Activity Recognition Using Accelerometer Data and Signal Processing

Ryan Louie, Meg McCauley, Jennifer Wei

INTRODUCTION

While researching potential directions for our final project, we were intrigued by a code implementation for activity recognition by Arshak Navruzyan[1]. In this implementation, Navruzyan references and uses data from a Fordham research paper[2] that looks into activity recognition using accelerometer data of a phone with four specified activities: walking, jogging, going upstairs, and going downstairs.

We were inspired to pursue this topic for our Signals and Systems final project. We implemented an identifier using accelerometer data and signal processing to predict which of the four activities mentioned above produced the data. The codebase was written in the Python programming language and relied on several tools in the Python scientific ecosystem, including IPython Notebooks, NumPy, Pandas, ThinkDSP, and HMMLearn.

HOW IT WORKS - HIGH LEVEL OVERVIEW

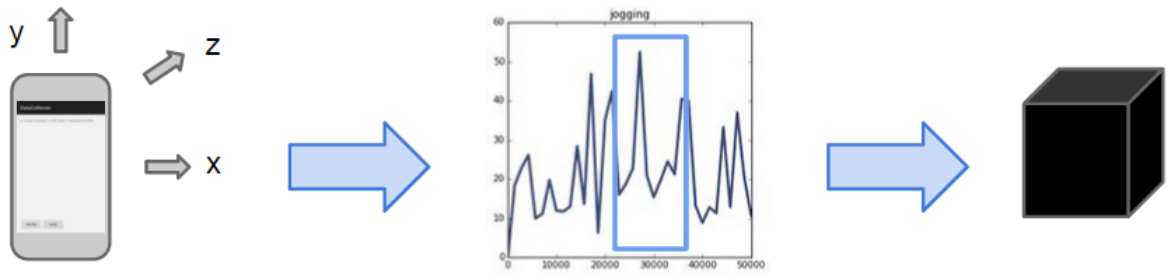


Figure 1: A pictorial representation showing (a) the axes of collected accelerometer data, (b) a segmented piece of the data, and (c) a "black box" representing the implementation of the Hidden Markov Model.

Here's a high level overview of how we went about our final project:

1. We collected data using an accelerometer app (developed by Chris Lee, Olin '15).
2. Using the accelerometer time series (where we look at the magnitude of the components), we applied a sliding window and extracted features (dominant frequency and amplitude) from this data.
3. We fed the features into the "black box" model (Hidden Markov Model) which returns the likelihood that the sequence of features demonstrates the activity model the data represents.

The idea of using consecutive windows to generate an observation sequence to feed into a Hidden Markov Model was inspired by Navruzyan's work. Our modifications, namely, using the magnitude of all three components, extracting multiple features, and

utilizing four different Hidden Markov Models for each activity, will be discussed below.

HOW IT WORKS - COLLECTING DATA AND PREPROCESSING

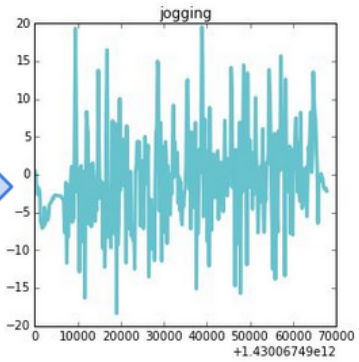
The accelerometer app collects data in the following format: x component, y component, z component, and time.

We collected data for four activities: walking, jogging, walking up stairs, and walking down stairs. For this data collection, the smartphone was placed upside down in the front pocket of the user. The upside down orientation allowed for easier access to the start, stop, and collect data buttons on the smartphone app. Exact placement of the smartphone varied due to differing sizes of pockets between users.

Four different users participated in data collection. Each user was asked to go for a short walk around for a fixed distance, a short jog of the same distance, and to walk up and down the same set of stairs.

x	y	z	time
0.493804	2.130241	8.994417	1430067490092
0.135272	1.395221	7.765593	1430067490265
-2.08535	2.178125	9.363723	1430067490445
-2.765303	1.742979	9.216479	1430067490625
-1.693299	-0.641047	10.671555	1430067490805

magnitude	time
9.25641973327	1430067490092
7.89109465122	1430067490265
9.38482031335	1430067490445
8.96295488497	1430067490625
10.5168380875	1430067490805



To graph the data, we took the magnitude of the three components and plotted it against time. In Figure 2 below, you can see the resulting acceleration magnitude of a subset of data graphed against time.

Figure 2: (Left) A subset of 5 data points as they were collected from the smartphone accelerometer data. (Middle) The magnitude of the x, y, and z components of the data and time. (Right) The graph that was created from an entire time series over 70 seconds (the x-axis is measured in milliseconds).

Two additional preprocessing steps were key to our final analysis pipeline: **interpolation** and **unbiasing**.

Interpolation of the acceleration magnitude was important in making sure that the data points we collected operated as if they were sampled at equal time intervals. The ThinkDSP “Wave” class we used assumes that the signal values given as input arguments come from an equally sampled sequence. Unbiasing is an operation where the values of a signal are subtracted by the mean of those values. In code, if vals is an array that represents the values of a time series signal:

```
vals -= vals.mean()
```

Unbiasing is important because it changes the characteristic of the frequency spectrum. For signals that are unbiased, the amplitude of frequency 0 is equal to 0. For signals that do not undergo unbiasing, this is not usually the case.

This can be explained by the mathematical definition of the Fourier Transform:

$$X(k) = \sum_{n}^{N} x(n)\, e^{-i\, 2\pi nk/N}$$

At frequency k = 0,

$$X(0) = \sum_{n}^{N} x(n)$$

This says that the amplitude of the fourier transform at frequency k =0 is the integral of the wave segment, or in other words, the area under the wave segment. Unbiased signals will have an area under the curve equal to 0.

See Figure 3 for the empirical reasons on how unbiasing affects the frequency spectrum.

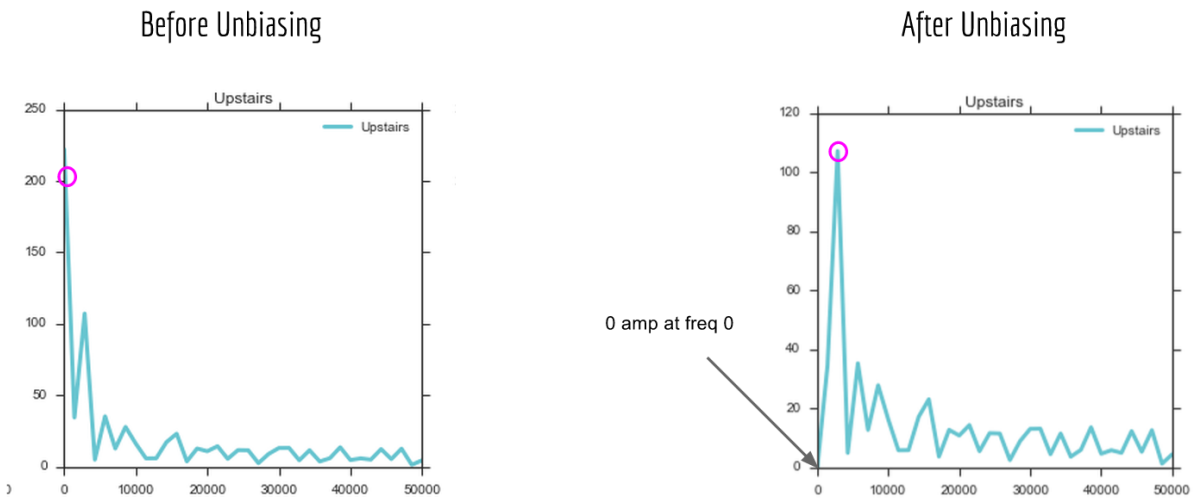


Figure 3: The frequency spectrum plots above show the before and after effects of unbiasing. The pink circles identify the dominant frequency in each plot. (Left) Before unbiasing, the dominant frequency is located at the zeroth point. (Right) After unbiasing, the frequency at the zeroth point now has an amplitude of 0 and the dominant frequency has shifted to represent the actual dominant frequency.

HOW IT WORKS - SLIDING WINDOW

In order to isolate a smaller piece of data to work with, we separated many overlapping segments, represented by each of the colored windows in Figure 4. ***Note: These boxes are the same color throughout the post, so you can follow them from here to the end.***

Below, you can see three windows that come from the larger data set. Each of these

segments then becomes the data from which we extract the dominant frequency and amplitude.

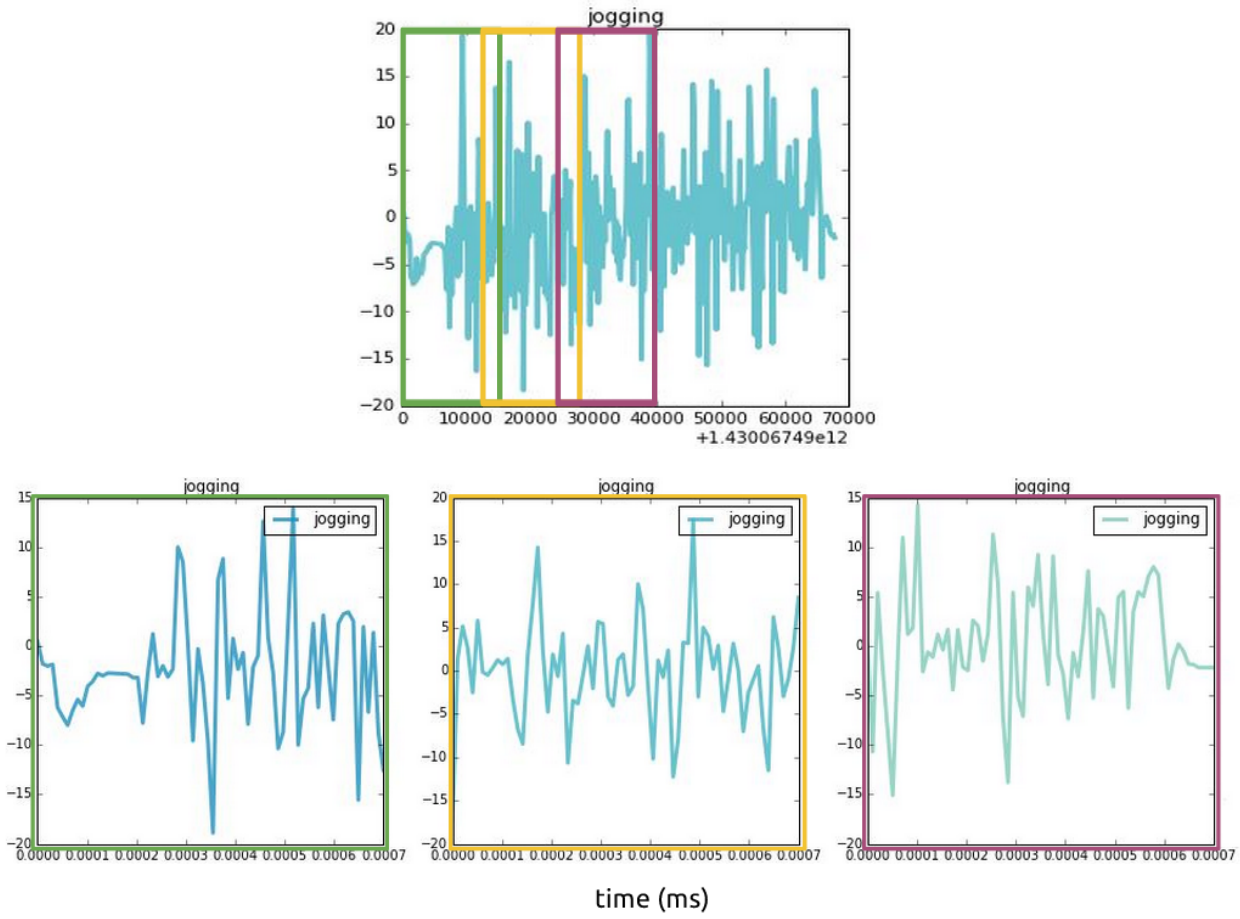


Figure 4: (Top, a) The entire data set from one user's jogging data, with three overlapping windows indicated. (Bottom, b,c,d) A zoom-in of the three indicated windows.

HOW IT WORKS - FEATURE EXTRACTION

We had a few options for distinct features to look into, like dominant frequency, fundamental frequency, or amplitude. We decided to tackle dominant frequency and amplitude. We chose not to find the fundamental frequency due to complications in dealing with the data. There was not one universally good way to find the fundamental frequency across a wide range of data frames. We decided on our direction because the dominant frequency is the first highest peak, and therefore, it proved to be more computationally feasible to find.

One common theme in Signals and Systems is that analyses can be done in both the time and frequency domain. For our feature extraction steps, this fact is absolutely true. While the autocorrelation function operates in the time domain, we can also extract the dominant frequency by analyzing the peaks in the frequency spectrum of the signal. Our

functions are written so that either method can be used.

Autocorrelation

First, we implemented the autocorrelation method using the built-in NumPy method, which returned lags and corrs. The variable lags is a list of numbers starting at 0 that goes to half of the time scale of the wave/segment you are working with. The variable corrs is the serial correlation that helps relate what you know about data at each lag value.

For the purposes of this project, autocorrelation was used to help us calculate the dominant frequency. Using the return value, lags, from the autocorrelation function, we were able to determine which point in time the signal experienced its highest peak. After getting this index value, we calculated the period - taking the lag and dividing it by the framerate of the segment. Once we had this, it was simple to find the frequency by taking the inverse of the period.

For the code that created this function, please see Figure 6.

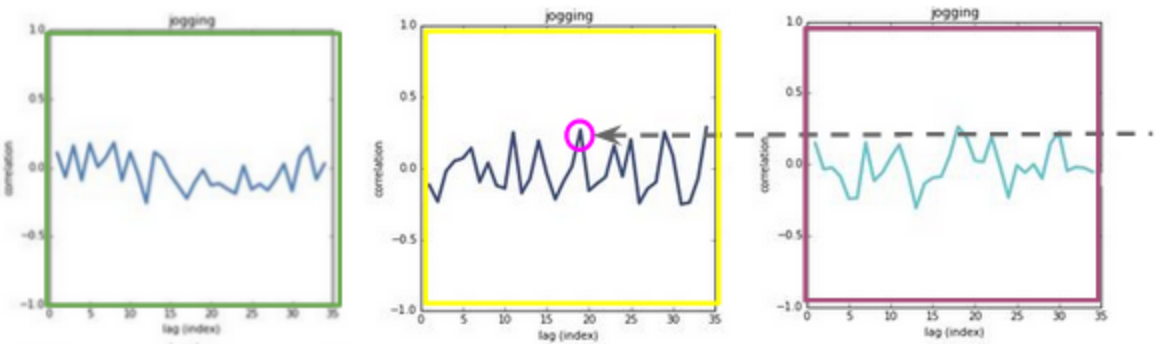


Figure 5: The three data segment windows after autocorrelation. The pink circle on the yellow window in the middle represents the extracted dominant frequency. This was calculated using the lag index and max value of the autocorrelation function.

```
def find_dominant(zseg):
    lags, corrs = autocorr.autocorr(zseg)

    # argmax() gets the maximum, [1:len(corrs)] cuts off the intial
    # 0 peak and +1 fixes the indices back to the correct place
    dom_freq_index = numpy.argmax(corrs[1:len(corrs)])+1

    period = dom_freq_index/zseg.framerate
    dom_freq = 1/period

    return dom_freq
```

Figure 6: The final function that calculates the dominant frequency of a windowed segment of data. The input is a single segment, and the output is a single value for dominant frequency. This code functions by performing autocorrelation with the segment and extracting the index of the maximum value frequency, calculating the period using the index and the framerate of the spectrum, using that to calculate the dominant frequency by taking the inverse of the period, and then returning the dominant frequency.

Frequency Spectrum

Dominant frequency, in addition to amplitude, can be calculated by analyzing the frequency spectrum too.

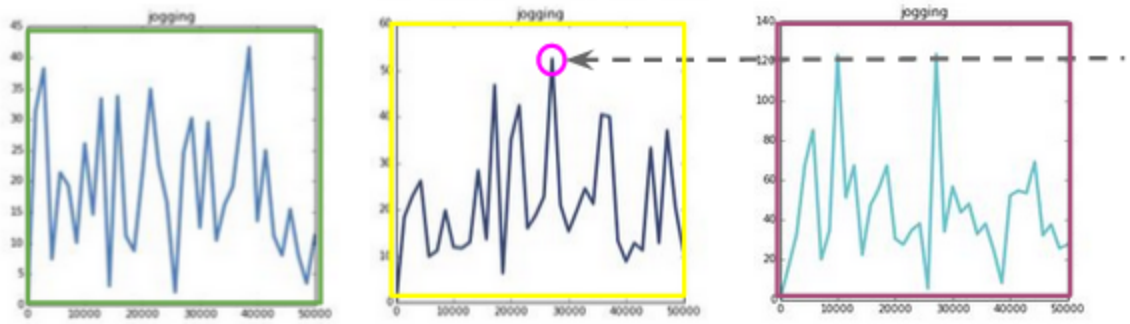


Figure 7: The spectrum of three data segment windows. The pink circle represents the extracted dominant frequency. This was calculated using the peaks method of the Spectrum class.

HOW IT WORKS - OBSERVATION SEQUENCES

Observation sequences are the end product of our feature extraction. In our model, the number of sliding window segments for any observation sequence was arbitrarily chosen to be 10. An observation sequence is an example that is presented to the model. For the sake of common language, we will refer to the 10 sequential windows as the window set.

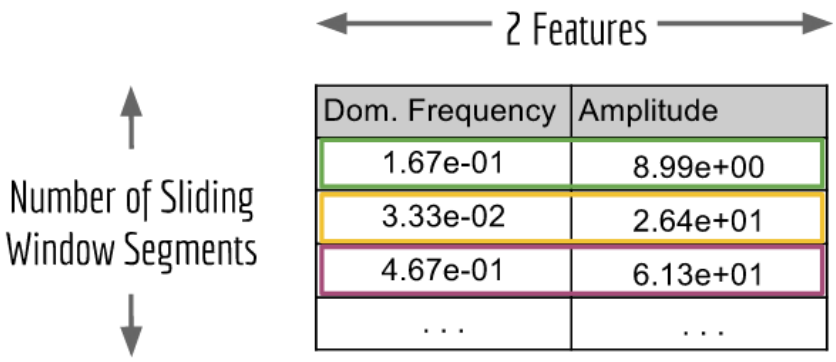


Figure 8: An example of an observation sequence. For our model, we used 2 features, dominant frequency and amplitude, and our window set was comprised of 10 window segments.

For any data set of a certain user doing a particular activity, their data can usually be broken down into more than 10 windows. Thus, we do a process of choosing sequential window sets from a single accelerometer time series, as shown in Figures 9 and 10.

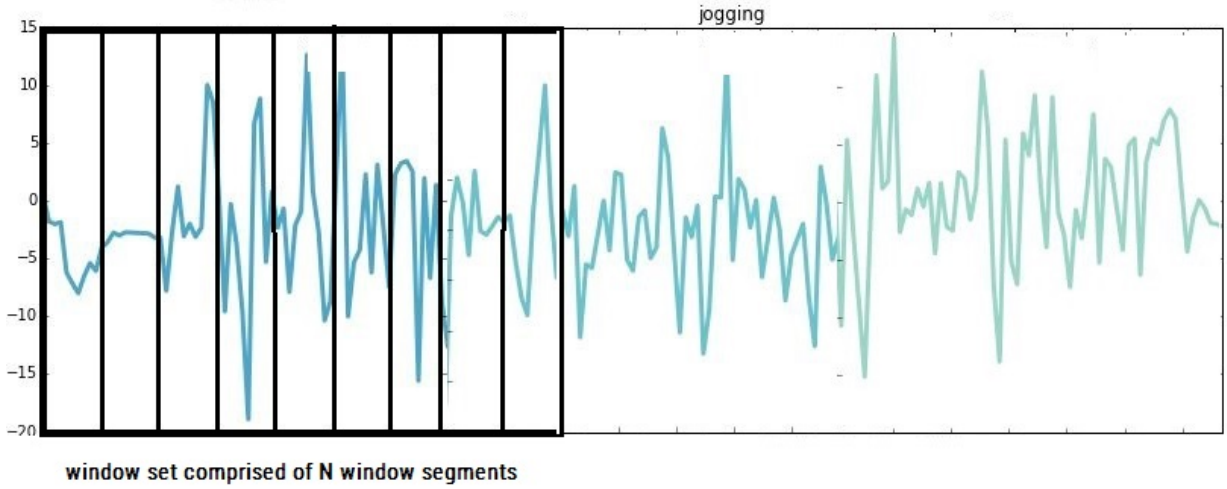


Figure 9: The first window set of the time series is comprised of N window segments. Extracting features from the N sequential windows yields an observation sequence.

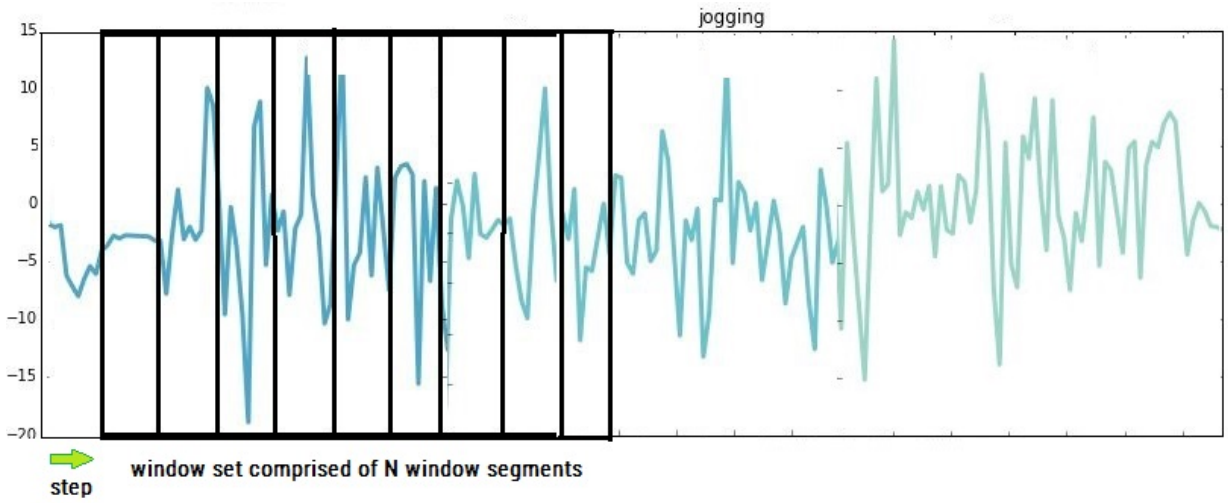


Figure 10: The next window set in the series is selected by choosing the next N consecutive window segments. The window segments shown here are adjacent for visual purposes; our model uses overlapping segments.

HOW IT WORKS - HIDDEN MARKOV MODEL

A Hidden Markov Model learns patterns in sequences of data. The patterns are modeled probabilistically. The model can produce a likely sequence as well as give a likelihood that a sequence could be produced by the model. For our project, one Hidden Markov Model learns the patterns from the accelerometer data for each activity, giving us four Hidden Markov Models.

Figure 11a represents how data flows for a particular activity into a model that learns exclusively on data for that activity.

Figure 11b shows how we can predict which activity produced an unseen sequence of observations.

We used a Python library called HMMLearn[3]. You can learn more about Hidden Markov Models online through this package. We decided to not dive too deep into the details given that our project was focused on signal processing, not on machine learning.

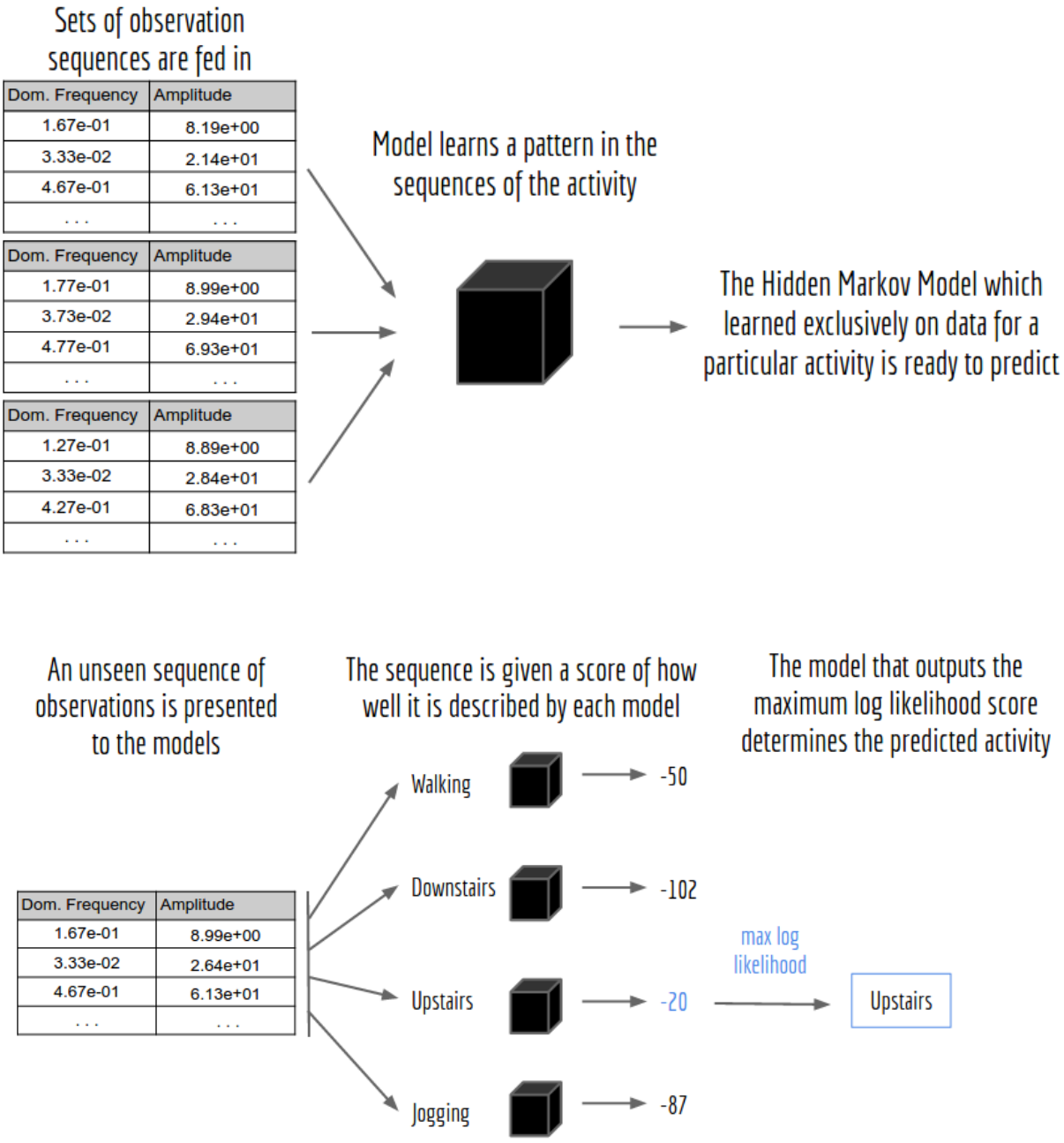


Figure 11:
(a) A Hidden Markov Model is trained by learning on a set of observation sequences for each respective activity.
(b) Predicting the activity for an unknown sequence works by querying the four models for each activity. The model that outputs the maximum likelihood determines the predicted activity.

RESULTS

To test our identifier, we ran it with all the user activity data the model had seen during train time.

Name	Activity	Prediction Accuracy
Ryan	Walking	142/177 (0.80)
	Jogging	10/50 (0.20)
	Upstairs	75/75 (1.0)
	Downstairs	6/8 (0.75)
Dennis	Walking	162/204 (0.79)
	Jogging	7/14 (0.50)
	Upstairs	20/223 (0.089)
	Downstairs	54/242 (0.22)

Figure 12. The table of results for the trained data against the HMM, where the prediction accuracy is the number of correct predictions out of the total number of window sets.

As shown in the table above, the model was able to correctly predict some sets of windows for each user and the four activities that they participated in. However, the accuracy varied from 8.9% to 100% across the eight different predictions. This is due to several possible factors, such as

- positioning of the phone in the user's pocket (though that should be taken care of since we look at the magnitude of the components and not the individual components)
- a small data set for the model to learn from
- not applying the optimal windowing technique
- differences in phone accelerometer sensitivity and calibration

		Predicted			
Actual	**Ryan**	Walking	Jogging	Upstairs	Downstairs
	Walking	***73.94***	-26.78	39.58	67.48
	Jogging	-41.66	-31.51	**-28.12**	-56.39
	Upstairs	-50.47	-46.41	**-15.31**	-132.10
	Downstairs	-30.85	-35.26	-39.32	**-29.04**

Figure 13. The confusion matrix above contains the average log likelihoods for each pair of activities (actual and predicted) for 'Ryan', a user activity data the model had seen during train time.

When we looked into each prediction via a confusion matrix, we noticed that the log likelihoods of failed predictions often had two that were close (as highlighted in yellow in Figure 13): that of the actual prediction and that of an incorrect prediction.

We later tested unseen data - data that was not fed into the model for training, against the model's predictions. As shown in Figure 14, one user had 100% accuracy for upstairs predictions, 2% accuracy for jogging, and 0% accuracy for walking and going upstairs.

Name	Activity	Prediction Accuracy
Meg	Walking	0/274 (0.0)
	Jogging	6/324 (0.02)
	Upstairs	103/103 (1.0)
	Downstairs	0/124 (0.0)

Figure 14. The table of results for the unseen data against the HMM, where the prediction accuracy is the number of correct predictions out of the total number of window sets.

To investigate this unexpected result, we created a confusion matrix of unseen data from one user, 'Meg'. The confusion matrix shows that the model predicts upstairs for all four actual activities. Unlike the results for the trained data, the second highest likelihood predicts downstairs for all four activities. As we mentioned earlier, a huge

contributing factor could be the small data set in terms of both data collected for individual users and also number of users to train the model with.

		Predicted			
Actual	**MEG**	Walking	Jogging	Upstairs	Downstairs
	Walking	*-44.49*	-30.92	**36.89**	-19.44
	Jogging	-42.36	-30.38	**32.37**	-19.60
	Upstairs	-44.17	-26.26	**57.32**	-14.35
	Downstairs	-44.25	-27.43	**51.34**	-19.04

Figure 15. The confusion matrix above contains the average log likelihoods for each pair of activities (actual and predicted) for unseen data.

CLOSING THOUGHTS

Overall, it was interesting to see how signal processing applies to the real world. The identifier we developed has a good structure that can be continued later. Our data and code are available from this Github repository. This project made us realize how important the quantity and quality of data are.

ACKNOWLEDGEMENTS

Special thanks to:

- Arshak Navruzyan for giving us suggestions on ways that he would move forward from his model.
- Chris Lee for building an Android app for us and rebuilding the app to our specifications a second time around.
- Jacob Riedel for allowing us to borrow his phone, install the app, and collect data with it.
- Paul Ruvolo for helping us gain a working understanding of Hidden Markov Models.
- Allen, Siddhartan, and Oscar, our Signals and Systems professors, who all provided useful tips regarding our project and how to present our findings (and specifically Allen for allowing us to guest post on his blog).

REFERENCES

1 http://nbviewer.ipython.org/github/StartupML/koan/blob/master/DSP_HMM.ipynb
2 http://www.cis.fordham.edu/wisdm/includes/files/sensorKDD-2010.pdf
3 https://github.com/hmmlearn/hmmlearn