ECE5242 - INTELLIGENT AUTONOMOUS SYSTEMS

PROJECT 2 - GESTURE RECOGNITION

MARCH 7, 2020

EVA ESTEBAN
CORNELL TECH

ABSTRACT

The aim of this report is to provide an in-depth analysis of the different Hidden Markov Models (HMMs) trained to recognize a set of arm motion gestures and classify them in real time. This document describes the approaches implemented, the results obtained and the performance of the models.

TABLE OF CONTENTS

1 Introduction

A Markov Model is a stochastic model used in probability theory to model systems that change randomly. A Hidden Markov Model (HMM) is a Markov model in which the system is a Markov process with states that cannot be observed i.e they are hidden states [1].

In this project, several HMMs were trained to identify different arm motion gestures in real time. The data used to train the models were sensor readings from the accelerometer and the gyroscope of an Inertial Measurement Unit (IMU). These readings corresponded to six different motions: Wave, Infinity, Eight, Circle, Beat3, Beat4. An HMM was trained for each of these motions. Once trained, given a set of new IMU measurements, the model that assigns the highest probability to the sequence of observations is selected as the motion that the measurements describe.

The process to train the HMMs consisted of first reading the raw training data, pre-processing it and discretizing it by using K-Means. This was followed by applying both the Expectation-Maximization [2] and the Forward-Backward algorithms [3] to train one HMM per motion according to the procedure described in the Rabiner paper [4], optimizing each HMM for the maximum log likelihood.

Given a new set of measurements, each HMM is used to calculate the probability that the measurements belong to each motion category by using the log likelihood and the Bayes Theorem. The top three motion guesses are then printed on screen.

2 Problem Formulation

The problem tackled in this project is the recognition and classification in real time of an arm motion gesture given a set of measurements which consist of a timestamp, accelerometer values and gyroscope values. The solution to the problem was developed using Hidden Markov Models and different machine learning techniques such as the Expectation-Maximization algorithm [2], and the Forward-Backward algorithm [3]. It was then evaluated using train-test splits with cross-validation and checking the log likelihood.

3 Technical Approach

3.1 Data Pre-processing and Discretization

The first step of the project consisted of reading the training data. The training data is stored in two folders: the folder **ECE5242Proj2-train** contains raw IMU readings for each motion repeated several times, while the folder **ECE5242Proj2_train_additional** contains readings for each motion performed only once.

The training data was split in order to train Hidden Markov Models (HMMs) and test them before the test data was released. For this, the data from the **ECE5242Proj2_train_additional** folder was considered test data, and the HMMs were trained using the data from the folder **ECE5242Proj2-train**.

To do this, first the data contained in the folder **ECE5242Proj2-train** was read into a single dataframe using the **pandas** library [5], and the timestamp value was removed. In order to discretize the data, K-Means clustering, which is the simplest form of a Gaussian Mixture Model (GMM), was applied to this data. To implement the **k_means()** function, the tutorial from [6] was completed and adapted to fit this project's needs. The resulting data consisted two vectors of values. For the first 1D vector, its length corresponds to the number of observations and each element is an integer number representing the cluster that particular observation belongs to. The second vector was a 75x6 array containing the cluster centers. Both arrays were stored in the **HMMs** folder as .npy files for future use. Once the data had been clustered and therefore discretized, it was split into the six separate motions in order to train one model per motion. The k-means functionality can be found in the function **k_means()**. The number of clusters k = 75 was found to be the most suitable after trying a set of different values within the range 50 to 100. Additionally, the distance measure used in the k-means function was the L2 norm or Euclidean distance, obtained by using the function **norm()** from the **linalg** module in the **numpy** library [7].

3.2 Hidden Markov Model Training and Saving

An HMM was trained for each motion category, giving a total of six HMMs. The trained HMMs were discrete and of type ergodic or fully connected, meaning that every state can be reached from every other state in a finite number of steps. The procedure to train an HMM implemented follows the tutorial in the Rabiner paper [4]. The resources from [8] and [9] were also consulted.

The first step to train an HMM consisted of initializing the number of hidden states N, the initial state distribution matrix π, the transition matrix A, and the emission matrix B, which are the parameters that fully describe the HMM. The number of hidden states N was set to 10 after being found the most appropriate value from a set of different values within the range 5 to 15. The number of observation types M for each motion was set to the number of clusters, which remained the same across motions. The matrix π was initialized as an NxN matrix with 1 for the first element and 0 for the rest of the elements. This is to indicate that we start from the first state at the beginning of every measurements file and to be able to keep π constant throughout the program. The transition matrix A was initialized as an MxN matrix with every element being a random double within the interval [0.0, 1.0). The matrix was then normalized in order for every row to add up to 1. Finally, the emission matrix B was initialized as an NxN matrix again with every element being a random double within the interval [0.0, 1.0). It was then normalized in order for every column to add up to 1.

The next step consisted of using the Expectation-Maximization algorithm [2] to tune the π, A, and B parameters until the log likelihood converged. The log likelihood is the log of the probability of a set of observations given an HMM model and is written as $\log(P(O|\lambda))$. The number of iterations for the EM algorithm was set to be 20, as this is the value at which the log likelihood was found to have converged for all the motions. The graphical results that show this converging behaviour can be found in the 'Results' section of this report.

During the expectation step, the log likelihood values were calculated by using the Forward-Backward algorithm [3]. The values of the forward variable α, the backward variable β, the gamma variable γ and the xi variable ξ were calculated together with the log likelihood by using the π, transition and emission matrices. This was done following the steps and mathematical formulas presented in the **Problem 1** and **Problem 2** sections of the Rabiner paper found in [4]. These formulas were not written in this report so as not to include redundant information easily accessible on the paper. To prevent elements in the emission matrix from being strictly zero which consequently causes problems when calculating the scaling alpha coefficients, while still keeping their statistical significance, the small value of 1^{-10} was added to every updated emission value. The code functions in charge of these calculations are **traim_hmm()**, **em_algorithm()**, **calculate_forward()**, **calculate_backward()**, and **calculate_gamma_xi()**.

During the maximization step, the transition and emission matrices were updated using the values of α, β, γ and ξ previously calculated during the expectation step. The mathematical expressions used can be found on the **Problem 3** section of the Rabiner paper [4]. The code function that implement these algorithms is **calculate_transition_emission()**.

The calculations in both the expectation and the maximization steps of the EM algorithm are scaled in order to avoid underflow. The scaling procedure consists of using a coefficient c which, for each observation in time, equals the inverse of the sum of the forward variables for all the observations in time. Each of the forward elements is then multiplied by this coefficient c to prevent it from increasing too fast, while each of the backward elements is then divided by the coefficient to prevent it from decreasing too fast. The mathematical expressions for this can be found in the **Scaling** section of the Rabiner paper [4].

Once the log likelihood converged, the values for the final transition and emission matrices together with the likelihoods corresponding to each of the EM iterations for the HMM were saved in the folder **HMMs** as .npy files using the function **save_model()**. This procedure was completed for each of the six motions, resulting in an optimized HMM saved for each of them.

3.3 Hidden Markov Model Loading and Application

To classify a new motion, the motion measurements are read into a dataframe. The dataframe values are then discretized in the function **k_means_predict()** using the clusters saved from k-means when training the data. This ensures that the value discretization is consistent throughout. The next step consists of loading the HMMs trained for each of the motions by using the function **load_model()**. Once this step is completed, the **calculate_forward()** function is applied to the new set of observations for the motion using each of the models, giving a log likelihood value for each model. These values and their corresponding motion names are stored in two lists, which are then sorted in ascending order according to the values. The names corresponding to the bottom three values are extracted, giving the top three predictions for the model that the motion belongs to. These are then printed. The code functionality for this can be found on the function **find_highest()**.

The reason for using the log likelihood value directly to classify the motion stems from Bayes Theorem, found in 1. The prior probability for each motion is calculated using 2. This gives a prior probability of P(class) = $\frac{1}{6}$ for each motion since each of them has approximately the same number of recordings. Since both the prior probability and the normalizing terms in the Bayes equation are equal for all the motions, the relationship between the different P(O|λ) and P(λ|O) values is proportional and the P(O|λ) value can be used to get the top three model guesses for a set of observations.

$$P(\lambda|O) = \frac{P(O|\lambda)P(\lambda)}{P(\lambda)} \tag{1}$$

$$P(\lambda) = \frac{\text{\# items in } \lambda}{\text{total \# items}} \tag{2}$$

The procedure described above was applied to the data in the **ECE5242Proj2-train** being used as testing data from the train-test split. Each set of measurements was classified using the algorithm. The results of this can be found in the **Results** section of this document.

The same procedure was applied to the test data in the folder **ECE5242Proj2-test** once it was released. The results can be found in the **Results** section of this report.

4 Results

4.1 Train-Test Split from Training Data

As mentioned in the previous section of this report, when training the models N was set to 10, M was set to 75, and the number of EM iterations was set to 20. The graphs on Figure 1, found at the end of this document, confirm the convergence of the log likelihood at 20 iterations for the six different HMMs trained. When using the train-test split described in Section 3 of this document, the algorithm achieved an accuracy of 100%. The results from classifying the test split of the data were as follows:

- beat3_31.txt - First guess: Beat3, Second guess: Beat4, Third guess: Wave.
- beat4_31.txt - First guess: Beat4, Second guess: Beat3, Third guess: Wave.

- circle31.txt - First guess: Circle, Second guess: Beat4, Third guess: Beat3.
- eight31.txt - First guess: Eight, Second guess: Inf, Third guess: Beat3.
- inf31.txt - First guess: Inf, Second guess: Eight, Third guess: Beat4.
- wave31.txt - First guess: Wave, Second guess: Beat3, Third guess: Beat4.

The time taken for the algorithm to run the predictions was measured by using the **default_timer()** function in the **timeit** library [10]. The algorithm took 11.52 seconds in total to classify the six sets of measurements. This gives 1.92 seconds to classify each motion.

4.2 Testing Data

Running the algorithm on the test data yielded the following results:

- test1.txt - First guess: Wave, Second guess: Inf, Third guess: Eight.
- test2.txt - First guess: Beat3, Second guess: Beat4, Third guess: Wave.
- test3.txt - First guess: Inf, Second guess: Eight, Third guess: Beat3.
- test4.txt - First guess: Beat4, Second guess: Beat3, Third guess: Wave.
- test5.txt - First guess: Circle, Second guess: Beat4, Third guess: Beat3.
- test6.txt - First guess: Inf, Second guess: Eight, Third guess: Beat4.
- test7.txt - First guess: Eight, Second guess: Inf, Third guess: Beat3.
- test8.txt - First guess: Beat3, Second guess: Beat4, Third guess: Wave.

Both these and the results from the train-test split can be found in the code file **epe25_project2_code.ipynb**. The algorithm took 19.44 seconds in total to classify the eight sets of measurements.

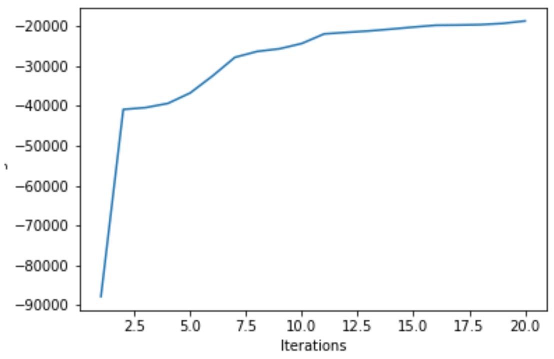
5 Discussion

The algorithm developed for this project achieved 100% accuracy in the first set of unknown data it was given to classify. It took the program 1.92 seconds on average to classify a single motion. The correct classification of the final test samples will be analyzed once the labels get released.

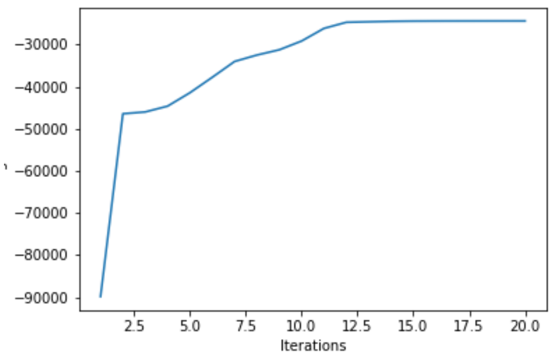
The algorithm was developed in an iterative fashion, improving it over the time of the project to both increase its accuracy and reduce its run time. All in all, the task was successfully completed and the material that corresponds to it was learnt in depth.

6 Future Work

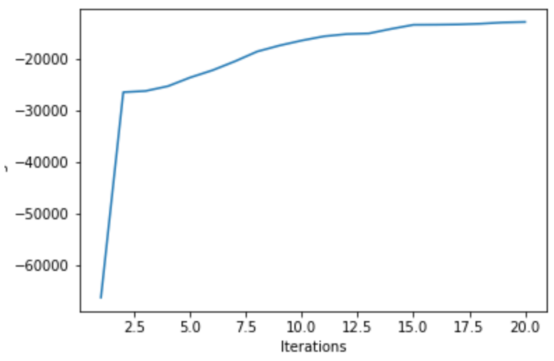
The next steps in this project include using cross-validation to better tune the number of clusters k and the number of hidden states N and increase the accuracy of the HMMs as a result. Moreover, the algorithms will be optimized in order to reduce the time it takes for the program to train the HMMs, even though it is currently within the expected and acceptable limits.



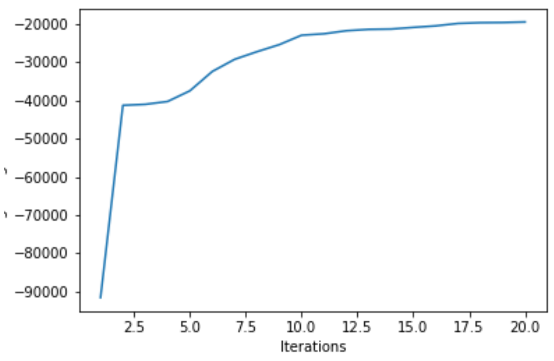
Log Likelihood Plot for Beat3.



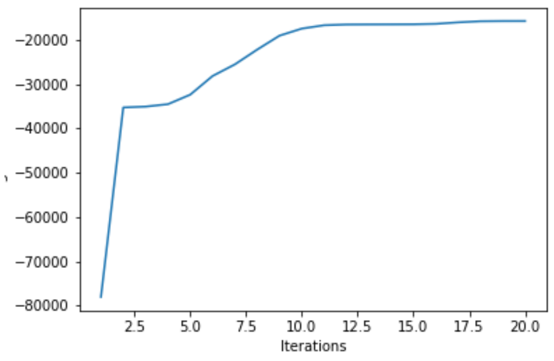
Log Likelihood Plot for Beat4.



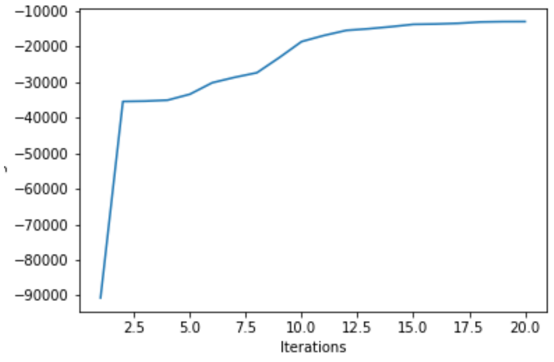
Log Likelihood Plot for Circle.



Log Likelihood Plot for Eight.



Log Likelihood Plot for Inf.



Log Likelihood Plot for Wave.

Figure 1: Log Likelihood Plots for HMMs.

REFERENCES

[1] P. A. Gagniuc, *Markov Chains: From Theory to Implementation and Experimentation.* USA, NJ: John Wiley Sons, 2017.

[2] Wikipedia. (2020) Em Algorithm. "https://en.wikipedia.org/wiki/Expectation\%E2\%80\%93maximization_algorithm"[Accessed 02-23-2020].

[3] ——. (2020) Forward-backward Algorithm. "https://en.wikipedia.org/wiki/Forward%E2%80%93backward_algorithm"[Accessed 02-23-2020].

[4] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, Feb 1989.

[5] W. McKinney, "Data Structures for Statistical Computing in Python," *Proceedings of the 9th Python in Science Conference*, pp. 51–56, 2010.

[6] M. Murat. (2020) Implementing K-Means Clustering from Scratch - in Python. "https://mmuratarat.github.io/2019-07-23/kmeans_from_scratch"[Accessed 03-01-2020].

[7] T. E. Oliphant, *A guide to NumPy.* Trelgol Publishing USA, 2006, vol. 1.

[8] E. Kang. (2017) Hidden Markov Model. "https://medium.com/@kangeugine/hidden-markov-model-7681c22f5b9"[Accessed 02-25-2020].

[9] ananthpn. (2015) pyhmm. "https://github.com/ananthpn/pyhmm"[Accessed 02-25-2020].

[10] P. S. Foundation. (2020) Timeit. "https://docs.python.org/3/library/timeit.html"[Accessed 03-05-2020].