Engineering Assignment Coversheet

Please note that you:

- Must keep a full copy of your submission for this assignment
- Must staple this assignment
- Must NOT use binders or plastic folders except for large assignments

Student Number(s)
1172562 1080982
Group Code (if applicable):

Assignment Title:	ELEN90052 Workshop 2
Subject Number:	ELEN90052
Subject Name:	Advanced Signal Processing
Student Name:	Mukul Chodhary, Joseph Rowan
Lecturer/Tutor:	Prof. Jingge Zhu
Due Date:	24/05/2024

For Late Assignments Only

Has an extension been granted? Yes / No (circle)

A per-day late penalty may apply if you submit this assignment after the due date/extension. Please check with your Department/coordinator for further information.

Plagiarism

Plagiarism is the act of representing as one's own original work the creative works of another, without appropriate acknowledgment of the author or source.

Collusion

Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or in part of unauthorised collaboration with another person or persons. Collusion involves the cooperation of two or more students in plagiarism or other forms of academic misconduct.

Both collusion and plagiarism can occur in group work. For examples of plagiarism, collusion and academic misconduct in group work please see the University's policy on Academic Honesty and Plagiarism: http://academichonesty.unimelb.edu.au/

Plagiarism and collusion constitute cheating. Disciplinary action will be taken against students who engage in plagiarism and collusion as outlined in University policy. Proven involvement in plagiarism or collusion may be recorded on my academic file in accordance with Statute 13.1.18.

STUDENT DECLARATION

Please sign below to indicate that you understand the following statements:

I declare that:

- This assignment is my own original work, except where I have appropriately cited the original source.
- This assignment has not previously been submitted for assessment in this or any other subject.

For the purposes of assessment, I give the assessor of this assignment the permission to:

- Reproduce this assignment and provide a copy to another member of staff; and
- Take steps to authenticate the assignment, including communicating a copy of this assignment to a checking service (which may retain a copy of the assignment on its database for future plagiarism checking).

Student signature Mukul Chodhary, Joseph Rowan ..... Date 24/05/2024 .....

Workshop 2

Mukul Chodhary and Joseph Rowan

Semester 1, 2024

1 Speech data preparation

1.1 Description and organisation of the data

The data is organised into seven classes: "Burger", "Meatball", "Pizza", "Rice", "Sandwich", "Sausage" and "Spaghetti". Each class contains 20 recordings of a male voice saying the associated English word. The recordings are 1 s long, sampled at 48 kHz and stored in the MP4 audio format as `.m4a` files. For any recording that is not exactly 1 s long, we trim the audio or add zero-padding as required.

1.2 Collection of custom data

We collect a further 20 samples for each class, with four different people, two male and two female, speaking the seven different words. The samples are cropped to remove silence at the beginning of the recording. As before, recordings longer than or shorter than 1 s are adjusted using cropping or zero-padding as appropriate at load time. These recordings are stored in a separate folder mirroring the organisation of the audio data provided in the course materials.

2 Feature extraction

There are several hyperparameters that need to be specified when generating the MFCC features:

1. Frame length: This controls the number of samples in each frame of the input audio signal and therefore the total number of frames in the MFFC features, i.e. the dimension N. Since the DFT of each frame is to be found separately, the choice of frame length is a trade-off between temporal resolution and frequency resolution. A shorter frame length provides better resolution in the time axis, since N is larger. On the other hand, it reduces the number of samples over which the DFT is calculated, making the frequency spacing larger. In this workshop, we use an intermediate value of 1440 samples, which is equal to $0.03 f_s$ where $f_s = 48\,\text{kHz}$.
2. Frame overlap: Because the window function decreases to zero at its edges, consecutive frames are usually overlapped before the DFT computation to avoid losing information at the beginning and end of each frame. Increasing the overlap increases the number of frames N, though if it becomes too large then the difference between the MFCCs of consecutive frames will be small. Large overlaps also create a higher computational cost because more frames need to be processed while the number of DFT points remains constant. Here, we use a large overlap of $0.02 f_s$, or 960 samples, to prevent distortions around the frame edges.
3. Window type: A smooth window function is usually applied to avoid edge effects when taking the DFT of each frame. Typical window functions are Hann or Hamming windows. In particular, we use a periodic Hann window of length 1440 samples, i.e. the same as the frame length, for the assignment tasks.
4. DFT length: The number of DFT points can be increased independently of the frame length by zero-padding. This gives a smoother DFT spectrum, equivalent to sinc interpolation, but it does not increase the amount of useful information. Therefore, the DFT length is usually the same as the frame length, often padded to the nearest power of two for computational efficiency. This gives a DFT length of 2048 in our case.

5. Frequency bands of the Mel filter bank: The Mel filter bank can be modified to have different frequency edges. We use the standard formulation from MATLAB which approximates human auditory perception with a logarithmic spacing between 133 and 6864 Hz.
6. Number of coefficients: The number of MFCC coefficients, i.e the dimension L of the feature bank, is determined after taking the discrete cosine transform (DCT) of the raw Mel features. Increasing the number of coefficients retains more information from the DCT step, but typical audio signals have most of their useful information concentrated in the lower-order few coefficients. Higher-order coefficients are likely to contain mostly noise, so increasing L past a certain point will not aid classification and may reduce performance instead by increasing the dimensionality of the feature space and making training more difficult. As specified in the assignment, we initially choose $L = 14$, although slightly smaller values of L would likely also work well.

3 Model training

We randomly split the extracted features into train and test data to train and evaluate a hidden Markov process (HMP) model for each class. The parameters that need to be estimated for each HMP model are

$$\lambda = (A, \pi_1, \Sigma_1, \mu_1, \ldots, \pi_K, \Sigma_K, \mu_K)$$

with transition matrix A, initial state probabilities $\pi_1, \ldots \pi_K$, state mean vectors $\mu_1, \ldots, \mu_K$ and state covariance matrices $\Sigma_1, \ldots, \Sigma_K$.

Algorithm 1 provides an overall structure to train the HMP associated with each class C. The expectation-maximisation (EM) step relies on the following posterior probabilities:

1. $\gamma_t(x_t) = p(x_t \mid y^n)$ for $t = 1, \ldots, N$ and for all $x_t \in \{1, \ldots, K\}$.
2. $\xi_{t,t+1}(x_t, x_{t+1}) = p(x_{t-1}, x_t \mid y^n)$ for $t = 1, \ldots, N-1$ and for all $x_t, x_{t+1} \in \{1, \ldots, K\}$.

These are calculated with the forward-backward algorithm via the intermediate quantities:

1. $\alpha_t(x_t) = p(x_t \mid y^t)$.
2. $\beta_t(x_t) = p(x_t \mid y^{t-1})$.

We also define $\gamma_t^{i,(r)} = p(x_t = i \mid y^n, \lambda^{(r)})$ and $\xi_{t,t+1}^{i,j,(r)} = p(x_t = i, x_{t+1} = j \mid y^n, \lambda^{(r)})$, i.e. the relevant quantities calculated using the estimated parameters $\lambda^{(r)}$, for convenience in the formulations below.

We use the sample mean and covariance of all samples in the class's training data to initialise μ_i and Σ_i for each state i before running the first step of the EM algorithm. The transition matrix and initial state probabilities are initialised randomly using a uniform distribution on $[0, 1]$. Before entering the training loop, each initial π_i is then normalised to form a valid probability distribution that sums to one, as are the individual rows of A.

We then train the model until the relative difference between the expected log likelihood for successive iterations is below 10^{-6}. This gives us a more controlled way to identify when the estimated parameters have converged.

The following section describes the mathematical formulation of the steps in the algorithm where we compute the forward recursion, backward recursion, EM parameter update and expected log likelihood. We provide a detailed derivation of all these formulations in the Appendix 1.

3.1 Algorithm implementation

3.1.1 Forward recursion

Initialise $\beta_1(x_1) = p(x_1)$. Then, for $t = 1, \ldots, n$ we compute:

$$\alpha_t(x_t) = \frac{\beta_t(x_t) p(y_t \mid x_t)}{\sum_{x_t'} \beta_t(x_t') p(y_t \mid x_t')}$$

$$\beta_{t+1}(x_{t+1}) = \sum_{x_t} \alpha_t(x_t) p(x_{t+1} \mid x_t)$$

where $p(y_t \mid x_t = i) = \mathcal{N}(\mu_i, \Sigma_i)$ and $p(x_{t+1} = j \mid x_t = i) = a_{ij}$.

Algorithm 1 HMM EM model training loop for one class

Input: y^n
Initialise λ_C using sample mean and covariance
$L \leftarrow \infty$
while $|\Delta L| < 10^{-6}$ **do**
    $\alpha, \beta \leftarrow$ Forward Recursion(y^n, λ_C)
    $\xi, \gamma \leftarrow$ Backward Recursion$(y^n, \lambda_C, \alpha, \beta)$
    $\lambda_C \leftarrow$ Parameter Update$(y^n, \lambda_C, \xi, \gamma)$
    $L \leftarrow$ Log Likelihood(y^n, λ_C)
end while

3.1.2 Backward recursion

Initialise $\gamma_n(x_n) = p(x_n \mid y^n) = \alpha_n(x_n)$ after solving for $\alpha_t(x_t)$ and $\beta_t(x_t)$ for $t = 1, \ldots, n$ using the forward recursion. Then, for $t = n-1, \ldots, 1$ we compute:

$$\begin{aligned}\xi_{t,t+1}(x_t, x_{t+1}) &= \frac{\gamma_{t+1}(x_{t+1})\alpha_t(x_t)p(x_{t+1} \mid x_t)}{\beta_{t+1}(x_{t+1})} \\ \gamma_t(x_t) &= \sum_{x_{t+1}} \xi_{t,t+1}(x_t, x_{t+1}).\end{aligned}$$

3.1.3 EM parameter update

We find the next iteration of the parameters $\lambda = (A, \pi_1, \Sigma_1, \mu_1, \ldots, \pi_K, \Sigma_K, \mu_K)$ using EM as follows.

$$\begin{aligned}a_{ij}^{(r+1)} &= \frac{\sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)}}{\sum_{t=1}^{n-1} \gamma_t^{i,(r)}} \\ \pi_i^{(r+1)} &= \gamma_1^{i,(r)} \\ \mu_i^{(r+1)} &= \frac{\sum_{t=1}^{n} \gamma_t^{i,(r)} y_t}{\sum_{t=1}^{n} \gamma_t^{i,(r)}} \\ \Sigma_i^{(r+1)} &= \frac{\sum_{t=1}^{n} \gamma_t^{i,(r)} (y_t - \mu_i^{(r+1)})(y_t - \mu_i^{(r+1)})^T}{\sum_{t=1}^{n} \gamma_t^{i,(r)}}.\end{aligned}$$

3.1.4 Expected log likelihood

To evaluate the complete log-likelihood $p(x^n \mid y^n)$, we would typically use the Viterbi algorithm. However, this can be computationally expensive, having time complexity $\mathcal{O}(KN^2)$ where K is the number of hidden states and N is the length of the sequence. To avoid this, we use the expected log likelihood $\mathrm{E}[\log p(x^n, y^n \mid \lambda)]$ as the convergence criterion, which is consistent with the EM approach. The expectation is calculated with respect to the posterior distribution $p(x^n \mid y^n, \lambda^{(r)})$ as usual.

The complete log likelihood is

$$\log p(x^n, y^n \mid \lambda) = \log p(x_1 \mid \lambda) + \sum_{t=1}^{n-1} \log p(x_{t+1} \mid x_t, \lambda) + \sum_{t=1}^{n} \log p(y_t \mid x_t, \lambda).$$

and its expectation, which is derived in detail in the appendix, is

$$\begin{aligned}\mathrm{E}[\log p(x^n, y^n \mid \lambda)] = &\sum_{i=1}^{K} \gamma_1^{i,(r)} \log \pi_i + \sum_{i,j=1}^{K} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)} \log a_{ij} \\ &- \frac{1}{2} \sum_{i=1}^{K} \sum_{t=1}^{n} \gamma_t^{i,(r)} ((y_t - \mu_i)^T \Sigma_i^{-1} (y_t - \mu_i) + \log\det(\Sigma_i) + L \log 2\pi).\end{aligned}$$

3.2 Multiple training sequences

In section 3.1, we considered training a HMP for a given class C using EM with one observation sequence y^n. However, in this project we have access to M training sequences, which we denote as $y^{n,1}, \ldots, y^{n,M}$. We need to fit a single model to each of these training sequences. An approach suggested in the workshop notes is to train individual models for each sequence to obtain the fitted parameters $\lambda_C^1, \ldots, \lambda_C^M$. The final parameters for class C are then taken as the mean

$$\lambda_C = \frac{1}{M} \sum_{m=1}^{M} \lambda_C^m.$$

There is however an implementation challenge involved with this approach. Because the final assignment of states to phonemes is arbitrary after running the EM algorithm, directly computing average parameters is not meaningful. For example, the phoneme /p/ may be associated with State 1 in one model but State 2 in another. It does not make sense to average both model's State 1 parameters, since they represent different things.

Instead, we make an iid assumption across each training sequence and consider ML estimation across all samples. Then, the complete log likelihood when taking all the samples into account is

$$\begin{aligned}
\log p(x^{n,1}, y^{n,1}, \ldots, x^{n,M}, y^{n,M} \mid \lambda) &= \sum_{m=1}^{M} \log p(x^{n,m}, y^{n,m} \mid \lambda) \\
&= \sum_{m=1}^{M} \log p(x_1^m \mid \lambda) + \sum_{m=1}^{M} \sum_{t=1}^{n-1} \log p(x_{t+1}^m \mid x_t^m, \lambda) \\
&\quad + \sum_{m=1}^{M} \sum_{t=1}^{n} \log p(y_t^m \mid x_t^m, \lambda).
\end{aligned}$$

Let us define $\gamma_t^{i,m,(r)} = p(x_t^m = i \mid y^{n,m}, \lambda^{(r)})$ and $\xi_{t,t+1}^{i,j,m,(r)} = p(x_t^m = i, x_{t+1}^m = j \mid y^{n,m}, \lambda^{(r)})$. The expectation is then taken with respect to the joint posterior $p(x^{n,1}, \ldots, x^{n,M} \mid y^{n,1}, \ldots, y^{n,M}, \lambda^{(r)})$, which again decomposes over $m = 1, \ldots, M$ to get

$$\begin{aligned}
\mathrm{E}[\log p(x^{n,1}, y^{n,1}, \ldots, x^{n,M}, y^{n,M} \mid \lambda)] &= \sum_{m=1}^{M} \sum_{i=1}^{K} \gamma_1^{i,m,(r)} \log \pi_i + \sum_{m=1}^{M} \sum_{i,j=1}^{K} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,m,(r)} \log a_{ij} \\
&\quad - \frac{1}{2} \sum_{m=1}^{M} \sum_{i=1}^{K} \sum_{t=1}^{n} \gamma_t^{i,m,(r)} ((y_t^m - \mu_i)^T \Sigma_i^{-1} (y_t^m - \mu_i) \\
&\quad + \log \det(\Sigma_i) + L \log 2\pi).
\end{aligned}$$

The EM algorithm is then similar to the case with a single training sequence. Using the forward-backward algorithm on each sequence individually, we first calculate

1. $\gamma_t^{i,m,(r)}$ for $m = 1, \ldots, M$, $t = 1, \ldots, n$ and for all $i \in \{1, \ldots, K\}$.
2. $\xi_{t,t+1}^{i,j,m,(r)}$ for $m = 1, \ldots, M$, $t = 1, \ldots, n-1$ and for all $i, j \in \{1, \ldots, K\}$.

Then, following the general outline of the derivations in the appendix, the EM parameter update equations are

$$\begin{aligned}
a_{ij}^{(r+1)} &= \frac{\sum_{m=1}^{M} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,m,(r)}}{\sum_{m=1}^{M} \sum_{t=1}^{n-1} \gamma_t^{i,m,(r)}} \\
\pi_i^{(r+1)} &= \frac{1}{M} \sum_{m=1}^{M} \gamma_1^{i,m,(r)} \\
\mu_i^{(r+1)} &= \frac{\sum_{m=1}^{M} \sum_{t=1}^{n} \gamma_t^{i,m,(r)} y_t}{\sum_{m=1}^{M} \sum_{t=1}^{n} \gamma_t^{i,m,(r)}} \\
\Sigma_i^{(r+1)} &= \frac{\sum_{m=1}^{M} \sum_{t=1}^{n} \gamma_t^{i,m,(r)} (y_t^m - \mu_i^{(r+1)})(y_t^m - \mu_i^{(r+1)})^T}{\sum_{m=1}^{M} \sum_{t=1}^{n} \gamma_t^{i,m,(r)}}.
\end{aligned}$$

4 Model evaluation

4.1 Inference procedure for multi-class classification

Consider a trained model M_C for class C. To run inference with M_C for observation y^n, we compute the expected log likelihood

$$M_C(y^n) = \mathrm{E}[\log p(x^n, y^n \mid \lambda_C)].$$

We now consider $M = \{M_1, \ldots, M_J\}$, which is a set of trained models for all classes $C = 1, \ldots, J$. The inference with M on y^n is

$$M(y^n) = \{M_1(y^n), \ldots, M_J(y^n)\}$$

and we classify y^n as belonging to the class that maximises the expected log likelihood using the maximum likelihood principle,

$$\hat{C}(y^n) = \underset{1 \leq C \leq J}{\arg\max}\, M(y^n).$$

4.2 Performance metrics

Standard performance metrics are shown for the HMP model in table 1 and the confusion matrix is shown in figure 1. We can see that the classifier performs very well on the provided test set with an accuracy of 0.94. From the confusion matrix, we see that the classifier struggled to identify the word "Pizza", which was once misclassified as "Rice". Depending on how these words are pronounced, they may contain similar sound sequences, causing the trained HMPs for the two associated classes to assign similar log likelihoods to the test input sequence. This may lead to erroneous outputs when choosing the most likely class.

Class	**Precision**	**Recall**	**F1 score**	**Support**
Burger	1.00	1.00	1.00	5
Meatball	1.00	1.00	1.00	5
Pizza	0.80	1.00	0.89	5
Rice	1.00	0.83	0.91	5
Sandwich	1.00	1.00	1.00	5
Sausage	1.00	1.00	1.00	5
Spaghetti	1.00	1.00	1.00	5
Accuracy	0.97			
Macro average	0.97	0.98	0.97	35
Weighted average	0.97	0.98	0.97	35

Table 1: HMP classification report using the original dataset

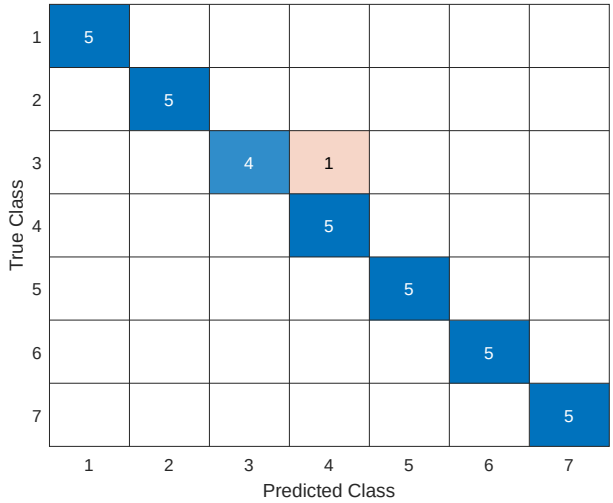

Figure 1: Confusion matrix for HMP using the original dataset

We next tested the trained model on different data, which we gathered ourselves as described in section 1.2. The tone was significantly different to the provided audio recordings, containing both male and female voices with a variety of accents. Unfortunately, the original trained model did not generalise well to these new inputs and the classification performance was poor, as shown in table 2 and figure 2. This was likely the result of the model overfitting on the provided training data, becoming too reliant on the specific allophones, accent or cadence of speech enunciated by the male speaker for making classification decisions.

Class	**Precision**	**Recall**	**F1 score**	**Support**
Burger	0.00	0.00	0.00	5
Meatball	0.00	0.00	0.00	5
Pizza	0.00	0.00	0.00	5
Rice	1.00	0.22	0.36	5
Sandwich	1.00	0.46	0.63	5
Sausage	0.00	0.00	0.00	5
Spaghetti	0.20	1.00	0.33	5
Accuracy	0.31			
Macro average	0.31	0.24	0.19	35
Weighted average	0.31	0.24	0.19	35

Table 2: HMP classification report using the custom dataset

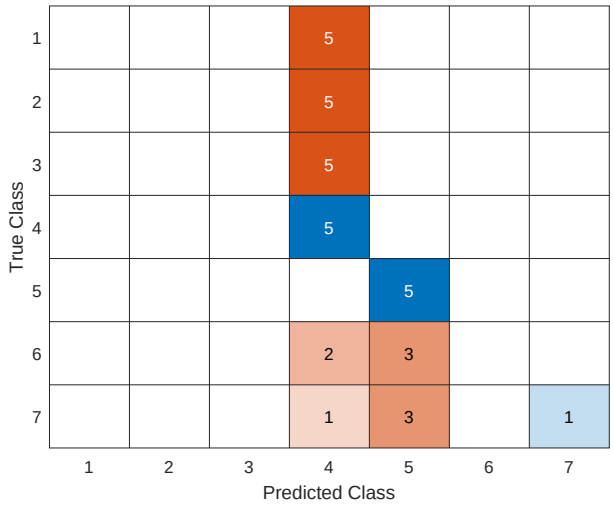

Figure 2: Confusion matrix for HMP using the custom dataset

One simple way to improve the generalisation ability of the classifier is to increase the amount of training data. Instead of training the classifier on the one male speaker only, we can use all five available people, who have a variety of accents. If we optimise the HMP models for all the classes with EM across the ensemble of these voices, they will be less likely to overfit to the specific features of one individual's speech patterns. Using this approach, we obtain better accuracy on the custom test data, even though the original male voice is still overrepresented in the dataset with 20 examples per class compared to just 5 for the new voices. The revised classification results with HMP models are shown in table 3 and figure 3. There were still difficulties when classifying certain words, in particular "Rice". We noted that the four speakers pronounced this word quite differently, which may have prevented the HMP from converging on a good representation of the related sound sequence and, limiting the classification accuracy. We also notice that because the model is no longer overfitting, the test set accuracy is not as high as it was with the original data.

Class	Precision	Recall	F1 score	Support
Burger	0.60	1.00	0.75	5
Meatball	0.80	0.67	0.73	5
Pizza	0.00	0.00	0.00	5
Rice	1.00	1.00	1.00	5
Sandwich	0.60	1.00	0.75	5
Sausage	0.60	0.75	0.67	5
Spaghetti	0.80	0.40	0.53	5
Accuracy	0.63			
Macro average	0.63	0.69	0.63	35
Weighted average	0.63	0.69	0.63	35

Table 3: HMP classification report using the custom dataset and more training data

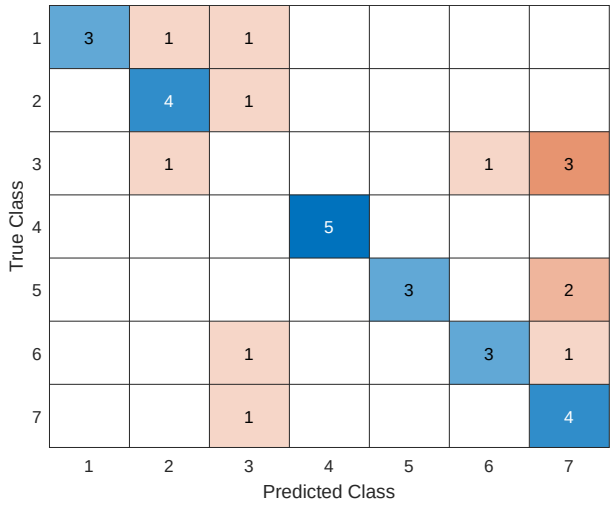

Figure 3: Confusion matrix for HMP using the custom dataset and more training data

5 Open-ended questions

5.1 Tuning model hyperparameters

We now consider the effect of the following hyperparameters:

1. N, the number of frames in each observation. This is controlled by changing the frame width during the DFT stage of MFCC generation.
2. L, the number of MFCC features per frame, which is equal to the number of retained DFT coefficients.
3. K, the number of states in the HMP. This is a direct parameter of the trained model.

The original parameters were $N = 98$, $L = 14$ and $K = 9$.

5.1.1 Tuning N

Each state in the HMP, which represents a phoneme, occupies a time period defined by the frame width. So, we should try to make the frame width similar to the typical length of a phoneme. Unfortunately, this is difficult to determine exactly, since people may speak at different speeds, even within the same word. The current value of N arises from a frame width of 1440 samples, as described in section 2, corresponding to 30 ms. This is somewhat shorter than the typical phoneme length in English, which is closer to 100 ms, so there may be room to increase the frame size. As well as modelling actual speech more closely without overfitting, this would decrease N and therefore the computational complexity since

each training sequence would be shorter. We do not try frame lengths shorter than 1440 samples since 30 ms is already short compared to most phonemes, and much shorter frame lengths are incompatible with the default settings of MATLAB's MFCC generation that were consistently used throughout our other tests.

However, we must be careful to ensure the frame size is not so large that shorter phonemes are modelled inaccurately. For this experiment, we try lengths of around 20, 30, 60 and 100 ms, corresponding to frame widths of 960, 1440, 2880 and 4800 samples. The resulting values of N are 98, 24 and 12, using MATLAB default settings for the other aspects of MFCC generation. Results are shown in tables 4 and 5. We observe that the largest value of $N = 98$ gave consistently better results in all the training situations, although these models took substantially longer to converge compared to the other scenarios. We can therefore conclude that using frame sizes shorter than average phonemes is not disadvantageous, though it is more computationally expensive. There is also a large performance drop when N is too small, likely because the HMP cannot capture the complexity of the speech pattern in such cases.

Class	$M = 98$	$M = 24$	$M = 12$
Burger	41	22	8
Meatball	80	28	6
Pizza	60	10	11
Rice	27	21	23
Sandwich	111	33	20
Sausage	500	32	11
Spaghetti	68	27	13
Average	126.71	24.71	13.14

Table 4: Training iterations for different values of M

Test, Train	$M = 98$	$M = 24$	$M = 12$
Provided, Provided	0.97	0.74	0.57
Custom, Provided	0.29	0.26	0.20
Custom, All	0.74	0.66	0.44
All, All	0.81	0.67	0.47
Average	0.70	0.58	0.42

Table 5: Overall accuracy for different values of M

5.1.2 Tuning L

As mentioned in section 2, we expect intermediate values for L to be most appropriate because higher-order coefficients are more likely to be dominated by noise, whereas the signal will not be represented accurately if only the lowest-order coefficients are retained. Perhaps more importantly, L is also the dimensionality of the Gaussian distributions in the HMP. If L is large, the dimension of the feature space increases and it becomes more difficult to estimate the correct Gaussian mean and covariance parameters, especially with limited training data.

Typical values are $L = 12$ or $L = 13$. We test values around this region, retaining $K = 9$ and $N = 98$ for simplicity. In particular, we try $L = 9$, 12, 14 and 16, recording both convergence speed and classification performance. The results are shown in tables 6 and 7. It appears that changing L does not have a noticeable effect on the convergence time of the algorithm, which suggests that EM is effective at estimating the parameters of higher dimensional Gaussian distributions. On the other hand, larger values of L provide better classification performance since less complexity is discarded during the preprocessing stage as discussed in section 2. We achieve the best average accuracy for $L = 16$, although the slightly worse performance when generalising from the provided to the custom dataset, dropping from 0.31 to 0.29 compared with $L = 14$, suggests overfitting is a problem if L, and therefore the the model dimensionality, is too high. Therefore, $L = 14$ is the most balanced choice overall.

Class	$L = 9$	$L = 12$	$L = 14$	$L = 16$
Burger	155	118	49	83
Meatball	123	73	34	40
Pizza	55	43	110	20
Rice	66	47	40	87
Sandwich	105	68	157	86
Sausage	43	101	71	156
Spaghetti	136	70	57	130
Average	97.57	74.29	74.00	86.00

Table 6: Training iterations for different values of L

Test, Train	$L = 9$	$L = 12$	$L = 14$	$L = 16$
Provided, Provided	0.91	0.89	0.91	0.94
Custom, Provided	0.14	0.26	0.31	0.29
Custom, All	0.71	0.71	0.74	0.80
All, All	0.76	0.76	0.84	0.84
Average	0.63	0.65	0.70	0.72

Table 7: Overall accuracy for different values of L

5.1.3 Tuning K

We next consider the hyperparameter K, which is the number of hidden states in our implementation. K intuitively represents the phonemes in each word. Therefore, we can tune this parameter for each class instead of setting a constant $K = 9$ for all classes. It can be seen that the ideal K_C for a class C, matching the number of phonemes in the word, cannot be larger than the number of letters in the word, which we denote as $|C|$, because a phoneme requires at least one letter. A more robust approach would evaluate the exact number of phonemes for each class instead of using this loose upper bound, but this might be more difficult when accounting for different accents and pronunciations.

Therefore, we use three approaches to tune K:

1. Method 1: Constant for all classes, $K = 9$.
2. Method 2: Varying, $K_C = |C|$.
3. Method 3: Varying, $K_C = \lceil 1.5|C| \rceil$.

To evaluate the performance of these methods, we consider the number of EM iterations before convergence and the overall accuracy on the provided and custom datasets.

We measure the convergence rate based on the number of iterations it took for a model to finish training for each class, keeping $N = 98$ and $L = 14$. These results are provided in table 8. We observed Method 2 converges roughly 1.5 times faster than Method 1, while Method 3 converges 1.4 times faster. Therefore, choosing a constant K is not ideal for quick model fitting.

Accuracies are shown in table 9, where we observe that Method 3 has the highest overall accuracy, followed closely by Method 1, while Method 2 has the lowest overall accuracy. In particular, Method 1 has the highest accuracy when testing solely on the provided dataset, which perhaps indicates overfitting, while Method 3 works best in the other configurations.

We can conclude that Method 3 has the best overall performance and balances accuracy with computational efficiency. Indeed, having too may states for a given word may quickly lead to overfitting, whereas the HMP will not be able to describe the true sequence of phonemes if K is too small. Therefore, better performance and training speed is achieved by tailoring K to the approximate number of phonemes in each word class, leaving a small overhead to ensure a good fit is achievable.

Class	Method 1	Method 2	Method 3
Burger	45	34	36
Meatball	65	26	52
Pizza	23	36	62
Rice	90	34	28
Sandwich	43	76	60
Sausage	127	27	33
Spaghetti	43	50	40
Average	62.29	40.43	44.43

Table 8: Training iterations for different methods for choosing K

Test, Train	Method 1	Method 2	Method 3
Provided, Provided	0.94	0.91	0.89
Custom, Provided	0.29	0.26	0.31
Custom, Trained	0.63	0.46	0.71
All, All	0.73	0.63	0.84
Average	0.65	0.56	0.69

Table 9: Accuracy comparison across methods for choosing K

5.2 Comparison to other ML models

We evaluate SVM and Naïve Bayes learning algorithms on the original dataset, as shown in tables 10 and 11. These models were trained on the existing MFCC features with $L = 14$. We observe a slightly lower accuracy of 0.89 for both these methods, which may occur because the HMP with Gaussian observations is an intuitively better description of the underlying phonological model. Our HMP classifier also benefits from having a separate trained model for each class, while most other classification approaches learn a single model that must account for the details of each class simultaneously. Therefore, the SVM and Naïve Bayes classifiers must learn a more complex representation compared to each HMP in the ensemble.

There are some distinctive advantages and disadvantages of EM algorithms, in particular HMPs, compared to the SVM and Naïve Bayes approaches. First, EM algorithms always increase the likelihood at each step, putting aside numerical issues. Therefore, they can be guaranteed to converge to a local optimum. On the other hand, methods such as SVM are based on convex optimisation and therefore have stronger optimality guarantees, converging to the global optimum in any case. More complex algorithms such as deep neural networks might not have any convergence guarantees at all.

EM algorithms also have the advantage of handling latent variables. This makes it a versatile technique for fitting more complex models such as HMPs that better describe underlying phenomena, compared to the more simple SVM and Naïve Bayes models. A final disadvantage of EM algorithms is that convergence is often slow. For some of our models, up to 500 EM steps were required to reach the convergence threshold of 10^{-6} relative difference between iterations. This manifested as slower training times compared to the other machine learning techniques explored in this section.

Class	Precision	Recall	F1 score	Support
Burger	1.00	1.00	1.00	7
Meatball	1.00	1.00	1.00	4
Pizza	1.00	1.00	1.00	3
Rice	1.00	1.00	1.00	5
Sandwich	0.43	1.00	0.60	3
Sausage	1.00	0.56	0.71	9
Spaghetti	1.00	1.00	1.00	4
Accuracy	0.89			
Macro average	0.92	0.94	0.90	35
Weighted average	0.95	0.89	0.89	35

Table 10: SVM classification report

Class	Precision	Recall	F1 score	Support
Burger	1.00	1.00	1.00	7
Meatball	1.00	0.75	0.86	4
Pizza	1.00	1.00	1.00	3
Rice	1.00	1.00	1.00	5
Sandwich	0.43	1.00	0.60	3
Sausage	1.00	0.67	0.80	9
Spaghetti	1.00	1.00	1.00	4
Accuracy	0.89			
Macro average	0.92	0.92	0.89	35
Weighted average	0.95	0.89	0.90	35

Table 11: Naïve Bayes classification report

6 Appendix 1: Derivation of EM algorithm for HMPs

In this section we derive the EM algorithm for HMPs. The model parameters are

$$\lambda = (A, \pi_1, \Sigma_1, \mu_1, \ldots, \pi_K, \Sigma_K, \mu_K).$$

with transition matrix A, initial state probabilities $\pi_1, \ldots \pi_K$, state mean vectors $\mu_1, \ldots, \mu_K$ and state covariance matrices $\Sigma_1, \ldots, \Sigma_K$. For convenience, we also define the posterior distributions $\alpha_t(x_t) = p(x_t \mid y^t)$, $\beta_t(x_t) = p(x_t \mid y^{t-1})$, $\gamma_t(x_t) = p(x_t \mid y^n)$ and $\xi_{t-1,t}(x_{t-1}, x_t) = p(x_{t-1}, x_t \mid y^n)$, together with $\gamma_t^{i,(r)} = p(x_t = i \mid y^n, \lambda^{(r)})$ and $\xi_{t,t+1}^{i,j,(r)} = p(x_t = i, x_{t+1} = j \mid y^n, \lambda^{(r)})$. The joint distribution of the HMP can be factored as

$$\begin{aligned} p(x^n, y^n) &= \prod_{t=1}^{n} p(x_t \mid x_{t-1}) p(y_t \mid x_t) \\ &= p(x_1) \prod_{t=2}^{n} p(x_t \mid x_{t-1}) \prod_{t=1}^{n} p(y_t \mid x_t) \\ &= p(x_1) \prod_{t=1}^{n-1} p(x_{t+1} \mid x_t) \prod_{t=1}^{n} p(y_t \mid x_t). \end{aligned}$$

From this, we can write the log likelihood function

$$\log p(x^n, y^n \mid \lambda) = \log p(x_1 \mid \lambda) + \sum_{t=1}^{n-1} \log p(x_{t+1} \mid x_t, \lambda) + \sum_{t=1}^{n} \log p(y_t \mid x_t, \lambda). \tag{1}$$

6.1 Expectation and maximisation

To carry out EM, we need to take the expectation of each term with respect to the posterior distribution $p(x^n \mid y^n, \lambda^{(r)})$. We can write the first term in equation (1) as

$$\log p(x_1 \mid \lambda) = \sum_{i=1}^{K} \mathbf{1}\{x_1 = i\} \log \pi_i.$$

Taking the expectation over x^n,

$$\begin{aligned} \mathrm{E}\left[\sum_{i=1}^{K} \mathbf{1}\{x_1 = i\} \log \pi_i\right] &= \sum_{x_1} p(x_1 \mid y^n, \lambda^{(r)}) \sum_{i=1}^{K} \mathbf{1}\{x_1 = i\} \log \pi_i \\ &= \sum_{i=1}^{K} \gamma_1^{i,(r)} \log \pi_i. \end{aligned}$$

We then pose the maximisation problem

$$\begin{aligned} &\text{maximise} \ \sum_{i=1}^{K} \gamma_1^{i,(r)} \log \pi_i \\ &\text{subject to} \ \sum_{i=1}^{K} \pi_i = 1. \end{aligned}$$

This can be solved using the method of Lagrange multipliers. The Lagrangian is

$$L(\pi_1, \ldots, \pi_K, \eta) = \sum_{i=1}^{K} \gamma_1^{i,(r)} \log \pi_i + \eta \left(\sum_{i=1}^{K} \pi_i - 1 \right).$$

Setting the derivative of L with respect to π_i to zero, we get

$$\frac{dL}{d\pi_i} = \frac{\gamma_1^{i,(r)}}{\pi_i} + \eta = 0 \implies \pi_i = -\frac{\gamma_1^{i,(r)}}{\eta}.$$

Applying the original equality constraint gives

$$\sum_{i=1}^{K} \pi_i = -\sum_{i=1}^{K} \frac{\gamma_1^{i,(r)}}{\eta} = 1 \implies \eta = -\sum_{i=1}^{K} \gamma_1^{i,(r)} = -1.$$

Combining, we obtain an iterative formula for π_i,

$$\pi_i^{(r+1)} = \gamma_1^{i,(r)}$$

We now consider the second term in equation (1), which can be written as

$$\sum_{t=1}^{n-1} \log p(x_{t+1} \mid x_t, \lambda) = \sum_{t=1}^{n-1} \sum_{i,j=1}^{K} \mathbf{1}\{x_t = i, x_{t+1} = j\} \log a_{ij}.$$

Taking the expectation over x^n,

$$\begin{aligned} \mathrm{E}\left[\sum_{t=1}^{n-1} \sum_{i,j=1}^{K} \mathbf{1}\{x_t = i, x_{t+1} = j\} \log a_{ij}\right] &= \sum_{t=1}^{n-1} \sum_{x_t, x_{t+1}} p(x_t, x_{t+1} \mid y^n, \lambda^{(r)}) \sum_{i,j=1}^{K} \mathbf{1}\{x_t = i, x_{t+1} = j\} \log a_{ij} \\ &= \sum_{i,j=1}^{K} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)} \log a_{ij}. \end{aligned}$$

Again, we now have the constrained maximisation problem

$$\begin{aligned} &\text{maximise} \ \sum_{i,j=1}^{K} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)} \log a_{ij} \\ &\text{subject to} \ \sum_{j=1}^{K} a_{ij} = 1, \ i = 1, \ldots, K. \end{aligned}$$

Writing out the Lagrangian, we get

$$L(A, \eta_1, \ldots, \eta_K) = \sum_{i,j=1}^{K} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)} \log a_{ij} + \sum_{i=1}^{K} \eta_i \left(\sum_{j=1}^{K} a_{ij} - 1 \right).$$

Setting the derivative of L with respect to a_{ij} to zero, we get

$$\frac{dL}{da_{ij}} = \frac{1}{a_{ij}} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)} + \eta_i = 0 \implies a_{ij} = -\frac{1}{\eta_i} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)}.$$

Applying the original equality constraint gives

$$\sum_{j=1}^{K} a_{ij} = -\frac{1}{\eta_i} \sum_{j=1}^{K} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)} = 1 \implies \eta_i = -\sum_{j=1}^{K} \sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)} = -\sum_{t=1}^{n-1} \gamma_t^{i,(r)}$$

since

$$\begin{aligned} \sum_{j=1}^{K} \xi_{t,t+1}^{i,j,(r)} &= \sum_{j=1}^{K} \mathrm{P}(x_t = i, x_{t+1} = j \mid y^n, \lambda^{(r)}) \\ &= \mathrm{P}(x_t = i \mid y^n, \lambda^{(r)}) \\ &= \gamma_t^{i,(r)}. \end{aligned}$$

Combining, we obtain an iterative formula for a_{ij},

$$a_{ij}^{(r+1)} = \frac{\sum_{t=1}^{n-1} \xi_{t,t+1}^{i,j,(r)}}{\sum_{t=1}^{n-1} \gamma_t^{i,(r)}}$$

Finally, we consider the third term in equation (1), which can be written as

$$\sum_{t=1}^{n} \log p(y_t \mid x_t, \lambda) = \sum_{t=1}^{n} \sum_{i=1}^{K} \mathbf{1}\{x_t = i\} \log \mathcal{N}(y_t \mid \mu_i, \Sigma_i)$$

Taking the expectation over x^n,

$$\begin{aligned} \mathrm{E}\left[\sum_{t=1}^{n} \sum_{i=1}^{K} \mathbf{1}\{x_t = i\} \log \mathcal{N}(y_t \mid \mu_i, \Sigma_i) \right] &= \sum_{t=1}^{n} \sum_{x_t} p(x_t \mid y^n, \lambda^{(r)}) \sum_{i=1}^{K} \mathbf{1}\{x_t = i\} \log \mathcal{N}(y_t \mid \mu_i, \Sigma_i) \\ &= \sum_{i=1}^{K} \sum_{t=1}^{n} \gamma_t^{i,(r)} \log \mathcal{N}(y_t \mid \mu_i, \Sigma_i) \\ &= -\frac{1}{2} \sum_{i=1}^{K} \sum_{t=1}^{n} \gamma_t^{i,(r)} ((y_t - \mu_i)^T \Sigma_i^{-1} (y_t - \mu_i) \\ &\quad + \log \det(\Sigma_i) + L \log 2\pi). \end{aligned}$$

This time, we have an unconstrained maximisation problem, which can be stated as

$$\text{maximise } J(\mu_1, \ldots, \mu_K, \Sigma_1, \ldots, \Sigma_K)$$

where

$$J(\mu_1, \ldots, \mu_K, \Sigma_1, \ldots, \Sigma_K) = \sum_{i=1}^{K} \sum_{t=1}^{n} \gamma_t^{i,(r)} ((y_t - \mu_i)^T \Sigma_i^{-1} (y_t - \mu_i) + \log \det(\Sigma_i)).$$

Assuming Σ is positive definite, Σ^{-1} is also positive definite and the optimisation problem is concave. Therefore, the maximum is attained when the derivatives of J with respect to μ_i and Σ_i are zero. First considering μ_i, we have

$$\frac{dJ}{d\mu_i} = 2 \sum_{t=1}^{n} \gamma_t^{i,(r)} \Sigma_i^{-1} (\mu_i - y_t) = 0$$

from which the iteration for μ_i is

$$\mu_i^{(r+1)} = \frac{\sum_{t=1}^n \gamma_t^{i,(r)} y_t}{\sum_{t=1}^n \gamma_t^{i,(r)}}.$$

To find the derivative of J with respect to Σ_i, we will use the following facts. For an invertible matrix X and vectors a and b,

$$\frac{d}{dX}(a^T X^{-1} b) = -(X^{-1})^T a b^T (X^{-1})^T$$

and

$$\frac{d}{dX}(\log|\det(X)|) = (X^{-1})^T.$$

Since the covariance matrix Σ_i is symmetric and positive semidefinite, we see that

$$\frac{d}{d\Sigma_i}((y_t - \mu_i)^T \Sigma_i^{-1}(y_t - \mu_i)) = -\Sigma_i^{-1}(y_t - \mu_i)(y_t - \mu_i)^T \Sigma_i^{-1}$$

and

$$\frac{d}{d\Sigma_i}(\log\det(\Sigma_i)) = \Sigma_i^{-1}$$

from which we can finally write the complete derivative

$$\frac{dJ}{d\Sigma_i} = \sum_{t=1}^n \gamma_t^{i,(r)}(\Sigma_i^{-1} - \Sigma_i^{-1}(y_t - \mu_i)(y_t - \mu_i)^T \Sigma_i^{-1}) = 0$$

and find the iterative update for Σ_i,

$$\Sigma_i^{(r+1)} = \frac{\sum_{t=1}^n \gamma_t^{i,(r)}(y_t - \mu_i^{(r+1)})(y_t - \mu_i^{(r+1)})^T}{\sum_{t=1}^n \gamma_t^{i,(r)}}.$$

6.2 Forward-backward algorithm

The updating equations found in the previous section require:

1. $\gamma_t(x_t)$ for $t = 1, \ldots, N$ and for all $x_t \in \{1, \ldots, K\}$.
2. $\xi_{t,t+1}(x_t, x_{t+1})$ for $t = 1, \ldots, N-1$ and for all $x_t, x_{t-1} \in \{1, \ldots, K\}$.

These are efficiently calculated using the forward-backward algorithm via the intermediate quantities $\alpha_t(x_t)$ and $\beta_t(x_t)$ as follows. We first consider the calculation of $\alpha_t(x_t)$, which is given by

$$p(x_t \mid y^t) = \frac{p(x_t, y^t)}{\sum_{x_t'} p(x_t', y^t)}.$$

Expanding the numerator,

$$\begin{aligned} p(x_t, y^t) &= p(x_t, y_t, y^{t-1}) \\ &= p(x_t, y^{t-1}) p(y_t \mid x_t, y^{t-1}) \\ &= p(y^{t-1}) p(x_t \mid y^{t-1}) p(y_t \mid x_t) \end{aligned}$$

where in the last step we have $p(y_t \mid x_t, y^{t-1}) = p(y_t \mid x_t)$ due to the memoryless channel property of the HMP. We then have

$$p(x_t, y^t) = \frac{p(x_t \mid y^{t-1}) p(y_t \mid x_t)}{\sum_{x_t'} p(x_t' \mid y^{t-1}) p(y_t \mid x_t')}$$

from which we can apply our definition of $\alpha_t(x_t)$ and $\beta_t(x_t)$ to get

$$\alpha_t(x_t) = \frac{\beta_t(x_t) p(y_t \mid x_t)}{\sum_{x_t'} \beta_t(x_t') p(y_t \mid x_t')}.$$

We next find a formula for $\beta_{t+1}(x_{t+1})$, which is equal to

$$\begin{aligned} p(x_{t+1} \mid y^t) &= \sum_{x_t} p(x_{t+1}, x_t \mid y^t) \\ &= \sum_{x_t} p(x_t \mid y^t) p(x_{t+1} \mid x_t, y^t) \\ &= \sum_{x_t} p(x_t \mid y^t) p(x_{t+1} \mid x_t) \end{aligned}$$

where in the last step we have $p(x_{t+1} \mid x_t, y^t) = p(x_{t+1} \mid x_t)$ due to the Markov property. Therefore,

$$\beta_{t+1}(x_{t+1}) = \sum_{x_t} \alpha_t(x_t) p(x_{t+1} \mid x_t)$$

Next we turn our attention to the calculation of $\gamma_t(x_t)$, which can be written as

$$\begin{aligned} p(x_t \mid y^n) &= \sum_{x_{t+1}} p(x_t, x_{t+1} \mid y^n) \\ &= \sum_{x_{t+1}} p(x_{t+1} \mid y^n) p(x_t \mid x_{t+1}, y^n). \end{aligned}$$

Now we use the Markov property to see that $p(x_t \mid x_{t+1}, y^n) = p(x_t \mid x_{t+1}, y^t)$, and so

$$\begin{aligned} p(x_t \mid y^n) &= \sum_{x_{t+1}} p(x_{t+1} \mid y^n) p(x_t \mid x_{t+1}, y^t) \\ &= \sum_{x_{t+1}} \frac{p(x_{t+1} \mid y^n) p(x_t, x_{t+1} \mid y^t)}{p(x_{t+1} \mid y^t)} \\ &= \sum_{x_{t+1}} \frac{p(x_{t+1} \mid y^n) p(x_t \mid y^t) p(x_{t+1} \mid x_t, y^t)}{p(x_{t+1} \mid y^t)}. \end{aligned}$$

Again we use the Markov property to see that $p(x_{t+1} \mid x_t, y^t) = p(x_{t+1} \mid x_t)$, from which we obtain

$$p(x_t \mid y^n) = \sum_{x_{t+1}} \frac{p(x_{t+1} \mid y^n) p(x_t \mid y^t) p(x_{t+1} \mid x_t)}{p(x_{t+1} \mid y^t)}.$$

Finally, using our definitions of $\gamma_t(x_t)$, $\alpha_t(x_t)$ and $\beta_t(x_t)$, we see that

$$\gamma_t(x_t) = \sum_{x_{t+1}} \frac{\gamma_{t+1}(x_{t+1}) \alpha_t(x_t) p(x_{t+1} \mid x_t)}{\beta_{t+1}(x_{t+1})}.$$

Also, we recall that

$$\xi_{t,t+1}(x_t, x_{t+1}) = p(x_t, x_{t+1} \mid y^n)$$

so we can reuse the work from above to get

$$\xi_{t,t+1}(x_t, x_{t+1}) = \frac{\gamma_{t+1}(x_{t+1}) \alpha_t(x_t) p(x_{t+1} \mid x_t)}{\beta_{t+1}(x_{t+1})}.$$

7 Appendix 2: MATLAB code

```
% main.m: Main program to run our experiments.
clearvars;
close all;

num_coeffs = 14;
num_states = 9;
train_ratio = 0.75;
train_fresh = true;

% Data from lecturer
```

```
dataset_1 = prepare_data('.\audio_1', '.\features_1', num_coeffs, train_ratio, '.m4a');
% Data recorded individually
dataset_2 = prepare_data('.\audio_2', '.\features_2', num_coeffs, train_ratio, '.wav');

num_classes = size(dataset_1, 1);

% Combine the datasets into dataset 3
dataset_3 = dataset_1;
for i = 1:num_classes
    dataset_3{i, 2} = [dataset_1{i, 2}; dataset_2{i, 2}];
    dataset_3{i, 3} = [dataset_1{i, 3}; dataset_2{i, 3}];
end

% Train the HMP on the recordings from lecturer
if train_fresh
    hmm_1 = HiddenMarkovEnsemble(num_classes, num_coeffs, num_states);
    llf_1 = hmm_1.train(dataset_1(:, 2), 500);
    save('model_1.mat', 'hmm_1');

    %hmm_2 = HiddenMarkovEnsemble(num_classes, num_coeffs, num_states);
    %llf_2 = hmm_2.train(dataset_2(:, 2), 500);
    %save('model_2.mat', 'hmm_2');

    hmm_3 = HiddenMarkovEnsemble(num_classes, num_coeffs, num_states);
    llf_3 = hmm_3.train(dataset_3(:, 2), 500);
    save('model_3.mat', 'hmm_3');
else
    load('model_1.mat', 'hmm_1');
    load('model_2.mat', 'hmm_2');
    load('model_3.mat', 'hmm_3');
end

% Evaluate the first model, trained on provided recordings only, on the recordings from lecturer
fprintf("\nProvided data, trained with provided recordings:\n");
fprintf("================================================\n");
test_results = hmm_1.test(dataset_1(:, 3));

figure;
confusion_matrix = confusionmat(test_results(1, :), test_results(2, :));
confusionchart(confusion_matrix);

[precision, recall, f1, accuracy] = classification_metrics(confusion_matrix);
for i = 1:num_classes
    fprintf("Class: %-10s  Precision: %.3f  Recall: %.3f  F1 Score: %.3f\n",...
        dataset_1{i, 1}, precision(i), recall(i), f1(i));
end
fprintf("Overall Accuracy: %.3f\n", accuracy);

% Evaluate the first model on the custom recordings
fprintf("\nCustom data, trained with provided recordings:\n");
fprintf("================================================\n");
test_results = hmm_1.test(dataset_2(:, 3));

figure;
confusion_matrix = confusionmat(test_results(1, :), test_results(2, :));
confusionchart(confusion_matrix);

[precision, recall, f1, accuracy] = classification_metrics(confusion_matrix);
for i = 1:num_classes
    fprintf("Class: %-10s  Precision: %.3f  Recall: %.3f  F1 Score: %.3f\n",...
        dataset_2{i, 1}, precision(i), recall(i), f1(i));
end
fprintf("Overall Accuracy: %.3f\n", accuracy);

% Evaluate the third model, trained with all recordings, on the custom recordings only
fprintf("\nCustom data, trained with all recordings:\n");
fprintf("========================================\n");
test_results = hmm_3.test(dataset_2(:, 3));

figure;
confusion_matrix = confusionmat(test_results(1, :), test_results(2, :));
confusionchart(confusion_matrix);
```

```
[precision, recall, f1, accuracy] = classification_metrics(confusion_matrix);
for i = 1:num_classes
    fprintf("Class: %-10s  Precision: %.3f  Recall: %.3f  F1 Score: %.3f\n",...
        dataset_2{i, 1}, precision(i), recall(i), f1(i));
end
fprintf("Overall Accuracy: %.3f\n", accuracy);

% Evaluate the third model on all recordings
fprintf("\nAll data, trained with all recordings:\n");
fprintf("======================================\n");
test_results = hmm_3.test(dataset_3(:, 3));

figure;
confusion_matrix = confusionmat(test_results(1, :), test_results(2, :));
confusionchart(confusion_matrix);

[precision, recall, f1, accuracy] = classification_metrics(confusion_matrix);
for i = 1:num_classes
    fprintf("Class: %-10s  Precision: %.3f  Recall: %.3f  F1 Score: %.3f\n",...
        dataset_3{i, 1}, precision(i), recall(i), f1(i));
end
fprintf("Overall Accuracy: %.3f\n", accuracy);
```

```
% prepare_data.m: Function to load audio files, convert to MFCC features
% and split the dataset into train and test partitions.
function dataset = prepare_data(src_folder, dst_folder, num_coeffs, train_ratio, ext)

features_suffix = sprintf('_%d.bin', num_coeffs);

% Read the subfolders, exclude '.' (current directory) and '..' (parent directory)
subfolders = dir(src_folder);
subfolders = subfolders(~ismember({subfolders.name}, {'.', '..'}));
num_classes = length(subfolders);

% The first column of the dataset gives the name of each class
dataset = cell(num_classes, 2);
dataset(:, 1) = {subfolders.name};

% Duplicate the directory structure for feature map storage
if ~isfolder(dst_folder)
    status = mkdir(dst_folder);
    if status == 0
        fprintf("Failed to create folder %s\n", dst_folder);
        return
    end
end

% Read files and create features
for i = 1:num_classes
    cur_src_subfolder = subfolders(i);
    cur_dst_subfolder = strcat(dst_folder, '\', cur_src_subfolder.name);

    if ~isfolder(cur_dst_subfolder)
        status = mkdir(cur_dst_subfolder);
        if status == 0
            fprintf("Failed to create folder %s\n", cur_dst_subfolder);
            return
        end
    end

    % Read audio from the source directory
    wildcard = strcat('\*', ext);
    src_files = dir(strcat(cur_src_subfolder.folder, '\', cur_src_subfolder.name, wildcard));
    num_examples = length(src_files);

    examples = cell(num_examples, 1);
    for j = 1:num_examples
        cur_src_file = src_files(j);
        audio_filename = strcat(cur_src_file.folder, '\', cur_src_file.name);

        features_filename = strcat(cur_dst_subfolder, '\',...
            replace(cur_src_file.name, ext, features_suffix));
        examples{j} = features_filename;
```

```
        if ~isfile(features_filename)
            features = get_features(audio_filename, num_coeffs);
            fid = fopen(features_filename, 'w');
            fwrite(fid, features, 'double');
            fclose(fid);
        end
    end

    % Split into train and test sets
    [train_indices, test_indices] = train_test_split(num_examples, train_ratio);
    dataset{i, 2} = examples(train_indices);
    dataset{i, 3} = examples(test_indices);
end

end

function features = get_features(filename, num_coeffs)

% Read the audio file
[waveform, fs] = audioread(filename);

% Crop audio to 1 second or add zero-padding
if length(waveform) > fs
    waveform = waveform(1:fs);
elseif length(waveform) < fs
    waveform(end + 1, fs) = 0.0;
end

assert(length(waveform) == fs);

% If necessary, resample the waveform now
if fs ~= 48000
    [p, q] = rat(48000 / fs);
    waveform = resample(waveform, p, q);
end

% Compute MFCC
features = mfcc(waveform, fs, 'LogEnergy', 'ignore', 'NumCoeffs', num_coeffs);

end
```

```
% train_test_split.m: Function to compute indices for a random train-test split.
function [train_indices, test_indices] = train_test_split(num_examples, ratio)

% Create a random permutation of the example indices
indices = randperm(num_examples);

% Find size of the train set
train_size = ceil(num_examples * ratio);

% Select training indices
train_indices = indices(1:train_size);
test_indices = indices((train_size + 1):end);

end
```

```
% load_features.m: Function to load pre-prepared MFCC features from disk.
function features = load_features(filename, num_coeffs)

fid = fopen(filename, 'r');
features = fread(fid, 'double');
features = reshape(features, [], num_coeffs);
features = features(3:end, :);
fclose(fid);

end
```

```
% HiddenMarkovEnsemble: Class to hold HMPs for multiple classes and perform
% classification using the ML principle. Includes methods for training and testing.
classdef HiddenMarkovEnsemble < handle
    properties (SetAccess = private)
```

```
        NumClasses
        NumCoeffs
        Models
    end

    methods
        % Constructor
        function obj = HiddenMarkovEnsemble(num_classes, num_coeffs, num_states)
            % Set the size of the model
            obj.NumClasses = num_classes;
            obj.NumCoeffs = num_coeffs;

            obj.Models = cell(1, num_classes);
            for i = 1:num_classes
                obj.Models{i} = HiddenMarkovModel(num_coeffs, num_states);
            end
        end

        % Train the hidden markov models on the training dataset and return the
        % log likelihood achieved
        function llf = train(obj, samples, max_iterations)
            assert(length(samples) == obj.NumClasses);

            llf = zeros(1, obj.NumClasses);
            for i = 1:obj.NumClasses
                fprintf("Training class %d:\n", i);
                llf(i) = obj.Models{i}.train(samples{i}, max_iterations);
            end
        end

        % Run the models on a sample and choose the class according to the
        % Maximum Likelihood principle
        function pred_label = run_models(obj, sample)
            llf = zeros(1, obj.NumClasses);
            for i = 1:obj.NumClasses
                llf(i) = obj.Models{i}.test(sample);
            end
            [~, pred_label] = max(llf);
        end

        % Test the HMM classifier and report results as [true_labels, pred_labels]
        function results = test(obj, samples)
            assert(length(samples) == obj.NumClasses);

            results = [];
            for i = 1:obj.NumClasses
                cur_samples = samples{i};
                for j = 1:length(cur_samples)
                    pred_label = obj.run_models(cur_samples{j});
                    results(:, end + 1) = 0;                %#ok<AGROW>
                    results(1, end) = i;
                    results(2, end) = pred_label;
                end
            end
        end
    end
end
```

```
% HiddenMarkovModel.m: Class to encapsulate a HMP model. It handles EM
% parameter estimation and the forward-backward algorithm for training
% over multiple observation sequences.
classdef HiddenMarkovModel < handle
    properties (SetAccess = private)
        % Fixed properties of the HMM
        NumStates
        NumCoeffs

        % Trainable parameters
        Prior
        LogPrior
        TransitionMatrix
        LogTransitionMatrix
        Means
```

```
    Covariances
end

methods
    % Constructor
    function obj = HiddenMarkovModel(num_coeffs, num_states)
        % Set the size of the model
        obj.NumStates = num_states;
        obj.NumCoeffs = num_coeffs;

        % Initialize with dummy parameters
        obj.reset(zeros(1, num_coeffs), eye(num_coeffs));
    end

    % Reset the trainable parameters
    function reset(obj, dataset_mean, dataset_covariance)
        % Use a randomized prior and transition matrix
        obj.Prior = rand(1, obj.NumStates);
        obj.Prior = obj.Prior ./ sum(obj.Prior);
        obj.LogPrior = log(obj.Prior);

        obj.TransitionMatrix = rand(obj.NumStates);
        obj.TransitionMatrix = obj.TransitionMatrix ./ sum(obj.TransitionMatrix, 2);
        obj.LogTransitionMatrix = log(obj.TransitionMatrix);

        % Initially assume state distributions are equal to the overall distribution
        obj.Means = repmat(dataset_mean.', 1, obj.NumStates);
        obj.Covariances = repmat(dataset_covariance, 1, 1, obj.NumStates);
    end

    % Compute the log probability of an observation in a given state
    function log_prob = log_obs_prob(obj, obs, state)
        log_prob = logmvnpdf(obs, obj.Means(:, state).', obj.Covariances(:, :, state));
    end

    % Forward recursion algorithm
    function [log_a, log_b] = forward(obj, obs)
        N = size(obs, 1);

        % Storage for the sequences a_t = p(x_t|y^t) and b_t = p(x_t|y^{t-1})
        log_a = zeros(N, obj.NumStates);
        log_b = zeros(N, obj.NumStates);

        log_b(1, :) = obj.LogPrior;

        % Forward recursion loop
        for t = 1:N
            % Measurment update
            for k = 1:obj.NumStates
                % Compute log probability of the observation in state k
                log_prob = obj.log_obs_prob(obs(t, :), k);

                % Update log probabilities
                log_a(t, k) = log_b(t, k) + log_prob;
            end

            % Normalize probabilities
            log_a(t, :) = log_a(t, :) - logsumexp(log_a(t, :), 2);

            % Time update
            if t < N
                for k = 1:obj.NumStates
                    log_b(t + 1, k) = logsumexp(log_a(t, :).' + obj.LogTransitionMatrix(:, k), 1);
                end
            end
        end
    end

    % Backward recursion algorithm
    function [log_y, log_z] = backward(obj, log_a, log_b)
        N = size(log_a, 1);

        % Storage for sequences y_t = p(x_t|y^n) and z_t = p(x_t, x_{t+1}|y^n)
```

```
    log_y = zeros(N, obj.NumStates);
    log_z = zeros(N - 1, obj.NumStates, obj.NumStates);

    % Initialization using results of the forward recursion
    log_y(N, :) = log_a(N, :);

    % Backward recursion loop
    for t = (N - 1):-1:1
        for k = 1:obj.NumStates
            log_z(t, k, :) = log_y(t + 1, :) + log_a(t, k)...
                + obj.LogTransitionMatrix(k, :) - log_b(t + 1, :);
            log_y(t, k) = logsumexp(log_z(t, k, :), 3);
        end
    end
end

% Calculate the log likelihood of an observation given the results of the
% forward-backward algorithm
function llf = log_likelihood(obj, obs, y, z)
    N = size(obs, 1);

    term1 = sum(y(1, :) .* obj.LogPrior);

    term2 = 0.0;
    for t = 1:(N - 1)
        term2 = term2 + sum(squeeze(z(t, :, :)) .* obj.LogTransitionMatrix, 'all');
    end

    term3 = 0.0;
    for t = 1:N
        for k = 1:obj.NumStates
            term3 = term3 + y(t, k) * obj.log_obs_prob(obs(t, :), k);
        end
    end

    llf = term1 + term2 + term3;
end

% Test a sample by calculating its log likelihood
function llf = test(obj, sample)
    obs = load_features(sample, obj.NumCoeffs);

    % Run forward-backward algorithm
    [log_a, log_b] = obj.forward(obs);
    [log_y, log_z] = obj.backward(log_a, log_b);

    y = exp(log_y);
    z = exp(log_z);

    % Return log likelihood
    llf = obj.log_likelihood(obs, y, z);
end

% Train the model on a series of samples
function llf = train(obj, samples, max_iterations)
    M = length(samples);

    % Load observations into memory
    obs = [];
    for i = 1:M
        obs(i, :, :) = load_features(samples{i}, obj.NumCoeffs);          %#ok<AGROW>
    end

    % Find dataset mean and covariance, then reset the model
    [mu, sigma] = get_stats(obs, obj.NumCoeffs);
    obj.reset(mu, sigma);

    N = size(obs, 2);
    L = size(obs, 3);

    reverse_string = '';

    prev_llf = 0.0;
```

```
    for iteration = 1:max_iterations
        % Run forward-backward algorithm for each observation
        log_a = zeros(M, N, obj.NumStates);
        log_b = zeros(M, N, obj.NumStates);

        log_y = zeros(M, N, obj.NumStates);
        log_z = zeros(M, N - 1, obj.NumStates, obj.NumStates);

        for i = 1:M
            [tmp_a, tmp_b] = obj.forward(squeeze(obs(i, :, :)));
            [tmp_y, tmp_z] = obj.backward(tmp_a, tmp_b);

            log_a(i, :, :) = tmp_a;
            log_b(i, :, :) = tmp_b;
            log_y(i, :, :) = tmp_y;
            log_z(i, :, :, :) = tmp_z;
        end

        y = exp(log_y);
        z = exp(log_z);

        % Compute log likelihood
        llf = 0.0;
        for i = 1:M
            llf = llf + obj.log_likelihood(squeeze(obs(i, :, :)),...
                squeeze(y(i, :, :)), squeeze(z(i, :, :, :)));
        end
        msg = sprintf('Iteration %3d, NLL: %.3e\n', iteration, -1.0 * llf);
        fprintf([reverse_string, msg]);
        reverse_string = repmat('\b', 1, length(msg));

        if abs(llf - prev_llf) / abs(llf) < 1e-6
            break
        end
        prev_llf = llf;

        % Update prior
        obj.LogPrior = squeeze(logsumexp(log_y(:, 1, :), 1)).' - log(M);
        obj.Prior = exp(obj.LogPrior);

        % Update transition probability
        for k = 1:obj.NumStates
            obj.LogTransitionMatrix(k, :) = logsumexp(log_z(:, :, k, :), [1, 2])...
                - logsumexp(log_y(:, 1:(N - 1), k), [1, 2]);
        end
        obj.TransitionMatrix = exp(obj.LogTransitionMatrix);

        % Update mean for each state
        for k = 1:obj.NumStates
            den = exp(logsumexp(log_y(:, :, k), [1, 2])) + 1e-12;
            num = zeros(1, L);
            for l = 1:M
                cur_obs = squeeze(obs(l, :, :));
                num = num + sum(y(l, :, k).' .* cur_obs, 1);
            end
            obj.Means(:, k) = num ./ den;
        end

        % Update covariance for each state
        for k = 1:obj.NumStates
            den = exp(logsumexp(log_y(:, :, k), [1, 2])) + 1e-12;
            num = zeros(L);
            for l = 1:M
                for t = 1:N
                    deviation = squeeze(obs(l, t, :)) - obj.Means(:, k);
                    outerprod = deviation * deviation.';
                    num = num + y(l, t, k) * outerprod;
                end
            end
            obj.Covariances(:, :, k) = (num ./ den) + 1e-6 * eye(L);
        end
    end
end
```

```
    end
end

% Compute log-sum-exp of a vector of log probabilities x
function y = logsumexp(x, dim)

c = max(x, [], dim);
if c == -Inf
    y = -Inf;
else
    y = c + log(sum(exp(x - c), dim));
end

end

% Calculate sample mean and covariance of the class samples
function [mean, cov] = get_stats(obs, num_coeffs)

M = size(obs, 1);

% Compute mean
mean = zeros(1, num_coeffs);
num_samples = 0;
for i = 1:M
    cur_obs = squeeze(obs(i, :, :));
    num_samples = num_samples + size(cur_obs, 1);
end
mean = mean ./ num_samples;

% Compute covariance
cov = zeros(num_coeffs);
for i = 1:M
    cur_obs = squeeze(obs(i, :, :));
    for k = 1:size(cur_obs, 1)
        cov = cov + (cur_obs(k, :).' * cur_obs(k, :));
    end
end
cov = cov ./ (num_samples - 1);

end
```

```
% logmvnpdf.m: Function to compute the log PDF of a multivariate
% normal distribution.
function logpdf = logmvnpdf(x, mu, sigma)

[V, D] = eig(sigma);
values = diag(D);
values_inv = 1.0 ./ values;
log_det = sum(log(values));

rank = length(values);
deviation = x - mu;

tmp_1 = V .* sqrt(values_inv).';
tmp_2 = tmp_1 * deviation.';
innerprod = tmp_2.' * tmp_2;

logpdf = -0.5 * (rank * log(2.0 * pi) + innerprod + log_det);

end
```