Lab 1: My First Front End

ELEN E6884/COMS 86884: Speech Recognition

Due: September 30, 2005 at 6pm

1 Overview

The goal of this assignment is for you, the student, to write a basic ASR front end and to evaluate it using a dynamic-time warping recognition system. It is meant to help you understand what basic signal processing steps are used in ASR, and why they are taken.

The lab consists of the following parts:

- **Part 0: Familiarization with the data (Required)** — Listen to a few utterances in the data set, until you believe you are processing actual speech signals.
- **Part 1: Write a front end (Required)** — Write a complete mel-frequency cepstral coefficient (MFCC) front end, except for the FFT, which will be provided.
- **Part 2: Implement dynamic time warping (Optional)** — Write a function that implements DTW.
- **Part 3: Evaluate different front ends using a DTW recognizer (Required)** — Run experiments on the TIDIGITS data set comparing the performance of different portions of the front end you implemented.
- **Part 4: Try to beat the MFCC front end (Optional)** — Try to develop a modification to the given MFCC front end (or do something completely different) to get better performance. The student achieving the best performance on a test set (that will not be released until after the assignment is due) will be awarded some sort of crappy prize.

All of the files needed for the lab can be found in the directory `~stanchen/e6884/lab1/`. Before starting the lab, please read the file `lab1.txt`; this includes all of the questions you will have to answer while doing the lab. Questions about the lab can be posted on Courseworks (`https://courseworks.columbia.edu/`); a discussion topic will be created for each lab. **Note:** The hyperlinks in this document are enclosed in square brackets; you need an online version of this document to find out where they point to.

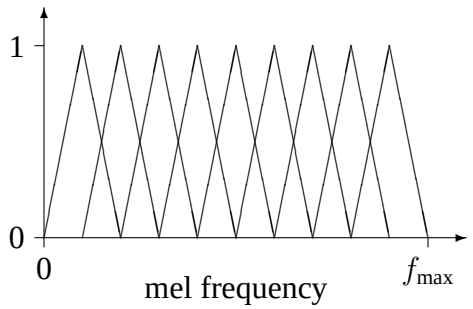

Figure 1: Mel binning

2 Part 0: Familiarization with the data (Required)

The data to be used in this lab is the TIDIGITS corpus, a standard ASR data set consisting of clean recordings of about 300 speakers reading digit sequences. In this lab, we are doing isolated digit recognition, so we will use only the utterances consisting of a single digit: each speaker was required to say each of the 11 digits (*one* through *nine*, *zero*, and *oh*) by itself two times.

For this part of the exercise, just listen to a few samples of the data. Here is one sample for each digit; click on the links to listen to each sample: [Sample 1], [Sample 2], [Sample 3], [Sample 4], [Sample 5], [Sample 6], [Sample 7], [Sample 8], [Sample 9], [Sample 10], and [Sample 11].

3 Part 1: Write a front end (Required)

In this part of the exercise, you will be writing an MFCC front end, except for the FFT which will be provided for you. (Pre-emphasis and adding deltas will be ignored.) To make things easier for you and to make it convenient to run experiments with different portions of the front end, we have written a whole slew of glue code for you. All you have to do is complete the code in three functions implementing three different parts of an MFCC. In a later part of the lab, the signal processing algorithms you implement here will be used in a primitive speech recognizer to evaluate the relative efficacy of different signal processing schemes. **Note:** There may be differences between the equations given here and the corresponding equations in the slides for the lectures. For the lab, use the versions of the equations given here!! (There is no one "correct" version of MFCC's; different sites have implemented different variations.)

To prepare for the exercise, create the relevant subdirectory and copy over the needed files:

```
mkdir -p ~/e6884/lab1/
cd ~/e6884/lab1/
cp ~stanchen/e6884/lab1/Lab1_FE.C .
cp ~stanchen/e6884/lab1/.mk_chain .
```

Your job will be to fill in the sections between the markers `BEGIN_LAB` and `END_LAB` in three different functions. Read `Lab1_FE.C` to see what input and output structures need to be accessed.

Computing MFCC's consists of the following steps: windowing, FFT, mel binning (and log), and DCT. The three different functions to be written correspond to these steps minus the FFT. Each function should take a matrix of values (representing the feature values output by the last signal processing module) and produce a new matrix of values. In the case of windowing, the input is a vector of waveform samples rather than a matrix of feature values.

3.1 Algorithms To Implement

3.1.1 Windowing

The first function to be completed takes a sequence of raw waveform samples for an utterance and performs windowing, returning a 2-D array containing the windowed samples for each frame.

That is, before we begin frequency analysis we want to chop the waveform into overlapping *windows* of samples, so that we can perform short-time frequency analysis on each window. For example, we might choose a window width of 256 samples with a shift of 100 samples. In this case, the first frame/window would contain samples 0 to 255; the second frame/window would contain samples 100 to 355; etc. In terms of the code, the values for the first frame would be written in the locations `outBuf[0][0..255]`, the values for the second frame in `outBuf[1][0..255]`, etc. In the lab, the total number of output frames has been computed for you (*i.e.*, `outFrames`).

Both rectangular and Hamming windowing should be supported. In rectangular windowing, sample values are unchanged. For Hamming windowing, the values in each frame/window should be modified by a Hamming window. Recall that for samples $\{s_i, i = 0, \ldots, N-1\}$, the samples $\{S_i\}$ after being Hammed are:

$$S_i = \left(0.54 - 0.46 \cos \frac{2\pi i}{N-1}\right) s_i$$

3.1.2 Mel binning

The second function to be completed should implement mel binning. Preceding this step, an FFT will have been performed for each frame on windowed samples $\{s_i, i = 0, \ldots, N-1\}$, producing an array $\{S_i, i = 0, \ldots, N-1\}$ of the same size. The real and imaginary parts of the FFT value for frequency $\frac{i}{NT}$ (in Hz) will be held in S_{2i} and S_{2i+1}, respectively, where T is the sampling period for the original signal. In terms of the code, for frame `frm` this corresponds to the locations `inBuf[frm][2*i]` and `inBuf[frm][2*i+1]`. The variable `samplePeriod` holds the value of T (in seconds). For each frame, the magnitude of the (complex) value for each frequency should be binned according to the diagram in Figure 1. (The number of bins in the diagram should not be taken literally; just the shapes of the bins.) The number of bins to use is held in `outDim`.

More precisely, the bins should be uniformly spaced in *mel* frequency space, where

$$\text{Mel}(f) = 1127 \ln(1 + \frac{f}{700})$$

The bins should be perfectly triangular (in *mel frequency space*!), with the right corner of each bin being directly under the center of the next, as in the diagram. The left corner of the left-most bin should be at mel frequency 0, and the right corner of the right-most bin should be at mel frequency $f_{\text{max}} = \text{Mel}(\frac{1}{2T})$. To decide the weight with which each FFT magnitude should be added to each bin, take the value of the curve corresponding to the given bin at the given frequency mapped to mel frequency space. That is, the output values $\{S_i\}$ (before the logarithm) should be

$$S_i = \sum_f |X(f)| \; H_i(\text{Mel}(f))$$

where $X(f)$ is the output of the FFT at frequency f, f ranges over the set of frequencies evaluated in the FFT, and $H_i(f_{\text{mel}})$ is the height of the ith bin at frequency f_{mel} in Figure 1 (where the left-most bin is the 0th bin). (Note that we are *not* squaring $|X(f)|$ in the previous equation, as is sometimes done.)

If this all seems very confusing, it's not as bad as it seems. Basically, you just have to implement the previous equation for each frame. For a frame `frm`, the $\{S_i\}$ are the output values `outBuf[frm][i]` and the $\{X_f\}$ are the input values `inBuf[frm][i]`, where you have to translate from indices i into frequencies f as described earlier in this section, and combine the real and imaginary parts for each frequency when computing $|X_f|$. The Mel function is given above. The trickiest part is figuring out the windowing function $H_i()$. For help, check out equation (6.141) on page 317 of HAH in the assigned readings. (Don't use the filter centers $f[m]$ suggested in the text as they do things differently. Instead, figure them out by studying Figure 1.)

3.1.3 Discrete Cosine Transform

The third function to be completed should implement the discrete cosine transform. For input values $\{s_i, i = 0, \ldots, N-1\}$, the output values $\{S_j, j = 0, \ldots, M-1\}$ are:

$$S_j = \sqrt{\frac{2}{N}} \sum_{i=0}^{N-1} s_i \cos\left(\frac{\pi(j+1)}{N}(i+0.5)\right)$$

where M is the number of cepstral coefficients that you want to keep (*i.e.*, the value `outDim`). This transform should be applied to the feature values for each frame.

3.2 Compiling and testing

Your front end code will be compiled into the program FETOOL, which is a tool that takes a waveform, applies a sequence of signal processing modules to it, and prints out the resulting feature vectors. To compile this program with your front end code, type

```
smk FETool
```

in the directory containing your source file (which must also contain the file `.mk_chain`). If successful, this will create the executable FETOOL in the current directory.

To print out feature values for a sample utterance after applying only your windowing module (with Hamming on), type

```
lab1p1win.sh
```

(This script can be found in `~stanchen/pub/exec/`, which should be on your path from Lab 0.) This shell script starts the executable FETOOL located in the current directory with the appropriate flags. To start up FETOOL in the debugger, add the flag `-debug` to the above line. The "correct" output can be found in the file `p1win.out` in `~stanchen/e6884/lab1/`. Don't sweat it if you don't match the "correct" answer exactly for every step; just try to get reasonably close so the word-error rate numbers that you will calculate later are sensible. For reference, the original waveform can be found in `p1wave.out`.

To print out feature values after applying the windowing, FFT, and mel-binning (w/log) modules, type

```
lab1p1bin.sh
```

To print out feature values after everything (*i.e.*, MFCC features), type

```
lab1p1all.sh
```

The correct outputs can be found in `p1bin.out` and `p1all.out`, respectively. (The correct output after just windowing and FFT can be found in `p1fft.out`.)

4 Part 2: Implement dynamic time warping (Optional)

In this optional exercise, you have the opportunity to implement dynamic-time warping. You may implement any variation you would like, but we advocate the version championed in the Sakoe and Chiba paper: symmetric DTW with $P = 1$. The distance measure used is Euclidean distance, and the distance calculation will be provided for you.

For this exercise, copy the file `Lab1_DTW.H` to the same directory as above. Read this file for instructions on what to do; look for the markers `BEGIN_LAB` and `END_LAB`.

Your DTW code (as well as front end code) will be compiled into the program DCDDTW, which is a tool that does DTW speech recognition given a set of templates. To compile this program using your code, type

```
smk DcdDTW
```

For a description of what this program does, see the next section.

To test your code, you can use the script

```
lab1p2.sh
```

This will run the version of DCDDTW in the current directory on a simple digits data set consisting of 11 training templates from a single speaker and 11 test utterances from the same speaker. The `-debug` flag is supported as before, and the correct output can be found in `p2.out` in `~stanchen/e6884/lab1/`. Note that the correct output will not be useful for those of you who implement alternate versions of DTW.

5 Part 3: Evaluate different front ends (Required)

In this part, you will evaluate different portions of the front end you wrote in Part 1 to see how much each technique affects speech recognition performance. In addition, you will do runs on different test sets to see the difference in difficulties of various testing conditions (speaker-dependent, speaker-independent, etc.).

You will need to be in the directory `~/e6884/lab1/`; all of the scripts will run the version of the program DCDDTW in the current directory. If you haven't done this already, type

```
smk DcdDTW
```

to compile this program with your front end code. This program is an (albeit primitive) speech recognizer, which is also known as a *decoder*. We supply it with a training example (or *template*) for each word that we would potentially like to recognize. To recognize a new utterance, it uses dynamic time warping to find the closest training example and returns the class of that training example.

When using DTW to compare an utterance with a training example, both waveforms are processed using the given signal processing modules, and DTW is performed on the resulting feature vectors. Thus, the quality of the signal processing will greatly affect the accuracy of decoding; the more salient the features that are extracted, the better performance should be. Given a set of templates, a list of signal processing modules to apply, and a set of waveforms to recognize, the program DCDDTW loops through each test utterance in turn and performs DTW to select which word it thinks it is. At the end of the run, it outputs the overall error rate.

The test set used in this part consists of 11 utterances (one of each digit) from each of 10 speakers. While this test set is not large enough to reveal statistically significant differences between some of the contrast conditions, we did not want to use a larger test set so the runs will be quick and it

should be large enough to give you the basic idea. For each test speaker, templates from a different training speaker are used. In the first set of experiments, the training and test speaker in each run are the same. Each script performs ten different runs of DCDDTW (using different training and test speakers) and averages the results.

First, let's see how much each processing step in the front end matters. Run each of the following scripts:

script	description
`lab1p3win.sh`	windowing alone
`lab1p3fft.sh`	windowing+FFT
`lab1p3mel.sh`	windowing+FFT+mel-bin (w/o log)
`lab1p3mellog.sh`	windowing+FFT+mel-bin (w/ log)
`lab1p3dct.sh`	windowing+FFT+mel-bin+DCT
`lab1p3noham.sh`	windowing+FFT+mel-bin+DCT (w/o Hamming)

Remember to keep track of the results, since you will need to fill in these numbers in `lab1.txt`. The first two scripts are quite slow (since the output features are of high dimension), so be patient. In case you are wondering what the accuracy of running DTW on raw (time-domain) waveforms are for this test set, it is 89.1% (accuracy, not error rate). You can run this for yourself if you figure out how, but this run took about 12 hours.

Now, let's see what happens if we relax the constraint that for each test speaker, we use DTW templates from the same speaker (*i.e.*, we no longer do speaker-dependent recognition). Run each of the following scripts (all use the full MFCC front end):

script	relation between test and template speaker
`lab1p3sd.sh`	same
`lab1p3dgd.sh`	same gender and part of US
`lab1p3gd.sh`	same gender
`lab1p3si.sh`	none (*i.e.*, speaker-independent recognition)

6 Part 4: Try to beat the MFCC front end (Optional)

In this optional exercise, you have the opportunity to improve on your MFCC front end. To get started, type the commands:

```
mkdir -p ~/e6884/lab1ec/
cd ~/e6884/lab1ec/
cp ~stanchen/e6884/lab1/.mk_chain .
```

Copy over your `Lab1_FE.C` from Part 1, and also `Lab1_DTW.H` from Part 2 if you have one and like it. If you want to add additional parameters to your front end and have figured out how to do this, then you can also grab a copy of `Lab1_FE.H` from `~stanchen/e6884/lab1/`.

For this exercise, you will need to compile DCDDTW like so:

```
smk DcdDTW
```

We have provided two development sets for optimizing your front ends, a mini-test set consisting of ~100 utterances and a larger test set consisting of ~1000 utterances. To run the DCDDTW in the current directory on these test sets, run

```
lab1p4small.sh
lab1p4large.sh
```

for the small and large test sets, respectively. These scripts are set up to use the full MFCC pipeline (windowing + FFT + melbin w/ log + DCT), and you can change what signal processing is done by modifying the modules you developed in Part 1 of the lab. If the algorithms you would like to implement cannot be realized within this framework (*e.g.*, you don't want to do an FFT), please contact one of the professors and we can tell you how to do this.

The task is set up to be speaker-independent: the speaker used to provide the templates for a test set may have no relation to the speaker of that test set.

The evaluation test set we will use to determine which front end wins the "Best Front End" award will not be released until after this assignment is due, to prevent the temptation of developing techniques that may only work well on the development test sets. This is consistent with the evaluation paradigm used in government-sponsored speech recognition competitions, the primary ones being the [NIST Spoken Language Technology Evaluations].

7 What is to be handed in

Make a copy of the ASCII file `lab1.txt` from the directory `~stanchen/e6884/lab1/`. Fill in all of the fields in this file and E-mail the contents of the file to `stanchen@watson.ibm.com`. (Please paste this file into the main body of the E-mail; *i.e.*, don't include this file as an attachment.)

For the written questions (*e.g.*, "What did you learn in this part?"), answers of a sentence or two in length are sufficient. Our answers for the written questions (as well as any interesting answers provided by you, the students) will be presented at a future lecture if deemed enlightening.