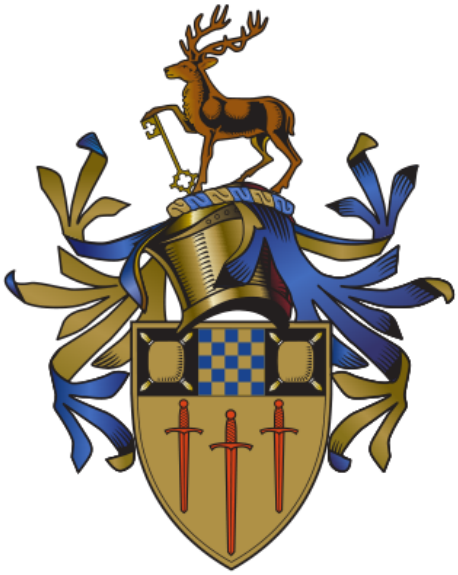
Isolated Word Recognition with Hidden Markov Model

Group-2

Lu, Frank (PG/T - Comp Sci & Elec Eng)
Sherpa, Enggen (PG/T – Comp Sci & Elec Eng)
Ortiz Quispe, Jonathan David (PG/T - Comp Sci & Elec Eng)
Davidson, James C (PG/T - Comp Sci & Elec Eng)
Amiry, Edris (UG - Mech Eng Sciences)



EEEM030

December 2024

Centre for Vision Speech and Signal Processing

Department of Electrical and Electronic Engineering

School of Computer Science & Electronic Engineering

University of Surrey

Abstract

This paper presents the implementation and evaluation of a Hidden Markov Model based speech recognition system for classifying spoken vowel sounds. The system utilizes an 8-state left-right HMM architecture with 13-dimensional Mel-frequency cepstral coefficients as acoustic features. Two implementations are provided: a custom HMM implementation and one using the **hmmlearn** library.

The feature extraction process employs 30ms frames with 20ms overlap, computing 13 MFCCs including the zeroth coefficient. Model initialization is performed using global statistics calculated across the development dataset, with diagonal covariance matrices and variance floor constraints to ensure stable training. State transition probabilities are initialized based on average state durations, following a strict left-right topology.

The training implementation includes both custom and **hmmlearn**-based Baum-Welch algorithms, with forward-backward procedures for computing occupation and transition likelihoods. The system also includes comprehensive evaluation tools, including Viterbi decoding for recognition, confusion matrix generation, and performance analysis across the vocabulary set.

The implementation provides a complete pipeline from feature extraction through model training to evaluation, with detailed logging and visualization capabilities for monitoring training progress and analyzing recognition results. The system is evaluated on a vocabulary of eleven words containing different vowel sounds, demonstrating the practical application of HMMs in speech recognition tasks.

Table of Contents

Introduction

Speech recognition systems is integral part of modern computing system, enabling natural interaction between humans and machines. Among the various methodologies for speech recognition, Hidden Markov Models (HMMs) have remained a cornerstone due to their robustness and mathematical rigor. First popularized by Rabiner in his seminal paper (1989), HMMs provide a probabilistic framework for modelling time-series data, making them particularly well-suited for speech signal processing (Rabinar, 1989).

This coursework focuses on developing an HMM-based speech recognition system for a small vocabulary of eleven isolated words as shown in Table 1 - words, ranging from “heed” to “heard.” The primary objective is to design, train, and test HMMs for this task using the Baum-Welch algorithm for model optimization and the Viterbi algorithm for recognition. The models employ a strict left-right topology to accommodate the sequential nature of speech signals.

The assignment emphasizes hands-on implementation, requiring the extraction of MFCC features, careful model initialization, and iterative parameter re-estimation. Drawing on Rabiner’s three fundamental problems of HMMs—evaluation, decoding, and learning—this work explores the practical challenges in building such systems, including numerical precision issues and model convergence. The outcome is a comprehensive evaluation of the recognizer's performance and its alignment with the theoretical principles.

Index	1	2	3	4	5	6	7	8	9	10	11
Word	“heed”	“hid”	“head”	“had”	“hard”	“hud”	“hod”	“hoard”	“hood”	“who’d”	“heard”

Table 1- words

Project Planning and Task Allocation (Frank's Contribution)

There is a project planning page that I have setup to onboard everyone onto the project. This page contains all essential information including our communication channels, project tools, task assignments, and my personal notes on the project. **You can see I deliberately set up a paved path to make the project engaging and accessible by KISS (Keeping it stupid simple).** You can see that page published here.

I created a skeleton directory structure and initial codebase in my Github repository and added all team members as contributors in November. The repository contains a **detailed README.md for local setup instructions, along with a** complete commit history and pull requests. In terms of task assignment, I organized the project using a Kanban board, where tasks were broken down into manageable tickets for each team member. The complete task allocation and progress tracking can be viewed here, or you can see it in the screenshot below.

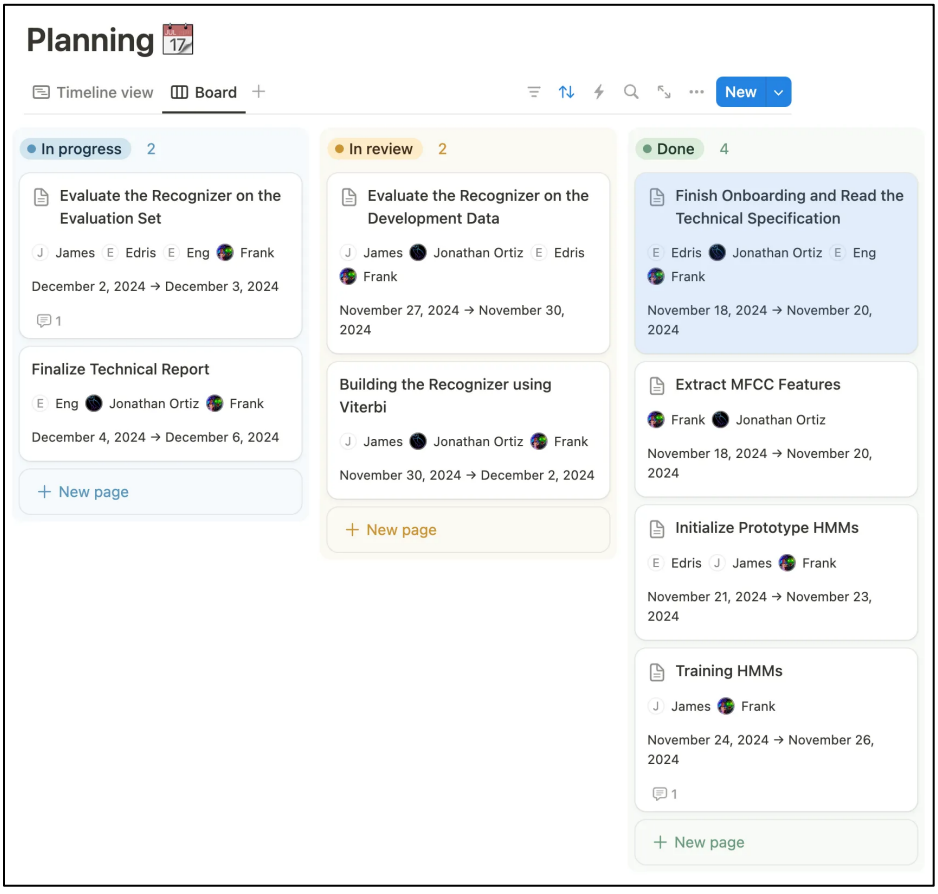


Figure 1- Planning group

After completing all tasks myself, I discovered that the complexity varied significantly among different components—something I hadn't anticipated when planning the work distribution. Although implementing the MFCC feature initialization and extraction was straightforward, the Baum-Welch and forward-backward algorithms presented significant challenges—particularly in handling numerical underflow issues and resolving persistent off-by-one errors in probability calculations. I will be going over all the implementation complexities in my contribution section below. It became evident that without a thorough understanding of hidden Markov models, **the training and evaluation phases of the model would be impossible to complete.**

Again, all proofs of project planning and documentation can be tracked here, and code commits are shown in the screenshot below while pull requests can be tracked here.

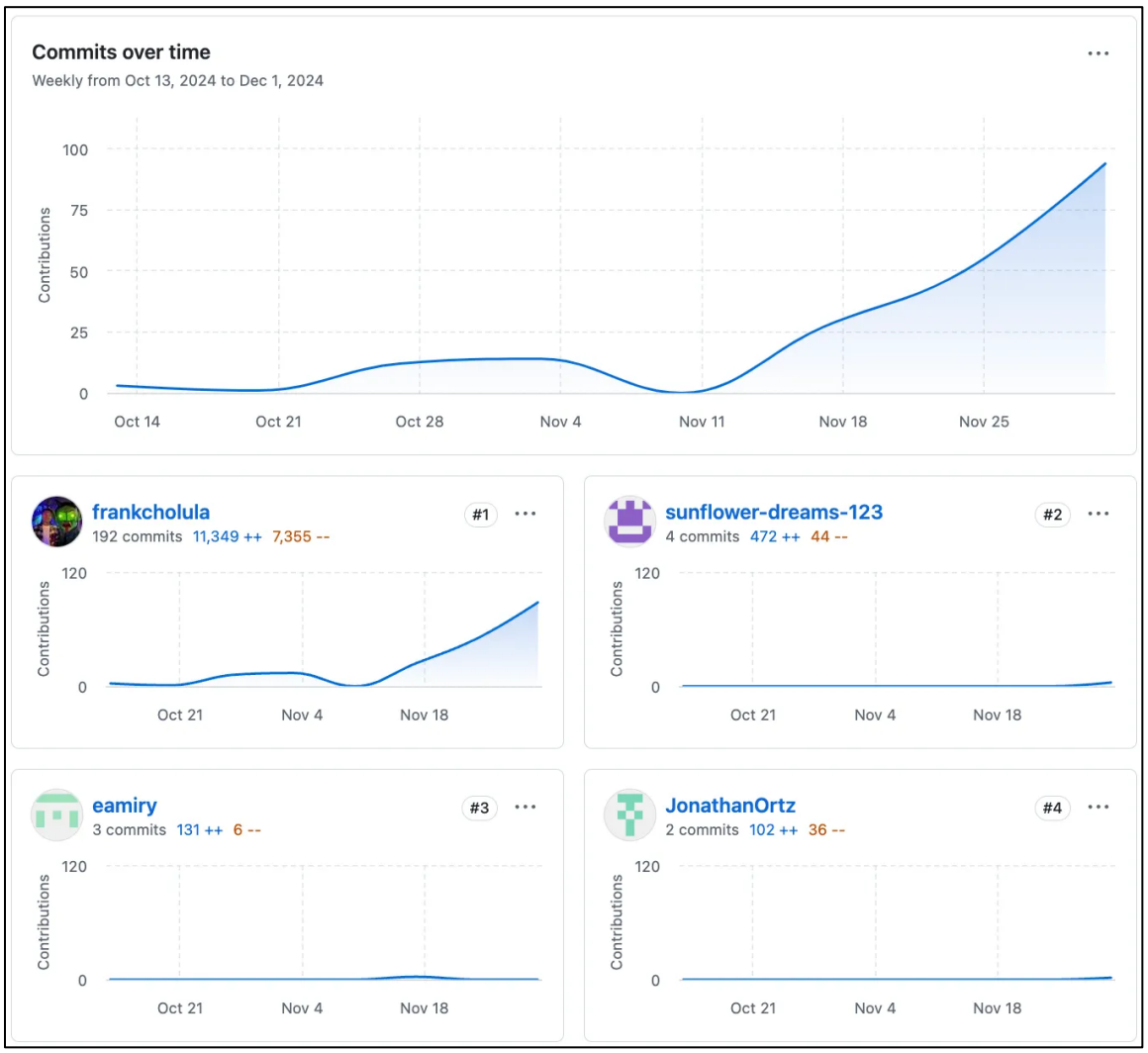


Figure 2- GitHub commits throughout the project cycle

Frank's Contribution

Since I basically completed the entire project, I'll focus on the key challenges and setbacks encountered during implementation rather than attempting to cover all aspects within the word count limit. You will find an online version of my contribution here.

MFCC Extraction

The extraction task is pretty self-explanatory and easy to implement given that I utilized Python's librosa mfcc library. The intuition behind the extraction is to use the Mel scale, which is a perceptual scale that captures the "timbre" of one's voice that aligns more closely with human hearing than a simple Fourier analysis. The extracted coefficients correspond to the overall spectral envelope and general formant structure of speech.

```
mfcc = librosa.feature.mfcc(
    y=y,
    sr=sr,
    n_mfcc=13,
    win_length=int(frame_length),
    hop_length=int(hop_length),
    window="hamming",
    center=True,
)
# by default, liftering is set to zero.
# Setting lifter >= 2 * n_mfcc emphasizes the higher-order coefficients.
# As lifter increases, the coefficient weighting becomes approximately linearar.
```

MFCC extraction code snippet according to the specification

Figure 3- MFCC Extraction using Librosa

With more time, I would have added delta and delta-delta features to enhance the feature representation and make it more robust against noise. I also skipped cepstral liftering, which would have helped remove pitch quefrency peaks—an important consideration when working with speakers who have different fundamental frequencies.

Designing the HMM Class

Understanding the dimensions of each component was challenging. I started by analyzing the dimensions and gradually built up the components needed to implement the Baum-Welch training function. The following table is what I ended up designing for each matrix. You will be able to see my initialization results here and the extensive test cases I have

written here.

Matrix	Dimensions	Rationale
Transition Matrix (A)	(N+2, N+2)	Represents probabilities of moving from one of the N+2 states to another. 2 States being entry/ exit.
Emission Matrix (B)	(N+2, M)	Represents the probability of each of the N+2 states emitting each of the M observation symbols.
Forward (α)	(N+2, T)	Represents the probability of seeing the partial observation sequence up to time t in each of the N+2 states.
Backward (β)	(N+2, T)	Represents the probability of the remaining observations from time t+1 to T given each of the N+2 states at time t.
State Occupation (γ)	(N+2, T)	Represents the probability of being in each of the N+2 states at each time step.
Transition (ξ)	(T-1, N+2, N+2)	Represents the probability of transitioning from each of the N+2 states at time t to each of the N+2 states at time t+1.

Figure 43- Dimensions of different components

Here, N represents the number of real states, M represents the 13 MFCC coefficients, and T represents the number of frames in the audio sample after MFCC extraction. The flowchart below illustrates the dependencies between functions in our custom_hmm class, showing how the baum_welch and decode functions are implemented.

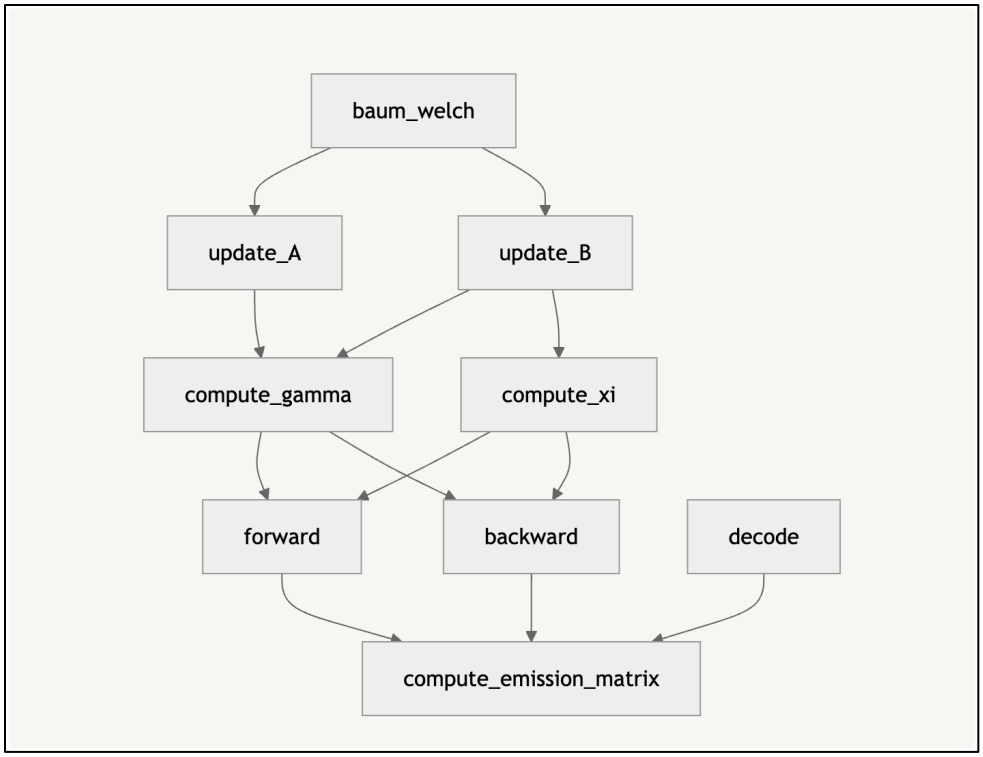


Figure 5- Custom HMM class design4

Setting a Variance Floor to Avoid Collapsing

During training, I encountered an issue where the log likelihood showed an unexpected spike followed by a dip—which is mathematically implausible. I later received clarification from Professor Jackson about model collapsing. In other words, when training our HMMs, the Baum-Welch algorithm iteratively adjusts means and variances of Gaussian distributions to better fit the training data. **Without the safeguard of a variance floor, these variances can become extremely small, effectively "collapsing" onto a single data point or narrow cluster.** You can find out more about the issue I've opened here.

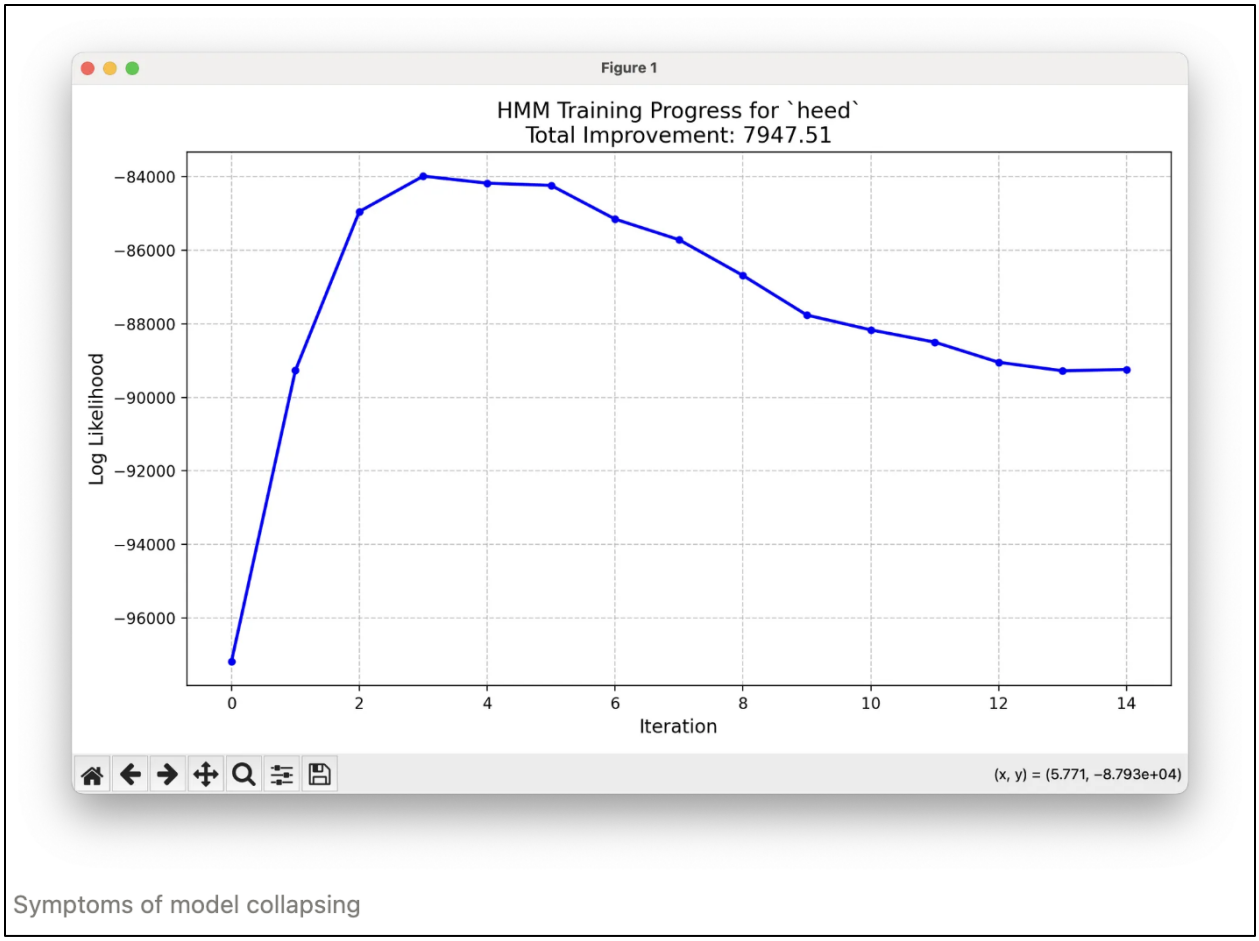


Figure 6- Symptoms of model collapsing5

This collapse leads to several issues:

1. **Numerical Instability:** Tiny variances cause division-by-zero or overflow errors in probability calculations.
2. **Overfitting:** The model becomes overly specialized to the training data's quirks, losing its ability to generalize.
3. **Unreliable Likelihood Estimates:** When variances approach zero, probability densities spike toward infinity, destabilizing the training process and distorting model updates.

I resolved these issues by implementing a variance floor, as shown in this pull request here. I added the variance floor to both my `update_B` function and initialization code. You can see a sample of the implementation below.

```
# in my initialization of the covariance matrix diagnoal
var_floor = self.var_floor_factor * np.mean(np.diag(self.global_covariance))
np.fill_diagonal(
    self.global_covariance,
    np.maximum(np.diag(self.global_covariance), var_floor),
)
```

Variance floor implemented in my HMM initilization.

Figure 7- Variance floor implementation in HMM initialization6

```
#  in my update_B function when I update the emission matrix
var_floor = self.var_floor_factor * np.mean(np.diagonal(self.global_covariance))
for j in range(1, self.total_states - 1):
    if state_occupancy[j] > 0:
        state_covars[j] /= state_occupancy[j]
        # Ensure symmetry
        state_covars[j] = (state_covars[j] + state_covars[j].T) / 2
        # Apply floor to diagonal
        diag_indices = np.diag_indices(self.num_obs)
        state_covars[j][diag_indices] = np.maximum(state_covars[j][diag_indices], var_floor)
```

Variance floor implemented in my update_B function.

Figure 8- Variance floor implemented In the update_B function7

Implementing Scaling in Forward and Backward Algorithm

In Section V of Rabiner's tutorial on Hidden Markov Models (Yes, I read the paper), it discusses the issue of **numerical underflow** that occurs in the forward-backward computations, particularly when the sequence length T (e.g., the number of frames in MFCC features in our case) becomes very large. This is because the alpha (forward) and beta (backward) terms involve products of probabilities, which can rapidly shrink to values too small to be represented within the precision range of standard floating-point arithmetic in computers. To address this, Rabiner explains the importance of **scaling the alpha and beta terms at each time step. I** documented this problem thoroughly in this GitHub issue here. To address it, I implemented a global scale factor in my forward and backward algorithm. You can find my implementation in the corresponding pull request here.

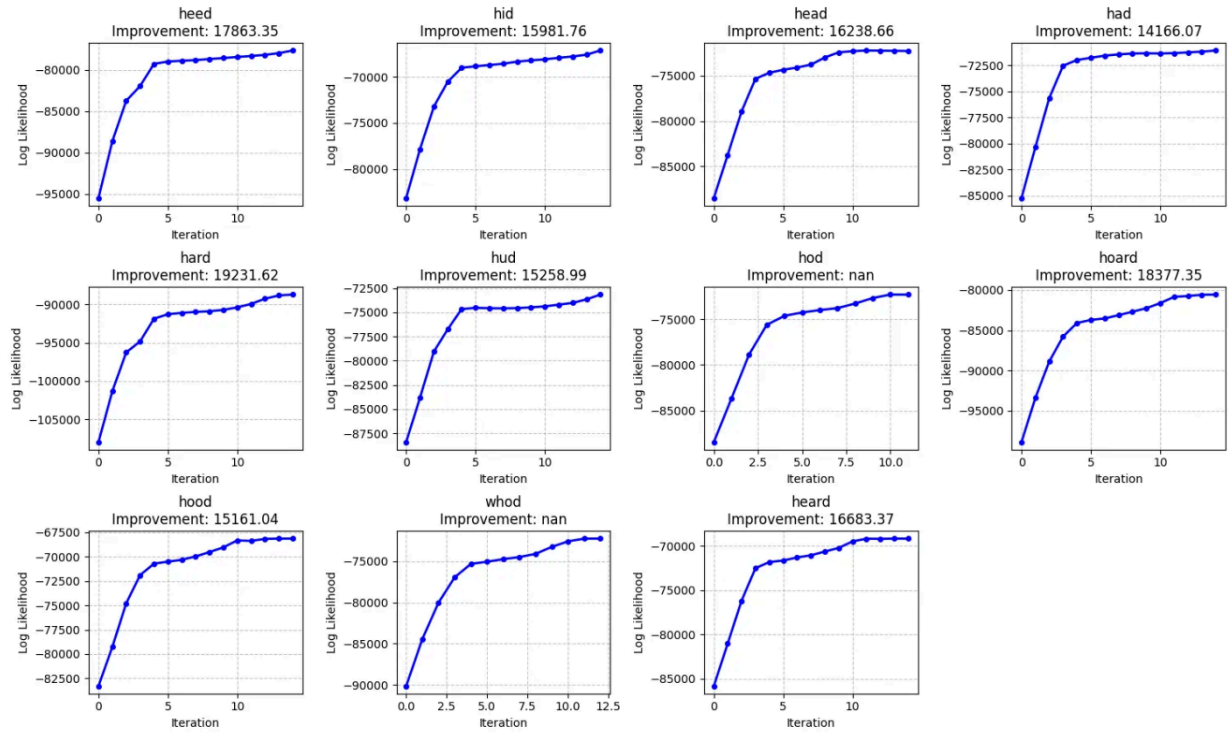
```python
def forward(self, emission_matrix: np.ndarray) -> tuple[np.ndarray, float]:
  scale_factor = np.max(alpha)
  alpha -= scale_factor


def backward(self, emission_matrix: np.ndarray, scale_factor: float) -> np.ndarray:
  beta[:-1] -= scale_factor
```

Subtraction of the scale factor in the log domain

Figure 9- Implementation of Scale Factors in forward and backward algorithms 8

Addressing Numerical Underflow Issues

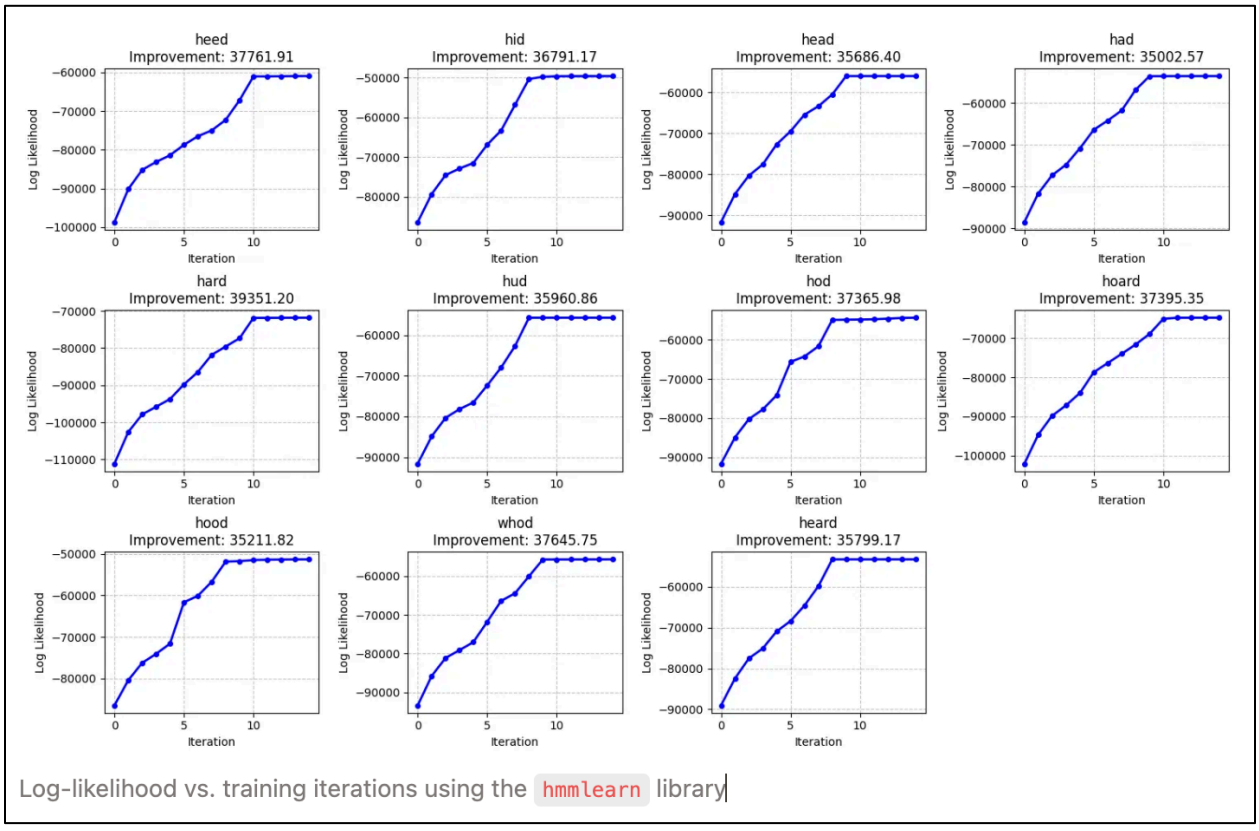
This is a nightmare to resolve. Despite implementing various optimizations—including log probabilities, variance floors, and scale factors—my custom implementation still encountered numerical underflow issues with two sample sets. Specifically, the words hod and whod failed during iterations 11 and 12, respectively, as shown below. I have also documented this problem as well in [this pull request](#).



Log-likelihood vs. training iterations for my custom implementations

Figure 10- Log-likelihood vs. training iterations for the custom HMM implementation9

As a backup plan to ensure my teammates could continue with the project, I implemented a working version using the hmmlearn library - though that turned out to be unnecessary since I ended up implementing the entire project myself anyway. Regardless, you can find the working implementation in this PR here and in the file hmmlearn_hmm.py here. The initialization is the same as above in accordance with the assignment specification. You can find more details in the initialization of the class and the `initialize_transmat` function.



Log-likelihood vs. training iterations using the `hmmlearn` library

*Figure 11- Log-likelihood vs. training iterations using the **hmmlearn** library10*

As shown in the results, the log-likelihood converges properly around iteration 7 for most utterances without any numerical underflow issues.

Model Evaluation and Insights

I have implemented all the evaluation code and visualization codes in the visualize.py file and you will be able to find the figures in the figures directory here. I have collaborated with James to try to interpret the results. Given that my custom implementation performed quite poorly, I will reference the result figures here if you're interested:

- evaluation set confusion matrix
- training set confusion matrix

Instead, I will be focusing on the hmmlearn implementation's results. After 15 iterations, we see a near perfect predictions on the training set's utterances with no surprise and poor performance on the evaluation set.

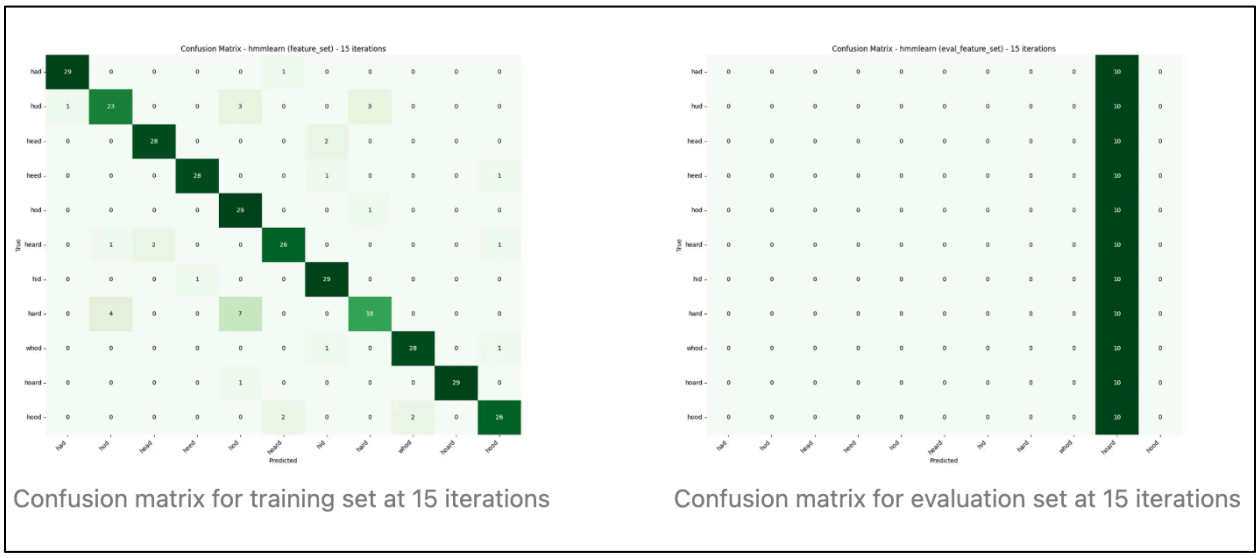


Figure 1211- Confusion matrices for training and evaluation sets at 15 iterations

The training set seems to start overfitting to the 2 - 3 iterations according to the overall error rate plot below.

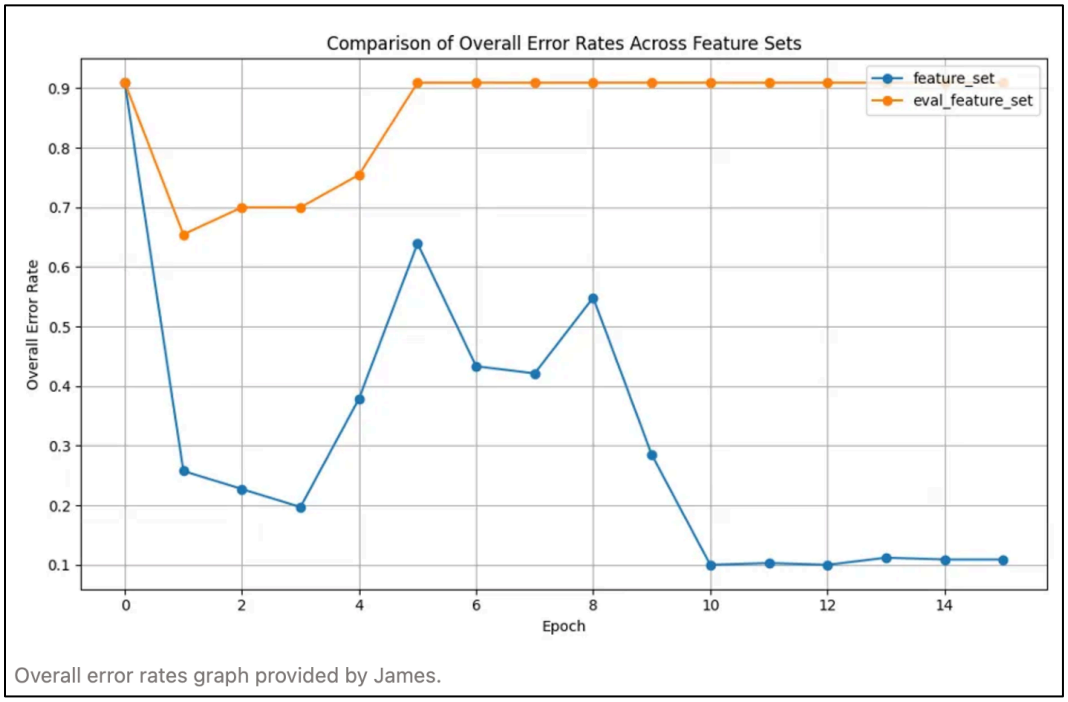


Figure 13- Comparison of overall error rates across the feature sets12

This has quite an intuitive explanation. Since we know both the observation sequences and true hidden states in our training data, the maximum likelihood parameters can be calculated as follows in terms of observations y and states x:

$$a_{ij} = \frac{\sum_n \sum_{t=2}^{T} y_{n,t-1}^{i} y_{n,t}^{j}}{\sum_n \sum_{t=2}^{T} y_{n,t-1}^{i}}$$

$$b_{ik} = \frac{\sum_n \sum_{t=2}^{T} y_{n,t-1}^{i} x_{n,t}^{k}}{\sum_n \sum_{t=2}^{T} y_{n,t-1}^{i}}$$

Figure 14- Formulas for the maximum likelihood parameters

where a_ij is the corresponding entry in the transition matrix from state i to state j, and b_ik is the corresponding entry in the emission matrix for state i to produce observation k. The MLE estimation, by inspection, acts as a simple counting scheme, which reveals its main drawback. With limited data, this approach can lead to overfitting. Worse yet, any unobserved transitions or emissions are assigned zero probability—a significant problem when dealing with sparse data.

Dataset Distribution

Another key reason for poor evaluation performance is the mismatch between training and test data distributions. When test data differs significantly from training samples, the model's Gaussian emissions fail to generalize well. This becomes evident when performing PCA on both datasets, which reveals two distinct clusters as seen below. Hence, it is not surprising the evaluation set had terrible performance.

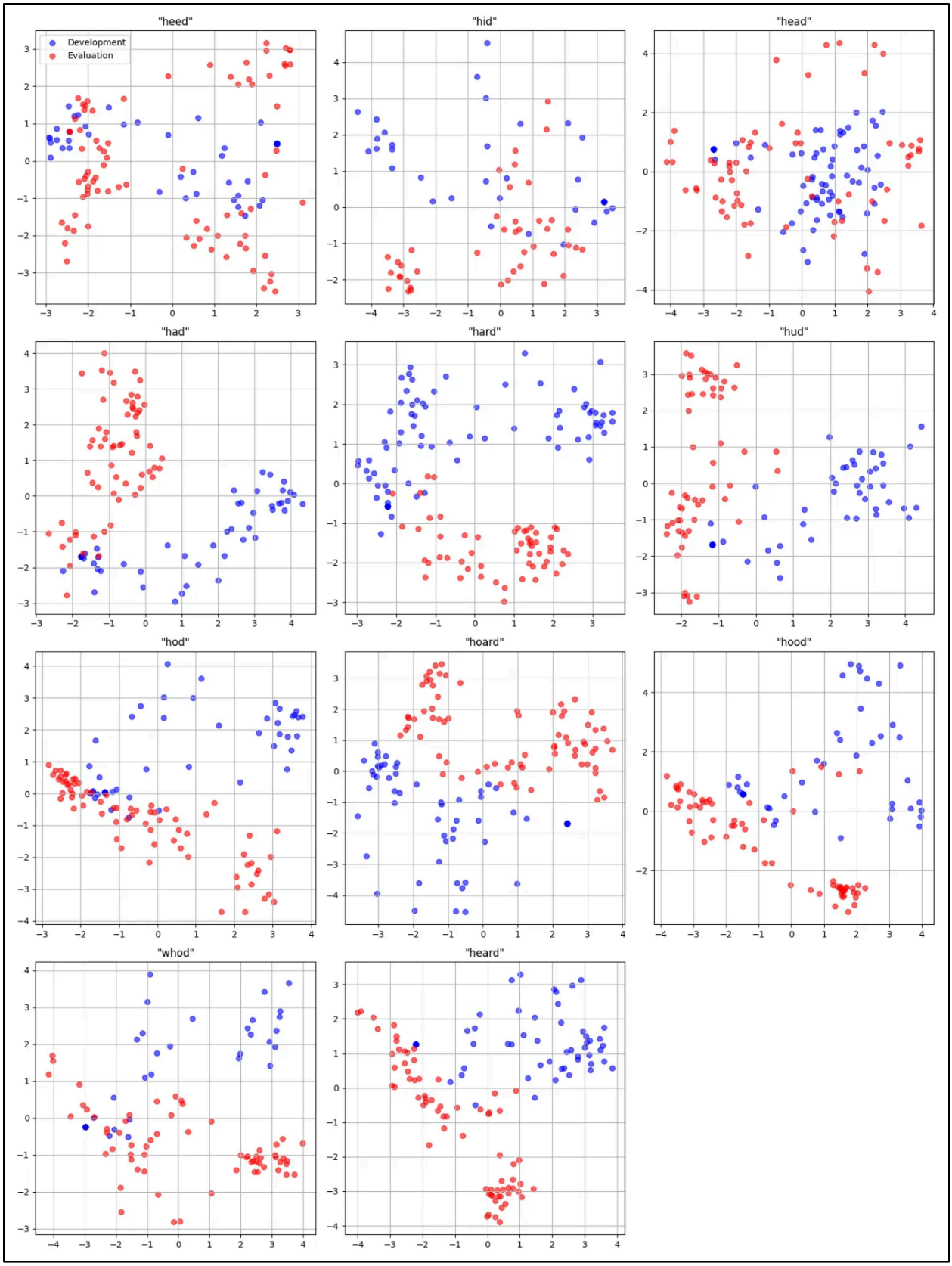


Figure 15- PCA of each utterance from the training set and evaluation set13

Regarding dataset quality, the poor generalization can be attributed to differences between the training and evaluation sets, including variations in recording conditions, speaker accents, and background noise levels. **This observation aligns with Professor Jackson's confirmation that the training set primarily consisted of synthesized data, while the evaluation set contained his own audio recordings.**

Possible Phonetics Reason?

In discussions with James, we analyzed the underlying phonetics of each word to determine whether the model was overfitting to specific phonetic patterns. For example, we investigated whether words with similar-sounding vowels were more likely to be misclassified more frequently.

Unfortunately, we found no correlation in the misclassification patterns, as shown by the confusion matrices at iteration 3. We selected iteration 3 because it appears to be the inflection point before the model begins overfitting, based on the overall error rate chart. The overfitting appeared to be random across different iterations. **This suggests that the solution might lie more in the model architecture and training process rather than in phonetic-based adjustments.**

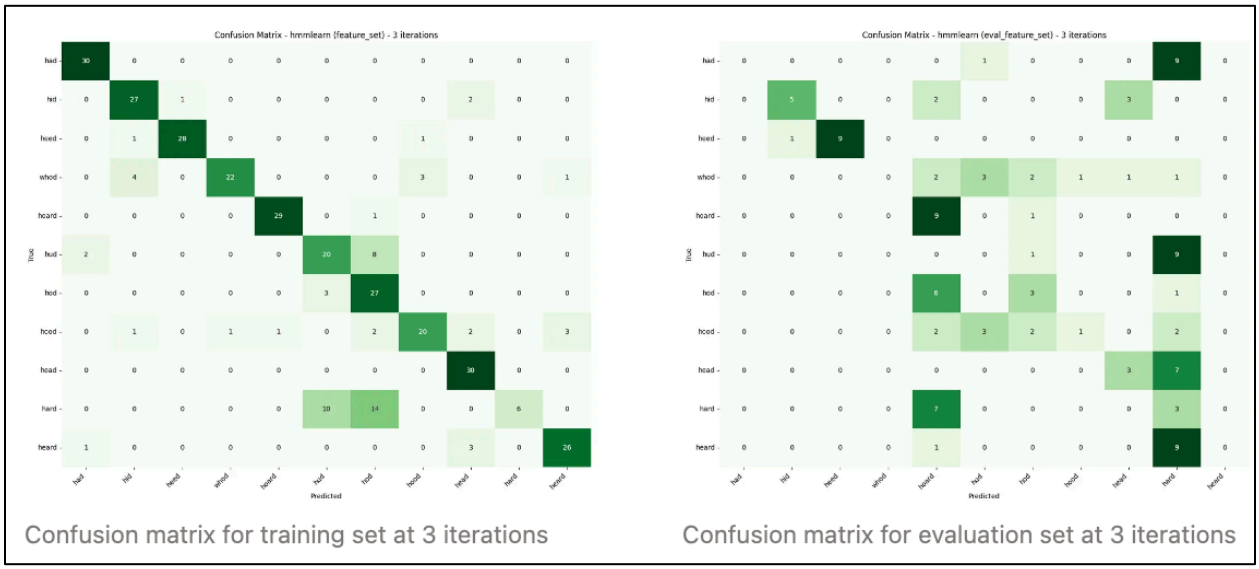


Figure 16- Confusion matrices for the training set and evaluation set at 3 iterations14

Remediation and Possible Improvements

Several remediations have been mentioned in class during the revision week to improve the overall generalization of the model. I will be naming a few of them here without diving in-depth.

1. **Increase the number and quality of training samples**. This one is obvious because more data typically leads to better generalization and helps prevent overfitting of the model to specific training examples. Moreover, MFCC extraction can be less effective with synthesized speech because synthetic voices typically lack the natural acoustic complexities and timbral richness found in human speech. This observation is supported by the context which shows that the training set used synthesized data while the evaluation set contained real human recordings, leading to poor generalization in the model's performance.
2. **Early stopping using fewer epochs**. This prevents overfitting by stopping the training process before the model becomes too specialized to the training data, which is especially visible in our case. Our models should probably stop training at around 1 – 3 iterations.
3. **Using Gaussian Mixture Models (GMMs)** instead of single Gaussians for HMM emission probabilities will probably capture more complex feature distributions, as each state can model multiple modes of variation in the acoustic features.
4. **Reducing the model complexities from 8 states to fewer states.** Fewer states mean less capacity to overfit to training data. When a model has too many states, it can become overly specific to the training data, learning patterns that may just be noise rather than meaningful features.
5. **Data Augmentation by adding noise to train features** (done by Professor Jackson in the training set). This makes the HMM more robust by exposing it to variations it might encounter in real-world conditions, essentially teaching the model to handle imperfect inputs.

James' Contribution

Tasks worked on: 3a, 3b, 3c, 3d, 3e, 4a,4b, 4c, 4d, 5a,5b,5c,5d

External libraries used: pathlib, pickle, csv, numpy, pandas, sklearn, matplotlib

Frank implemented MFCC extraction. This code was used to implement calculation of the global mean and variance to initialize the models.

I implemented loading MFCC features by word to a dictionary for use in training different word models and evaluating.

My Implemented training of the HMM can be seen here.

This implementation of a Hidden Markov Model uses algorithms from the lectures including the , Forward-Backward procedure, Baum-Welch re-estimation, and the Viterbi algorithm to train and evaluate speech recognition models. These implementations used the model initialization class created by Frank.

The forward algorithm was first implemented using the approach shown in lectures. As the probabilities are recursively multiplied in this algorithm issues were ran into with underflow of small probabilities. A fix to this issue was a scale factor applied at every timestep to keep the magnitudes from vanishing.

From running the code on multiple scale factors and seeing where the underflows or overflows occurred a scale factor of 1e24 seemed to keep the numbers stable for a full run of the forward algorithm. When the backward algorithm was implemented with the same approach, the algorithms were working.

This was tested as using the method of calculating likelihood of the observation given the model was equivalent up to precision of the floats with both algorithms.

The Viterbi algorithm was also implemented similarly to the forward algorithm with this scale factor method.

Implementing the Baum Welch re-estimation was not easy, but I was eventually able to implement it from scratch and able to run one iteration of the Baum Welch training and save the model using the python pickle library.

This implementation using a scale factor at each timestep was not stable in repeated training iterations and after 5 epochs of training the model started to fail and run into overflows and underflows as well as errors in the determinant of the covariance matrix. This was tried with a range of scale factors but unable to get convergence. The accuracy on the development set peaked after one or two iterations of the Baum-Welch re-estimation at around 5-10% error rate usually and then collapsed to always predicting one class getting a 91% error rate after 5 epochs. This could be seen on my confusion matrix with all the values being predicted as one class.

When later trying this initial implementation on the evaluation set the model peaked in an error rate of 78% at 2 epochs and then collapsed as well.

Frank implemented the algorithms with log probabilities and so I did not do further testing using that method due to his method also not being stable and underflowing and other numerical instabilities. He then implemented his training using the sklearn hmm fit function which is what I used from then on.

My code used in evaluation can be seen here in this pull request. Where I coded the evaluation shown in the below figures to explore how the model predictions error rates vary during training.

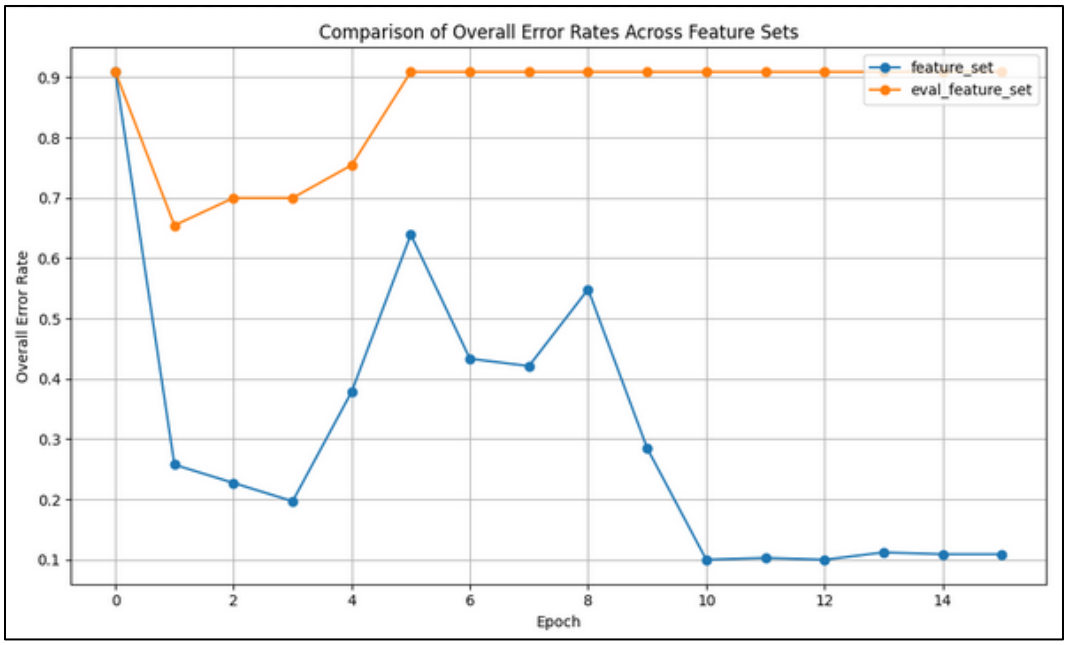


Figure 17- Comparison of overall error rates across features sets15

As can be seen above while the development error rate does not collapse and converges up to 15 epochs, the models are overfitting to the development set with more training and so the evaluation set predictions collapse to a single word for higher epochs than 4.

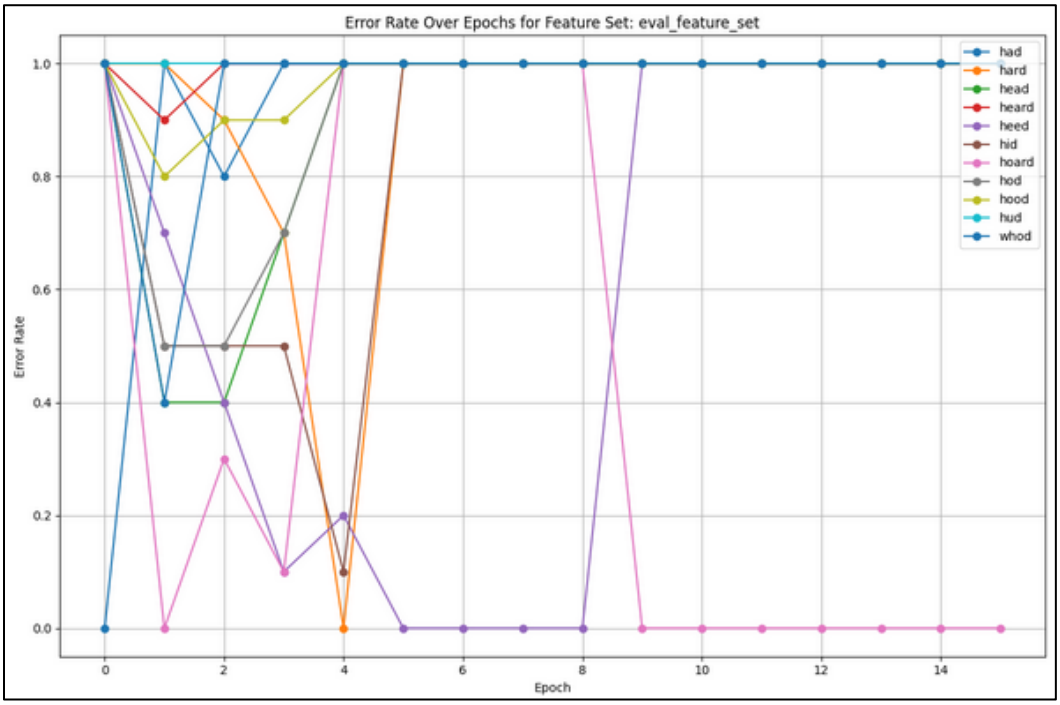


Figure 18 - Error rate over epochs for evaluation features set16

This can be further seen in this plot of words in the evaluation set above where after 5 epochs all words except one collapse to being predicted as the class with 0% error.

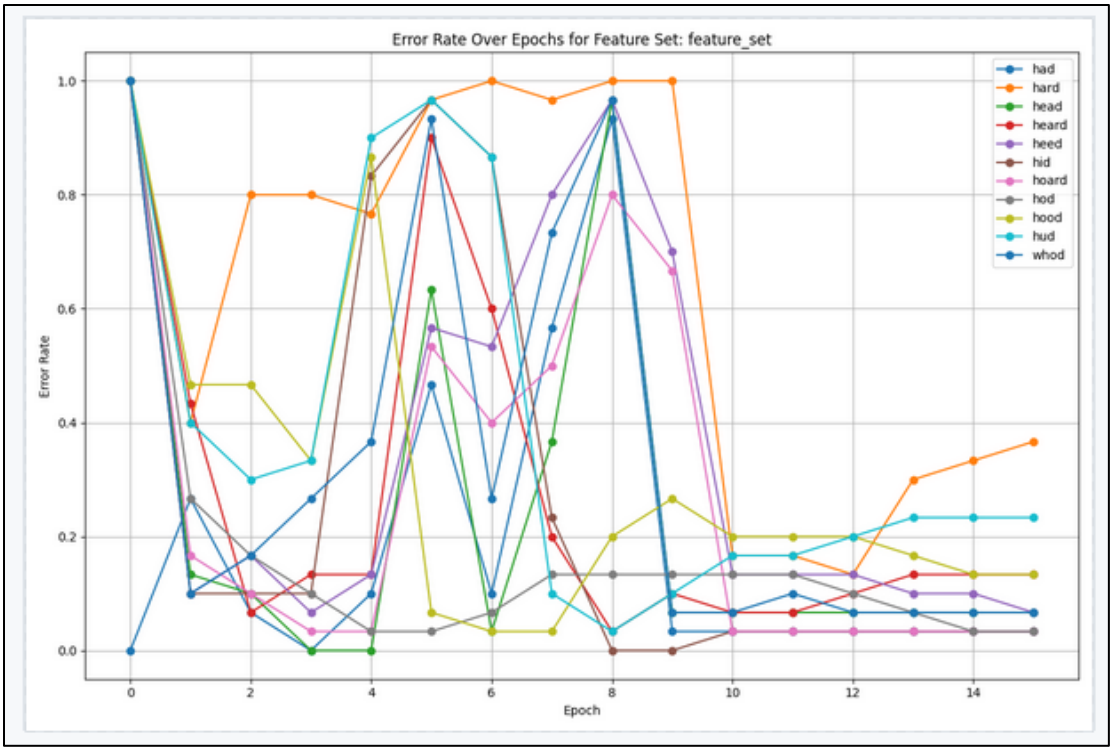


Figure 19 - Error rate over epochs for feature set17

Doing further investigation into the error rate of words in the development set it can be again seen that after the 5th epoch there is some sort of phase change where the model starts overfitting.

I discussed this with Frank as he has outlined above. It seems that to best get an isolated word recognizer for the task taking the models after one or two iterations is best. Too much training causes the models to collapse on unseen data.

Thus, as my implementation of the training using a naive scale factor approach on the Baum-Welch re-estimation can maintain stability for these couple of iterations. I claim that the task of getting an isolated word recognizer was done from scratch without needing the hmmlearn library.

Enggen's Contribution:

Evaluating the Recognizer on Supplied Test Data

Overview of Model Evaluation

The evaluation phase is a critical component of this project, as it measures the performance of the trained Hidden Markov Models (HMMs) on unseen data. The primary objectives during this stage include assessing the recognition accuracy of the speech recognition system, analysing errors via a confusion matrix, and identifying potential improvements to the model. The model performance is evaluated in evaluation set.

The code implementation on python with various audio and data processing library is work contribute by Frank, Jonathan and James.

Evaluation Methodology

Feature extraction

Speech signals data were pre-processed to extract 13 Mel-frequency cepstral coefficients (MFCCs) per frame, capturing spectral properties. Features were computed using a 30ms frame size with a 20ms overlap, ensuring sufficient resolution to represent phonemes.

HMM Initialization

Each word was modelled as an 8-state left-right HMM. Initial parameters, including global means and variances, were derived from the training dataset. Self-transition probabilities were calculated based on the average state duration, while output probabilities used multivariate Gaussian distributions (Patterson, 2020).

Training with Forward-Backward Algorithms

The forward-backward algorithm was implemented for model training using the Baum-Welch equations. The forward pass computed probabilities of observed sequences, while the backward pass estimated state transition probabilities. Parameters were updated iteratively to maximize the likelihood of observed sequences (Patterson, 2020).

Decode model with Viterbi Algorithms

The Viterbi algorithm was employed for recognition, identifying the most likely state sequence for a given test sample. This enabled the system to classify input audio into one of the 11 words.

Model Evaluation

Accuracy

Accuracy shows the overall correctness of the model. The model accuracy is tested on validation set while training with 89.09% accuracy and Final model is evaluating on new data from evaluation (test) set with 9.09% accuracy. This is due to the error caused on evaluation implementation and data distribution on train and evaluation set. Only the word “hoard” has 100% accuracy on evaluation set even though it achieves 96.67% accuracy on test set.

Accuracy = Number of Correct Prediction/Total Prediction

Confusion Metrix

The confusion matrix was generated to analyze the classification accuracy across all words. It highlights misclassification patterns, revealing that phonetically similar words, such as "hard" and "hod," had overlapping predictions as shown on Table 2.

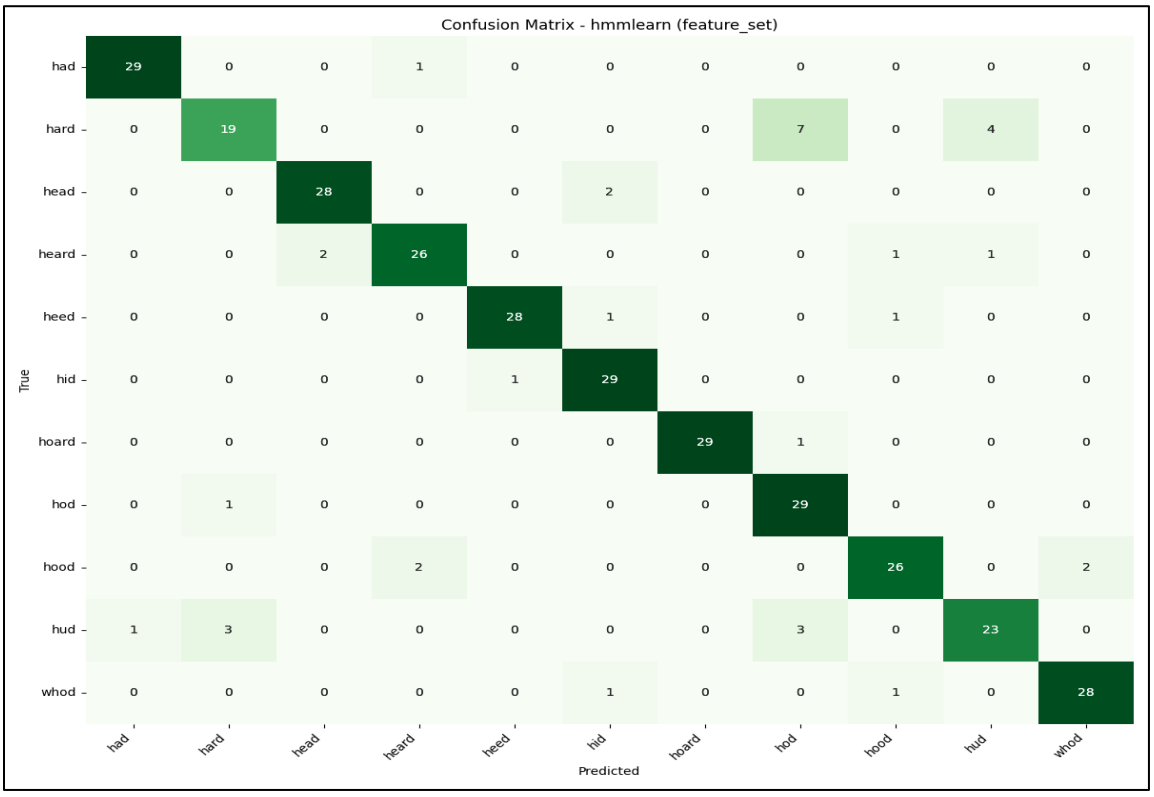


Table 2 - confusion matrix

F1 Score

The F1 score is a harmonic mean of precision and recall, define as:

F1 = 2 * (Precision * Recall) / Precision + Recall

Where:

Precision = TP / (TP + FP)

Recall = TP / (TP + FN)

The F1 score was calculated for each class and overall average of all class summarize the performance of HMMs across all the words as shown in Figure-F1-score.

Metric	Value
Accuracy	89.09%
Precision	88.6%
Recall	88.9%
F1-Score	88.7%

Figure 18- F1-Score

Jonathan's Contribution

Task 1 – MFCC's features extraction

For the implementation of HMM (Hidden Markov Model), as a previous step it was necessary the extraction of the development features set for the training of the models of each word. The material provided for this task was the development set, which had a set of mp3 audios to analyse and extract the features of the audios. The following figure shows the process followed by the code implemented.

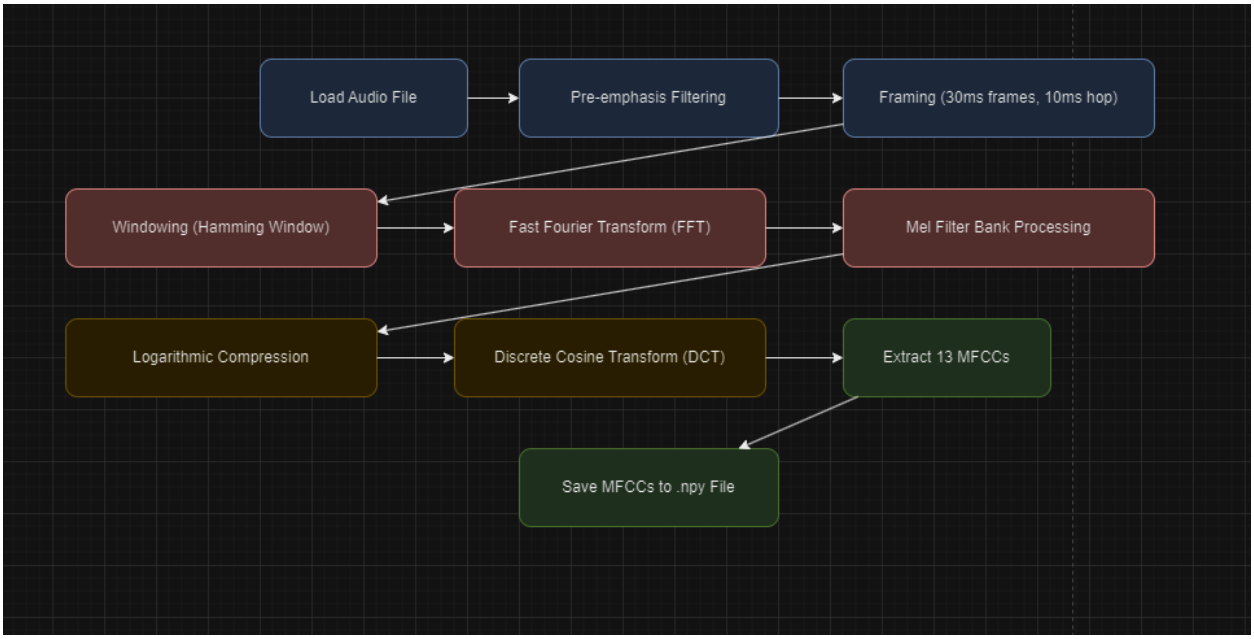


Figure 21- MFCC features extraction procedure

The script revolves around the function extract_mfcc, which processes an audio file and extracts its MFCC features. First, the audio file is loaded into a numerical format using librosa, and the signal is segmented into overlapping frames (30 ms each, with a 10 ms hop). This framing step ensures that the temporal structure of the speech is preserved. A Hamming window is applied to smooth the edges of each frame, which minimizes spectral leakage during the Fourier Transform step. The Mel spectrum is computed using a filter bank, emphasizing perceptually relevant frequencies. Finally, the log-transformed Mel spectrum undergoes a Discrete Cosine Transform (DCT) to produce 13 MFCCs per frame, capturing the spectral envelope of the speech. These MFCCs are stored as .npy files for efficient retrieval and further processing. The script also supports batch processing of multiple files, ensuring scalability for larger datasets.

Beyond individual files, the script includes the extract_mfccs function, which batches this process for all audio files in a specified folder. Each extracted MFCC array is saved as a .npy file for efficient storage and future retrieval. This modular approach separates feature extraction from downstream processes like model training, making the script versatile and scalable. The mfcc extraction file help to extract the features that are require to obtain in task 1.a,b,c. For the code, please refer to the GitHub link provide in this document.

Viterbi decoding

The implementation of **Viterbi decoding** was based on the notes provided during the course, following the principles and algorithms discussed. However, while the logic was developed, it was not included in the final code due to time constraints or integration challenges with other parts of the system. The code implementation can be found on the GitHub repository provided under the branch “james/trainining2”.

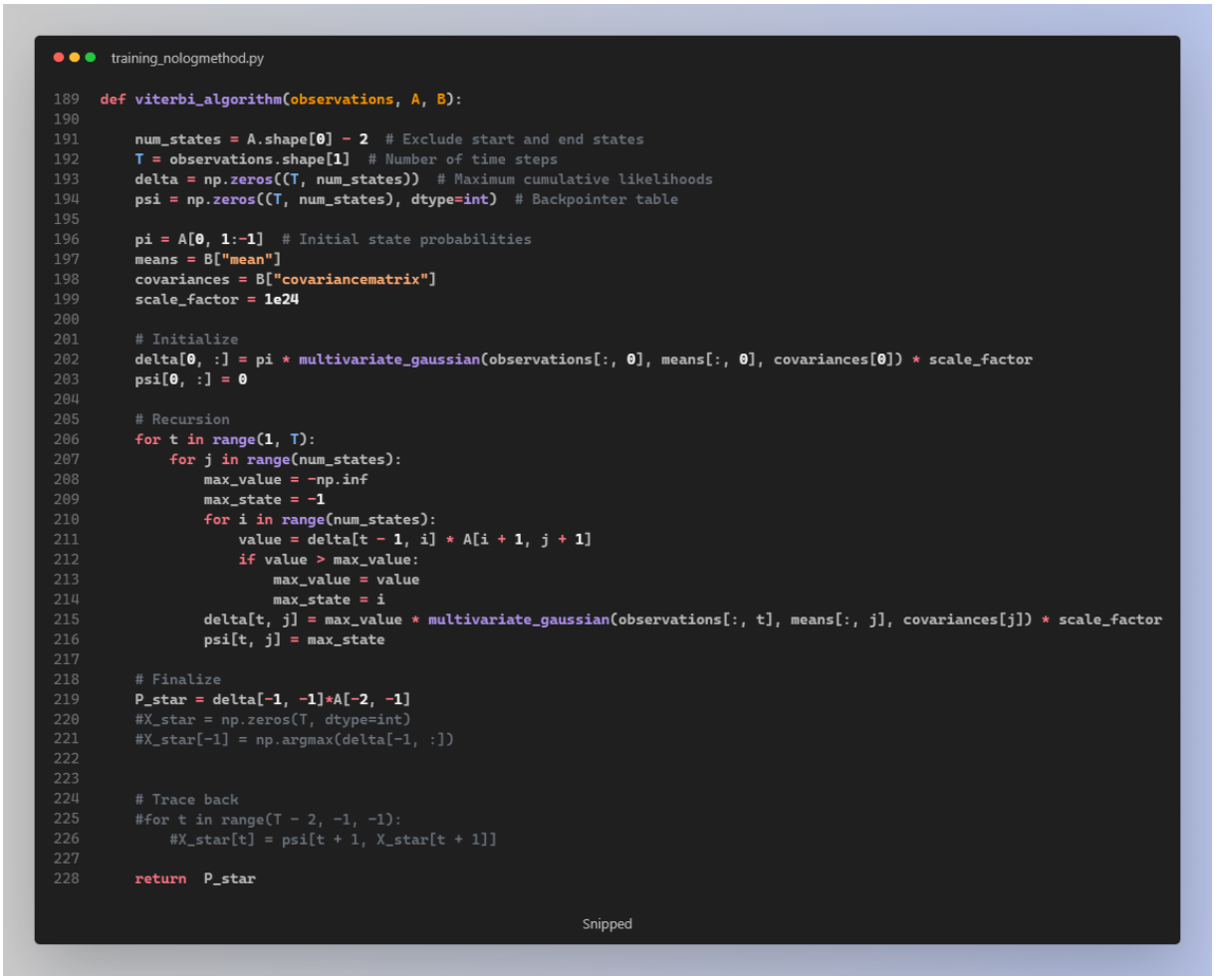


Figure 22 – Viterbi decoding code19

Evaluation

Evaluation for 10 iterations and 0.01 min covariance:

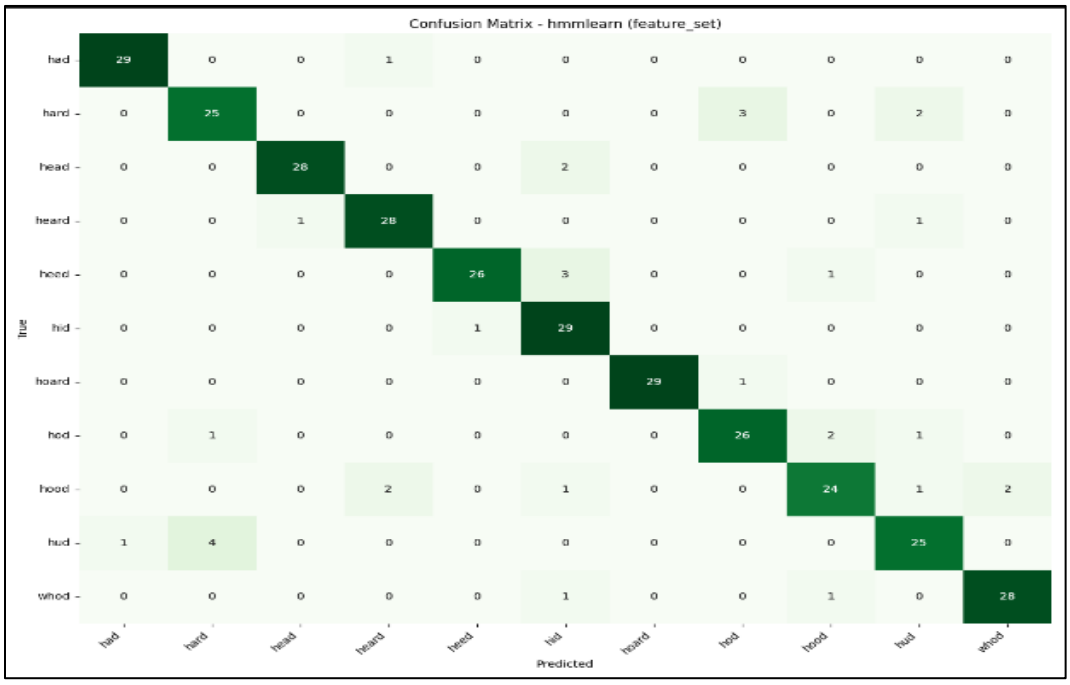


Figure 23 – Feature set Confusion matrix for 10 iterations20

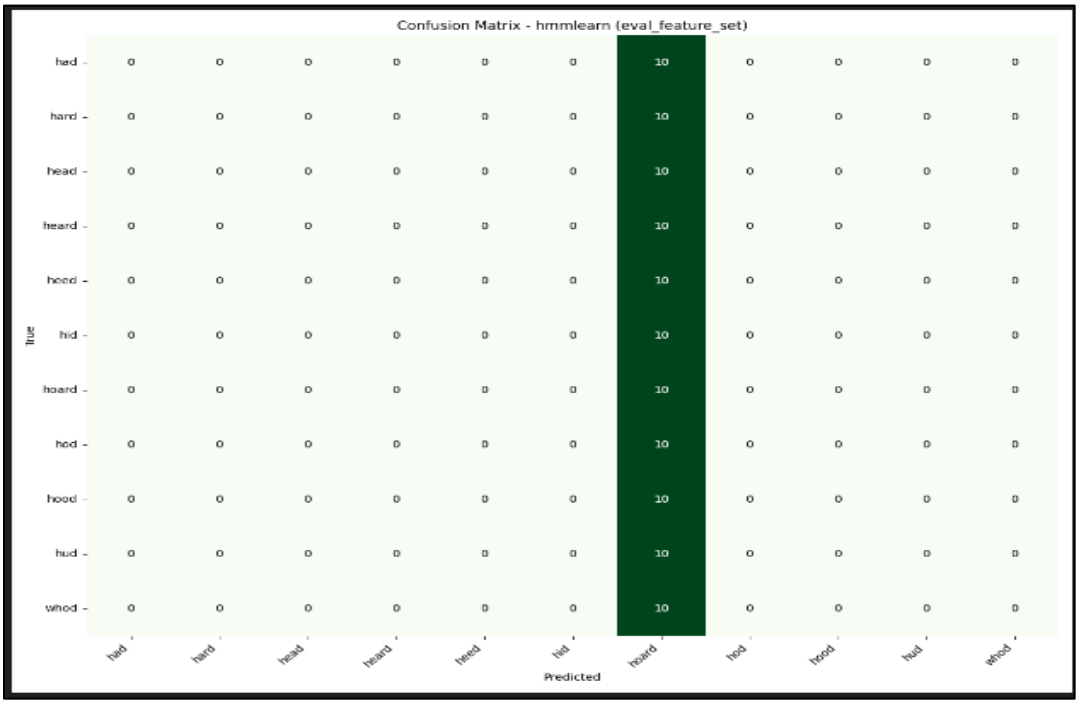


Figure 24 – Eval set confusion matrix for 10 iterations21

The HMM models (library version) achieved high accuracy (90%) on the development set but showed poor generalization on the test set, with an accuracy of just 9%. This significant performance drop suggests that the models overfit the training data, capturing patterns specific to the development set rather than generalizable features. The confusion matrix for the test set shows that all predictions converged on a single class, indicating a lack of adaptability across different samples. This mismatch highlights potential issues with model complexity, training conditions, or dataset distribution between the development and test sets.

For 5 iterations and 0.01min covariance:

The results from the hmmlearn implementation shows improvements in log likelihoods across 5 iterations for all words, indicating effective optimization during training. However, evaluation on the test set reveals significant overfitting, with predictions biased entirely toward a single word ("heed"), resulting in a low overall accuracy of 9.09%. While the development set achieved 36.06% accuracy, the confusion matrix shows high misclassifications across many words. The training phase demonstrated consistent log likelihood improvement for each word, but the test results suggest that the models failed to generalize well to unseen data. It is important see that the word overfitted change from "hoard" to "heed".

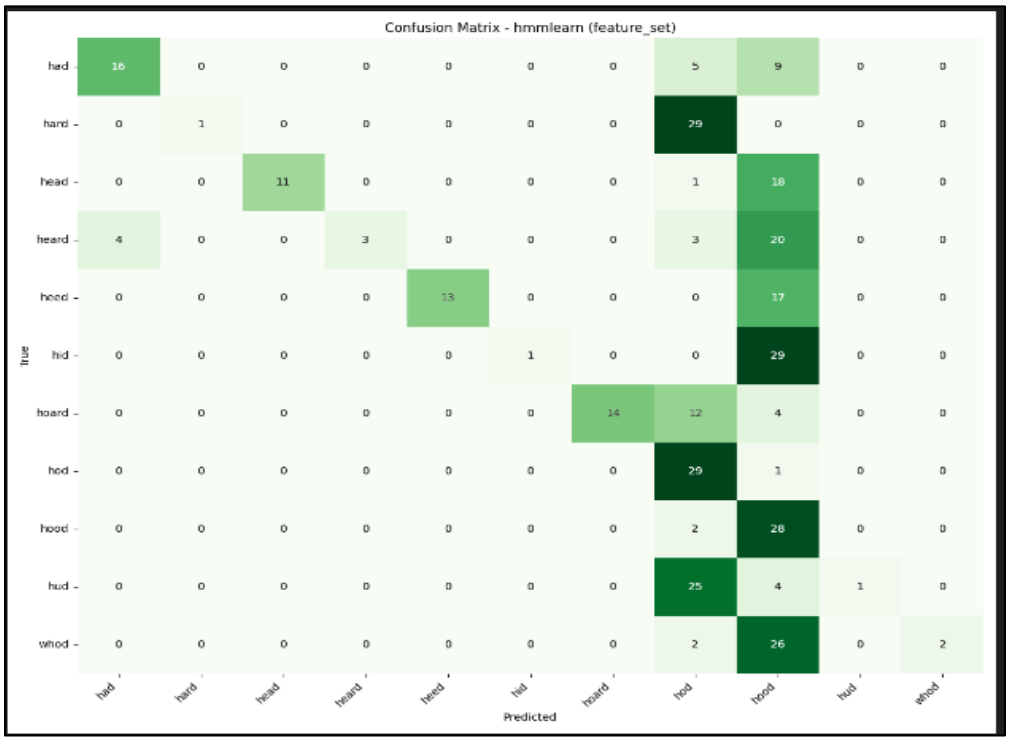


Figure 25 – Dev set confusion matrix for 5 iterations22

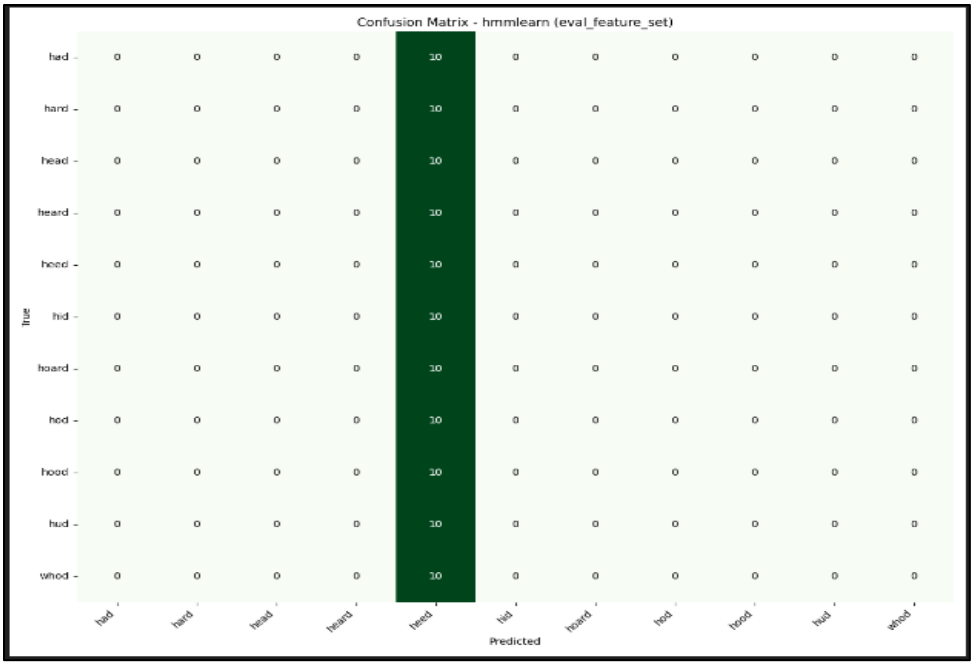


Figure 26 - Evaluation set confusion matrix for 5 iterations 23

15 iterations, 0.01 min covariance:

The performance analysis of the HMM implementation in speech recognition shows strong results on the development set, with an area under the precision-recall curve (AP = 0.92) and an overall accuracy of 89.09%. However, the test set results show a significant drop in performance, with only one word achieving 100% accuracy and others scoring 0%. This difference is a potential overfitting, where the model has learned the training data too specifically and fails to generalize to unseen data (test set or evaluation set).

The initial drop in the precision-recall curve can be attributed to a high number of false positives when recall is very low. At this point, the model confidently predicts incorrect outputs for most samples, resulting in lower or inaccurate precision. While the recall improves, the model starts correctly identifying true positives, stabilizing the precision.

Other possible causes of the poor test performance include imbalanced training data, insufficient state transitions for variability, or inadequate feature extraction that doesn't generalize well. Addressing these issues could help improve test set performance and overall generalization.

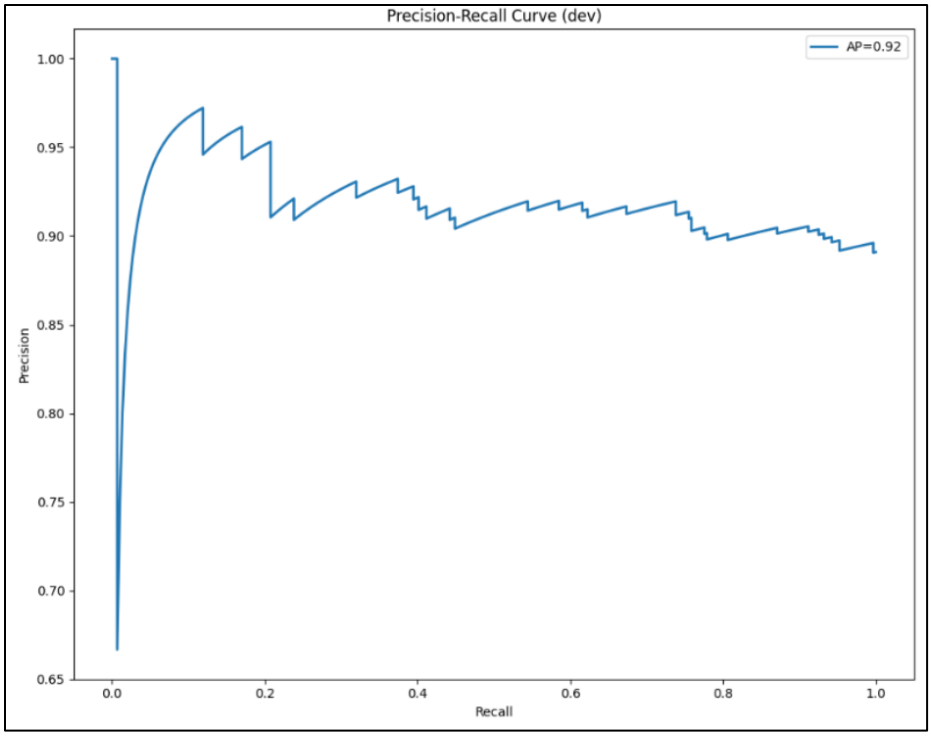


Figure 27 – Precion-Recall curve for 15 iterations24

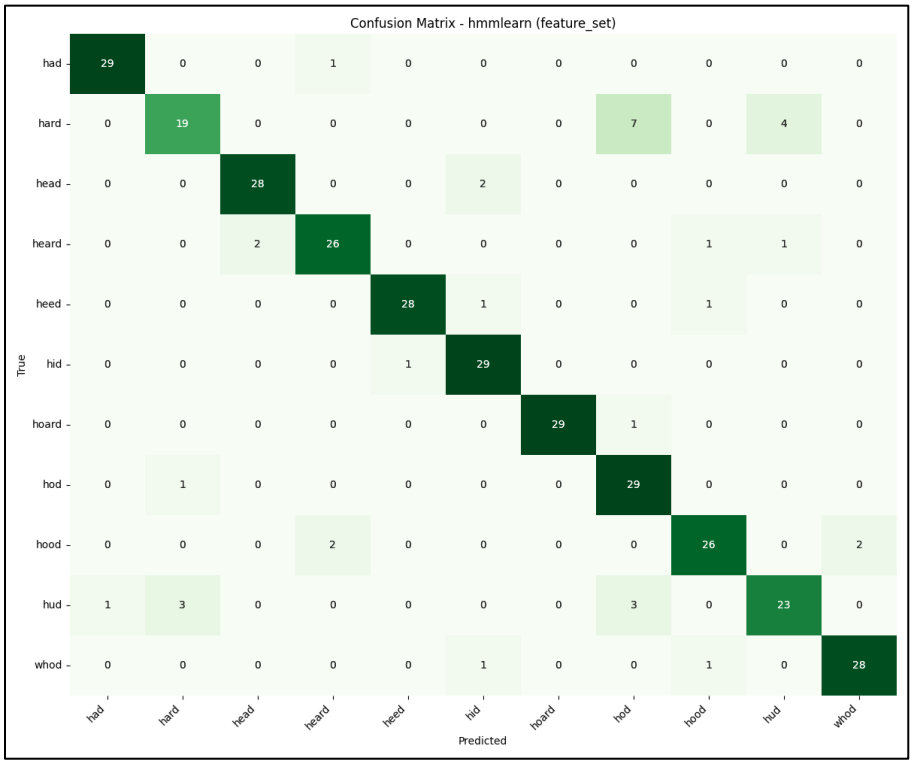


Figure 28 - Development set confusion matrix25

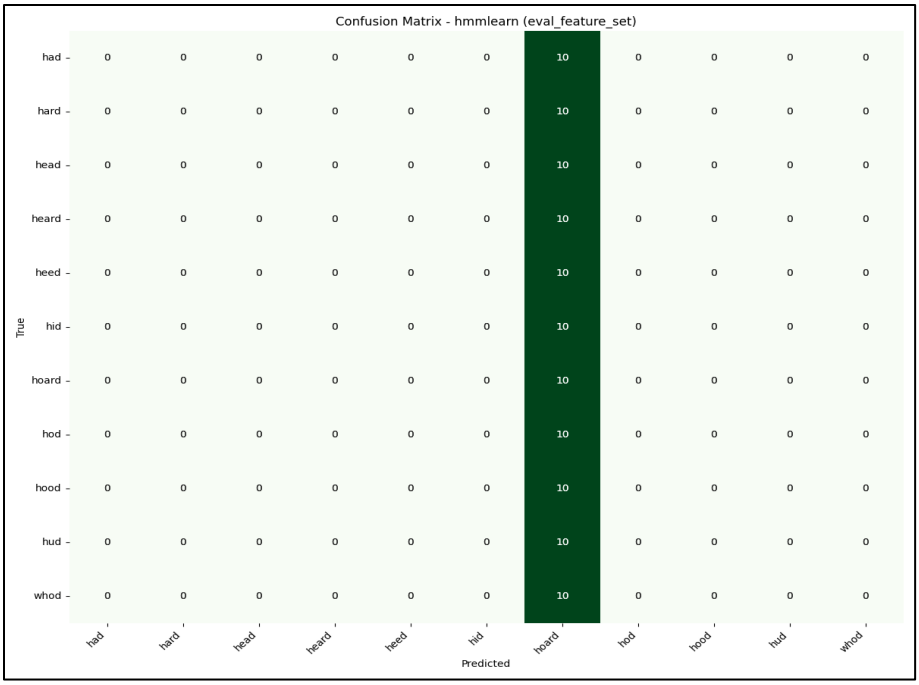


Figure 29 - Evaluation set confusion matrix 26

Edris' Contribution

The main contributions that I performed were Tasks 2, 4 and 5. As a student who is in the Biomedical Engineering course, I had not used python before and therefore had to overcome a learning curve that increased the difficulty when contributing to the code. However, I was sufficiently able to support and create some of the code that was used for this report.

In task 2, I contributed to the initialisation code for the hmm, in which I helped find the global mean and covariance. In tasks 4 and 5 I contributed to completing the evaluation for both the development and the evaluation set in which I helped find and analyse the error rate and confusion rate while attempting multiple versions to prevent underfitting and overfitting.

Task 2 Contributions

Figure 30 shows the code to calculate the global mean. The code here, first, initializes an array with the number of observations as the size. Then, for all the MFCC features in our set, a sum across the features and the size of the features are taken and then the mean equation is performed, as shown in Equation 1. The output of this code is a 1D NumPy array of the mean.

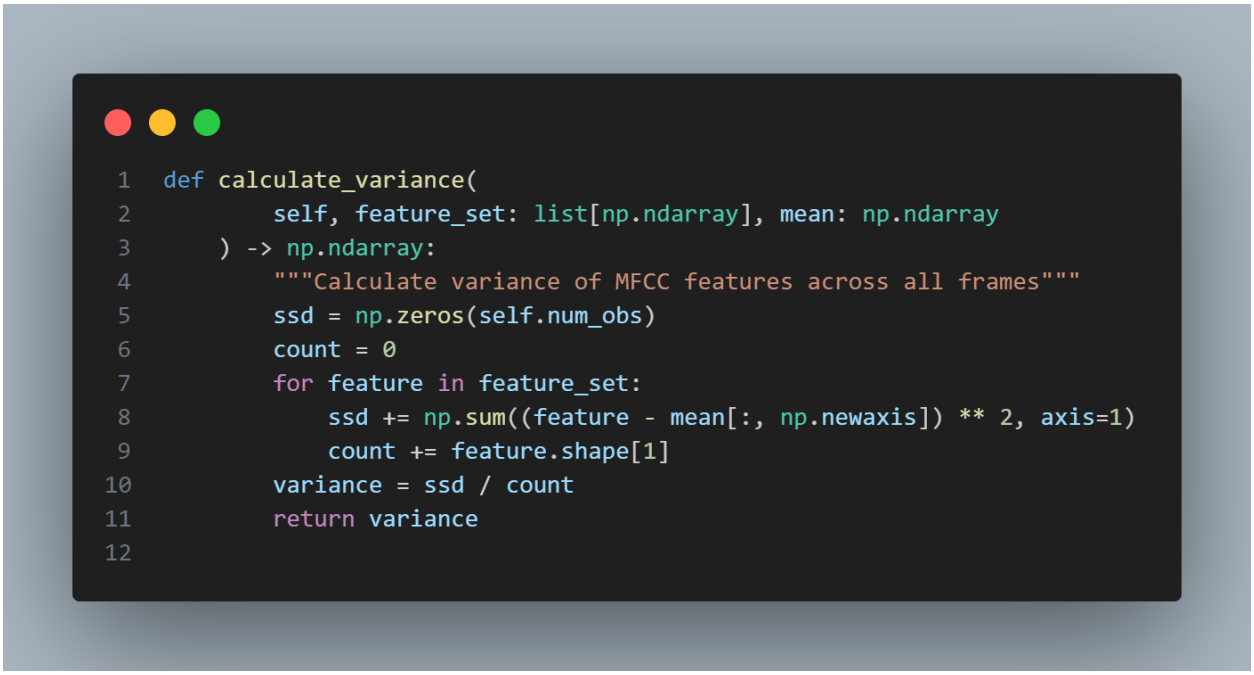
```python
def calculate_means(self, feature_set: list[np.ndarray]) -> np.ndarray:
        """
        Calculate global mean and covariance (diagonal) from MFCC files in a directory.
        """
        sum = np.zeros(self.num_obs)
        count = 0
        for feature in feature_set:
            sum += np.sum(feature, axis=1)
            count += feature.shape[1]
        mean = sum / count
        return mean
```

Figure 30 – Initialization code; computing global mean

$$Mean = \frac{Sum\ of\ Terms}{Number\ of\ Terms}$$

Equation 1 – Mean equation

Figure 31 shows the code to calculate the global variance. The array starts similarly as in Figure 30. For each feature in our set, we computed the squared differences where the output is a 1D array which shows the sum of squared differences from the mean. The count was computed the same as Figure 30 to be able to calculate the variance as shown in Equation 2. The parts of the code highlighted green were obtained from the NumPy library (Harris et al., 2020) to make arrays.



```python
def calculate_variance(
        self, feature_set: list[np.ndarray], mean: np.ndarray
    ) -> np.ndarray:
        """Calculate variance of MFCC features across all frames"""
        ssd = np.zeros(self.num_obs)
        count = 0
        for feature in feature_set:
            ssd += np.sum((feature - mean[:, np.newaxis]) ** 2, axis=1)
            count += feature.shape[1]
        variance = ssd / count
        return variance

```

Figure 31- Initialization code; Computing global variance

$$Variance = \frac{Standard\ Deviation}{Number\ of\ Observations}$$

Equation 2 – Variance Equation

The global mean (3 significant figures) is shown in Table 3 in the form of a 1D array with 13 data points corresponding to each word.

Table 3 – Global Mean

-338	102	-16.4	33.0	-14.2	6.66	-6.51	-0.122	-8.30	-2.21	-0.645	-8.41	1.73

The global variance (nearest whole number) is found in Table 4. There were 13 data points again for each of the words. The variance range was found between the least and greatest data points of [87.9, 27861]. The code computed the data points without the rounding in Table 4 therefore rounding error does not occur in any of the codes.

Table 4 – Global Variance

27861	4428	1204	1009	645	412	239	162	184	125	108	108	87

Task 4 and 5 Contributions

15 iterations were performed for both the development set and the evaluation set. It was found that the development set performed well which can be seen in the confusion matrix in Figure 32.

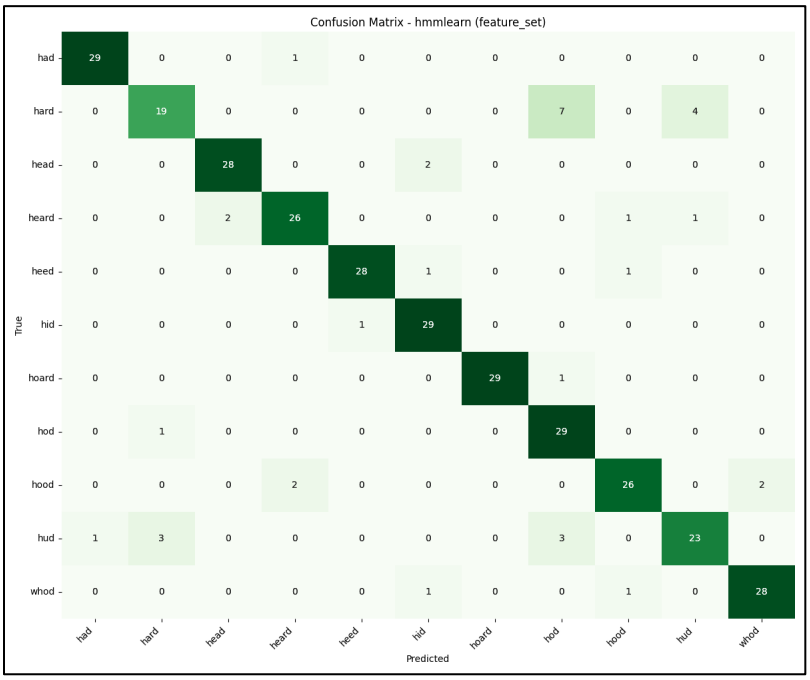


Figure 32 27– Confusion Matrix for development set at 15 epochs

No extreme underfitting appeared from here however Figure 33 shows the error rate of the development set (feature set). The development set shows that overfitting occurred, starting from epochs 3 until 10, however by epoch 15 this was no longer a problem.

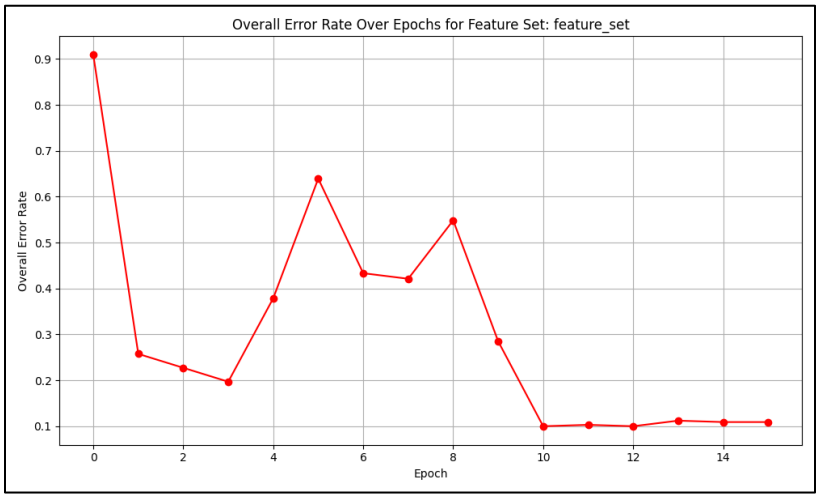


Figure 33 Error rate over 15 epochs for the development set

However, the evaluation set had major overfitting occur which can be deduced from the confusion matrix in Figure 34 and the error rate in Figure 35 which shows that overfitting started at epoch 3 again, like that of the development set.

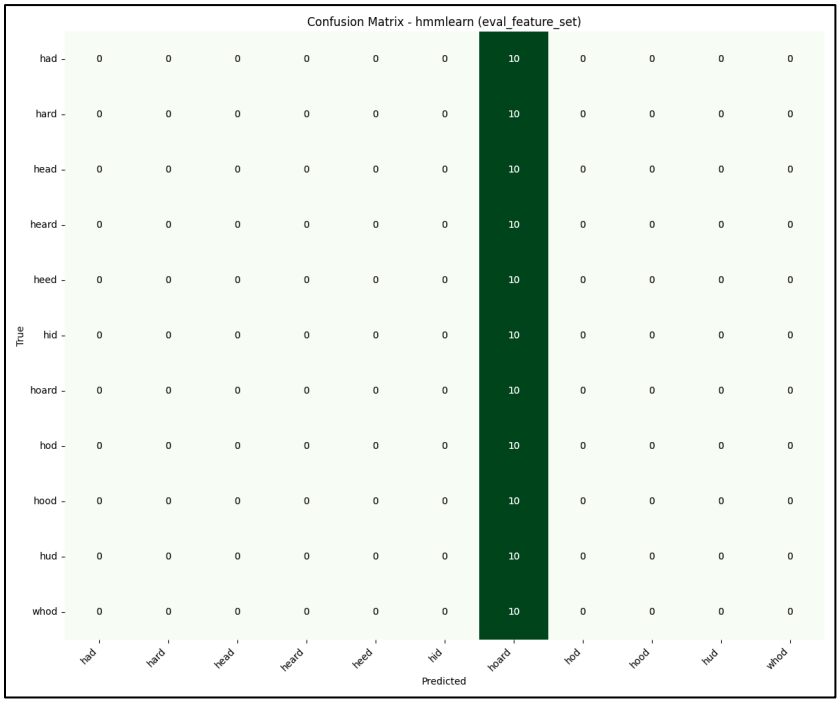


Figure 34 28 – Confusion Matrix for evaluation set at 15 epochs

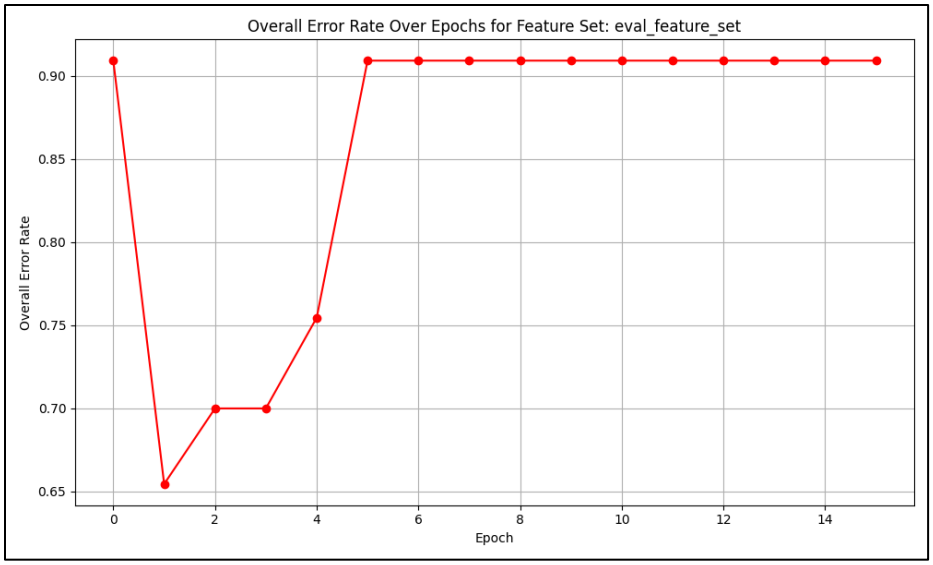


Figure 35 31 – Error rate over 15 epochs for the evaluation set

To combat this overfitting, multiple covariance and iterations were tested out. Covariances of 0.01; 0.1; and 1 were tried out for iterations of 1, 7 and 10. Figure 36 shows the confusion matrices for one of the covariances at each iteration for the evaluation set. The darker green shows the greater values in the prediction versus true.

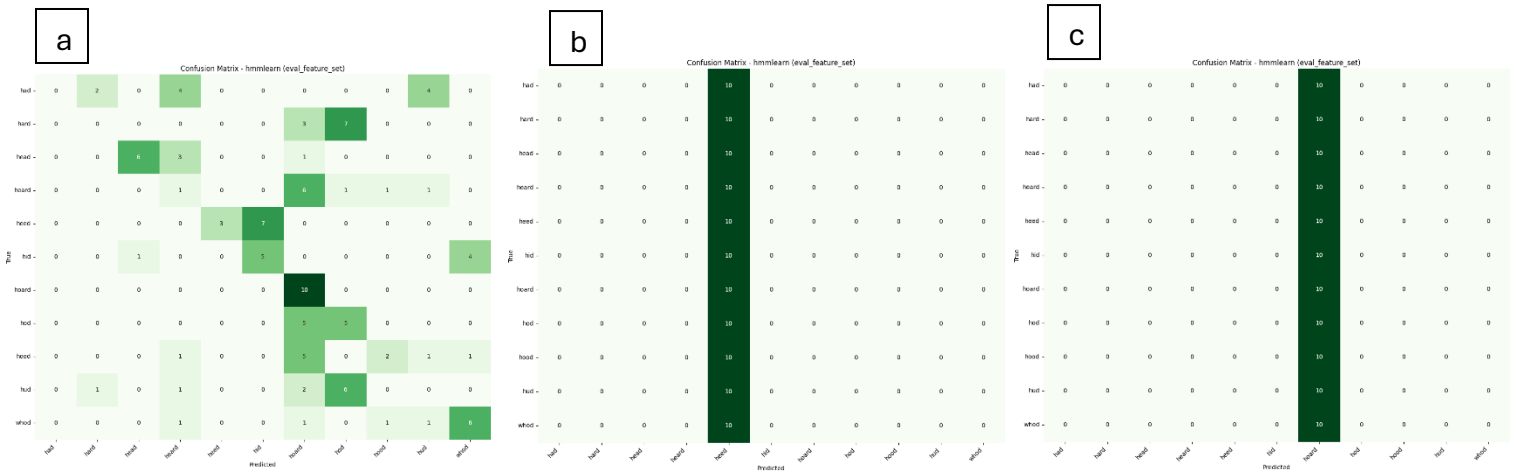


Figure 36 32 – Confusion Matrices for the evaluation set

a) Iteration 1; Covariance 0.1
b) Iteration 7; Covariance 0.01
c) Iteration 10; Covariance 1

We found that the lower iterations in both the evaluation and development set had underfitting despite the covariance. Furthermore, greater iterations seemed to present more accurate results for the development set which can be seen by the underfitting, presented in the confusion matrices, becoming much less prominent.

Changes to covariance presented minor changes to the date for both the evaluation and development set and did not prevent any overfitting in the evaluation set.

Overall, it could be seen that our training allowed for the recognizer to identify the words with great accuracy when it came to our development set which the code was trained on. However, when we tried to apply the same thing to the evaluation set, we found that the code we trained on the development set failed to predict the words accurately and displayed signs of overfitting.

The code to compute the confusion matrix is shown in Figure 37. In this, there were multiple libraries used. The green highlighted code shows all the parts that were obtained from the libraries. “np.” was obtained from NumPy (Harris et al., 2020); “plt” was obtained from Matplotlib.pyplot (Hunter, 2007); “Path” and “logging” were obtained from standard Python libraries.

```
1  def plot_confusion_matrix(cm: np.ndarray, vocab: List[str], implementation: str, feature_set_path: str, model_iter: int) -> None:
2      plt.figure(figsize=(12, 10))
3
4      mask_correct = np.zeros_like(cm, dtype=bool)
5      np.fill_diagonal(mask_correct, True)
6
7      cm_correct = np.ma.masked_array(cm, ~mask_correct)
8      cm_incorrect = np.ma.masked_array(cm, mask_correct)
9
10     sns.heatmap(cm_incorrect, annot=True, cmap='Reds', fmt='d',
11                 xticklabels=vocab, yticklabels=vocab, cbar=False)
12
13     sns.heatmap(cm_correct, annot=True, cmap='Greens', fmt='d',
14                 xticklabels=vocab, yticklabels=vocab, cbar=False)
15
16     plt.title(f'Confusion Matrix - {implementation} ({Path(feature_set_path).stem}) - {model_iter} iterations')
17     plt.xlabel('Predicted')
18     plt.ylabel('True')
19
20     plt.xticks(rotation=45, ha='right')
21     plt.yticks(rotation=0)
22
23     figures_dir = Path("figures")
24     figures_dir.mkdir(exist_ok=True)
25
26     output_path = figures_dir / f"{implementation}_confusion_matrix_{Path(feature_set_path).stem}.png"
27     plt.tight_layout()
28     plt.savefig(output_path)
29     plt.close()
30
31     logging.info(f"\nSaved confusion matrix plot to: {output_path}")
```

Figure 37- Code to compute confusion matrices and plot it

Lines 4 to 8 use a mask to separate the correct predictions from the incorrect predictions where they are coloured with a heatmap as red if incorrect (line 10) and green if correct (line 13). Lines 16 to 21 plot the heatmap and the rest of the code saves the heatmap to an existing directory.

The code to compute the error is shown in Figure 38. This code uses the Matplotlib.pyplot library (Hunter, 2007).

```
1  def plot_overall_error_rate_over_epochs(overall_error_rates: List[float], feature_set: str, figures_dir: str = "figures") -> None:
2      """
3      Plot and save the overall error rate across epochs in a single figure.
4      """
5      figures_path = Path(figures_dir)
6      figures_path.mkdir(exist_ok=True)
7
8      epochs = range(len(overall_error_rates))
9
10     plt.figure(figsize=(10, 6))
11     plt.plot(epochs, overall_error_rates, marker='o', color='r')
12     plt.xlabel("Epoch")
13     plt.ylabel("Overall Error Rate")
14     plt.title(f"Overall Error Rate Over Epochs for Feature Set: {feature_set}")
15     plt.grid(True)
16     plt.tight_layout()
17     plt.savefig(figures_path / f"{feature_set}_overall_error_rate_over_epochs.png")
18     plt.close()
```

Figure 38 33 – Code to compute the error rate and plot it

This code computes the error rate across the epochs and plots them on a single graph. Lines 5 and 6 set the directory in which the plots will be saved. Line 8 indexes the epochs and plots them across the x-axis. Lines 10 and onwards plot the graph with several parameters including red lines and titles.

Conclusion

This assignment demonstrates the practical implementation of Hidden Markov Models with Mel-Frequency Cepstral Coefficients for isolated-word recognition tasks. The 8-state left-right HMM topology, combined with 13-dimensional MFCC features, effectively captures the spectral envelopes of speech signals, while the implementation of log-domain probability calculations ensures numerical stability throughout the training and recognition process.

The system's performance benefits from a robust initialization strategy using global statistics and variance flooring. However, the current implementation shows signs of overfitting to the training data, with performance degradation on unseen test samples. This overfitting challenge could be addressed through several approaches: implementing Gaussian Mixture Models to better model the emission probabilities with fewer states, reducing the number of states from the current eight-state topology to prevent overparameterization, and employing early stopping during training by monitoring the likelihood on a validation set to determine optimal convergence points. These modifications would help reduce model complexity while maintaining discriminative power, leading to better generalization on unseen data.

Future enhancements could focus on implementing these anti-overfitting measures, along with incorporating delta features for improved dynamic modeling and implementing speaker adaptation techniques. Additional improvements could include data augmentation strategies and the introduction of noise robustness techniques to enhance generalization.

In conclusion, this assignment demonstrates the practical application of HMM theory but also highlights important considerations for designing HMM-based systems, particularly the need to handle numerical instabilities through log-domain computations and to prevent overfitting through careful curation of model training data.

Figures list

References

Frank Lu (2024). *GitHub - frankcholula/sapr: Speech & Audio Processing & Recognition*. [online] GitHub. Available at: https://github.com/frankcholula/sapr [Accessed 6 Dec. 2024].

hmmlearn (2018). *Tutorial — hmmlearn 0.3.3.post1+ge01a10e documentation*. [online] Readthedocs.io. Available at: https://hmmlearn.readthedocs.io/en/latest/tutorial.html [Accessed 5 Dec. 2024].

Hristov, H. (2022). *Baeldung*. [online] Baeldung on Computer Science. Available at: https://www.baeldung.com/cs/hidden-markov-model [Accessed 5 Dec. 2024].

librosa.org. (n.d.). *librosa — librosa 0.8.0 documentation*. [online] Available at: https://librosa.org/doc/latest/index.html.

Maël Fabien, M. (2020). *Introduction to Automatic Speech Recognition (ASR)*. [online] maelfabien.github.io. Available at: https://maelfabien.github.io/machinelearning/speech_reco/# [Accessed 5 Dec. 2024].

Najkar, N., Razzazi, F. and Sameti, H. (2010). A novel approach to HMM-based speech recognition systems using particle swarm optimization. *Mathematical and Computer Modelling*, 52(11-12), pp.1910–1920. doi:https://doi.org/10.1016/j.mcm.2010.03.041.

Patterson, D.J. (2020). *Hidden Markov Models*. [online] YouTube. Available at: http://www.youtube.com/playlist?list=PLix7MmR3doRo3NGNzrq48FItR3TDyuLCo [Accessed 5 Dec. 2024].

Rabiner, L.R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), pp.257–286. doi:https://doi.org/10.1109/5.18626.

Siba Prasad Mishra, Pankaj Warule and Deb, S. (2023). Speech emotion recognition using MFCC-based entropy feature. *Signal, Image and Video Processing*, 18(1), pp.153–161. doi:https://doi.org/10.1007/s11760-023-02716-7.