Fall Detection

Michael Yang, Shivay Wadhawan, Ali Naveed

1. Introduction

Our project is focused on fall detection using a wearable sensor. With a user wearing a sensor on a specific location (e.g. the chest), we attempt to detect when they take a sudden fall. Fall detection seemed like a natural extension to the IMU gesture data collection done in class. We were familiar with machine learning algorithms, so we can use collected motion data from falls to create a model to determine whether or not a specific motion was a fall. Fall detection is an important topic with regard to human safety- if someone is alone and has nobody to assist them when they take a fall, an automated system that detects it could call emergency services to the scene. Many populations may benefit from such a system, like the elderly or solo travelers. Methods used for fall detection may also prove useful for discerning other movements, such as crash detection or sudden stops, which can be applied in many other fields.

2. Project Overview

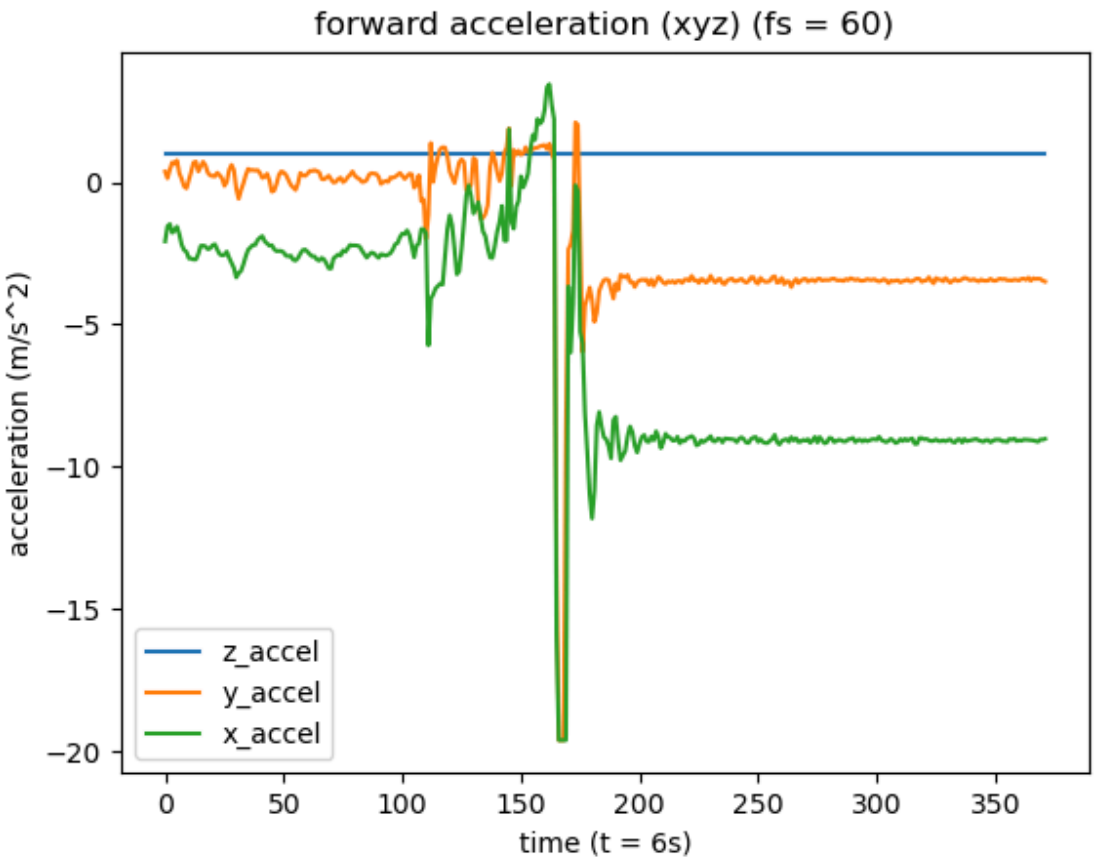
Our system consists of an ESP32-S3 microcontroller that reads from an MPU6050 IMU. The ESP32 is powered by a small portable battery, which helps eliminate some of the constraints of a wired power supply, making the device more mobile and sampling falls easier. The MPU6050 IMU contains an accelerometer and gyroscope, the measurements for which are communicated to the microcontroller via I2C, processed, and then sent wirelessly via WiFi to our computers to be output as data files. Four fall directions are recorded- forward, backward, left and right. We feed these data samples into a machine learning model to train it to predict when certain movements are falls or not.

3. Project Implementation

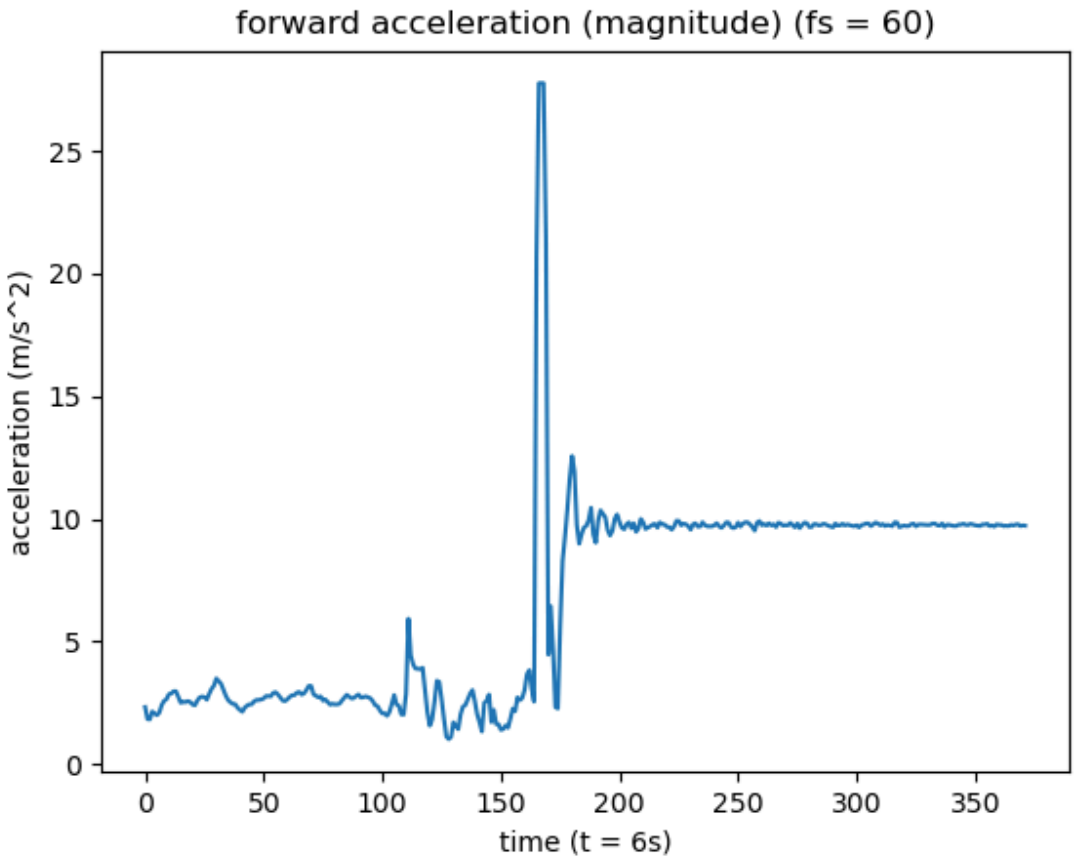
Data collection is done with an ESP32-S3 and MPU6050. The MPU6050 sensor is communicated with from the microcontroller using I2C. A Python program then collects the accelerometer and gyroscope readings, the measurements consist of readings along 3 axes for both the accelerometer and gyroscope.

Falls are collected by attaching the sensors to a stick, then letting it fall in a given direction. Each fall sample is around 6 seconds, and 20 falls were taken in each direction. Each movement (left, right, forward, backwards) has an associated label, 1 for a forwards fall, 2 for a backward fall, 3 for a left fall and 4 for a right fall. These readings along with their label are sent to supabase and stored in a relational database table with columns id, aX, aY, aZ, gX, gY and gZ, representing the type of fall and the reading

along each axis for that data point.



An example fall is shown above. There is a clear spike when the fall takes place- which axis the spike is in varies based upon the direction. Another way of interpreting this is by taking the magnitude of the acceleration: $\sqrt{x^2 + y^2 + z^2}$



This provides a more unified/consistent graph between the directions, while still maintaining the characteristic spike in acceleration seen by a fall.

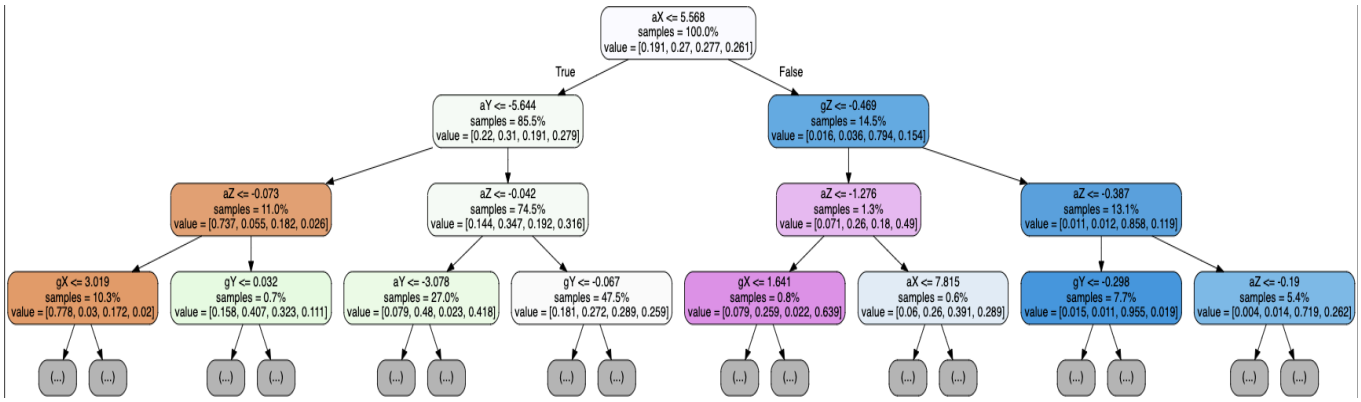
The fall data is then communicated to Supabase over WiFi and extracted for analysis. An abridged version of the database looks like the following table:

id numeric	t(s) numeric	aX numeric	aY numeric	aZ numeric	gX numeric	gY numeric	gZ numeric
1	5.87988	-3.35428	-9.05726	-2.67911	-0.0452986	0.0221164	-0.0245145
1	5.89697	-3.46201	-9.06684	-2.72221	-0.0423675	0.021317	-0.0231822
1	5.91284	-3.37582	-9.07642	-2.61207	-0.0438331	0.0190521	-0.0145222
1	5.92896	-3.39977	-9.10036	-2.66714	-0.0442328	0.0209173	-0.00932619
1	5.94482	-3.39019	-9.12191	-2.65038	-0.0463645	0.0153216	0.00319755
1	5.96094	-3.39737	-9.0429	-2.65038	-0.0431669	0.0141225	0.00586217
1	5.97681	-3.45962	-9.04768	-2.56419	-0.046098	0.0214502	-0.0342404
1	5.99292	-3.49074	-9.02135	-2.71502	-0.0463645	0.0227825	-0.0341072
2	0.000976563	0.337582	-2.00155	8.52096	-0.0776738	0.00519602	0.0237152
2	0.0168457	0.653617	-2.16196	8.43716	-0.0728775	0.0218499	-0.0042634
2	0.032959	0.562637	-2.16675	8.49702	-0.0706125	0.0367718	-0.035706

The final dataset consisted of over 50,000 data points which were used to train the machine learning model. The models were developed in Python using machine learning libraries, notably, SKLearn. The motivation behind using a pre-built library instead of developing our own custom models was that the pre-built models were more robust and would likely result in better performance and stability.

Initially, a KNearestNeighbor classification model was deployed. In this model, training examples are points or vectors in a multidimensional space, and then, test data points are classified based on the classification of the K nearest neighbours, a common metric to find the nearest neighbours is Euclidean distance. As stated before, the data set was split into testing and training data with an 80:20 split. A pipeline was used to streamline the pre-processing steps. The most important pre-processing step in this case was scaling the data to be on a consistent scale, which was achieved using the SKLearn standard scaler. Then, an SKLearn component known as GridSearchCV was used to find the best hyperparameters, in this case, the choice was between the number of neighbours.

Later, a Random Forest Classifier was used. Random Forest combines the output of multiple decision trees to reach a single result. A decision tree is a tree where each node represents a test on an attribute and each branch represents the outcome of a test, each leaf node then represents the label of that class. Below is a sample decision tree, however, the depth is restricted to 3 since these can get very large.



The color of each node represents the majority label of that node, and the depth of that color represents how prevalent that class is in the node. For example, in the left-most leaf, label 1 has a value of 0.778, which is why orange is quite dark. Similarly, the left second from the right has a value of 0.955 for label 3 which is why it is very blue.

Again, the hyper-parameters were optimised. This time, RandomizedSearchCV was used instead of GridSearchCV, since there are a lot more parameters and doing an exhaustive search for the best ones would take a considerable amount of time. The difference is that randomizedSearchCV samples hyperparameters randomly in some number of iterations, returning the best ones. The parameters that we are tuning over here are n_estimators and max_depth, both of which apply to the decision trees that the forest will be formed from.

There were many challenges in getting the entire setup to work. At first, learning to flash the chip and finding what packages to modify was difficult due to the many variations in the ESP board, as well as inconsistent documentation. Finding the correct registers and determining their units to convert to was also challenging. For data collection, there were difficulties at first trying to write data files directly onto

the ESP32 or connected computer through a USB connection as there is only one serial communication port. At first we attempted bluetooth communication, but ran into issues with setup and found that WiFi was easier. The hardware also presented some challenges, both components used (MPU6050 and ESP32-S3) did not have a protective housing, which meant that dropping the hardware to collect fall data was a risky procedure. The programming interface for the hardware was also entirely new, which meant that there were always aspects of the programming that seemed unfamiliar and prone to error. Another challenge was selecting the most appropriate machine-learning model for the classification problem at hand. Each model had its own set of parameters that needed to be read about and tuned before they could be implemented.

4. Evaluation & Demonstration

The KNearestNeighbor classifier achieved a mean test score of 85.64% in the best case. This means that this model could predict the type of fall with a reasonable accuracy. Below are the 8 best runs for this model.

split0_test_score	split1_test_score	mean_test_score	std_test_score	rank_test_score
0.858924	0.854015	0.856470	0.002454	1
0.826870	0.821323	0.824097	0.002773	2
0.825594	0.817151	0.821372	0.004221	3
0.820636	0.813960	0.817298	0.003338	4
0.816709	0.813175	0.814942	0.001767	5
0.811015	0.805763	0.808389	0.002626	6
0.809641	0.801983	0.805812	0.003829	7
0.807039	0.798400	0.802719	0.004320	8

The random forest model on the other hand managed a best accuracy of 90.29%. This accuracy is based on the SKLearn accuracy score metric, which computes the percentage of labels which were predicted correctly by the model.

Sample a fall, transfer the file, then feed it into model for classification

Repository link:
https://github.com/michaelyg18/528-Project

Short demonstration of an example fall:
https://youtube.com/shorts/D9naR7ADbFE

We would like to add the following demonstration videos/photo to the course website.

5. Task Assignment and Contribution

Michael:

- Group organization
- Project proposal and final report writeup
- Presentation slide creation
- Fall data analysis and graphing
- Fall data metrics

Shivay:

- Data collection
- Hardware setup
- Database population
- Data labeling

Ali:

- ML Model training, evaluation
- Final Presentation, final report writeup
- ML Model Development