Project Fishscale: Designing a Texture Classifier

Akhil Chandra Panchumarthi
Tangible Media Group (TMG), MIT

Advisor: Joao Wilbert
Professor Supervisor: Hiroshi Ishii

With the assumption of there being a sensor that can provide data of different textures, the goal of this project is to develop the software tools that can efficiently pipeline the sensor's data and use it to classify texture. The end goal was to have trained model, that, given input of an instance of sensor data to be classified, can accurately predict the required class.

1. Understanding the targeted problem

Project fishscale refers to, at a fundamental level having a device that could sense, encode and classify textures. Texture is a defining feature of the sensation of an object. The goal of the present implementation is, with the assumption that such a texture sensing device exists, one would like to develop software that can encode and use it for texture classification. The sensor collects data by scanning a surface. In practice, this is physical contact between the sensor and the surface whose texture we wish to sense, simply by swiping the sensor across the required surface.
For the sake of implementation, the sensor is understood as a square matrix where at each Aij entry we expect a piezoelectric unit. Each unit captures data as a time series.

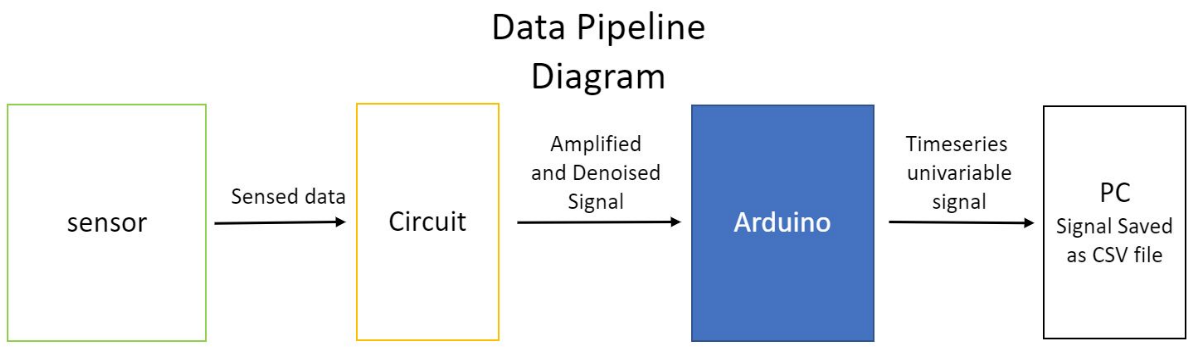
Collectively for each swipe, one would have i*j features of the swipe. The fluctuation in mechanical pressure on the device is a unique function of many variables, such as time of data sample, speed of swipe, texture that it was swiped against, etc. The idea is then to use this data to determine what class the texture belongs to at a very generic level. For simplicity, one assumes that all swipes are done within constant time. Also, we shall only work with one such feature, i.e. only one such piezoelectric unit.

2. Overview

The fundamental question we are trying to solve is, therefore, texture classification. To actually formally make an attempt at such classification, one has to develop a data pipeline that for a given sensor, could capture meaningful data to be eventually used to design a classifier.
We will start by discussing the data pipeline, followed by the classifier. We then explain the obtained results, and then conclude with a discussion on present limitations, possible fixes followed by how to take the project forward.

3. The data pipeline

The crucial part of this project revolves around the data pipeline. We will use this term to refer to the following structure.

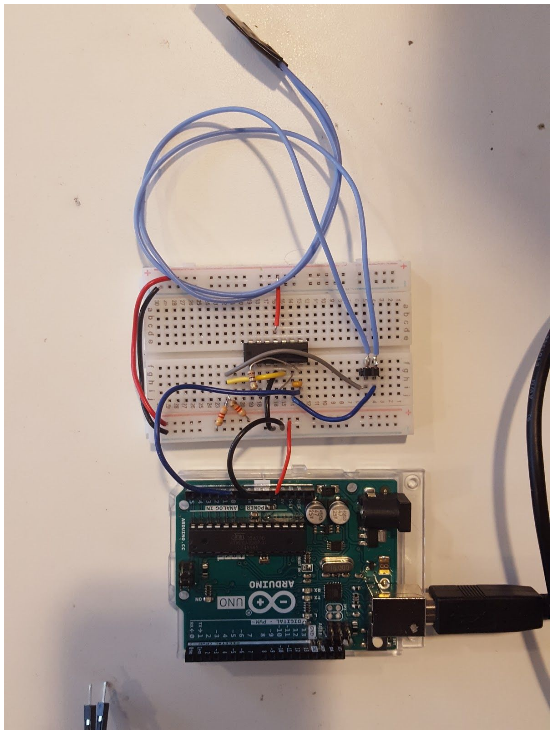


MEAS Piezo Vibration Sensor

The sensor that was used is a MEAS piezo vibration sensor. One important characteristic is that the type of charge depends on which direction one bends the sensor. Bending in on direction will generate positive charges and bending it in the opposite direction gives negative values. We measure the voltage rather than charge. We measure the output of this sensor in millivolts, and it is this data that we capture feed to the circuit. We see the MEAS piezo vibration sensor below.

Circuit

The circuit amplifies the signal that is received from the sensor and removes unwanted noise. The breadboard in the following image is the circuit. We can also see the sensor as well as the Arduino board.



Arduino Specifications:

The main part of the data pipeline lies in the Arduino and PC. The Arduino board's operating voltage is set as 5V. Analog Pin A0 is the input port of the Arduino Uno board, from which all outputs of the circuit are read. The values of the circuit, the input voltage, which now varies from 0 to 5V is then mapped to 0 to 1023(could explain this better). Any input that would be read from the circuit will now be scaled between 0 to 1023, which is the corresponding value that will be displayed. It is important to note, however, the units for this scaling is not mV. 1 unit on this scale is approximately 4.8875 mV.

Pyserial and PC Buffer:

Pyserial is used as the medium of communication between the Arduino and PC. When data is sent from the Arduino to PC, it is initially stored in the PC buffer. Any read operation in the python script is done from the PC buffer. The same is true for communication from the PC to arduino. The byte is read and flushed out. Note that this is like the Queue data structure, following FIFO architecture. The first byte sent from the Arduino would be the first to be read in the python script(provided a read operation is called). If the PC attempts to read without there being data in the buffer, one can expect problems. Another key fact is that the PC can read values at a much faster rate if it is reading directly from the PC buffer, rather than waiting for data to be transmitted from the Arduino. These two facts led to the failure of an initial attempt to design a communication method between the PC and the Arduino, which relied on asynchronicity.

Asynchronous Communication:

In the asynchronous method, the Arduino would constantly produce a stream of data points. It is up to the python script to decide when to read data. Because of oversight of how a buffer works, this method is fundamentally flawed. The buffer would be filled with values irrespective of whether data was being read by the PC or not. This meant that when the PC did start reading, it captures data in the buffer, when it really wants to capture data being written into the buffer at that exact moment. We now would end up with data that we don't want.

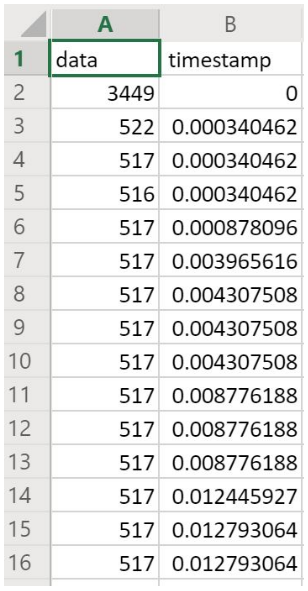
Handshake Protocol:

A more robust protocol was defined, where each byte was individually tracked to prevent miscommunication. In this protocol, communication between Arduino and the python script was enabled so that one can acknowledge reading a byte from the other.

This protocol employed an FSM in the arduino, a checker that makes sure that the PC knows which state the Arduino is in, and careful manipulation of data reading from the PC buffer was developed. Data is now only read when it was available in the PC buffer, and regular checks were done to make sure no unwanted bytes crept into the PC buffer. This fixed the issues presented by the asynchronous method.

Timestamps(Python Clock):

Any data that was read from the Arduino required a timestamp. Initial methods of generating the timestamp as the time it was received (read) by the PC proved to be flawed. While the time.time() function of the time library did have quite a remarkable precision, the fact that data was communicated asynchronously meant that the buffer would be full of values to be read. One should remember that the data rate of reading from the buffer is very large, and as it was seen in our implementation, faster than the rate the clock was updated. This resulted in data being read much faster than a significant change in the timestamp was seen, resulting in the following problem.

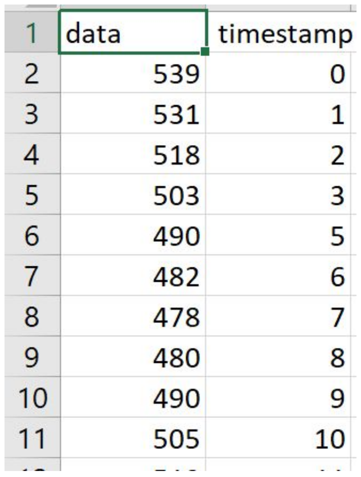


	A	B
1	data	timestamp
2	3449	0
3	522	0.000340462
4	517	0.000340462
5	516	0.000340462
6	517	0.000878096
7	517	0.003965616
8	517	0.004307508
9	517	0.004307508
10	517	0.004307508
11	517	0.008776188
12	517	0.008776188
13	517	0.008776188
14	517	0.012445927
15	517	0.012793064
16	517	0.012793064

Notice that while the read data value(we shall use units to define its unit, 1 unit = 4.8875 mV) does fluctuate, implying that the same byte was not being read repeatedly, the timestamp remains the same, even at such high precision! This confirmed the fact that reading bytes from the PC buffer reads data at a faster rate than the time.time() update.

Timestamps(Arduino Clock):

A solution would be having arduino sending the timestamp with the data. After all, a more logical conclusion would be the since the arduino was what was taking in data, it would have a more relevant timestamp value. The timestamps are generated using an arduino time library, millis(), which measured in milliseconds. As a matter of fact, it was also extended to micros() to give higher precision. Data now has different timestamp values for different readings, as it should be. The following figure shows the occurrence



1	data	timestamp
2	539	0
3	531	1
4	518	2
5	503	3
6	490	5
7	482	6
8	478	7
9	480	8
10	490	9
11	505	10

One should note in the Arduino device, each loop iteration also seems to vary in the time requires to run. Assume that each iteration is tracked by a counter value. An iteration with a smaller counter value seems to run quicker than a much later one. Larger counter valued iterations did seem to saturate at a constant iteration time though. This is one of Arduino's limitation, one that was carefully kept in mind while avoiding to capture the same timestamp for different readings, as discussed earlier. Because of this reason, to make sure the arduino is working properly, it is sufficient to sample only the first few sample timestamps.

Sampling Rate (ms):

The final problem lies in the sampling rate. This defines the resolution that we capture the data signal. The following is discussed with timestamps defined in milliseconds and the sample count being in a 5-second timeframe. Timestamps defined by milliseconds is upper bounded by 5000 samples in a 5-second timeframe. In a practical case, the actual value turns out to be slightly smaller, because of the delay generated by the arduino per sample. One should also factor in baud rates! In any data sampling mechanism, the sampling rate is depended on the delay between each sample and the transfer rate between the two communicating systems.   With a baud rate of 9600 bps (bits per second) and delay of 100 ms, the samples measured was 204 samples. A delay of 10ms gave 493 samples and a delay of 1ms gave 1219 samples (keeping other values constant). A smaller delay can't be specified, and a zero delay would give an inconsistent data measuring pattern, with time between samples being defined by arduino's looping speed. With a delay of 1 ms and a baud rate of 38400 bps and 57600 bps gave a reasonable number, 4279 or 4280 samples. One should note this as a saturation point, and this is the case. Higher baud rates did not improve the sample count. An important point to mention is that the data is sometimes of different lengths. This is expected, and not worrisome as the sample count doesn't vary a lot. A fix is discussed in the data processing section.

Sampling Rate(μs):

An important discussion is in the case of timestamps being defined with microseconds. The upper bound now turns out to be 5 million samples in a 5-second frame. Each loop iteration would require a larger number of bytes to be transmitted to the PC. One should see this as having a greater dependence on the baud rate. A baud rate of 9600 bps and a delay of 1ms gave 693 samples. In theory, increasing baud rates will greatly increase sample rates. However, since millisecond timestamps give a sample count comparable to the maximum possible (4279 vs 5000) this was not further pursued. Another reason is that about 5000 in a 5-second timeframe is relatively sufficient to capture time series data. This did prove to be the case.

Data Recording:

Data recording is a relatively straightforward task. The PC opens up the serial port and waits for an acknowledgment from the arduino. After that, using the spacebar (followed by and enter) as a trigger, it initiates the sampling process. The process lasts for 5 seconds, and data is saved as a comma separated values, in a text file. CSV format is 'data,timestamp' where each row is a data value. The naming scheme of the files is 'class1_sample_1.txt' implying it is the first sample of class 1. Note that sample here defines an identifier for the 4279 data points stored in the CSV named as defined previously. Each run of the python script records 10 files, and class number, starting sample number, and number of files to record per iteration can be modified.

Data Processing:

Data collected in raw form needs to be modified before it can be used by the KNN algorithm. The required data format is as an np.array of the following shape: [n,s] where n represents number of data points and s the sample size. Recall that sample size is with respect to a 5 second timestamp. Signal data when extracted directly from pandas dataframes is of the shape [s,1]. However, a very important key point is that s need not be of a standard count. Remember that s can be 4279 or 4280 in our implementation (baud 57600 bps and 1ms delay). One needs to reshape all CSV files so that they will all be of the same number of samples. Data now can be rearranged as an np.array of shape [n,s] without any other hindrance(by first generating a list of all CSV file values and reshaping it in np.array format).

4. The Classifier

KNN (K Nearest Neighbors):

To classify data, a simple implementation using KNN is used. KNN refers to the algorithm where for a query data point, one attempts to classify it based it's distances to all known points. We count the frequency of all the K data points that it is nearest to. K refers to the number of neighbors we use to make a decision on a particular point. K is essentially a hyperparameter that can be tuned based on different training datasets. An important thing to not is that KNN is essentially a lazy algorithm. No such 'training' of a model occurs until a query is made. When a query is made, that is when it does all of it's work. In the brute Force implementation, the query time is directly proportional to training days size. KNN's query time, in general, depends on the training data size.

The Distance Metric:

When considering one dimensional time series signals(we do capture both data and timestamps in a CSV sample file, however only data values matter) we need a metric to define how to rank certain signals over the other, in terms of their 'nearness' to all known values, the training dataset. Essentially we need to define a distance metric. In such cases, DTW proves to be invaluable. DTW refers to distance time warping. Unlike the euclidean distance between two signals, which requires that the two signals be of the same number of data points, DTW provides flexibility of having different length signals to compare with. The length of a signal is the number of samples in it. It should be noted that since signals are of the same length, this isn't too much of an advantage. DTW also provided flexibility in terms of translational invariance. Euclidean distance take the sum of the squares of the difference of the two signal’s ith samples. However, DTW generated and o(nm) matrix (n is length of first signal and m is length of second), where each sample in the first signal is compared with, using the l2 norm, all the points in the second signal. This is what brings out the advantage we wanted, in terms of translational invariance.

Time Complexity:

An immediate limitation is the time complexity. O(nm) is too much time for comparing two signals, especially when one compares the DTW of a query signal with all point in the training dataset(this refers to the brute Force implementation of KNN). However, intelligent algorithms like the ball-tree for KNN classification and the use of a faster approximation of dtw, implemented by the fastdtw library of python, make the time to run reasonable.

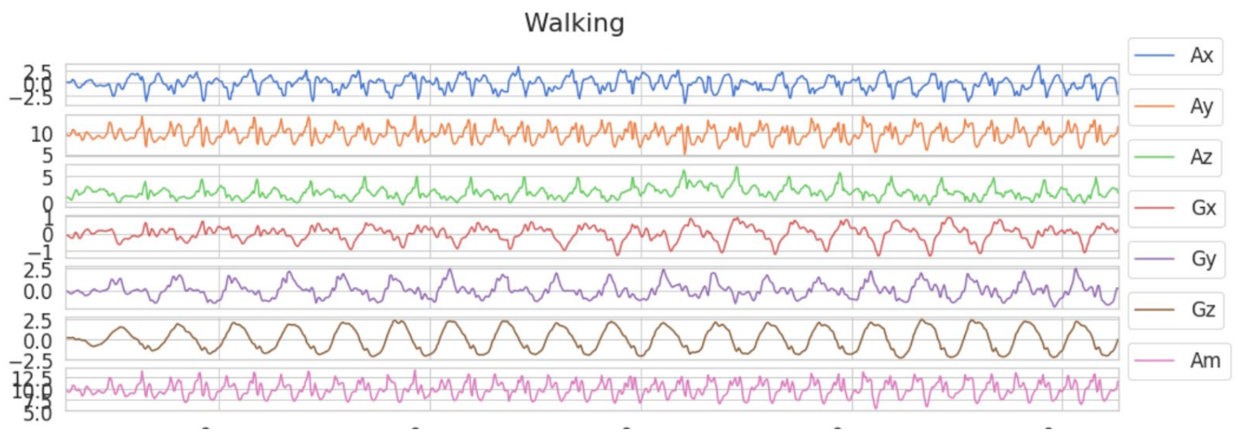
Classifier Expectation:

DTW and KNN is a working solution to our classification problem, provided one uses the approximations and assumptions stated above. DTW is a near perfect distance metric for two signals. KNN allows easy implementation and simple reasoning to classify a query signal.

5. Testing the Classifier

Testing with Dummy Data:

To first test this(DTW and KNN) combination's validity as a classification method, the algorithm was run on an accelerometer dataset available at:
https://github.com/wisdal/Deep-Learning-for-Sensor-based-Human-Activity-Recognition
The initial project was an attempt to figure out in which class a person was in based on his accelerometer values. The 6 classes are walking, running, standing, sitting, going upstairs and going downstairs. The dataset contained 4 complete data files, of which the first 3 was used to create the training data and the last to test the data. We focus on Am, a calculated parameter, that depends on A, Ay and Az as follows: Am^2 = Ax^2 + Ay^2 + Az^2. Am is the measure of the A vector. The graph of such Am values can be seen here, in pink. Note that this is taken from the github implementation as given in the above link. We see only here the case of walking.



The training dataset was broken up into 3000 sample parts which defined the size of a data point. This resulted in 159 data points, of size 3000. Each sample was from one class, i.e. each sample had the label of a class. The same was done to the query dataset, which gave 51 test cases. With a K value of 10, all 51 test cases were classified with an accuracy of about 60.78%. COnfining our study to only the first four classes gives an accuracy of 78.95%. The first list is the predicted classes of the ith testing datapoint, and the second list is of the actual classes.

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 1, 1, 1, 1, 1, 0, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 2, 2, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 5,
0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5,
5, 5]
```

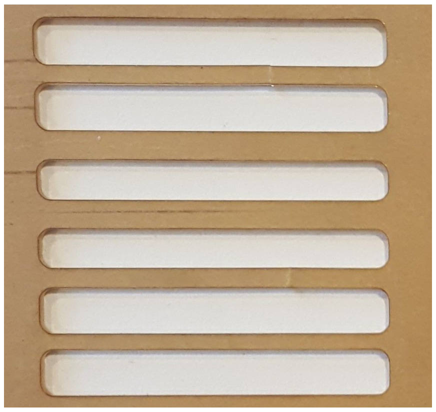
We can see the classifier's confidence in them with this array representation. The ith index represents how many data points of the 10 nearest neighbors were of class i. This is for the first 5 datapoints in the test data. Consider the first example. Of the 10 nearest neighbors, 8 of them are of walking data, and 2 of them are going upstairs data. Hence, the predicted class is printed below the confidence values, as 0, referring the index of walking.

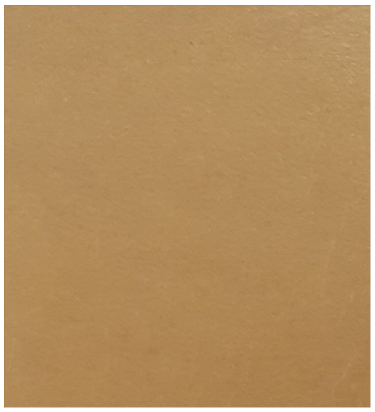
```
[[8. 0. 0. 0. 2. 0.]]
0
[[9. 0. 0. 0. 1. 0.]]
0
[[8. 0. 1. 0. 1. 0.]]
0
[[7. 0. 0. 0. 3. 0.]]
0
[[9. 0. 0. 0. 1. 0.]]
0
```

In conclusion, while it did struggle to classify classes 4 and 5, which refers to going upstairs and downstairs, it did classify the remaining 4 classes with about 80 percent accuracy. This proves it capability to be a useful classifier.

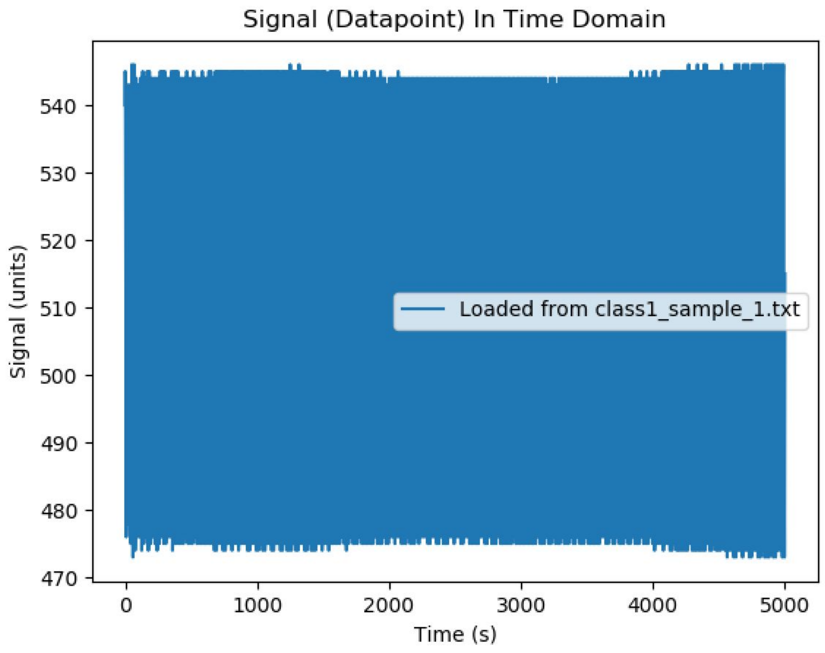
Testing with Real Data:

We now go into the real implementation. The sensor and the surface who's texture is to be classified is shown. There are two surfaces, the one with wide slits and the smooth surface. The surfaces are shown on the next page. 30 CSV files of each surface were made, where the sensor was swiped across the surface thrice. Each CSV stores information on a datpoint over a time frame of 5 seconds, the standard of the time frame defined throughout this entire document.

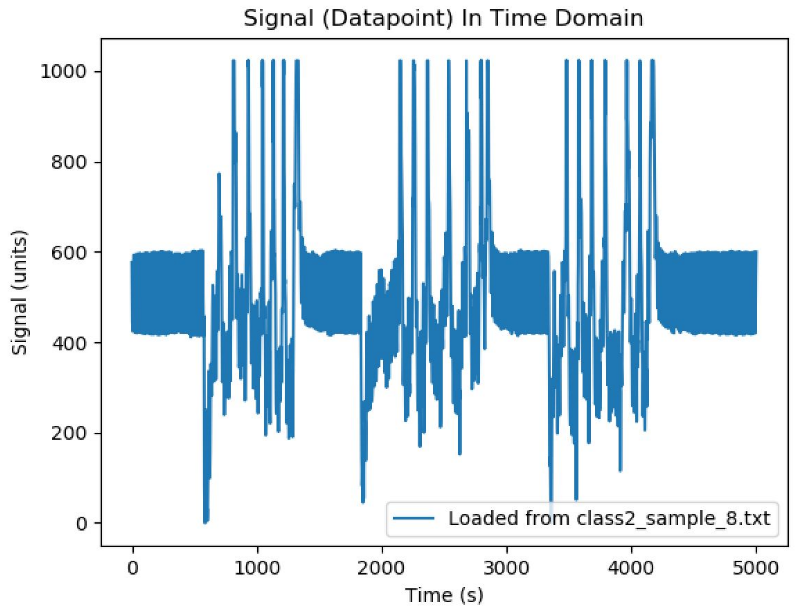




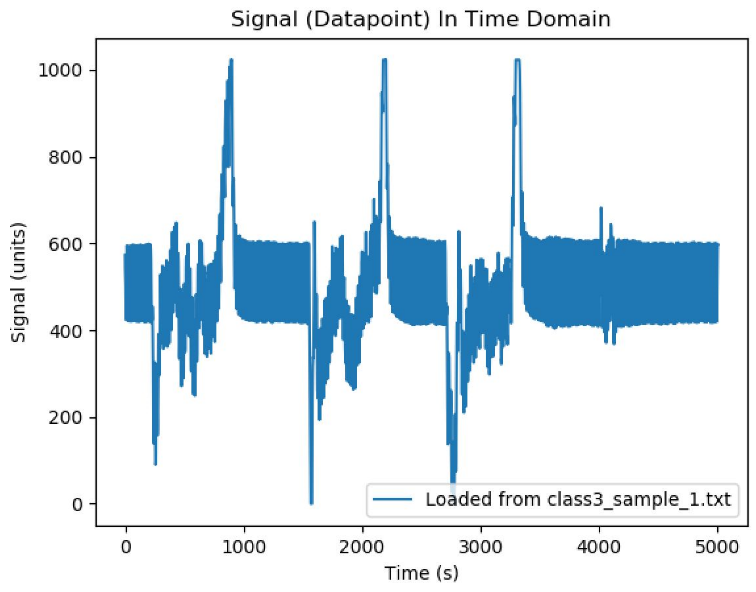
There are two classes of 30 points now, and 25 of them from each was taken as training data and the remaining 5 as testing data. We now have 50 training data points and 10 testing. The following is the noise that is generated. Note the Y axis values, ranging from 480 units to 540 units. Noise is defined as class 1, but this isnt the official class naming scheme.



The next diagram is of a sample signal (datapoint) of the rough surface. Notice the 3 different peaks, that deviate from the noise, and note that each peak is an oscillation between values less than and greater than 0.



The final diagram is of the smooth surface. The sensor doesn't really oscillate much, except when it snaps back to the mean value in between each of the 3 swipes. The peak occurs at the end of each swipe.



Running the classification algorithm with K as 5 gave this:

```
[[ 9  3 24  6 16]
 [12 16 24 23 13]
 [ 3 12  6 20  5]
 [ 6 21 14 11 20]
 [19  9  7 24  3]
 [49 39 44 45 38]
 [38 48 40 41 36]
 [39 42 44 35 37]
 [38 39 45 42 48]
 [38 47 39 36 43]]
```

The ith row represents the indices of the 5 nearest neighbors to the ith test datapoint. Note that there are 10 rows, as there are 10 data points in the test data. A value less than 25 implies we are near a point classified as the first surface(class 1) and the final is of the smooth surface (class 2). Greater than or equal to 25 means the same for a class 2 point. A simple iteration tells us that we have 100 percent accuracy.

```
(1, array([[0., 5., 0.]]))
(1, array([[0., 5., 0.]]))
(1, array([[0., 5., 0.]]))
(1, array([[0., 5., 0.]]))
(1, array([[0., 5., 0.]]))
(2, array([[0., 0., 5.]]))
(2, array([[0., 0., 5.]]))
(2, array([[0., 0., 5.]]))
(2, array([[0., 0., 5.]]))
(2, array([[0., 0., 5.]]))
```

Each row now represents the predicted class and the confidence array. The first index of the confidence array corresponds to the noise class (not assigned a class number), the second is the rough surface class (class 2) and the last is the Consider the first row. We know that its actual class is 1. It matches with the predicted class. We can see that this is actually 100% accurate.
Obviously we haven't built a perfect classifier. All of the data points were obtained with an equal number of surface swipes. Similarly they were done at the relatively some speed(I was the constant here). DTW allows flexibility in speed of collecting the swiped, provided it doesn't disorient the shape of the signal too much. This flexibility comes from the fact a stretched signal and the original signal's DTW is 0(or very close to zero). However the number of swipes does affect it. Variations in the pressure that is manually applied also do the same. A fluctuation that displaces the analog signal by a

digital component also causes problems in DTW, since DTW can only handle changes in signal shape and not changes in amplitude. It's important to remember that all of these conditions were nullified in the current experiment, this giving results we wanted.

Limitations, Potential Fixes and the Step Forward

Hardware Limitations and Fixes:

Any hardware based implementation will have some sort of limitation specifically to the hardware. The arduino board’s variations in the time that it takes to make a reading even with a constant delay (remember this arises from different Serial.write times) proves to be a problem with timestamps. Using milliseconds as a the timestamp unit limits the signal resolution to 5000 samples(in practice 4279) in a 5 second time frame, and there may be a need to have a higher resolution to clearly pick out the differences in the signal waveform of different classes.
Using microseconds as a timestamp without any delay, i.e. letting the arduino loop time define the sampling rate, is a possible fix. Remember that with a higher precision of timestamps we are now at the liberty of doing this. We won’t have the problem of two samples having the same timestamps.
Another thing to note is that using a vibration sensor won’t let it pick up the minute details texture. This, though not a problem in our implementation, will show up when one starts testing with softer surfaces, and ones with finer differences. Essentially, the sensor needs to be able to read a surface and convert it into a precise signal that is of a characteristic shape of that texture. This ideally should be done at a very small scale. This is not seen in the vibration sensor, as it can only measure differences in surfaces of a large scale (centimeter scale). Swiping it across two extremely fine surfaces will not find a difference in them. The classifier would then fail as the data will be not precise enough. This can be seen in the testing of the classifier where classes 2 and 3 were being confused.
A highly sensitive sensor could be a fix. It needs to detect differences of at least the millimeter scale to start making differences. Micrometers would be the best, though such precision depends on the sensor’s design.
Reducing noise couldn’t hurt, though amplification that happens in the circuit might make this hard to bypass. A better SNR is always better for classification.

Software Limitations and Fixes:

A standardized measure of collecting data is required and is perhaps the biggest limitation. While DTW can accommodate differences in the pace that one swipes a surface(provided we have a good enough signal resolution) it can't accommodate a change in amplitude, whether it is a shift of the signal by some units or scaling of the signal. Another metric that can accommodate this would be useful, however, since it would also have to test for scaling, its time complexity will be worse than DTW itself. It is recommended to stick with DTW and make sure the signal isn't scaled or shifted.
In theory, on could have a sensor that would swipe across a surface, where the pressure applied and direction of swipes is kept constant. This would require a robotic implementation that would have a surface to place the textures and the sensor fixed to a moving arm.
Knn also has a fundamental limitation that arises from it's laziness, it's not incredibly scalable. DTW brings out the problem of time, an accurate DTW measure is quadratic in it's input. Fastdtw is a limitation as by approximating DTW, we are hitting against a wall, defined by in general how good the DTW approximation is.
One could play around with the many different ways KNN runs, since currently it is implemented with ball tree. However, this may also reach a performance limit that may not be scalable.

The Step Forward:

As we are focussed on software, our discussion will be the next step to the software of this texture classification problem.
We have shown that with one sensing unit in the m*n matrix of piezoelectric units we can classify data to some precision. This same classification algorithm can now be extended to accommodate all m*n sensing units.
Each data point would then have m*n single variable signals, so we can understand this as an m*n variable signal. To compare the distance between the data points, we measure the DTW of each of the two signals of the same (i,j) location of the sensor matrix. Then we can take the sum of squares of all such DTW over all i and j to get a proper distance metric. This will allow now to classify in the case we have multiple sensing units.
Using CNNs followed by a fully connected neural network with softmax activation for classification would be the better way to go. We would treat signals as a 1D image, and would run many filters to get characteristic features of the signals. An example would be number of peaks. We can then get a confidence level since this is the meaning of softmax activation.

Another idea would be to treat the time series data as a time series of matrices. This would essentially mean we would have a tensor for reach datapoint and the problem now becomes tensor classification. To do so, recurrent neural networks, such as LSTMs can be used.