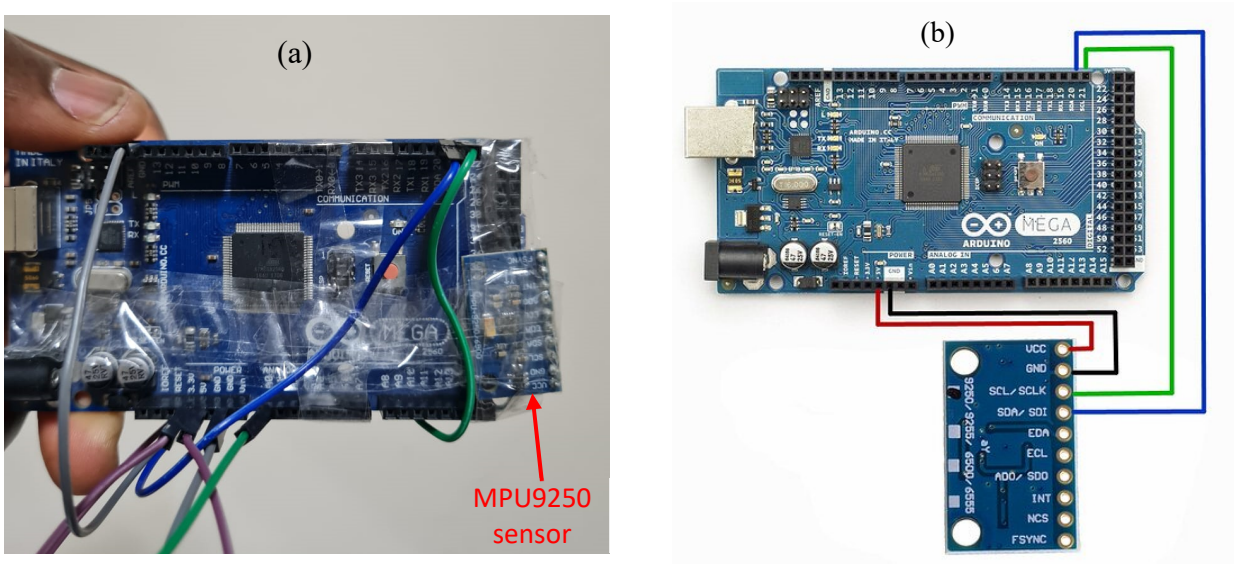

Fig. 1. (a) Arduino Mega 2560 board and the MPU9250 sensor. (b) Connection diagram.

We captured data with the sensor configuration shown in Fig. 1 for the following gestures:

- Rest – when no motion/gesture was being performed,
- Vertical motion of the accelerometer sensor,
- Horizontal motion of the accelerometer sensor, and
- Circular motion of the accelerometer sensor.

The data was captured for an average time of 30 seconds for each gesture, and the captured data was used to train two Machine Learning classifiers, viz. Support Vector Machines (SVM) and Random Forest classifier. The training process is described in the next subsection.

2.2 Training the ML classifier

When working with motion data (or any time-series data), we cannot just pick the accelerometer values at a given instant and tell what gesture the user is performing. One has to look at a *window* of data. In other

words, with time series data our aim is to analyze windows of data, not single values. How long this window should last will depend on the kind of gestures you want to recognize. So, we need a way to reframe the data we have in a format suitable to train a ML classifier on. This step is super-easy thanks to a small library called *Embedded Window* which is publicly available on GitHub.

Now we can apply the windowing to our data with a single line of Python code. 80% data was used for training, and 20% data was reserved for validation and testing. Once the captured data is loaded, we can see a plot of the data as shown in Fig. 2.

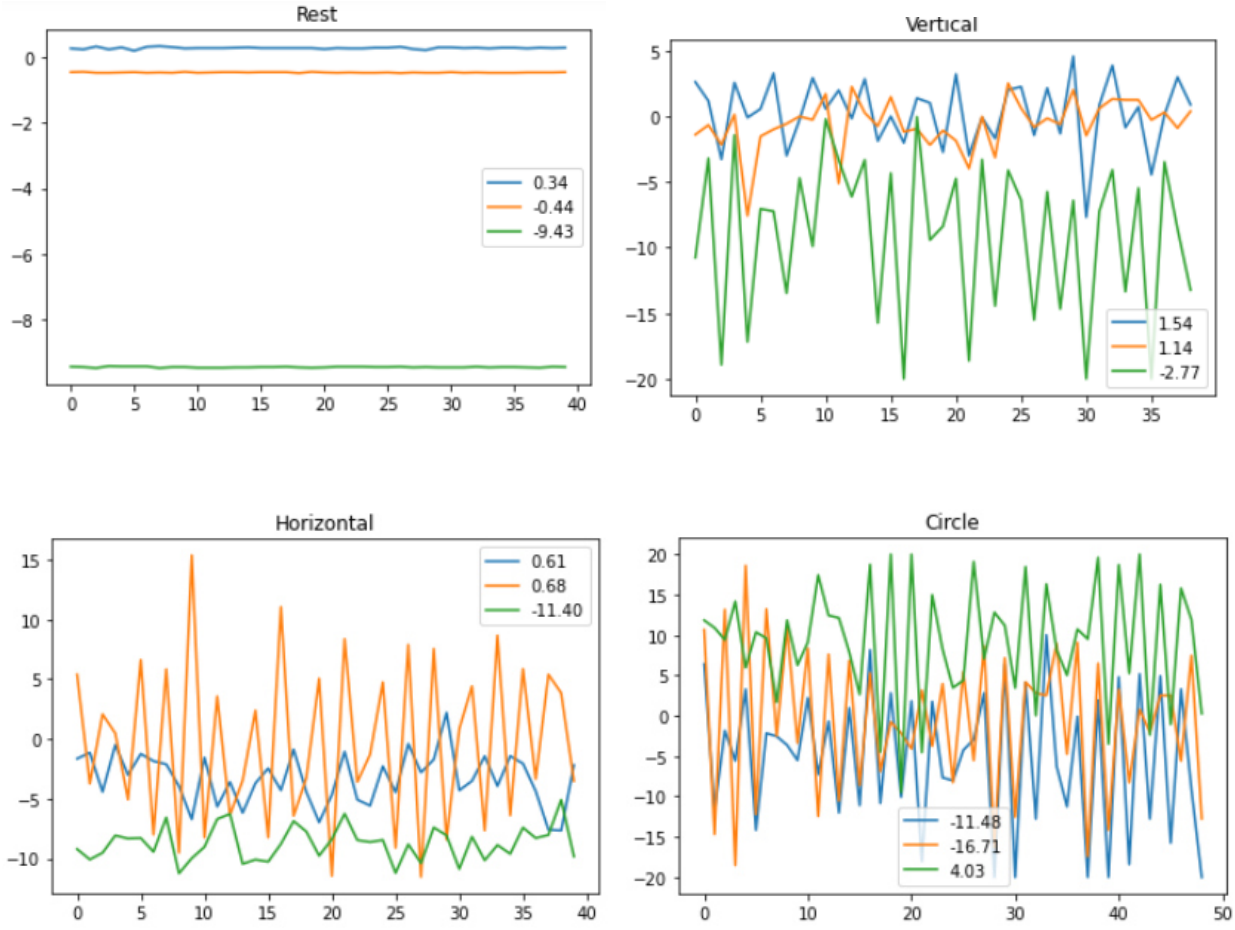

Fig. 2. Plots of each gesture and shape of the captured data.

After training our ML classifiers, we got good results as shown in the confusion matrix of Fig. 3. As can be seen, the values around the matrix diagonal are very high, and the others very low, which is a good result in most practical scenarios.

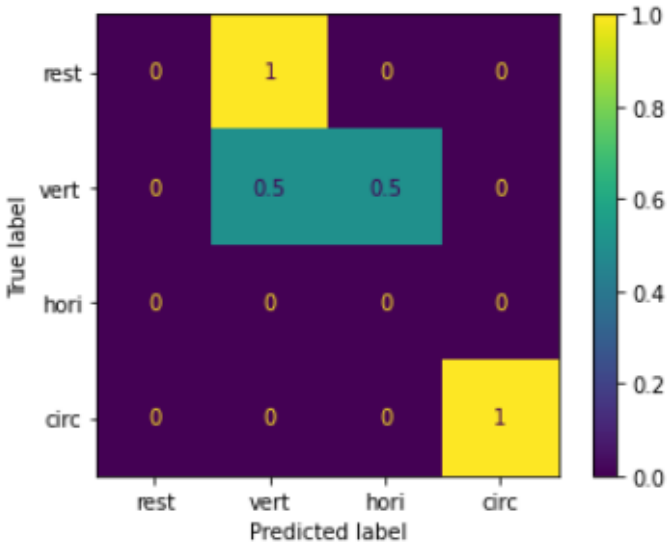

Fig. 3. Confusion matrix of the ML classifiers.

2.3 Porting the classifier to C++ (Arduino IDE)

At the end of the training, we ported the trained classifiers to C++ with a single line of Python code. As stated earlier, the *Embedded Window* library has two main functions: (1) it reframes our Python data to windows, and (2) extracts its features to train a ML model. It does exactly the same in C++ during the testing phase.

2.4 Results

We tested the trained classifiers on Arduino IDE to see their ability to recognize the gestures in real-time. Fig. 4 shows the printed gestures during real-time testing on Arduino. On the other hand, Fig. 5 shows the accuracy of the classifiers for each of the predicted gestures.

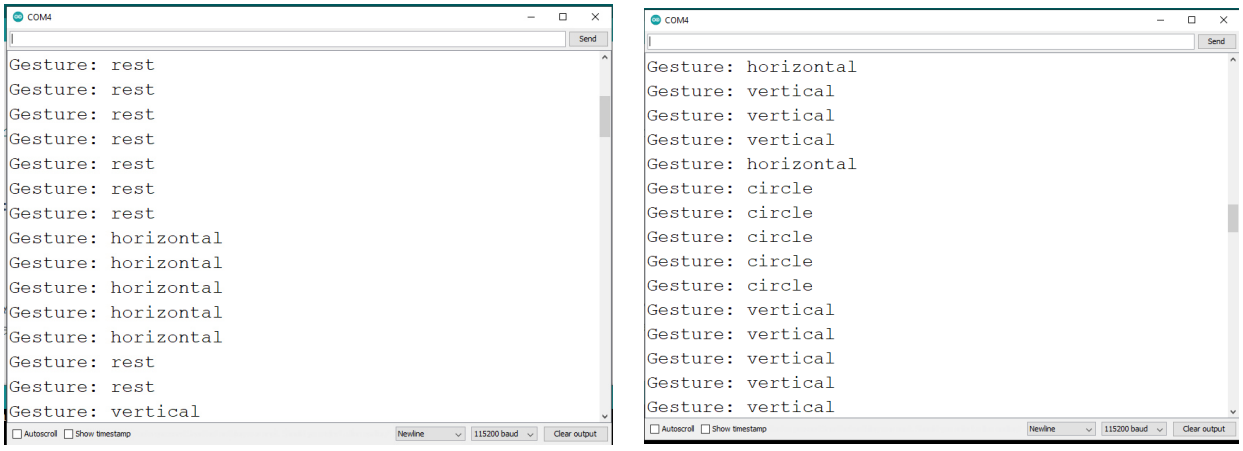

Fig. 4. Predicted gestures in Arduino IDE serial monitor.

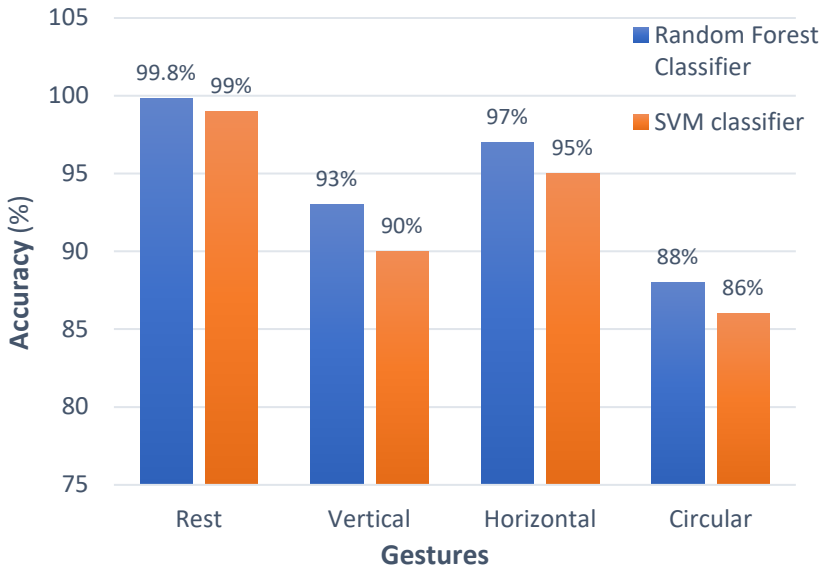

Fig. 5. Accuracy of the Random Forest classifier vs. SVM classifier for each predicted gesture

Accuracy metrics:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Sensitivity = \frac{TP}{TP + FN}$$

$$Specificity = \frac{TP}{TP + FP}$$

$$FPR = \frac{FP}{FP + TN}$$

$$F1Score = 2 \times \frac{sensitivity \times precision}{sensitivity + precision}$$

where TP = True Positive, TN = True Negative, FP = False Positive, FN = False Negative.

From Fig. 5, we can see that the Random Forest classifier has a higher accuracy for each predicted gesture compared to the SVM classifier. The highest accuracy of 99.8% was achieved with the Random Forest classifier which occurred at rest gesture. On the other hand, the highest accuracy of 99% was achieved with the SVM classifier at rest gesture. Lowest accuracy occurred at circular motion gesture with 88% and 86% for the Random Forest and SVM classifier respectively.

The project has been able to achieve its aim for the first part. The second part of the project (the anomaly detection) will be discussed in section 3.

3. Anomaly Detection

3.1 Description

In this project, we used the Arduino Mega board and the sound sensor for capturing the data that we used to train the classifiers for anomaly detection.

The microphone sensor was then connected to the Arduino board and plugged into a PC where Arduino IDE runs in order to start audio recording of the raw data. The background sound was first recorded and we set a threshold between the background sound and the actual captured sound data. A threshold of 3 was set (refer to the attached Arduino codes) in order for our sensor to differentiate between noise and actual data. A total of 15 numerical samples were recorded for every sound detected and this was collected continuously for a total time of 30 seconds.

3.2 Experimental setup

Fig. 6 shows the connection diagram of the sound sensor and a typical Arduino board (same connection applies in any board, be it Mega, Uno, etc.). Fig. 7 shows the overall setup of the embedded system.

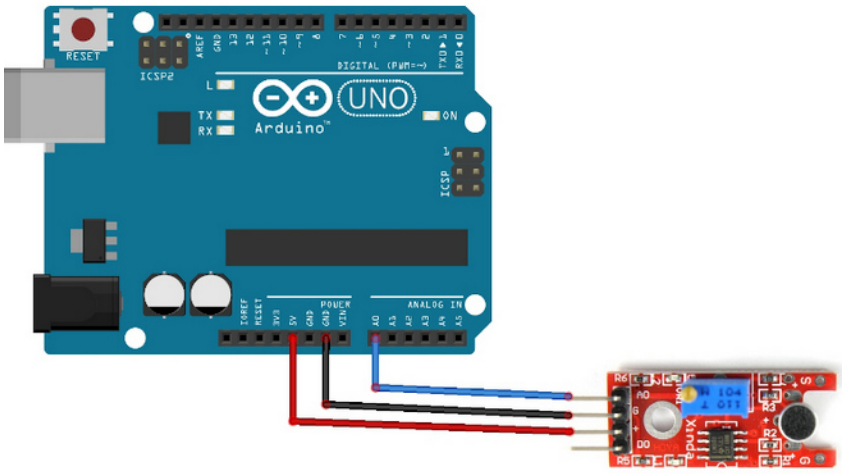

Fig. 6. Board setup (source: http://arduinolearning.com/code/ky038-microphone-module-and-arduino-example.php).

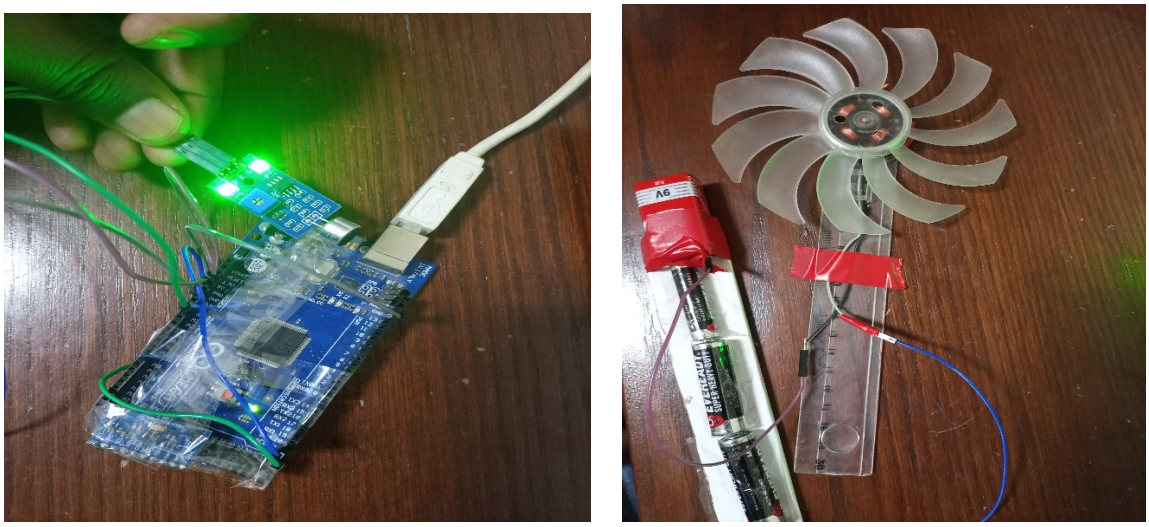

Fig 7. (a) Setup and test components. (b) Fan setup as data generator.

We used a fan that was obtained from a damaged small cooling system as the source of data generation; remember we intended to classify two types of sounds (namely, normal and abnormal fan conditions). The Fan requires 12 V supply to properly work, hence we used 3 batteries having the voltages of 9 V, 1.5 V, and 1.5 V, respectively, making a total of 12V.

After all the components be in place as shown in Fig. 8, we started the fan by connecting the batteries at the same time. Soon we started recording the data for 'normal fan condition' for 30 seconds with 15 samples generated. Thereafter, we put a piece of paper into the running fan making it to change the sound condition of the fan, and the generated data was saved as 'abnormal fan condition'.

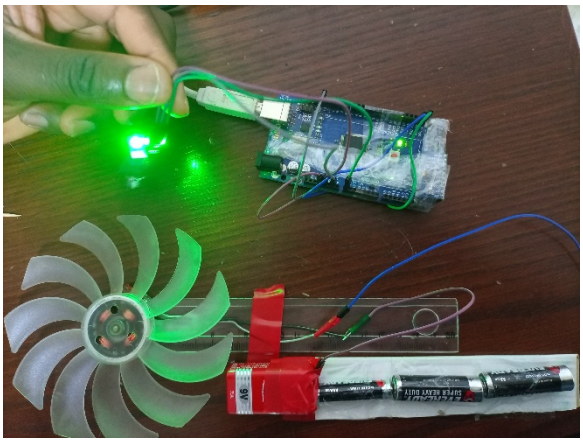

Fig. 8. Complete setup ready for data capturing.

3.3 Training process

The captured data were saved as '.csv' format, put in a folder, and trained using the following steps:

- **Load the data**

The obtained data was loaded into Jupyter Notebook, ready for training, with help of below code you can load easily:

```
import numpy as np
from glob import glob
from os.path import basename


def load_features(folder):
    dataset = None
    classmap = {}
    for class_idx, filename in enumerate(glob('%s/*.csv' % folder)):
        class_name = basename(filename)[:-4]
        classmap[class_idx] = class_name
        samples = np.loadtxt(filename, dtype=float, delimiter=',')
        labels = np.ones((len(samples), 1)) * class_idx
        samples = np.hstack((samples, labels))
        dataset = samples if dataset is None else np.vstack((dataset, samples))
    return dataset, classmap
```

- **Train the classifier**

Micromlgen supports different types of machine learning algorithms such as decision tree, SVM, Random Forest, XGBoost and others. But in this project, we used the Random Forest and SVM classifiers to train our data. We used the below code for the classifiers:

1). from sklearn.ensemble import RandomForestClassifier

```
def get_classifier(features):
    X, y = features[:, :-1], features[:, -1]
    return RandomForestClassifier(20, max_depth=10).fit(X, y)
```

2). from sklearn.svm import SVC

```
clf = SVC(kernel='poly', degree=2, gamma=0.1, C=100)
clf.fit(X_train, y_train)
print('Accuracy', clf.score(X_test, y_test))
print('Exported classifier to plain C')
print(port(clf, classmap=classmap))
```

- **Export to plain C**

After the Python code finishes running, it will generate an output in C code, that should be saved as 'model.h', and this saved C code will be imported into the Arduino IDE for testing of the model in the Arduino board.

Next, we run the prediction of the anomaly classification with the sound sensor in the Arduino Mega board by capturing sound data in real-time. Finally, we open the serial monitor to see the outputs.

3.4 Results

Fig. 9 shows a plot of the data during the testing phase while Fig. 10 shows the predicted fan conditions in real-time testing. On the other hand, Fig. 11 (a) shows the accuracy of the training phase and Fig. 11(b) shows the accuracy of the testing phase. As can be seen, the accuracy is higher for the Random Forest (RF) classifier in both training and testing phases compared to the SVM classifier. For the training phase, the highest accuracy attained was 75% for the RF classifier whereas the SVM achieved 65% (for each fan condition). However, the accuracy dropped a little during testing (prediction) phase and the classifier was able to easily detect abnormal fan conditions more correctly than normal operating condition. Maximum prediction accuracy was 70% for the RF classifier and 65% for the SVM and this occurred at the abnormal fan condition. Minimum prediction accuracy was 65% and 60% for the RF and SVM classifiers respectively and this occurred at the normal fan condition.

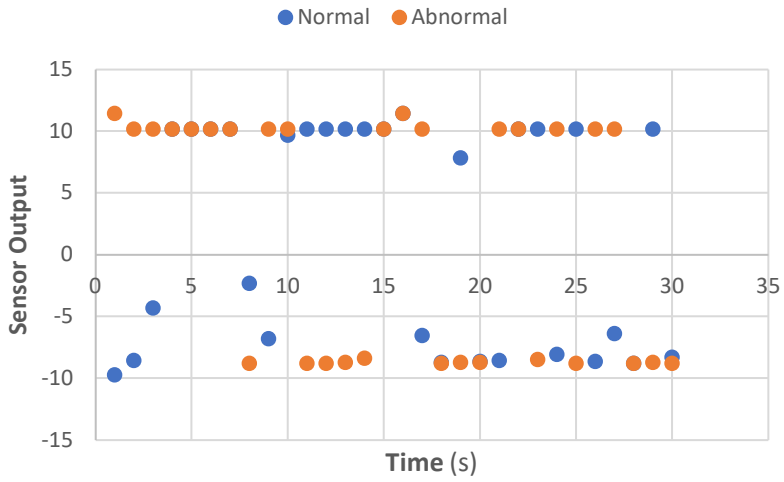

Fig. 9. Plots of test data.

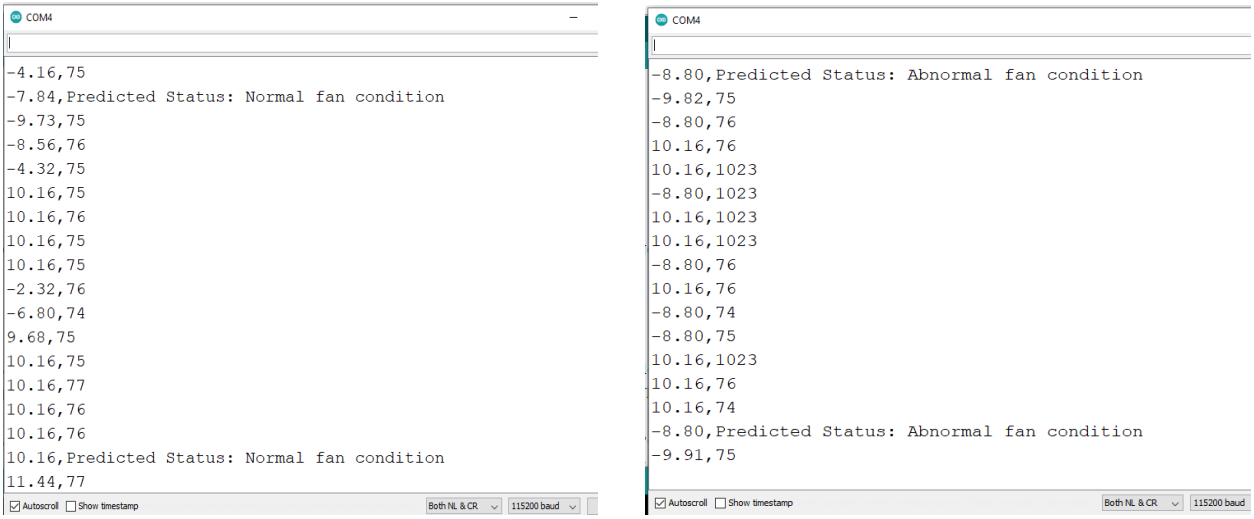

Fig. 10. Predicted fan conditions in Arduino IDE serial monitor.

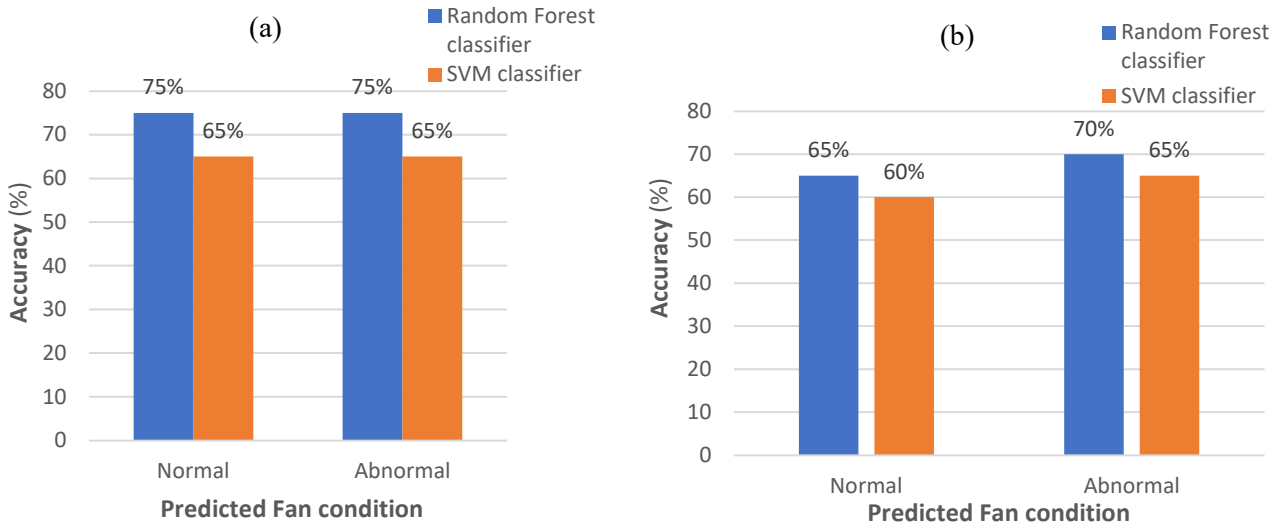

Fig. 11. Accuracies: (a) Training accuracy, (b) Test accuracy.

4. Conclusion

This report has been able to demonstrate the use of machine learning algorithms for both gesture recognition and anomaly detection tasks in an embedded system. We demonstrated this using the SVM and RF classifiers with Python and C++ programming languages. Overall, we achieved higher accuracy for the RF classifier compared to the SVM classifier in both training and testing phases. The results presented are potential in developing real-life practical recognition techniques for embedded platforms.

References

Cohen, I. (2022). What is Anomaly Detection? Examining the Essentials. Retrieved from https://www.anodot.com/blog/what-is-anomaly-detection/

Simone. (2021). How to deploy an Arduino Machine learning classifier in 4 easy steps. Retrieved from https://eloquentarduino.github.io/2019/11/how-to-train-a-classifier-in-scikit-learn/

Virtusa Corp. (2022). Gesture recognition. Retrieved from https://www.virtusa.com/digital-themes/gesture-recognition