Lab 1 - Basic feature extraction and classification (2013)

Sunday, June 23, 2013
11:37 PM

PURPOSE

This lab will introduce you to the practice of analyzing, segmenting, feature extracting, and applying basic classifications to audio files. Our future labs will build upon this essential work - but will use more sophisticated training sets, features, and classifiers.

We'll first need to setup some additional Matlab folders, toolboxes, and scripts that we'll use later.

DIRECTORY

Go to the folder: /usr/ccrma/courses/mir2013

MATLAB SETUP

1. Launch Matlab
2. Configure your Path: Add the folder /usr/ccrma/courses/mir2013/**Toolboxes** to your local Matlab path (including all subfolders).
3. Set the "Java Heap Memory" to 900 MB via : File>Preferences>General> Java Heap Memory
   This allows us to load large audio files and feature vectors into memory.
   Click on "OK"
   Click Apply.
4. Restart Matlab.

Why are the Paste / Save keys different? Why does Paste default to Control-Y?

On Linux, Matlab defaults to using Emacs key bindings. If you want Mac or Windows bindings, go:

File menu > Preferences > Keyboard
Switch the Editor/Debugger key bindings to "Windows"

MP3READ

To read **MP3 files** into Matlab, we have a function called **mp3read.** It is used just like **wavread**.

AUDIO FILES

A large collection of audio files for your experimentation are located at /usr/ccrma/courses/mir2013/audio

SECTION 1

Purpose: We'll experiment with the different features for known frames and see if we can build a basic understanding of what they are doing.

1. Make sure to save all of your development code in an .m file. You can build upon and reuse much of this code over the workshop.

   To create a new .m file, chose:
   File> New > Script...
   Save the file as Lab1.m
   You can execute the code in Lab1.m via any of the below options:
   - Type Lab1.m in the command window
     Or…
     press F5 in the Editor to execute the current selected script.
     Or…

- You can execute 1 or more commands selected in the Editor window at a time. Select the code and press F9. Note the Command Window will update.

2. Tab Completion.

   Tab Completion works in Command Window and the Editor. After you type a few letters, hit the Tab key and a popup will appear and show you all of the possible completions, including variable names and functions. This prevents you to mistyping the names of variables - a big time (and aggravation) saver!

   For example, in the Command Line or Editor , try typing wavr and then hitting Tab! ("wavread" should appear)

3. Load the audio file **simpleLoop.wav** into Matlab, storing it in the variable ***x*** and sampling rate in ***fs***.

```
[x,fs]=wavread('/usr/ccrma/courses/mir2013/audio/simpleLoop.wav');
```

4. In this course, we will convert all stereo files to mono. Include this code after your read in WAV files to automatically detect if a file is stereo and convert it to mono.

```
% MAKING MONO
% If your audio files (x) are stereo, here's how to make them mono:
    if size(x,2) == 2
        x= (x(:,1)+x(:,2) ) ./ max(abs(x(:,1)+x(:,2))) ;
        disp('Making your file mono...');
    end
```

5. You can play the audio file by typing using typing

```
sound(x,fs)
```

   To stop listening to a long audio file, press Control-C. Audio snippets less than ~8000 samples will often not play out Matlab. (known bug on Linux machines)

6. Run an onset detector to determine the approximate onsets in the audio file.

```
[onsets] = onset_times(x,fs);  % leighs onset detector with signal and  sample_rate as input
onsets=round(fs*onsets);    % convert onset times in seconds to to samples - round to nearest integer sample
numonsets = length(onsets);
```

   For debugging, we have function which generates mixes of the original audio file and onset times.
   This is demonstrated with test_onsets.m

One of Matlab's greatest features is its rich and easy visualization functions. Visualizing your data at every possible step in the algorithm development process not only builds a practical understanding of the variables, parameters and results, but it greatly aids debugging.

8. Plot the audio file in a figure window.

```
plot(x)
```

9. Now, add a marker showing the position of each onset on top of the waveforms.

```
plot(x); hold on; plot(onsets,0.2,'rx')
```

10. Adding text markers to your plots can further aid in debugging or visualizing problems. Label each onset with it's respective onset number with the following simple loop:

```
for i=1:numonsets
  text(onsets(i),0.2,num2str(i));  % num2st converts an number to a string for display purposes
end
```

Labeling the data is crucial. Add a title and axis to the figures. (**ylabel**, **xlabel**, **title**.)

```
xlabel('seconds')
ylabel('magnitude')
title('my onset plot')
```

11. Now that we can view the various onsets, try out the onset detector and visualization on a variety of other audio examples located in /usr/ccrma/courses/mir2013/audio. Continue to load the various audio files and run the onset detector - does it seem like it works well? If not, yell at Leigh.

Segmenting audio in Frames

As we learned in lecture, it's common to chop up the audio into fixed-frames. These frames are then further analyzed, processed, or feature extracted. We're going to analyze the audio in 100 ms frames starting at each onset.

12. Create a loop which carves up the audio in fixed-size frames (100ms), starting at the onsets.
13. Inside of your loop, plot each frame, and play the audio for each frame.

```
% Loop to carve up audio into onset-based frames
frameSize = 0.100 *fs;      % sec
for i=1:numonsets
   frames{i}= x(onsets(i):onsets(i)+frameSize);
   figure(1);
   plot(frames{i}); title(['frame ' num2str(i)]);
   sound(frames{i} ,fs);
   pause(0.5)
end
```

Feature extract your frames

14. Create a loop which extracts the Zero Crossing Rate ***for each frame***, and stores it in an array.
    Your loop will select 100ms (in samples, this value is = ***fs*** * 0.1) , starting at the onsets, and obtain the number of zero crossings in that frame.

    The command **[z] = zcr(x)** returns the number of zero crossings for a vector x.
    Don't forget to store the value of **z** in a feature array for each frame.

```
clear features
% Extract Zero Crossing Rate from all frames and store it in "features(i,1)"
for i=1:numonsets
   features(i,1) = zcr(frames{i})
end
```

For simpleLoop.wav, you should now have a feature array of 5 x 1 - which is the 5 frames (one at each detected onset) and 1 feature (zcr) for each frame.

Sort the audio file by its feature array.

Let's test out how well our features characterize the underlying audio signal.
To build intuition, we're going to sort the feature vector by it's zero crossing rate, from low value to highest value.

15. If we sort and re-play the audio that corresponds with these sorted frames, what do you think it will sound like? (e.g., same order as the loop, reverse order of the loop, snares followed by kicks, quiet notes followed by loud notes, or ???) Pause and think about this.

16. Now, we're going to play these sorted audio frames, from lowest to highest. (The pause command will be quite useful here, too.) How does it sound? Does it sort them how you expect them to be sorted?

```
[y,index] = sort(features);

for i=1:numonsets
  sound(frames{index(i)},fs)
  figure(1); plot(frames{index(i)});title(i);
  pause(0.5)
End
```

You'll notice how trivial this drum loop is - always use familiar and predictable audio files when you're developing your algorithms.

17. Now that you have this file loading, playing , and sorting working, try this with out files, such as: /usr/ccrma/courses/mir2013/audio/CongaGroove-mono.wav, and /usr/ccrma/courses/mir2013/audio/125BOUNC-mono.WAV.

SECTION 2 - Spectral Features & k-NN

PURPOSE

My first audio classifier: introducing K-NN! We can now appreciate ***why*** we need additional intelligence in our systems - heuristics can't very far in the world of complex audio signals. We'll be using Netlab's implementation of the k-NN for our work here. It proves be a straight-forward and easy to use implementation. The steps and skills of working with one classifier will scale nicely to working with other, more complex classifiers.

We're also going to be using the new features in our arsenal: cherishing those "spectral moments" (centroid, bandwidth, skewness, kurtosis) and also examining other spectral statistics.

TRAINING DATA

First off, we want to analyze and feature extract a small collection of audio samples - storing their feature data as our "training data". The below commands read all of the .wav files in a directory into a structure, **snareFileList**.

1. Use these commands to read in a list of filenames (samples) in a directory, replacing the path with the actual directory that the audio \ drum samples are stored in.

```
snareDirectory = ['/usr/ccrma/courses/mir2013/audio/drum samples/snares/'];
snareFileList = getFileNames(snareDirectory ,'wav')

kickDirectory = ['/usr/ccrma/courses/mir2013/audio/drum samples/kicks/'];
kickFileList = getFileNames(kickDirectory ,'wav')
```

2. To access the filenames contained in the cell array, use the brackets **{ }** to get to the element that you want to access.

   For example, to access the text file name of the 1st file in the list, you would type **snareFileList{1}** Try it out:

```
snareFileList{1}
```

When we feature extract a sample collection, we need to sequentially access audio files, segment them (or not), and feature extract them. Loading a lot of audio files into memory is not always a feasible or desirable operation, so you will create a loop which loads an audio file, feature extracts it, and closes the audio file. Note that the only information that we retain in memory are the features that are extracted.

3. Create a loop which reads in an audio file, extracts the zero crossing rate, and some spectral statistics. The feature information for each audio file (the "feature vector") should be stored as a feature array, with columns being the features and rows for each file.

   Or in Matlab, for example:

   featuresSnare =

   1.0e+003 *

0.5730	1.9183	2.9713	0.0004	0.0002
0.4750	1.4834	2.4463	0.0004	0.0012
0.5900	2.2857	3.1788	0.0003	0.0041
0.5090	1.6622	2.6369	0.0004	0.0051
0.4860	1.4758	2.2085	0.0004	0.0021
0.6060	2.2119	3.2798	0.0004	0.0651
0.4990	2.0607	2.7654	0.0004	0.0721
0.6360	2.3153	3.0256	0.0003	0.0221
0.5490	2.0137	3.0342	0.0004	0.0016
0.5900	2.2857	3.1788	0.0003	0.0012

   In your loop, here's how to read in your wav files, using a structure of file names:

   ```
   [x,fs]=wavread([snareDirectory snareFileList{i}]);    %note the use of brackets for snareFileList
   ```

   Here's an example of how to feature extract for the current audio file..

   ```
   frameSize = 0.100 * fs;   % 100ms
   currentFrame = x(1:frameSize)
   featuresSnare(i,1)   = zcr(currentFrame);
   [centroid, bandwidth, skew, kurtosis]=spectralMoments(currentFrame,fs,8192)
   featuresSnare(i,2:5) = [centroid, bandwidth, skew, kurtosis];
   ```

4. First, extract all of the feature data for the kick drums and store it in a feature array. (For my example, above, I'd put it in "featuresKick")
5. Next, extract all of the feature data for the snares, storing them in a different array.
   Again, the kick and snare features should be separated in two different arrays!

   OK, no more help. The rest is up to you!

BUILDING MODELS

Building Models

1. Examine the feature array for the various snare samples. What do you notice?

2. Since the features are different scales, we will want to normalize each feature vector to a common range - storing the scaling coefficients for later use. Many techniques exist for scaling your features. We'll use linear scaling, which forces the features into the range -1 to 1.

   For this, we'll use a custom-created function called **scale.** Scale returns an array of scaled values, as well as the multiplication and subtraction values which were used to conform each column into -1 to 1. Use this function in your code.

```
[trainingFeatures,mf,sf]=scale([featuresSnare; featuresKick]);
```

Building a k-NN

3. Build a k-NN model for the snare drums in Netlab, using the function **knn.**

We'll the implementation of from the Matlab toolbox "netlab":

```
>help knn
NET = KNN(NIN, NOUT, K, TR_IN, TR_TARGETS) creates a KNN model NET
        with input dimension NIN, output dimension NOUT and K neighbours.
        The training data is also stored in the data structure and the
        targets are assumed to be using a 1-of-N coding.

        The fields in NET are
          type = 'knn'
          nin = number of inputs
          nout = number of outputs
          tr_in = training input data
          tr_targets = training target data
```

Here's an example...

```
labels=[[ones(10,1) zeros(10,1)]; [zeros(10,1) ones(10,1) ]];
```

Which is an array of ones and zeros to correspond to the 10 snares and 10 kicks in our training sample set:

```
labels=
     1     0
     1     0
     1     0
     1     0
     1     0
     1     0
     1     0
     1     0
     1     0
     1     0
     0     1
     0     1
     0     1
     0     1
     0     1
     0     1
     0     1
     0     1
     0     1
     0     1
```

```
[trainingFeatures,mf,sf]=scale([featuresSnare; featuresKick]);

model_snare = knn(5,2,1,trainingFeatures,labels);
```

This k-NN model uses 5 features, 2 classes for output (the label), uses k-NN = 1, and takes in the

feature data via a feature array called trainingFeatures.

These labels indicate which sample in our feature data is a snare, vs. a non-snare. The k-NN model uses this information to build a means of comparison and classification. It is ***really important*** that you get these labels correct - because they are the crux of all future classifications that are made later on. (Trust me, I've made many mistakes in this area - training models with incorrect label data.)

4. **Create a script which extracts features for a single file, re-scales its feature values, and evaluates them with your kNN classifier.**

Evaluating samples with your k-NN

Now that the hard part is done, it's time to throw some feature data through the trained k-NN and see what it outputs.

RESCALING.

In evaluating a new audio file, we need to extract it's features, re-scale them to the same range as the trained feature values, and then send them through the knn.

Some helpful commands:

```
featuresScaled = rescale(features,mf,sf) ;   % This uses the previous calculated linear scaling parameters to adjust the incoming features to the same range.
```

EVALUTING WITH KNN

[voting,model_output]=knnfwd(model_snare **,** featuresScaled **)**

The output **voting** gives you a breakdown of how many nearest neighbors were closest to the test feature vector.

The **model_output** provides a list of whether output is Class 1 or Class 2.

```
output = zeros(size(model_output),2)
output(find(model_output==1),1)=1
output(find(model_output==2),2)=1
```

Now you can visually compare the **output** to **trainlabels**

Once you have completed function, first, test it with your training examples. Since a k-NN model has exact representations of the training data, **it will have 100% training accuracy** - meaning that every training example should be predicted correctly, when fed back into the trained model.

Now, test out with the examples in the folder "test kicks" and "test snares", located in the drum samples folder. These are real-world testing samples…

If the output labels "1" or "0" aren't insightful for you, you can add an if statement to display them as strings "snare" and "kick".

NEED HELP?

Tricks of the trade

Select code in Matlab editor and then press F9. This will execute the currently selected code.

To run a Matlab "cell" (multiline block of code), press Control-Enter with the text cursor in the current cell.

The **clear** command re-initializes a variable. To avoid confusion, you mind find it helpful to clear arrays and structures at the beginning of your scripts.

Common Errors

>??? Index exceeds matrix dimensions.

Are you trying to access, display, plot, or play past the end of the file / frame?

For example, if an audio file is 10,000 samples long, make sure that the index is not greater than this maximum value. If the value is > than the length of your file, use an **if** statement to catch the problem.