CSC401
St. George Campus

13 March 2022
University of Toronto

Homework Assignment #3
Due: Friday, 8 April 2022 at 19h00 EST, electronically

Speech

TA: Amanjit Kainth (`amanjitsk@cs.toronto.edu`), and Shuja Khalid (`skhalid@cs.toronto.edu`)

1 Introduction

This assignment introduces you to Gaussian mixture modelling, recurrent neural networks with acoustic data, and three basic tasks in speech technology: *speaker identification*, in which we try to determine *who* is talking, *speech recognition*, in which we try to determine *what* was said, and basic *sequence modelling.*

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 evaluate two speech recognition engines.

The data 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.

Data are in `/u/cs401/A3/data/`; 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.

---

2 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 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 ω_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 $0 \le \omega_m \le 1$. Each component is a multivariate Gaussian distribution, which is characterized by that component's mean, μ_m, and covariance matrix, Σ_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.

2.1 Utility functions [10 marks]

We start with implementing three utility functions in `/u/cs401/A3/code/a3_gmm.py`. For your convenience, there is also another starter code, `a3_gmm_structured.py`, with slightly different structures. You can start with either one.

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 θ, 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 θ and precomputed $b_m(\vec{x}_t)$ to this function.

2.2 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 Σ_m are diagonal. Start with $M = 8$. You'll be asked to experiment with that in Section 2.4. Complete the function `train` in `/u/cs401/A3/code/a3_gmm.py`.

2.3 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, $\hat{s}$:

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

Complete the function `test` in `/u/cs401/A3/code/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 `gmmLiks.txt`.

2.4 Experiments and discussion [10 marks]

Experiment with the settings of M and $maxIter$ (or ϵ 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 `gmmDiscussion.txt`.

2.5 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 `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:
  ```
  python train.py --source <path/to/data/folder> --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').

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 `detectionDiscussion.txt`.

3 Automatic 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 ran 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**.

3.1 Evaluation with WER [10 marks]

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.

Complete the file at `/u/cs401/A3/code/a3_levenshtein.py`. Specifically, in the `Levenshtein` function, accept lists of words r (reference) and h (hypothesis), and return a 4-item list containing the floating-point WER, the number of substitution errors, the number of insertion errors, and the number of deletion errors.

$$WER = \frac{\text{numSubstitutions} + \text{numInsertions} + \text{numDeletions}}{\text{numReferenceWords}}$$

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.

Note: During implementation, if multiple possibilities are possible, you can "break the tie" by arbitrary orders. The autograder accepts different answers. An example is r="recognize speech". The hypothesis h="not recognize" can have either of {numSubstitutions=2} or {numInsertions=1, numDeletion=1}.

In the `main` function, iterate through each of the speakers, and iterate through each line i of their transcripts. For each line, preprocess these transcripts by removing all punctuation, setting the text to lowercase, and any tags such as [laughter], [noise], <LG>, <BR> etc. Output the following to stdout:

```
[SPEAKER] [SYSTEM] [i] [WER] S:[numSubstitutions], I:[numInsertions], D:[numDeletions]
```

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

Save this output and put it into `asrDiscussion.txt`.

On the second-to-last line of `asrDiscussion.txt`, in free text, summarize your findings by reporting the average and standard deviation of WER for each of Kaldi and Google, separately, over all of these lines. If you want to be fancy, you can compute a statistical test of significance to see if one is better than the other, but you don't need to.

On the last line of `asrDiscussion.txt`, add a sentence or two describing anything you observe about the types of errors being made by each system, by manually examining the transcript files.

3.2 Hands-on ASR pipeline [5 marks]

The goal of this question is to get familiar with the ASR pipeline end-to-end (from voice to text). There are several steps to building this pipeline.

Step 1. Prepare 3 transcripts of different lengths. Read out the transcripts, record the sound files yourself. Here are some possible examples:

- One word (around 1 second)
- One sentence (around 5 seconds)
- A sentence, but read very slowly (around 10 seconds)

Step 2. Find an ASR system of your choice. There are many systems that come with pre-trained models:

- Kaldi – the most popular system for ASR. There are many small (i.e., computationally-efficient) pre-trained models with good WERs. Kaldi also provides many bash scripts ("recipes") and tutorials to train and use their pre-trained models.
- PyKaldi – a Python wrapper for Kaldi.
- SpeechBrain – a Python package as a Kaldi replacement. You don't need to play around with the bash scripts (except for the `pip install speechbrain`). Note that many SpeechBrain models are large. Transcribing with their models can take multiple minutes.
- Online ASR transcription services like AWS, Google, or Microsoft. Some of them only have limitations regarding the free acoustic data that you can transcribe. After that, users will need to pay for the transcription. Note that if you choose to use online transcription services, we can *not* support any associated cost.

Step 3. Transcribe the three sound files using the ASR system of your choice. Remember to delete the sound files after you finish this problem.

In `asrPipeline.txt`, briefly summarize the transcripts, your procedure to collect the sounds, and the ASR system you use. Comment on the performance of the speech recognition.

4 Bonus [up to 5 marks]

We will give up to 5 bonus marks for innovative work going substantially beyond the minimal requirements. These marks can make up for marks lost in other sections of the assignment, but your overall mark for this assignment cannot exceed 100%. You may decide to pursue any number of tasks of your own design related to this assignment, although you should consult with the instructor or the TA before embarking on such exploration. Certainly, the rest of the assignment takes higher priority. Some ideas:

4.1 Improving the truth-lie detection algorithm [5 marks]

Make the models better than the GMM or GRU would be a nice improvement. Some ideas include:

- Balance the training samples in different classes by, e.g., subsampling or giving different weights to some data samples.
- In addition to using only the MFCC features, you can also extract engineered features, such as those extracted in Assignment 1, from the text transcripts and classify using discriminative models in **scikit-learn**. Are words more discriminative than the audio?
- Consider how errors in transcripts affect those extracted features and therefore overall system accuracy. See Zhou L, Fraser KC, Rudzicz F. (2016) Speech recognition in Alzheimer's disease and in its assessment. In Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH.

Write the discussions to the file `bonusDetectionDiscussion.txt`.

4.2 Dimensionality reduction [5 marks]

Principal components analysis (PCA) is a method that converts some multivariate representation of data into a lower-dimensional representation by transforming the original data according to mutually orthogonal principal components.

Implement an algorithm that discovers a $d \times d'$ matrix W that transforms a d-dimensional vector, $\vec{x}$ into a d'-dimensional vector $\vec{y}$ through a linear transformation based on PCA, where $d' < d$. Repeat speaker identification using data that has been transformed by PCA and report on what you observe, e.g., for different values of d'.

Write the findings in the file `bonusPCA.txt`. Submit all code and materials necessary to repeat your experiments.

4.3 ASR with sequence-to-sequence models [5 marks]

Try to do better than Kaldi or Google by implementing, for example:

Chiu C-C, Sainath TN, Wu Y, *et al.* (2017) State-of-the-art Speech Recognition With Sequence-to-Sequence Models. http://arxiv.org/abs/1712.01769.

Consider using open-source end-to-end ASR using PyTorch or TensorFlow, e.g., deepSpeech.

Write your procedure and the results in the file `bonusBetterASR.txt`. Submit all code and materials necessary to repeat your experiments.

---

5 General specification

We may test your code on different training and testing data in addition to those specified above. Where possible, do not hardwire directory names into your code. As part of grading your assignment, the grader may run your programs using test harness Python scripts. It is therefore important that your code precisely meets the specifications and formatting requirements, including arguments and file names.

If your code uses a file or helper script, it must read it either from the directory in which that code is being executed (i.e., probably 'pwd'), or it must read it from a subdirectory of `/u/cs401` whose absolute path is completely specified in the code. Do **not** hardwire the absolute address of files in your home directory; the grader does not have access to that directory.

All your programs must contain adequate internal documentation to be clear to the graders. External documentation is not required.

This assignment is in **Python 3**.

5.1 Submission requirements

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

1. All your code for `a3_gmm.py`, `a3_gmm_structured.py`, `model.py` (from Sec 2.5), and `a3_levenshtein.py` (including helper files, if any).

2. The output file `gmmLiks.txt`

3. Your discussion files `gmmDiscussion.txt`, `asrDiscussion.txt`, `asrPipeline.txt`, and `detectionDiscussion.txt`

4. The `ID` file available from the course web-site.

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

5.2 Academic integrity

This is your last assignment of this semester. In the past years, some students posted their implementations to their own GitHub repositories after the semesters. We have requested that they change the visibility to private. Please do NOT publicize your solutions. **Copying the solutions of other students violates the academic integrity** – your codes should be the results of your honest efforts. The questions change from year to year anyways.

6 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.** Question 3.2 is an exception – it is more convenient that you use a system with admin access.

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*