Assignment 2

Sanket Shah, ss4228

A Markov Model describes a graph structure of possibilities, with each node representing a state. Each node-to-node transition has some probability. At each time step there is some new current state (node). If the model ran indefinitely, there would be a natural convergence of total probabilities for each state. (what is the probability of any given state at this time?).

Suppose that the system is totally visible, and we can observe which state it is in. If we were to decouple the observation from the state, we can say that at time t, for a given state s^i, our observation o^i corresponds to that state and that state only. So if we could not see the state, but only an observation (ie signal), we'd know which state with 100% probability. However, if there were multiple possible states per observation, then we'd have a distribution of possible observations for each state, and accordingly, of possible states per observation.

This is a Hidden Markov Model. The states are hidden. This is a parametric model $X^k := f(t; N, M, \lambda)$ where N is the number of possible states, M is the number of possible observations, and λ are our state and observation probability matrices. In a classification task, each class of Markovian sequences will have a model X^k. Our goals are to:

1. Predict the state of the Markov Process
2. Figure out which Markov process we're looking at (classification)

We're trying to calculate $P(O|X^k)$ where O is our sequence of observations (time series). We can re-write this as $P(O|Q, \lambda^k)$, where Q is the state-sequence and λ^k is our main parameter. This can be interpreted as a "generative" model. The Markov Model X generates the series of Observations O. As we are only seeing a sample of the theoretical total Observation space, we cannot directly compute the $P(O)$. However, we can reason about the joint probabilities of Observations and States.

In other words, we'd like to know:

$$P_\lambda(O_1, O_2, O_3, O_4, O_5, \ldots, O_T)$$

But we cannot compute it directly. Therefore we must use this relationship:

$$P_\lambda(O, Q) = P_\lambda(Q) * P_\lambda(O|Q)$$

$$P_\lambda(O) = \sum_{\forall Q} P_\lambda(Q) * P_\lambda(O|Q)$$

However, the number possible state sequences is extremely large, at N^T. For this reason, we use the Forward-Backward algorithm. Note that the $P_\lambda(Q)$ term is represented by our A matrix in the algorithm, and the $P_\lambda(O|Q)$ term is represented by the B matrix in the algorithm. Matrices A and B are part of our parameters λ. In the event we *do not know* this information, we can iteratively compute the matrices, as well as the marginalized probabilities, using the Baum-Welch Algorithm. The Baum-Welch uses the forward-backward algorithm components. Essentially, these iteratively compute the joint probability of the states in the state sequence before and after a given time step *t*. We can use this to reestimate our A, B, and pi parameters.

The algorithm works generally as follows:

1. Initialize a random λ
2. Compute new A, B, and pi
3. Compute new forward/backward, and related probabilities.
4. Repeat 2/3 until stopping criteria.

In my implementation, the algorithm could only handle single observation sequences. I did not implement ability to account for multiple observation sequences. The main challenge here is dealing with "underflow" errors, in which the joint probabilities become so small that they can no longer be represented in 64bits. There are 2 ways to deal with this:

1. Compute everything in “Log-Space”. This is acceptable since we only care about the relative probabilities. This is the method I used. Although the results of calculations were always converted back to standard probability space when storing. They were “normalized” where required (not in the forward/backward) within log-space using Scipy’s *logsumexp* function
2. Normalize on the forward/backward calculations. Rabiner argues that this is possible, although I was not able to get it to produce reasonable results.

Overall, results were poor. Despite what seems to be a correct implementation, predicted classes (gestures) performed about as good as random chance, using the recommended hyperparameters (# States, # Observations). Below are the results for testing data:

The predictions of the test data are as follows, in order of decreasing probability:

{

‘observation’: [predicted classes]

'./test/test1.txt': ['circle14', 'inf18', 'eight08'], './test/test2.txt': ['circle14', 'eight08', 'beat4_08'], './test/test3.txt': ['beat3_02', 'wave07', 'inf18'], './test/test4.txt': ['beat4_08', 'beat3_02', 'inf18'], './test/test5.txt': ['wave07', 'beat3_02', 'inf18'], './test/test6.txt': ['beat4_08', 'inf18', 'wave07'], './test/test7.txt': ['wave07', 'eight08', 'circle14'], './test/test8.txt': ['beat4_08', 'eight08', 'wave07']

}

And here are the results for the training data:

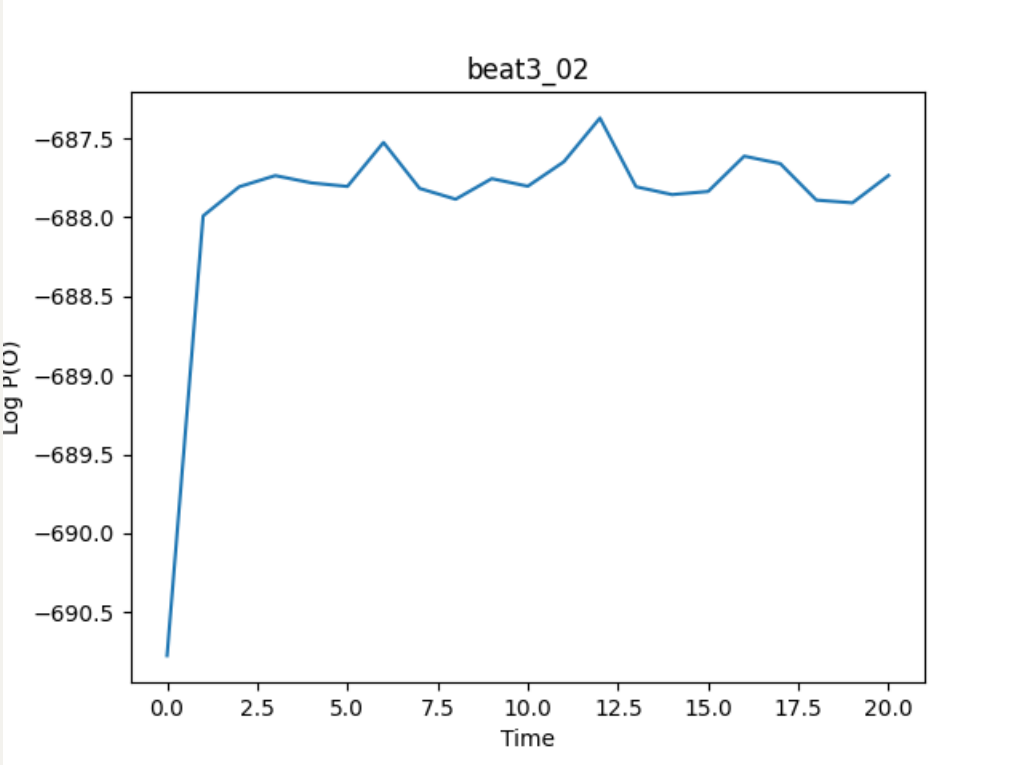
{

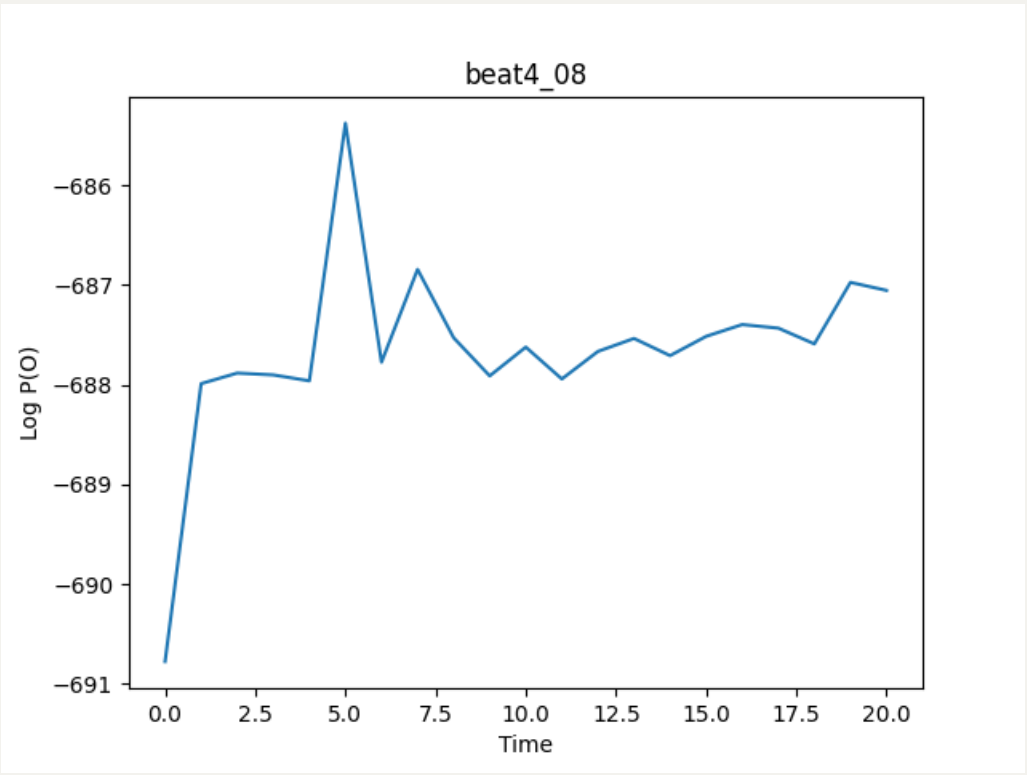
‘observations’: ‘class prediction’, prob

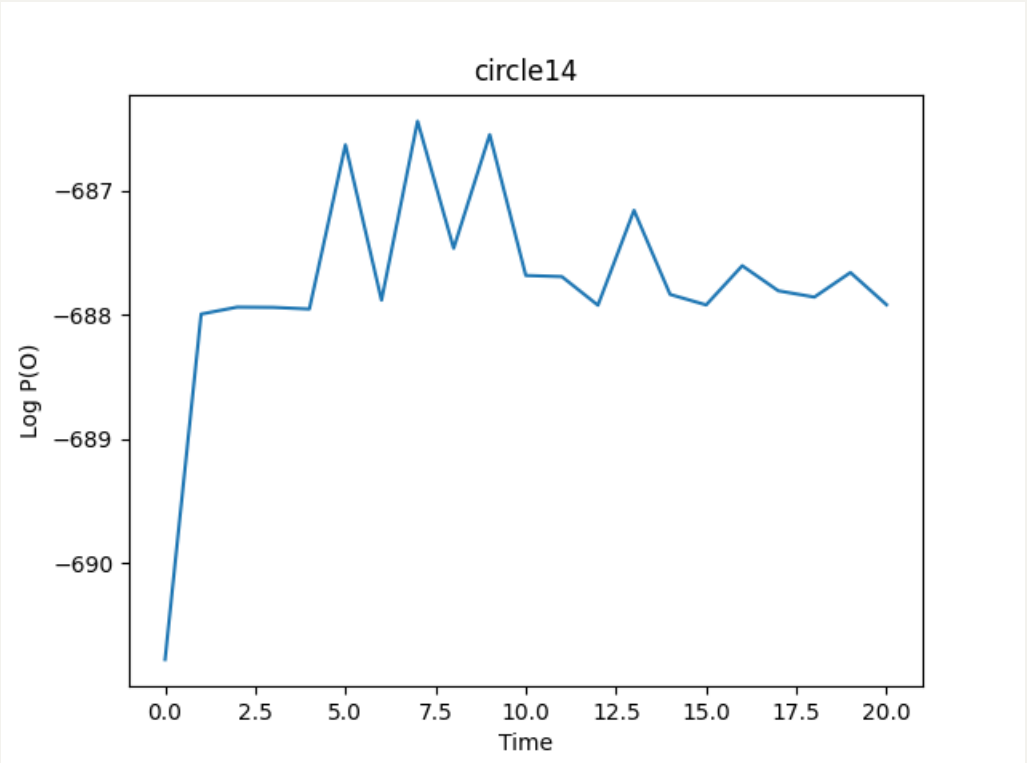
'./train/wave07.txt': wave07 1.521971e-299 './train/circle14.txt': beat3_02 1.555590e-299 './train/eight08.txt': beat4_08 2.080732e-299 './train/inf18.txt': circle14 1.519275e-299 './train/beat4_08.txt': wave07 1.532866e-299 './train/beat3_02.txt': beat4_08 1.524305e-299

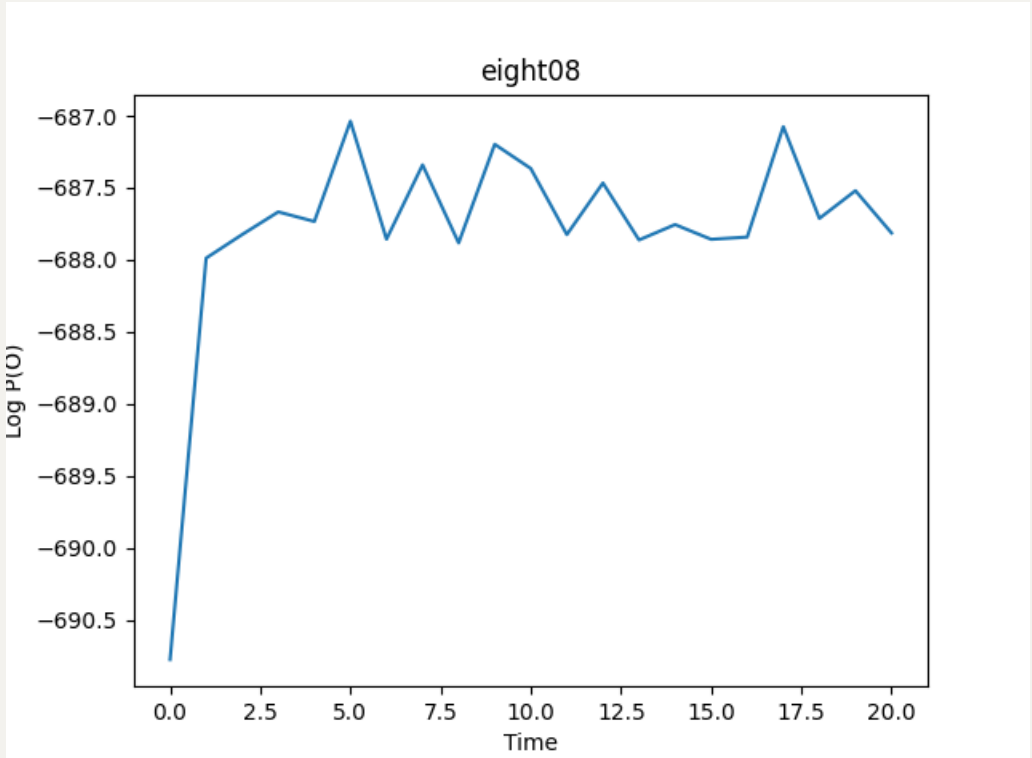
}

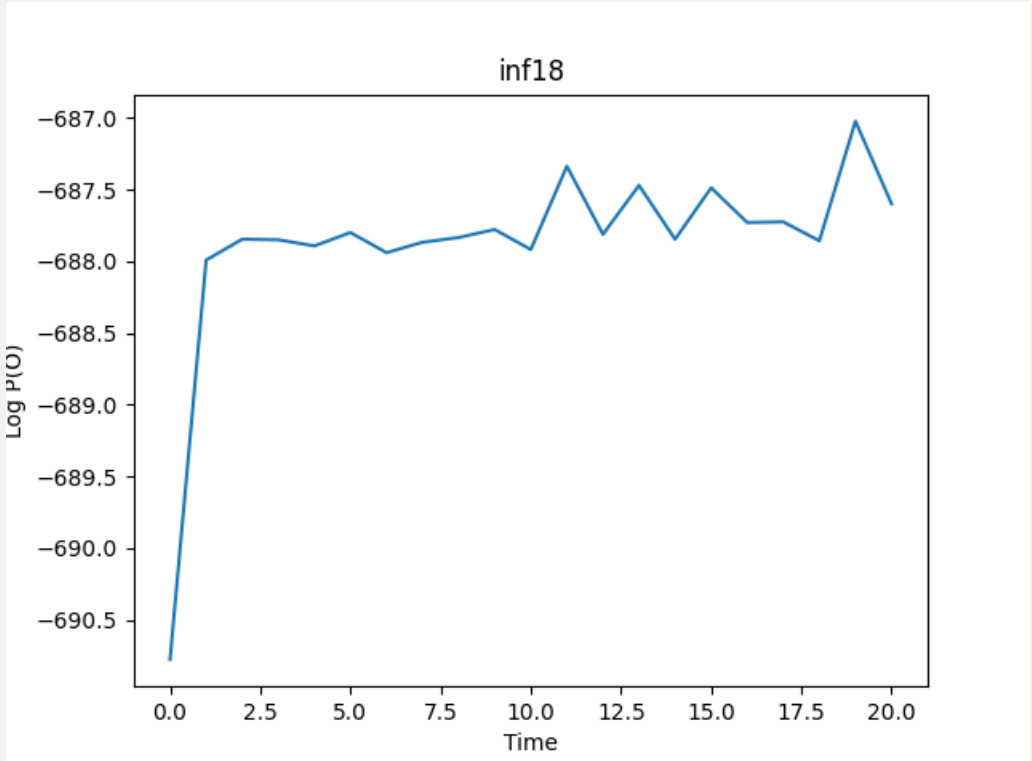
Log-Probability Training plots below. They represent the marginalized probabilities of the observations. As you can see, they fail to consistently improve.




beat4_08
−686
−687
−688
−689
−690
−691
Log P(O)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time


circle14
−687
−688
−689
−690
Log P(O)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time


eight08
−687.0
−687.5
−688.0
−688.5
−689.0
−689.5
−690.0
−690.5
Log P(O)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time


inf18
−687.0
−687.5
−688.0
−688.5
−689.0
−689.5
−690.0
−690.5
Log P(O)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time