Computer Science 401 — 24 Feb 2026
St. George Campus — University of Toronto

Homework Assignment #3
Due: Thursday, 2 April 2026 at at 23h59 (11:59pm)

**Email**: `csc401-2026-01-a3@cs.toronto.edu`. Please prefix your subject with `[CSC401_W26-A3]`.
All accessibility and extension requests should be directed to `csc401-2026-01-a3@cs.toronto.edu`.
**Lead TAs**: David Wei.
**Instructors**: Gerald Penn.

## Overview

This assignment introduces you to Gaussian mixture modelling, recurrent neural networks, and dynamic programming for speech data. The assignment is divided into two sections. In the first, you will experiment with acoustic-based classification for 1) speaker identification, and 2) deception detection. In the second section, you will 1) evaluate two speech recognition engines, and 2) perform a naive speaker verification.

Data for the deception detection task come from the **CSC Deceptive Speech** corpus, which was developed by Columbia University, SRI International, and University of Colorado Boulder. It consists of 32 hours of audio interviews from 32 native speakers of Standard American English (16 male, 16 female) recruited from the Columbia University student population and the community. The purpose of the study was to distinguish deceptive speech from non-deceptive speech using machine learning techniques on extracted features from the corpus.

All starter code is available on **MarkUs**.

Data are on the `teach.cs` servers in the `/u/cs401/A3/data/` directory. Each sub-folder represents speech from one speaker and contains raw audio, pre-computed MFCCs, and orthographic transcripts. Further file descriptions are in Appendix A.

All code should be compatible with the default Python 3 environment on `teach.cs`. You should call your scripts with the command `python3.12`.

**Please check Piazza regularly for assignment updates and help.**

---

## 1 Sequence classification: Speakers and lies [35 marks]

Speaker identification is the task of correctly identifying speaker $s_c$ from among $S$ possible speakers $s_{i=1..S}$ given an input speech sequence $X$, consisting of a succession of $d$-dimensional real vectors. In the interests of efficiency, $d = 13$ in this assignment. Each vector represents a small 25 ms unit of speech called a *frame*. Speakers are identified by training data that are ascribed to them. This is a discrete classification task (choosing among several speakers) that uses continuous-valued data (the vectors of real numbers) as input.

### Gaussian Mixture Models

*Gaussian mixture models* (GMMs) are often used to generalize models from sparse data. They can tightly constrain large-dimensional data by using a small number of components but can, with many more components, model arbitrary density distributions. Sometimes, they are simply used because the domain being modelled appears to have multiple modes.

Given $M$ components, GMMs are modelled by a collection of parameters, $\theta = \{\omega_{m=1..M}, \mu_{m=1..M}, \Sigma_{m=1..M}\}$, where $\omega_m$ is the probability that an observation is generated by the $m^{th}$ component. These are subject to the constraint that $\sum_m \omega_m = 1$ and $0 \leq \omega_m \leq 1$. Each component is a multivariate Gaussian distribution, which is characterized by that component's mean, $\mu_m$, and covariance matrix, $\Sigma_m$. For reasons of computational efficiency, we will reintroduce some independence assumptions by assuming that every component's covariance matrix is diagonal, i.e.:

$$\Sigma_m = \begin{pmatrix} \Sigma_m[1] & 0 & \cdots & 0 \\ 0 & \Sigma_m[2] & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & \Sigma_m[d] \end{pmatrix}$$

for some vector $\vec{\Sigma}_m$. Therefore, only $d$ parameters are necessary to characterize a component's (co)variance.

## 1.1 Utility functions [10 marks]

We need three utility functions for our GMM. Your first task is to complete the implementation of those functions in `a3_gmm.py`.

First, implement `log_b_m_x`, which implements the log observation probability of $x_t$ for the $m^{th}$ mixture component, i.e., the log of:

$$b_m(\vec{x}_t) = \frac{\exp\left[-\frac{1}{2}\sum_{n=1}^{d}\frac{(x_t[n]-\mu_m[n])^2}{\Sigma_m[n]}\right]}{(2\pi)^{d/2}\sqrt{\prod_{n=1}^{d}\Sigma_m[n]}} \tag{1}$$

Next, implement `log_p_m_x`, which is the log probability of $m$ given $x_t$ using model $\theta$, i.e., the log of:

$$p(m|\vec{x}_t;\theta) = \frac{\omega_m b_m(\vec{x}_t)}{\sum_{k=1}^{M}\omega_k b_k(\vec{x}_t)} \tag{2}$$

Finally, implement `logLik`, which is the log likelihood of a set of data $X$, i.e.:

$$\log P\left(\tilde{X};\theta_s\right) = \sum_{t=1}^{T}\log p(\vec{x}_t;\theta_s) \tag{3}$$

where

$$p(\vec{x}_t;\theta) = \sum_{m=1}^{M}\omega_m b_m(\vec{x}_t) \tag{4}$$

and $b_m$ is defined in Equation 1. For efficiency, we just pass $\theta$ and precomputed $b_m(\vec{x}_t)$ to this function.

Please follow the instruction given in your starter code carefully. It should provide you a very detailed guideline on expected input, output and function behaviour.

## 1.2 Training Gaussian mixture models [5 marks]

Now we train an $M$-component GMM for each of the speakers in the data set. Specifically, for each speaker $s$, train the parameters $\theta_s = \{\omega_{m=1..M}, \mu_{m=1..M}, \Sigma_{m=1..M}\}$ according to the method described in Appendix B. In all cases, assume that covariance matrices $\Sigma_m$ are diagonal. Start with $M = 8$. You'll be asked to experiment with that in Section 1.4. Complete the function `train` in `a3_gmm.py`.

## 1.3 Classification with Gaussian mixture models [5 marks]

Now we test each of the test sequences we've already set aside for you in the `main` function. I.e., we check if the *actual* speaker is also the most likely speaker, $\hat{s}$:

$$\hat{s} = \underset{s=1,\ldots,S}{\arg\max} \; \log P\left(\tilde{X}; \theta_s\right) \tag{5}$$

To complete the function `test` in `a3_gmm.py`. Run through a train-test cycle, and save the output that this function writes to stdout, using the $k = 5$ top alternatives, to the file `a3_gmmLiks.txt`.

## 1.4 Experiments and discussion [10 marks]

Experiment with the settings of $M$ and *maxIter* (or $\epsilon$ if you wish). For example, what happens to classification accuracy as the number of components decreases? What about when the number of possible speakers, $S$, decreases? You will be marked on the detail with which you empirically answer these questions and whether you can devise one or more additional valid experiments of this type.

Additionally, your report should include short hypothetical answers to the following questions:

- How might you improve the classification accuracy of the Gaussian mixtures, without adding more training data?
- When would your classifier decide that a given test utterance comes from none of the trained speaker models, and how would your classifier come to this decision?
- Can you think of some alternative methods for doing speaker identification that don't use Gaussian mixtures?

Put your experimental analysis and answers to these questions in the file `a3_gmmDiscussion.txt`.

## 1.5 End-to-end truth-and-lie detection with GRUs [5 marks]

Each of the utterances has been labelled as either truthful or deceitful (see Appendix A). Your task is to train and test models to tell these utterances apart using the provided data.

- Using the acoustic sequences from the previous GMM questions as input, use *torch.nn.GRU* to create a simple **unidirectional** GRU with **one hidden layer**. This GRU takes in MFCC vectors as inputs, and output a single output (truth or false). The starter code labels lies as 1 and truth as 0.
- For this part, modify the `__init__` method of the `LieDetector` in `a3_model.py` to create a GRU as specified above. In addition, fill in the code to create a linear layer to project the GRU's outputs to logits.
- Experiment by training the model using different hidden sizes of 5, 10, and 50.
- To train, run:
  ```
  srun -p csc401 --pty python3.12 train.py --source /u/cs401/A3/data/ --hidden_size <value>
  ```
  If you wish, you can experiment with the hyperparameters by specifying `--batch_size`, `--lr`, `--epochs` or `--optimizer` (either 'adam' or 'sgd'; by default, it is 'adam').

The `srun -p csc401 --pty` part of the command requests a compute node on teach.cs. This will provide you with dedicated CPUs and reduce congestion during peak times. The `--pty` flag enables pseudo-terminal mode, which allows your breakpoints to work properly. However, if you want to run the assignment locally, you should remove it.

Is there a trend in performance with different hidden sizes? Explain this trend or lack thereof in 1-2 sentences. Write your model configurations, the detection performance (accuracy) and answers to the discussion questions in the file `a3_detectionDiscussion.txt`.

---

## 2 Dynamic Programming in Speech [35 marks]

### 2.1 Word Error Rate in Speech Recognition [15 marks]

Automatic speech recognition (ASR) is the task of correctly identifying a word sequence given an input speech sequence $X$. To simplify your lives, we have run two popular ASR engines on our data: the open-source and highly customizable **Kaldi** (specifically, a bi-directional LSTM model trained on the Fisher corpus), and the neither-open-source-nor-particularly-customizable **Google Speech API**.

We want to see which of Kaldi and Google are the most accurate on our data. For each speaker in our data, we have three transcript files: `transcripts.txt` (the gold-standard transcripts, from humans), `transcripts.Kaldi.txt` (the ASR output of Kaldi), and `transcripts.Google.txt` (the ASR output of Google); see Appendix A.

For this part of the assignment, we want you to complete the file `a3_levenshtein.py`. Specifically, we define the `Levenshtein` function that accepts lists of words $r$ (Reference) and $h$ (hypothesis), and return a 4-item list containing the floating-point WER, the number of substitutions, the number of insertions, and the number of deletions where

$$WER = \frac{\#\text{Insertions} + \#\text{Deletions} + \#\text{Substitutions}}{\#\text{ReferenceWords}}$$

Recall from lecture how Dynamic Programming comes into the play. You should feel familiar with how the modules are structured based on the lecture material. You should complete the pre-defined modules `initialize`, `step` and `finalize` and also fill in the rest of `Levenshtein` function.

For this part, we assume that the cost of a substitution is 0 if the words are identical and 1 otherwise. The costs of insertion and deletion are both 1. **Keep this assumption in mind.**

The `main` function iterates through each of the speakers and each line $i$ of their transcripts as well. You do not have to worry about this part because we wrote the preprocessing and the entire pipeline for you. In the output file `a3_levenshtein.out`, you should see something like this for each line:

```
[SPEAKER] [SYSTEM] [i] [WER] I:[#Insertions], D:[#Deletions], S:[#Substitutions]
```

where [SYSTEM] is either 'Kaldi' or 'Google'.

You should also see the average and standard deviation of WER for each of Kaldi and Google, respectively.

### 2.2 Dynamic Time Warping in Speaker Verification [20 marks]

In speaker verification, the goal is to confirm or deny a speaker's claimed identity based on their voice. This involves comparing a spoken sample (the "candidate" sample) against a stored template or model of the claimed speaker's voice (the "reference" sample). Given the natural variations in speech—even from the same speaker—due to factors like emotion, health, environment, and context, exact matches between samples are improbable. DTW addresses this by aligning the temporal sequences of feature vectors (often MFCCs extracted from the speech samples) in a way that minimizes the overall distance between them, accounting for variations in speaking rate and pronunciation.

Dynamic Time Warping (DTW) is another DP algorithm used in the field of speech processing, particularly in applications like speaker verification, speech recognition, and voice activity detection. Its primary function is to measure the similarity between two temporal sequences, which could vary in speed or duration, making it especially suitable for dealing with the temporal variabilities inherent in human speech.

For this part of the assignment, we want you to complete the file `a3_dtw.py`. Specifically, we define the `DTW` function in a very similar manner to `Levenshtein` from last part. The goal is to relate two pieces of MFCC data with DTW. In DTW, the similarity between two sequences $X \in \mathbb{R}^{N \times L}$ and $Y \in \mathbb{R}^{M \times L}$ is defined by the DTW distance $\texttt{DTW}(X, Y)$:

$$\texttt{DTW}(X, Y) = D(N, M)$$

Where $D$ is the **Accumulative Cost Matrix** filled by the following DP value function:

$$D(n,m) = \begin{cases} c(x_1, y_1) & n = 1, m = 1 \\ \sum_{k=1}^{n} c(x_k, y_1) & n \in [2:N], m = 1 \\ \sum_{k=1}^{m} c(x_1, y_k) & n = 1, m \in [2:M] \\ \min\{D(n-1, m-1), D(n, m-1), D(n-1, m)\} + c(x_n, y_m) & 1 < n \le N, 1 < m \le M \end{cases}$$

and $c(x, y)$ is the cost function. For this assignment, since $x, y \in \mathbb{R}^L$, **we use the Euclidean distance between two vectors as the cost function**.

In addition to $\texttt{DTW}(X, Y)$, we are also interested in tracing the Optimal Warping Path (OWP) $p^* = \{p_1, ..., p_L\}$, where each element $p_l$ for $l \in [1:L]$ is defined as:

$$p_l = \begin{cases} (1,1) & l = 1 \\ \begin{cases} (1, m-1) & n = 1, m \in [2:M] \\ (n-1, 1) & n \in [2:N], m = 1 \\ \arg\min\{D(n-1, m-1), D(n, m-1), D(n-1, m)\} & 1 < n \le N, 1 < m \le M \end{cases} & 1 < l < L, p_{l+1} = (n, m) \\ (N, M) & 1 = L \end{cases}$$

We have selected a few voice segments as a simple test for your code. You should be able to identify the speaker that matches the reference speaker. You can find the right answer by observing the path we used to load the data. Getting an intuitive answer with the default test should be a good verification for your code.

You can play around the values in the main function. For example, you can modify the reference speaker to see what happens when you switch to a different speaker; you can try to edit the timestamps to navigate to a different segment of the same speaker and see how that affects your result. You can also listen to the raw audio of those selected segments. Write briefly about 1) what you did, 2) what you have observed for each change you made and 3) why you think the result is the way it is, in `a3_dtwDiscussion.txt`. You will be marked mainly on the effort rather than the results.

## Submission requirements

This assignment is submitted electronically. Submit your assignment on MarkUs. You should submit:

1. All your code for `a3_gmm.py`, `a3_model.py`, `a3_levenshtein.py` and `a3_dtw.py`.
2. The output file `a3_gmmLiks.txt`.
3. Your discussion files `a3_gmmDiscussion.txt`, `a3_detectionDiscussion.txt` and `a3_dtwDiscussion.txt`.
4. The `ID.txt` file available on the course website, with your details filled in.

Do not `tar` or `compress` your files, and do not place your files in subdirectories.

## Using your own computer

If you want to do some or all of this assignment on your laptop or other computer, you will have to do the extra work of downloading and installing the requisite software and data. You take on the risk that your computer might not be adequate for the task. You are strongly advised to upload regular backups of your work to teach.cs, so that if your machine fails or proves to be inadequate, you can immediately continue working on the assignment at teach.cs. When you have completed the assignment, you should try your programs out on teach.cs to make sure that they run correctly there. **A submission that does not work on teach.cs will get zero marks.**

## A Appendix: Details on CSC data set

Each utterance is represented by the following file types:

| | |
|---|---|
| `*.wav` | The original speech waveform sampled at 16kHz. |
| `*.mfcc.py` | The Mel-frequency cepstral coefficients obtained from an analysis of the waveform, in numPy format. Each row represents a 25ms frame of speech and consists of 13 floating point values. |
| `*.txt` | Label and orthographic transcription of each utterance, for each of Kaldi and Google ASR, and human gold-standard. |

Participants were told that they were participating in a communication experiment which sought to identify people who fit the profile of top entrepreneurs in America. To this end, participants performed tasks and answered questions in six areas; they were later told that they had received low scores in some of those areas and did not fit the profile. The subjects then participated in an interview where they were told to convince the interviewer that they had actually achieved high scores in all areas and that they did indeed fit the profile. The interviewer's task was to determine how he thought the subjects had actually performed, and he was allowed to ask them any questions other than those that were part of the subjects' tasks. For each question from the interviewer, subjects were asked to indicate whether the reply was true or contained any false information by pressing one of two pedals hidden from the interviewer under a table.

Interviews were conducted in a double-walled sound booth and recorded to digital audio tape on two channels using Crown CM311A Differoid headworn close-talking microphones, then downsampled to 16 kHz. Interviews were orthographically transcribed by hand using the NIST EARS transcription guidelines. Labels for local lies were obtained automatically from the pedal-press data and hand-corrected for alignment, and labels for global lies were annotated during transcription based on the known scores of the subjects versus their reported scores.

MFCCs were obtained using the `python_speech_features` module using default parameters, i.e., 25 ms windows, 13 cepstral coefficients, and 512 fast Fourier transform coefficients

Each **transcript file** has the same format, where the $i^{th}$ line is:

```
[i] [LABEL] [TRANSCRIPT]
```

where $i$ corresponds to `i.wav` and `i.mfcc.npy`, [LABEL] is the Global Lie label, and [TRANSCRIPT] is the actual transcript orthography. Global Lie valence and the version of the pre-interview task for the utterance appears before the colon (e.g., T/H) and the section name appears after the colon (e.g., INTERACTIVE).

**Global Lie valence** is indicated as: T == Truth; LU == Lie Up (subject claims better performance than was actually achieved); and LD == Lie Down (subject claims worse performance). The task version is indicated as: H == Hard; and E == Easy. So, for example, T/H:INTERACTIVE indicates that the subject is telling the truth based on having performed the hard version of the Interactive task.

## B Appendix: Training Gaussian mixture models

```
Input: MFCC data $X$, number of components $M$, threshold $\epsilon$, and $maxIter$
begin
    Initialize $\theta$ ;
    $i := 0$ ;
    $prev\_L := -\infty$ ; $improvement = \infty$;
    while $i < maxIter$ and $improvement >= \epsilon$ do
        ComputeIntermediateResults ;
        $L :=$ ComputeLikelihood $(X, \theta)$ ;
        $\theta :=$ UpdateParameters $(\theta, X, L)$ ;
        $improvement := L - prev\_L$ ;
        $prev\_L := L$ ;
        $i := i + 1$ ;
    end
end
```

**Algorithm 1:** GMM training algorithm.

For `ComputeIntermediateResults`, it is strongly recommended that you create two $M \times T$ numPy arrays – one to store each value from Equation 1 and the other to store each value from Equation 2. In fact, we've set up the function `logLik` to encourage you to do this, to avoid redundant computations. You will use these values in both `ComputeLikelihood` and `updateParameters`, where the latter is accomplished thus:

$$\begin{aligned}
\hat{\omega}_m &= \frac{\sum_{t=1}^{T} p\left(m|\vec{x}_t;\theta\right)}{T} \\
\hat{\vec{\mu}}_m &= \frac{\sum_{t=1}^{T} p\left(m|\vec{x}_t;\theta\right)\vec{x}_t}{\sum_{t=1}^{T} p\left(m|\vec{x}_t;\theta\right)} \\
\hat{\Sigma}_m &= \frac{\sum_{t=1}^{T} p\left(m|\vec{x}_t;\theta\right)\vec{x}_t^2}{\sum_{t=1}^{T} p\left(m|\vec{x}_t;\theta\right)} - \hat{\vec{\mu}}_m^2
\end{aligned} \tag{6}$$

In the third equation, the square of a vector on the right-hand side is defined as the component-wise square of each dimension in the vector. *Note that you don't need to break up Algorithm 1 into separate functions as implied – it is only written that way above to emphasize the sequence of steps*