IFT6269 Homework 4

Vicki Anand (20137598)

November 21 2018

1 Entropy and Mutual Information

Done on hand-written sheets.

2 HMM Implementation

2.1

Implemented in alpha_beta.py.

2.2

Figure 1 has the asked plot.

2.3

Implemented in hmm_em.py as m_step() function of hmm_gaussian() class.

2.4

Implemented in hmm_gaussian() class in hmm_em.py.

2.5

Figure 2 plots the log-likelihood values for training and test models against the iteration count of the EM algorithm as it progresses. Some comments based on this figure -

- Both training and test likelihood values increases rapidly for initial iterations and them becomes almost stable. This is the expected behaviour from EM algorithm.
- Consistently the test likelihood values are lesser than the training likelihood values. This is due to the fact that the HMM is a very flexible model and such models generally have higher likelihood on training data than on test data.

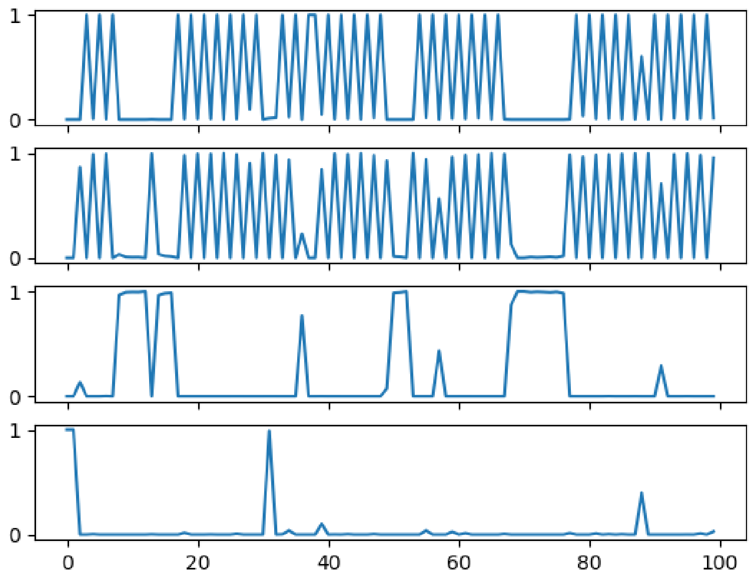

Figure 1: $p(z_t|x_1, ..., x_T)$ for each of the 4 states as a function of t for the 100 first datapoints in the test file. States from top to bottom are 1 to 4.

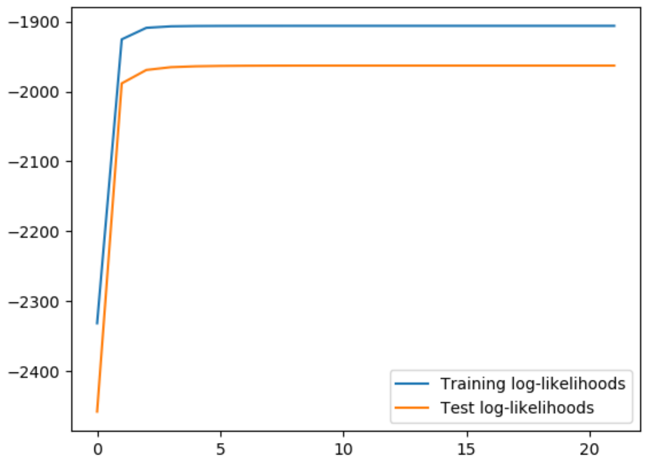

Figure 2: Log likelihood values on training and test data with EM iterations

Model	Train log-likelihood	Test log-likelihood
HMM	-1906.0810	-1962.9260
GMM (full covariance)	-2370.8840	-2453.5631
GMM (isotropic)	-2739.2643	-2718.8795

Table 1: Comparing log-likelihood of train and test data with different models.

2.6

Table 1, shows the log-likelihood values for HMM model and the GMM models from previous homework. We see that the log-likelihood of the both the training data and the test data is more (better) for HMM than the GMM (full-covariance).

Yes, it makes sense to compare these log-likelihood values because here both HMM and GMM are probabilistic graphical models with same emission density assumption (full-covariance gaussian), just that they have different assumptions about conditional independence of the hidden states. By comparing these two we can evaluate that which of the two model is closer to the true underlying distribution of the data.

Further adding the third model - GMM with isotropic-covariance - to the comparison, we see that the HMM is better than both the GMM models and amongst the GMM models, the full-covariance one is better. This ordering holds same on training and test data. Also as explained in previous homework, the more flexible models generally gives mode likelihood on training data compared to that on test data. We observe the same pattern here.

2.7

Viterbi algorithm for HMM is a MAP inference algorithm for the hidden states of the model (i.e. for finding $argmax_z p(z_1, ..., z_n|x_1, ..., x_n)$). It uses the dynamic programming method to find this efficiently. This algorithm breaks the problem into similar sub-problems by taking advantage of factorizing the probability formula.

Below is a Pseudo Code of Viterbi decoding. Notations have been kept similar to those of python language (specially important for python list operations and numpy array operations), except comments have been made in style of C language and not python.

```
/*
Here we assume gaussian distribution for the emission probability.
This can be changed by changing the values that are being assigned
to the e_0 and e_t emission probabilities.

model has following values stored in it:
    model.pi : array with pmf for first latent variable(z_0) (size: k)
    model.A : Transition probability matrix (size: k X k)
    model.mus : array of k means of the k clusters (size: k X d)
    model.sigmas : array of k sigmas of the k clusters (size: k X d X d)

Using dynamic programming -
    dp : stores the array of k max values of sub problem
    dp_chains (list of k lists, each of length t): store the
    4 chain of most-likely states
*/
function Viterbi(model, data):
n, d = data.shape
k = model.k

// emission probabilities (gaussian) for 0th time
e_0 = k_gaussians(data[0], model.mus, model.sigmas)

// DP initializations
dp = log(model.pi) + log(e_0)
dp_chains = [[i] for i in range(k)]

for t in range(1, n):
    // e_t : emission probabilities (gaussian) for step t
    e_t = k_gaussians(data[t], model.mus, model.sigmas)

    argmax_states = argmax(log(model.A.T) + dp , axis=1)

    dp_chains = [copy.deepcopy(dp_chains[ami]) for ami in argmax_states]
    for i in range(k): dp_chains[i].append(i)

    dp = log(e_t) + max( log(model.A.T) + dp , axis=1)


viterbi_decoding = dp_chains[argmax(dp)])
return viterbi_decoding
```

2.8

Figure 3 has the plot of clustering found using the Viterbi decoding.

2.9

Figure 4 has the required plot.

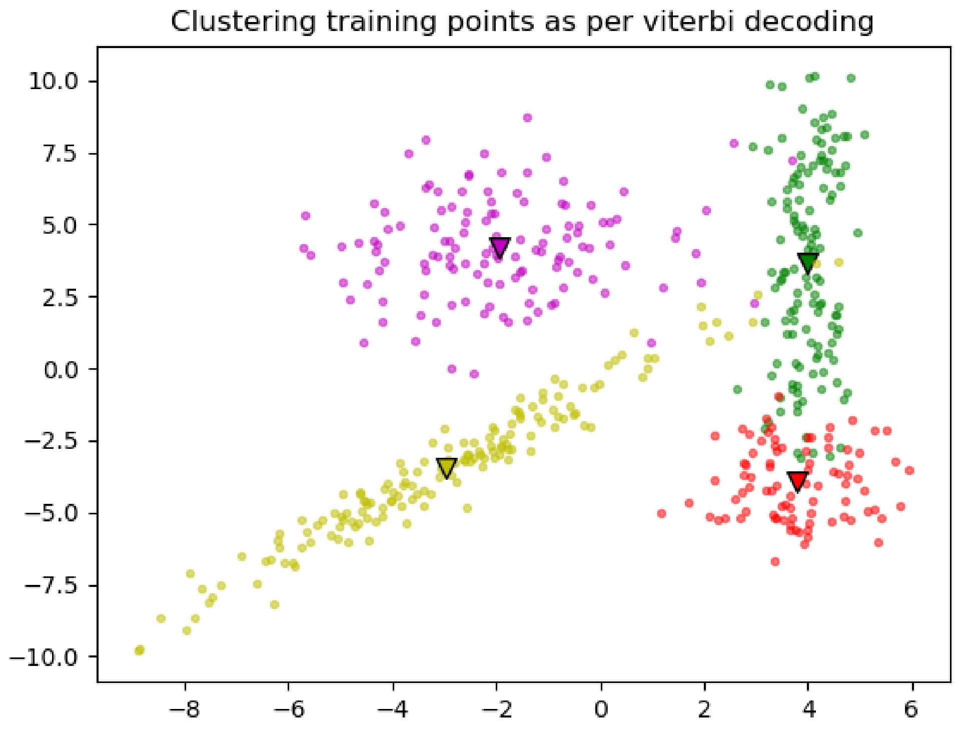

Figure 3: the training data in 2D with the cluster centers and with markers of different colors for the datapoints belonging to different classes.

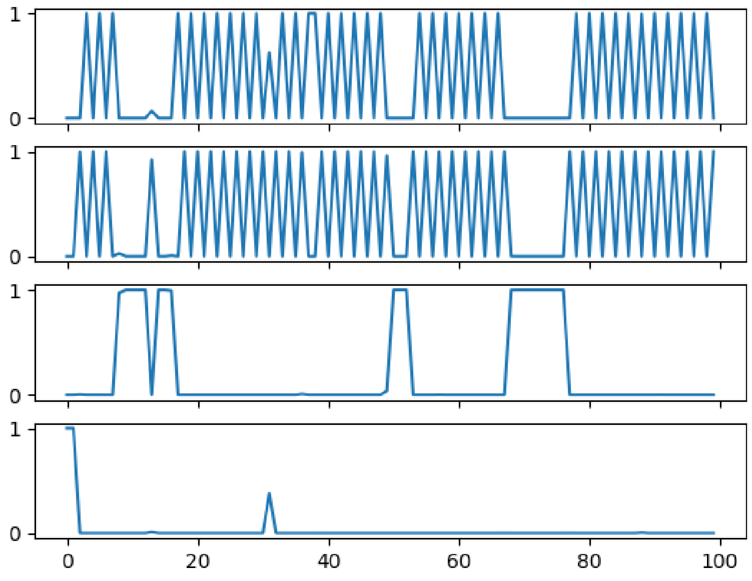

Figure 4: For each state plot the probability of being in that state as a function of time for the 100 first points in test data

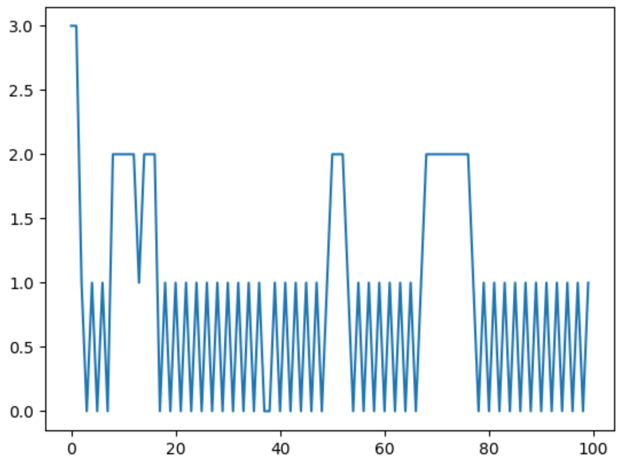

Figure 5: Plot representing the most likely state in {1, 2, 3, 4} as function of time for first 100 points in test data.

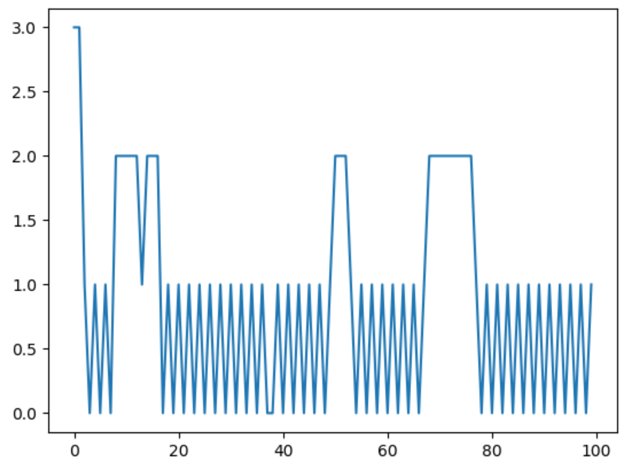

Figure 6: The most likely sequence of states obtained using viterbi decoding for the first 100 test data points.

2.10

2.11

In figure 6, we plot the most likely sequence of states obtained using Viterbi decoding for the first 100 test data points. This is same as the the plot in figure 5.

These two can be same or different. Here we are seeing them to be same for the first 100 states but they may be different.

2.12

We treat K as a hyper-parameter for the model. We tune this by trying different values of K and running EM algorithm to find one that maximizes the likelihood on the training data.