**Ebisu: An Innovative User-Input Solution**

By Brittany Barnes and Yusef Ghouth

COMP 495

Dr. Sat Garcia

Advisor: Dr. Eric Jiang

5/22/17

**1. Abstract**

Ebisu is a Machine Learning Algorithm that aims to recognize and categorize wearable device data, such as accelerometer data, into gestures. These gestures open up a new avenue in user-machine interaction, allowing customers to have a form-fitting experience with their technology.

**2. Background**

Ebisu seeks to function as a User-Input solution. Replacing or augmenting the experience of using a traditional keyboard and mouse, or the cutting edge of voice command recognition. With either of those two experiences there are significant advantages of using Ebisu instead of those systems and significant synergies that occur when Ebisu is used with those systems. In General, an input system that is based on a Machine Learning Algorithm has several advantages over ones that are not. Ebisu can mold itself to it’s user over time by picking up on the user’s unique gestures and incorporating them into a command scheme. The expandable library of inputs made possible with Ebisu opens up the possibility for extremely specialized version of Ebisu. Versions that are trained with specialized data like medical jargon or programming terms yield an experience that is different to and arguably better than than catch-all input systems, such as mouse/keyboard set ups.

**3. Implementation Details**

***3.1 The Glove***

To classify the gestures, the prototype must be created to find the data into the machine learning algorithm. The glove takes in two different dimensions of data, the first being the degrees in which your fingers are bent, and the movement of your hand in a 3-D space. To read the degree of bend from each finger, you will need to download code that prints out the data in a readable format. A snippet of the code for each finger is as follows:

int tADC = analogRead(thumb);

float tV = tADC \* VCC / 1023.0;

float tR = R\_DIV \* (VCC / tV - 1.0);

float tAngle = map(tR, tS, tB, 0, 90.0);

This is an example of what the code for each finger looks like where we begin by reading in the data at that particular pin for the thumb. Calculating its voltage where VCC is a predefined variable at the top of the code, then determining the resistance of the flex sensor. Finally, we map these values to a range between 0 and 90 degrees which will be used for the print statement at the end of the code. These lines of code are done for each finger or the number of flex sensors for your specific implementation. Next, we will need code to determine the data from the accelerometer and given a 3-D output of the positioning of our hand in space. Other than setting up the accelerometer, and ensuring that there is data coming from the accelerometer for the arduino to read, the code boils down to these three main lines:

mpu.dmpGetQuaternion(&q, fifoBuffer);

mpu.dmpGetGravity(&gravity, &q);

mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

Here we are getting the quaternion, which is a four-dimensional vector space, the gravity vector, and then finally the coordinates of the hand along with the gravity vector. The “ypr” values are multiplied by 180/pi and then returned to give the coordinates of the hand.

***3.2 Machine Learning Algorithm I/O***

The Algorithm takes in the training file name and opens it using buffered reader. An Int is parsed out of the first line of the file and stored as a variable (n) representing the number of lines in the file. Then a loop executes that number of times that does the following in pseudocode

n = Integer.parseInt(buffer.readLine())

x = number of inputs

for (n times){

new DataPoint p;

String temp = buffer.readLine()

for (i for x times){

p[i] = Float.parseFloat(temp.split(“,”)[i])

}

trainingData.add(p)

}

***3.3 Perceptron***

The initial algorithm used a model based for a multilayer perceptron that was implemented to act single layer. This resulted in a working Supervised-Learning Algorithm but was abandoned to move Ebisu into an Unsupervised model. The Single layer was initialized with a neuron for each class, each with a number of inputs (“Previous Layer Size”) equal to the number taken in through numInputs(5 for the initial prototype, 8 for the final one). This single layer classified data by feeding it through the weight matricies of each neuro. The neurons output a number for each class,

***3.4 DBSCAN***

My Implementation of DBSCAN was based on the following Pseudocode:

DBSCAN(D, eps, MinPts) {  
 C = 0  
 **for each** point P in dataset D {  
 **if** P is visited  
 **continue** next point  
 mark P as visited  
 NeighborPts = regionQuery(P, eps)  
 **if** sizeof(NeighborPts) < MinPts  
 mark P as NOISE  
 **else** {  
 C = next cluster  
 expandCluster(P, NeighborPts, C, eps, MinPts)  
 }  
 }  
}  
  
expandCluster(P, NeighborPts, C, eps, MinPts) {  
 add P to cluster C  
 **for each** point P' in NeighborPts {   
 **if** P' is not visited {  
 mark P' as visited  
 NeighborPts' = regionQuery(P', eps)  
 **if** sizeof(NeighborPts') >= MinPts  
 NeighborPts = NeighborPts joined with NeighborPts'  
 }  
 **if** P' is not yet member of any cluster  
 add P' to cluster C  
 }  
}  
  
regionQuery(P, eps)  
 **return** all points within P's eps-neighborhood (including P)

**4. Future Work**

***4.1 Software Integration***

In the future we plan on integrating ebisu into a platform that will translate the gestures into commands that allow you to control various apps and functions on your devices. We may even implement an OS that is fully integrated with Ebisu Input.

***4.2 Machine Learning Algorithm***

We plan on upgrading DBSCAN with a sub-space clustering algorithm called SUBCLU. This will greatly increase the ability of the algorithm to deal with higher dimensional data, allowing for more gestures to be trained. If enough data can be collected and analyze, we might attempt to advance the algorithm to read into human body language and make assumptions about their needs at that moment.

***4.3 Hardware Prototype***

We plan on switching the prototype from a glove with elements hard wired into it to a modular design composed of rings or bracelets that communicate their relative positions to one another and to a central computing hub, which would contain the algorithm, and then transmit the output to a computer or phone wirelessly.

**5. Conclusion**

Upon completion of the project we could implement the gathering and classification of gesturing data which was the main feature we originally wanted to accomplish. Before beginning the project, we had the idea to solely use accelerometer data to determine the gestures being made, but as we began designing the prototype, it became necessary to include the use of flex sensors. By adding the flex sensors, we could include another dimension of data that increased the accuracy of the gestures being made. Since gestures can include the positioning of each finger, the flex sensors were able to help classify two similar gestures that occurred in the same 3-D space because the fingers were bent in a different manner. In terms of the software component, our idea was to use a multiclass perceptron to classify the gestures, but as we moved into unsupervised learning, we saw that clustering would fair better than the perceptron. This was crucial as we could now feed the DBSCAN gestures, and their names, and it would be able to group those gestures together without explicitly training it on these gestures.

**6. References**

1. DBSCAN Algorithm : Ester, Martin; [Kriegel, Hans-Peter](https://en.wikipedia.org/wiki/Hans-Peter_Kriegel); Sander, Jörg; Xu, Xiaowei (1996). Simoudis, Evangelos; Han, Jiawei; Fayyad, Usama M., eds. *A density-based algorithm for discovering clusters in large spatial databases with noise*. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96). [AAAI Press](https://en.wikipedia.org/wiki/AAAI_Press). pp. 226–231. [CiteSeerX](https://en.wikipedia.org/wiki/CiteSeerX) [10.1.1.121.9220](https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.9220) Freely accessible. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [1-57735-004-9](https://en.wikipedia.org/wiki/Special:BookSources/1-57735-004-9)
2. JIMB0. "Flex Sensor Hookup Guide - Learn.Sparkfun.Com". *Learn.sparkfun.com*. N.p., 2017. Web. 23 May 2017.
3. Rupasinghe, Kinath. "MPU6050 (GY-521 Breakout) + Arduino Mega 2560 Accelerometer And Gyroscope Application". *Dummyscodes.blogspot.com*. N.p., 2017. Web. 23 May 2017.

**7. User Manual**

***7.1 The Glove Prototype***

*Hardware*:

* 5 x Flex sensors
* 1 x MPU-6050 Accelerometer and Gyroscope
* Jumper wires,
* A Glove
* 1 x Arduino Mega,
* 1 x mini breadboard,
* 5 x 22k resistors
* Velcro

*Setting up the glove*:

1. Measure the voltages of each flex sensor. Do this while the flex sensor is both straight and bent around 90 degrees. Write down these measurements as they will be needed in the code.
2. Attach Velcro to the fingers of the glove, and the other half to the back of the flex sensor. This will allow the flex sensor to move freely for accurate readings.
3. For each flex sensor hookup, connect a wire from one of the analog pins of the Arduino to the breadboard. Then connect a resistor to the same column of the wire coming from the analog pin. On the opposite end of the resistor, connect a wire from the GND pin to a space in the breadboard. Finally, connect a wire from the 5V port to the breadboard in a row next to the previously connected wires. Finally, connect a wire from the analog pin side to the left of the flex sensor, then connect a wire from the 5V side to the right of the flex sensor. See figure 1.1.

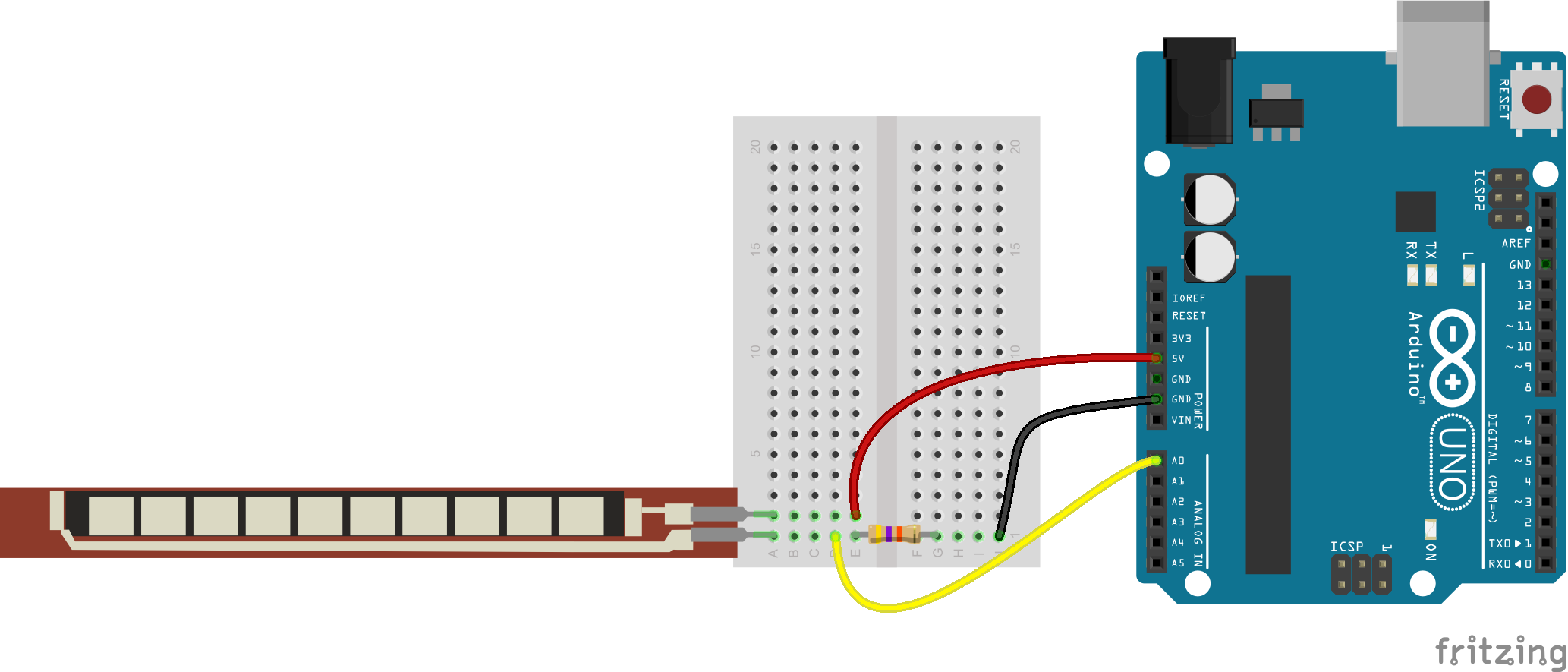


Figure 1.1 Connecting flex sensor to Arduino

1. To connect the accelerometer, you will need to first solder the ends of the pins to hold them in place for a secure connection.
2. Next, place small amount of Velcro to the back of the accelerometer and to a section on the glove, preferably the back center of the hand.
3. To wire the accelerometer you will first connect a wire to the section on the breadboard that holds the 5V charge and to the VCC section on the accelerometer. Then you can directly connect a wire from the GND on the USB port side of the Arduino Mega to the GND port on the accelerometer. Next, directly connect the SCL and SDA sections of the accelerometer to their respective pins on the Arduino. Finally, connect the INT section to the number 2 digital port on the Arduino. See Figure 1.2.

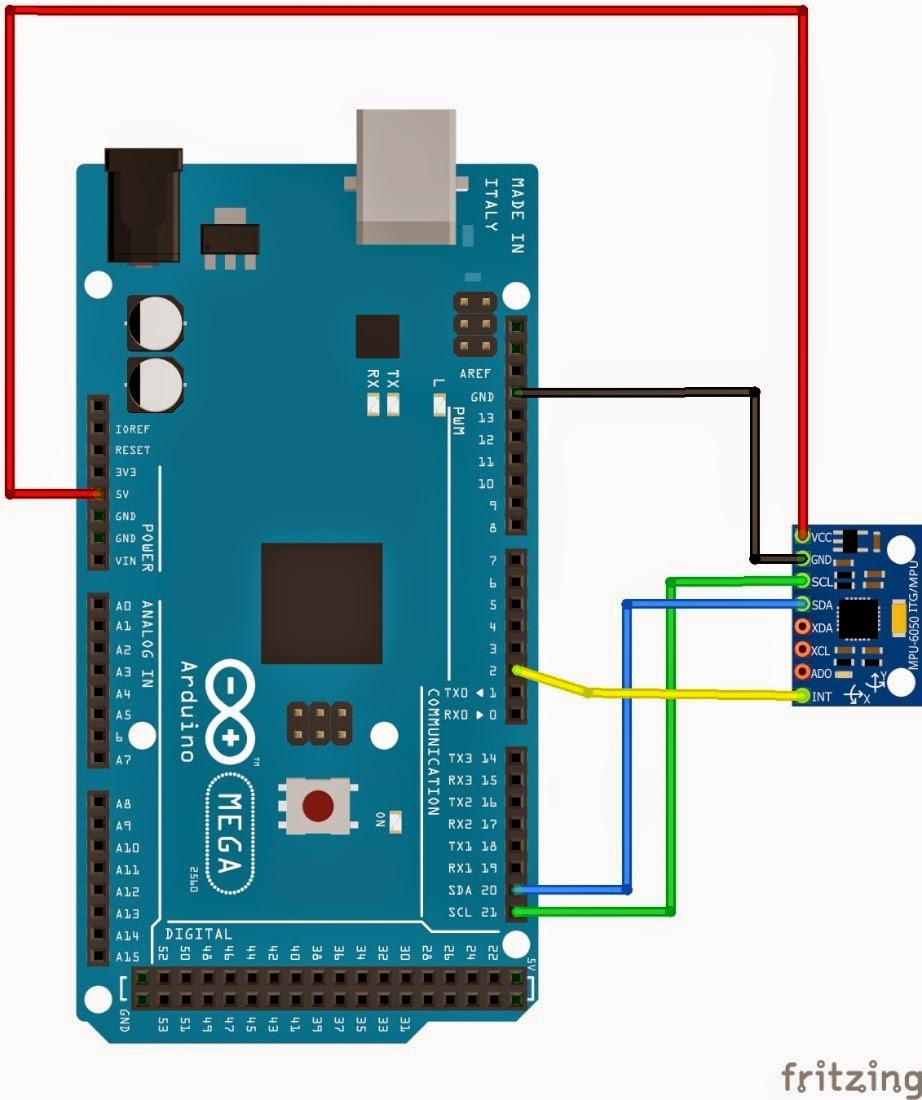


Figure 1.2 Connecting MPU-6050 to Arduino Mega

The code is found within the FlexNAccel folder to run both the accelerometer and flex sensors together, and individually in the FlexSensor and Accelerometer folders. I would suggest running solely the flex sensor code for each new finger connection to ensure the wiring is correct and outputs accurate values. This will be helpful finding bugs or misconnected wires. Then I would suggest running the MPU-6050 code to ensure the values are outputting correctly and making sure the wiring is correct. Once all six sensors are running separately, I would suggest running the code with everything connected to ensure the Arduino is able to output data from all six wires at the same time.

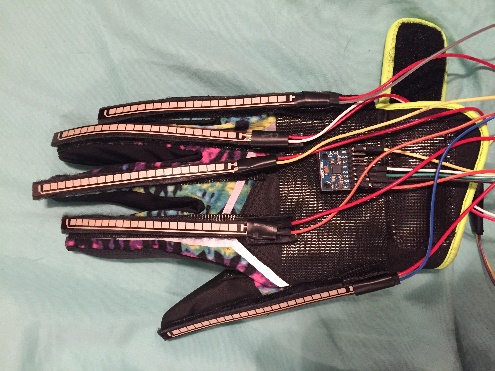


Figure 1.3. Glove with Accelerometer and Flex Sensors

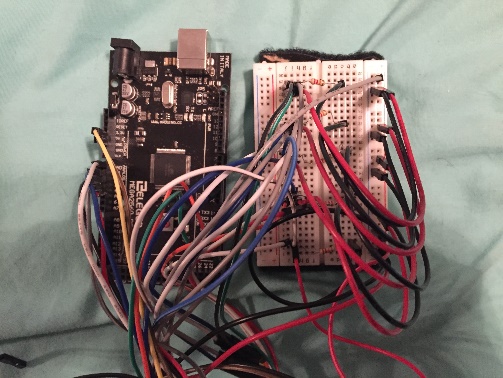


Figure 1.4. Arduino Mega and Breadboard Wiring

***7.2 Software***

Correct Parameters: Training Set (File), Testing Set (File), No. of Inputs(Int), No. of Starting Classes (int), Config (File)

The Training Set should be formated with the number of training samples at the beginning of the file followed by the data printed line by line, with commas separating the inputs.

Testing set and no. of Starting Classes are obsolete with the update to unsupervised learning

The config file should be a list of the expected classes in order of appearance line by line.

No. of Inputs is self explanatory.