Lab 1: My First Front End

EECS E6870: Speech Recognition

Due: September 30, 2009 at 6pm

SECTION 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. Before you start this lab, make sure you set up your account as described at `http://www.ee.columbia.edu/~stanchen/fall` You need to do this to get some environment variables set correctly.

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 `/user1/faculty/stanchen/e6870/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.

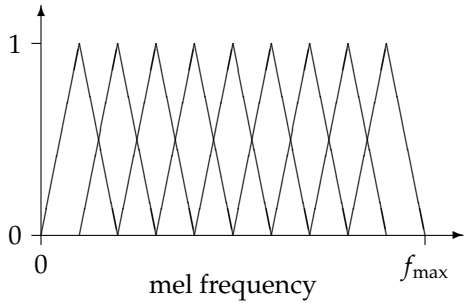

Figure 1: Mel binning

SECTION 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]. (You'll need to use the HTML version of this document to do this part.)

SECTION 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.) In particular, you will be writing a program that reads a file containing the digitized audio signal for a sequence of utterances, and outputs a file containing the acoustic feature vectors for each of those utterances. The format for the input and output files will be Matlab/octave text matrix format; routines for reading/writing this format are provided in the course class library (which can be accessed from C++, Java, and Python). In other words, your program will read a file that looks like this:

```
% name: utt1
% type: matrix
% rows: 15104
% columns: 1
 10
 8
 4
 -2
```

```
 -6
...
% name: utt2
% type: matrix
% rows: 14080
% columns: 1
 -2
 -1
 0
 -2
 -3
...
```

and write a file that looks like this:

```
% name: utt1
% type: matrix
% rows: 74
% columns: 12
 -1.35008 1.70376 1.05819 0.891178 ...
 -0.735043 2.22319 1.23097 1.01904 ...
 0.354016 1.77824 0.0652225 -0.369128 ...
...
% name: utt2
% type: matrix
% rows: 68
% columns: 12
 -0.940911 2.29026 0.922323 1.37886 ...
 -0.46562 2.9617 1.4576 1.01031 0.222223 ...
 -1.08419 2.00002 0.628429 0.890354 ...
...
```

To make things easier for you, we provide a bunch of C++ and Java skeleton code which does all the I/O and stuff, and you just have to fill in the key algorithms. 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 or in the readings. While it's fine not to do things exactly the way we have here, we strongly recommend trying to implement the versions of the equations given in the lab, so you can check your output against the target output we provide. (There is no one "correct" version of MFCC's; different sites have implemented different variations.)

As alluded to in the slides, we can decompose the MFCC computation into a series of primitive processing operations: windowing, FFT, mel binning (and log), and DCT, each taking a matrix as input and a matrix as output. The output matrix of the last step is used as the input matrix of the next. The rows of each matrix correspond to samples or frames, and the columns correspond to the values at each frame. (In the first step, windowing, the input is a waveform which can be viewed as a matrix of width 1.) You'll have to fill in functions that do each of these steps, except for the FFT, which we have provided. We describe the algorithms

to implement in detail in the next section, and in Section 3.2, we give information on how to actually complete the lab in C++ or Java.

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 (see Section 3.2), the values for the first frame would be written in the locations `outFeats(0, 0..255)`, the values for the second frame in `outFeats(1, 0..255)`, etc. In the lab, the total number of output frames has been computed for you (*i.e.*, `outFrameCnt`).

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 (rounded up to the next power of 2). 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 `frmIdx` this corresponds to the locations `inFeats(frmIdx, 2*i)` and `inFeats(frmIdx, 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 `outDimCnt`.

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 $f_{\max} = \text{Mel}(\frac{1}{2T})$. 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 `frmIdx`, the $\{S_i\}$ are the output values `outFeats(frmIdx, i)` and the $\{X_f\}$ are the input values `inFeats(frmIdx, i)`, where you just 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 `outDimCnt`). This transform should be applied to the feature values for each frame.

3.2 Logistics

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

```
mkdir -p ~/e6870/lab1/
cd ~/e6870/lab1/
cp -d /user1/faculty/stanchen/e6870/lab1/* .
```

Be sure to use the `-d` flag with `cp` (so the symbolic links are copied over correctly).

We provide a C++ library which can be accessed from C++ as well as from Java and Python. It includes functions for handling command-line parameters, for the input and output of matrices in Matlab format, and for FFT's. Also, we use the `matrix` class supplied in the Boost C++ libraries (`http://www.boost.org`) as well as the STL `vector` class. For documentation of this stuff, check out `http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/util_`
`http://www.ee.columbia.edu/~stanchen/fall09/e6870/classlib/classmatrix.html`,
and `http://www.sgi.com/tech/stl/Vector.html`.

Here's a small example of matrices and vectors in C++:

```
matrix<double> mat;
//  Resize matrix to 3 rows, 4 cols; values uninitialized.
mat.resize(3, 4);
//  Set values to 0.
mat.clear();
```

```
for (int frmIdx = 0; frmIdx < mat.size1(); ++frmIdx)
    for (int dimIdx = 0; dimIdx < mat.size2(); ++dimIdx)
        mat(frmIdx, dimIdx) += 2.5;

//  Allocate vector with 5 items; values initialized to 0.
vector<int> vec(5);
for (int idx = 0; idx < 5; ++idx)
    vec[idx] *= 3;
```

Here's the analogous example in Java:

```
DoubleMatrix mat = new DoubleMatrix();
mat.resize(3, 4);
mat.clear();
for (int frmIdx = 0; frmIdx < mat.size1(); ++frmIdx)
    for (int dimIdx = 0; dimIdx < mat.size2(); ++dimIdx)
        mat.set(frmIdx, dimIdx, mat.get(frmIdx, dimIdx) + 2.5);

IntVector vec = new IntVector(5);
for (int idx = 0; idx < 5; ++idx)
    vec.set(idx, vec.get(idx) * 3);
```

Also, for C++, we recommend using the `format` construct from Boost instead of `printf()`; this is automatically included when you include `util.H`. Instead of

```
printf("%d %d\n", 4, 6);
```

you would do something like

```
cout << format("%d %d\n") % 4 % 6;
```

This is safer and works correctly with C++ strings.

3.2.1 C++ instructions

The files `util.H` and `util.C` contain routines for reading and writing Matlab-format files, as well as an FFT routine. The file `lab1.C` contains the `main()` program and need not be changed; this program loops through each input utterance and calls the `FrontEnd` class to do signal processing on each and outputs the result. The file `front_end.H` contains the declarations for the file `front_end.C`, which is the file you need to edit. Read the file and fill in all the algorithms that need filling; your job is to fill in the sections between the markers `BEGIN_LAB` and `END_LAB` in three different functions.

To compile, you can just type `make lab1`, which produces the executable `lab1`. The basic usage of this program is

```
lab1 --audio_file in-file --feat_file out-file
```

By default, all signal processing modules are applied, but you can specify flags to exclude certain modules. For example,

```
lab1 --audio_file in-file --feat_file out-file \
    --frontend.window false --frontend.fft false \
    --frontend.melbin false --frontend.dct false
```

excludes every single module, so the output will be the same as the input. Flags can be specified in any order.

In ASR, it is important to be able to specify parameters to programs, because the underlying algorithms typically have lots of parameters that can be tuned to optimize performance on particular domains. For our purposes, command-line parameters let us easily run contrast experiments to reveal how important each signal processing module is to overall performance.

3.2.2 Java instructions

These are similar to the C++ instructions, except that instead of editing `front_end.C`, you need to edit `FrontEnd.java`. (For the later labs, we probably won't be providing Java skeleton code.) The file `Lab1.java` contains the main loop and need not be edited. To compile, you can type `make Lab1`. To run, the basic usage is

```
java Lab1 --audio_file in-file --feat_file out-file
```

The command-line arguments are the same as for C++.

As templates are not supported in Java, to access the C++ template classes in Java we need to rename them. Here is the mapping from C++ template class names to Java (and Python) class names:

C++	Java/Py
vector<int>	IntVector
vector<double>	DoubleVector
vector<string>	StringVector
matrix<int>	IntMatrix
matrix<double>	DoubleMatrix
map<string, string>	StringMap

See the earlier example to see how to set and get elements in matrix and vector objects. To see all the methods that are available for a given C++ class, you can look in the corresponding `.java` wrapper file in the directory `/user1/faculty/stanchen/e6870/lib/edu/columbia/asr/`. Global C++ functions and variables are attached to the class `asr_lib`, *e.g.*, `asr_lib.get_bool_param(...)`.

3.3 Testing

To aid in testing, we provide the "correct" output for the input file `p1in.dat`. All of these files should have been copied into your directory earlier. (The originals can be found in `/user1/faculty/stanchen/e6870/lab1/`, in case you overwrite something.)

The correct output after applying just windowing (with Hamming on) can be found in the file `p1win.dat`. To produce the corresponding output, you can do something like (for C++)

```
lab1 --audio_file p1in.dat --feat_file out-file \
    --frontend.fft false --frontend.melbin false --frontend.dct false
```

For Java, replace `lab1` with `java Lab1`.

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

```
lab1 --audio_file p1in.dat --feat_file out-file --frontend.dct false
```

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

```
lab1 --audio_file p1in.dat --feat_file out-file
```

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

You may find it useful to visualize your output and/or the target output using Matlab. For example, you can look at the input waveform by running `matlab` and typing

```
load p1in.dat
plot(p1in)
```

It is trickier to visualize the other output stages since the data is 2D instead of 1D, but it is straightforward to view the data at a single frame. So, we could do

```
load p1win.dat
plot(p1win(37,:))
```

to view the data for the 37th frame. The output of the FFT alternates real and imaginary components of a value; to visualize this we can plot both types of values together:

```
load p1fft.dat
plot(1:256,p1fft(37,1:2:512),1:256,p1fft(37,2:2:512))
```

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. One check you can do is to run the following script:

```
cat sd.10.list | b018t.run-set.py p018h1.run_dtw.sh %trn %tst
```

This does a little speaker-dependent speech recognition experment (see Section 5 for more details) using the program `lab1` to do the front end processing, or if that file doesn't exist, it uses `Lab1.class`. Try to get the accuracy above 90%, say (it should be 100% if you match the target algorithm exactly). In terms of speed, don't worry abou it, but if running the preceding script takes more than ten minutes, say, you might want to speed things up or Part 3 will take forever.

SECTION 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 $P = 1$.

Here is the big picture: we can construct a simple speech recognizer as follows. First, we get one (or more) training (or *template*) examples for each target word (i.e., digits in our case). Then, for each test example, we find the nearest training example using dynamic time warping; this tells us what digit we think it is. We apply dynamic time warping on the processed acoustic feature vectors rather than the original waveform as this works better. (This is the same technology that was used for voice dialing in early cell phones.)

Here, we use our front end program from Part 1 to generate the acoustic feature vectors for all of our training and test utterances. Then, we can write a program that reads in acoustic feature vectors for training and test utterances and computes the nearest training utterance for each test utterance.

We give C++ skeleton code in `lab1_dtw.C`; this can be compiled by doing `make lab1_dtw`. Fill in the part delimited by `BEGIN_LAB` and `END_LAB`. For Java, you're on your own. One option is to port `lab1_dtw.C` to Java; compare `lab1.C` and `Lab1.C` for hints on how to do this.

To test your code, you can run something like

```
lab1_dtw --verbose true --template_file template.feats \
    --feat_file devtest.feats --feat_label_list devtest.labels
```

(The mentioned files should have been copied into your directory earlier.) This will run your program on a simple digits data set consisting of 11 training templates from a single speaker and 11 test utterances from the same speaker using acoustic feature vectors generated by our front end code. The target output can be found in `p2.out`. Note that the target output will not be useful for those of you who implement alternate versions of DTW.

To make things run faster, you have the option of compiling your code with optimization. You can do this for C++ by doing

```
touch lab1_dtw.C
OPTFLAGS=-O2 make lab1_dtw
```

The `touch` command forces recompilation to occur.

SECTION 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.). We will be performing speech recognition with various data sets using the speech recognition setup described in Part 2. For this exercise, you can either use the DTW engine you wrote in Part 2, or you can use the one supplied by us. If the file `lab1_dtw` is present in the current directory, the script will use that version; otherwise, our version will be called. If you decide to use your own DTW engine, be sure to compile with optimizations for this part. You will need to be in the directory `~/e6870/lab1/`; our scripts will run the version of the program `lab1` or `Lab1.class` in the current directory (the script looks for both) to do signal processing. (The program `lab1` takes priority, so delete this if it exists and you are using Java.)

In each individual experiment, we choose a particular front-end configuration and particular training set. Each experiment consists of 10 runs of `lab1_dtw` over different test speakers where we average the results over runs. In each run, we have a test set of 11 utterances (one of each digit) from the given speaker, and 11 training examples from either the same speaker or a different speaker depending on the condition. By comparing the results between experiments, we can get insight on the importance of various front-end modules, and the difficulties of different data sets. While these test sets are 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 these data sets should be large enough to give you the basic idea.

To do this part, run the following command:

```
lab1p3.sh | tee lab1p3.log
```

This script will take a while to run (hopefully less than an hour total). If you don't see any accuracies printed within a few minutes, you may need to speed up your front end. You can examine the script to see what it is doing; it is located in the directory `/user1/faculty/stanchen/pub/exec/`. The `tee` command takes its input and writes it to both standard output and the given file.

The script `lab1p3.sh` calls the following command repeatedly

```
cat speaker-pair-file | b018t.run-set.py command
```

What this does is run *command* on each of the speaker pairs listed in *speaker-pair-file* and averages the results. The *command* that we run is

```
p018h1.run_dtw.sh [FE-parameters] train-spkr test-spkr
```

What this script does first is extract utterances for the training and test speakers from the TIDIGITS corpus and convert them to Matlab text format. Then, it calls `lab1` or `java Lab1`

to compute acoustic features for all of these utterances. Finally, it runs `lab1_dtw` to do decoding using dynamic time warping and to compute word accuracy.

In the first set of experiments, the training and test speaker in each run are the same, and we compare the performance of using various front-end modules, such as windowing alone; just windowing and FFT; etc. The first two experiments 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.

In the second part of the script, 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). In these experiments (which all use the full MFCC front end), we see how much performance degrades as the variation between training and test speaker increases.

SECTION 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. The instructions for this part are the same as for Part 1, except do everything in a different directory. To get started, type

```
mkdir -p ~/e6870/lab1ec/
cd ~/e6870/lab1ec/
cp -d /user1/faculty/stanchen/e6870/lab1/* .
```

You'll probably want to copy over your existing front end from Part 1 as a starting point.

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 on these test sets, you can do something like:

```
cat si.10.list | b018t.run-set.py p018h1.run_dtw.sh %trn %tst
cat si.100.list | b018t.run-set.py p018h1.run_dtw.sh %trn %tst
```

for the small and large test sets, respectively. 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. If you want to play with parameter settings, you can put `--`*name val* flags immediately after `p018h1.run_dtw.sh` in the commands above. To add new parameters, just add calls to `get_bool_param()`, etc., as in the existing code.

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].

SECTION 7

What is to be handed in

You should have a copy of the ASCII file `lab1.txt` in your current directory. 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.