fingers have touch sensors connected to 5V while the middle, ring and pinky fingers have touch sensors connected to a resistor and ground. The Arduino Nano reads the voltage at the middle, ring, and pinky finger touch sensor node with its digital pins. When the thumb or index fingers touches one of the touch sensors on the middle, ring and pinky finger, the Arduino will detect a 1 from the touch sensor.

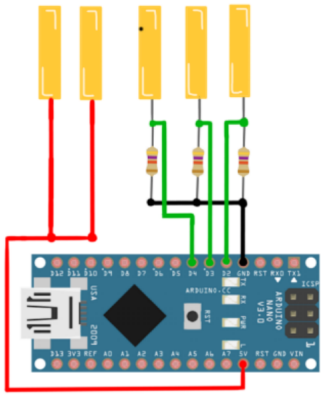

Figure 7: Touch Sensors and Arduino Nano Connections

The circuit for connecting the IMU to the Arduino Nano for I2C communication is shown in Figure 8. Although the Adafruit TDK InvenSense ICM-20948 9-DoF IMU came with a QWiiC connection, we plan to communicate with it through I2C using its SCL and SDA ports in our PCB to reduce the amount of wires on the devices.

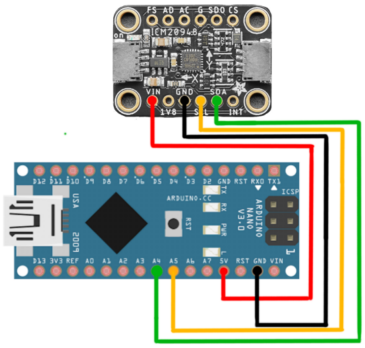

Figure 8: InvenSense ICM-20948 9-DoF IMU and Arduino Nano Connections

5.3 Subsystem B - Computer

In terms of the Computer subsystem, we start off by reading the data that's passed to the Arduino Nano using Python. As the Arduino Nano will be connected to a computer using its serial port, we use pySerial, a Python module for accessing serial port, to read data from the Arduino Nano. This module was chosen because it is frequently updated library with ample documentation and testing.

While the sensor data has been pre-processed on the Arduino to be more human readable, such as converting values from flex sensors to actual bending angles, further normalization and scaling will be done to ensure that the models can be trained correctly. This is a crucial step as certain models can have different results if data is not properly processed. This may also help in speeding up the calculations for model. The scaled data will then be passed through the classification model, which will output the most likely letter based on the given data.

After having tested our classification problem on five different machine learning models: SVM, KNN, perceptron, random forest, and neural network. After several iterations, we found that the random forest classifier consistently performs with the highest accuracy both in real-time testing as well as testing on a testing dataset.

We believe that perceptron did not perform well because it is simply linear regression and our data is not linearly separable in many instances. Random forest uses subset of data to train decision trees and takes the majority vote. The reason it works so well on our data may be because it uses subsets of data which can filter out the less relevant data in training and that decision trees tend to use the most important features to determine an outcome.

As for the other models that we considered, neural network, a widely used machine learning model, is not working as well as we expected. We deduct this is because our data does not contain many features since they are all 17 by 1 vectors. If we were to capture data over a window of time, neural network's performance may improve. As for KNN, k-nearest neighbor, this model finds the nearest neighbors of a data point and takes the majority value. This model has an accuracy of 82% on our first set of real data, which is actually quite good. However, as there are a few similar ASL letters and we are collecting more data in the future, this accuracy may decrease and its time on testing will increase, which will not be ideal if we want to meet our requirement for latency, so we ruled out this model early on in our project timeline. SVM, support vector machine, is a linear model, but it is capable of performing non-linear classifications through data transformation. This is the model that performs the best after the random forest classifier. From our previous assumptions of nonlinear data, this may be the reason why it is doing well.

Our system is able to make around 20 classifications per second, so we require our model to make 8 of the same classifications in a row before we output the audio. This helps avoid incorrect intermediate/transition predictions. We pre-generated audio files of the 26 letters using the