ECE 417 MP7 Walkthrough - Shot-Boundary Detection

Dan Soberal (ECE 417 TA)

Spring 2014

Introduction

1. We have now dealt with a number of machine learning and multimedia signal processing techniques. We've done nonparametric and parametric models, various classifiers, and image retrieval techniques.
2. In terms of techniques and concepts, there is not a great deal of material that is new for this machine problem. It is more of a natural extension of what you already know about features in general.
3. The goal of this machine problem is to implement a shot-boundary detector. In other words, given a vide, we want to detect when there is a shot-change (in other words, a change in the scene).
4. Most of the existing methods detect shot boundaries by employing some kind of distance measure and by measuring the frame-to-frame content change. A predefined threshold for the value of these distances is mostly used to detect shot boundaries.

Overview

1. Two videos are given on the webpage. In both we give you the overall video file, the image files, and the audio file. You will only really need the image files and audio files, the video is just for reference. In other words, if you write your code in such a way that it specifically depends on where these images/audio are in your directory, then don't include the videos themselves when you submit your deliverables.
2. Watch the video files, and look through the given image frames to determine where the true shot boundaries are.
3. Write code to evaluate a series of image and audio features for each frame (combined features will be the concatenation of these two matrices).
4. Compute inter-frame distance for each frame's feature vector
5. Judge whether or not a scene change has occured by seeing if inter-frame distance is above a certain threshold. Where's the threshold, you ask? You decide.
6. Evaluate the detected frames against your true-shot boundaries you have recorded.

Taking a step back, and thinking about things...

1. Video 1 should have extremely high accuracy for boundary detection using image features (use it as a reference for your algorithm). Video 2 is more complicated as far as images are concerned.
2. Common sense: is it easier to detect shot changes with images or audio samples? Under what conditions? (HINT: ever watched the news?, OTHER HINT: how do you transition between frames?)
3. These are the question you should think about while designing your algorithm (for the code) and while interpreting your results (for your report).
4. < Videos watched at this point in the lecture >

Evaluating the True Transitions

1. Sometimes it's easy to evaluate a where they are, and sometimes it's not... (see previous slide)
2. It's up to **you** as the designer to figure out where the acceptable ranges are for ambiguous transitions and keep these decisions in mind when deciding whether or not the transition you recognized is correct.

Audio Processing (Cepstrum)

1. Back to Cepstral features!
2. To avoid what happened in MP3, use these parameters:
   - 2.1 Offset your cepstral feature results so that they don't contain the DC ($c[1]$) component. In other words, when saving all your coefficients for each frame take results[2:(NumComponents+1)] instead of results[1:(NumComponents)]
   - 2.2 Hamming Window of length L (you will have to design L)
   - 2.3 DO NOT RESIZE YOUR SIGNAL
   - 2.4 50% overlap
   - 2.5 Incorrect implementation will result in loss of points
   - 2.6 See MP3 walkthrough for the main part of cepstrum code (if yours is still wrong somehow).
3. You will need to determine the window size based on the number of frames to make the audio frames align with the image frames. How do you do this? Well, there is a deterministic equation for determining the number of frames: $N = 1 + \text{floor}\left[\frac{T-W}{W-\text{ceil}[P \times W]}\right]$. Algebraically solve this equation for W, given that T is the signal length, W is the window size, P is the overlap such that $P \in (0, 1)$.

Audio Processing (New Features)

1. We'll introduce a couple of new features into our feature vector: Energy and Zero-Crossing Rate (ZCR).
2. Energy follows the usual definition from Signal Processing theory. However, it's better to normalize the energy by dividing by N, though, so let's do that: $E_n = \frac{1}{N} \sum_{k=1}^{N} |x[k]|^2$
3. The zero-crossing rate is literally exactly what it sounds like: count the number of time the signal cross the x-axis. Like the energy, we'll normalize it for our implementation: $ZCR_n = \frac{1}{N} \sum_{k=1}^{N-1} |\text{sign}\left(x_{k+1}\right) - \text{sign}\left(x_k\right)|$
4. Both of these can be written in 1 line each in Matlab.

Audio Processing (Putting it all together)

1. For each frame, compute the 12 cesptral coefficients (again, excluding the DC component), the zero-crossing rate, and the energy of the signal. Store them all in a vector.
2. Thus, for each frame $1 \leq n \leq N_{frames}$, make a vector $\mathbf{a}_n$ such that $\mathbf{a}_n = \begin{bmatrix} E_n & ZCR_n & c[2] & \dots & c[13] \end{bmatrix}^T$.
3. Store all such feature vectors into a matrix that $\mathbf{A}$ that is $\dim(\mathbf{A}) = 14 \times N_{frames}$, or in more general terms, $\dim(\mathbf{A}) = (N_{cepstrum} + 2) \times N_{frames}$.
4. Now you can see how this is a relatively easy modification of your current cepstrum function (ie, add two lines of code, and maybe change one existing line, and then you are done).

Image Processing (Raw Vectors)

1. To do the image processing, we'll make use color, but only 2 components of it. We'll use normalized R and G values (which should give us a feature representation that is robust to changes in illumination).
2. Read the images, and convert the images to doubles (and normalize by dividing by 255) for processing
3. Separate the image into 4 quadrants: $I[1 : N/2, 1 : N/2]$, $I[(N/2+1) : N, (N/2+1) : N]$, $I[(N/2+1) : N, 1 : N/2]$, $I[1 : N/2, (N/2+1) : N]$
4. Compute the normalized R and G components components of each quadrant: $\hat{\mathbf{R}}_{i,n} = \frac{\mathbf{R}_{i,n}}{\mathbf{R}_{i,n}+\mathbf{G}_{i,n}+\mathbf{B}_{i,n}}$ and $\hat{\mathbf{G}}_{n,i} = \frac{\mathbf{G}_{i,n}}{\mathbf{R}_{i,n}+\mathbf{G}_{i,n}+\mathbf{B}_{i,n}}$ for $1 \leq i \leq 4$ and $1 \leq n \leq N_{frames}$.
5. HINT: at this point we no longer care about the image indices themselves, so feel free to unroll each matrix $\hat{\mathbf{R}}_{i,n}$ into a vector $\hat{\mathbf{r}}_{i,n}$ and each matrix $\hat{\mathbf{G}}_{i,n}$ into a vector $\hat{\mathbf{g}}_{i,n}$. In fact, this will make it much easier to compute this histogram.

Image Processing (Histogram Implementation)

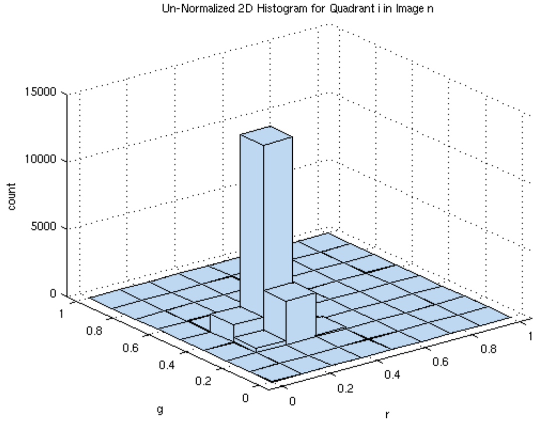
1. Next, we create a 2D histogram $h(\hat{\mathbf{g}}_{i,n}, \hat{\mathbf{r}}_{i,n})$ for each frame n and each quadrant i.
2. We want the 2D histogram to have 8 bins for r and 8 bins for g for a total of 64 bins
3. Compute the 2D histogram in one line (per each n and i) in matlab using the hist3() function. My call was essentially:

   ```
   h = hist3([r_in g_in],'Edges',edges);
   ```

   where edges is a 2×1 cell containing the edges we want to evaluate the histogram over (remember that we want 8 bins per axis, and at this point the vectors should only have values between 0 and 1). Be sure to read the documentation for this function to fully understand it before implementing it
4. Once you do this for each of the 4 quadrants, you will have 4 vectors that are 64×1. Concatenate them together to create a 256×1 vector for that frame.
5. Finally, create a matrix of all the feature vectors $\mathbf{G}$ such that $\dim(\mathbf{G}) = 256 \times N_{frames}$.

Image Processing (Histogram Example)

The histograms should look something like this.



For actually getting the features, turn the histogram into a column vector (again, by unrolling the matrix) and divide by the number of elements (in other words, the length of *r* and the length of *g*).

Image Processing Hints/ Warnings

1. I'm sure some of you will be tempted to read every image and store all the matrices. This is a very bad idea and will result in memory inefficiency. It would be much smarter to process the images one at a time and only store the matrix of features. This will allow you store all your data in a space that is $256 \times N_{frames}$ instead of a space that is $D_1 \times D_2 \times N_{frames}$
2. Coding-wise, think about coding this as a for loop over the frames, and then a for loop for each quadrant. Obviously it is neither wise nor neccessary to iterate through the image indices

Fusion

There's not much to fusion. Literally just concatenate your audio features matrix and your image feature matrix.

$$\mathbf{F} = \begin{bmatrix} \mathbf{A} \\ \mathbf{G} \end{bmatrix} \tag{1}$$

$$\dim(\mathbf{F}) = (N_{cepstrum} + 2 + 256) \times N_{frames} \tag{2}$$

Inter-Frame Distances

Let the feature matrix you are currently working with (either **A**, **G**, or **F**) be denoted as **X**. You can compute the interframe distance as:

$$d_{n,n-1} = \|\mathbf{x}_n - \mathbf{x}_{n-1}\|_2, \quad d_{0,-1} = 0, \quad \mathbf{x}_n \in \mathbf{X}, \quad d_{n,n-1} \in \mathbf{d} \tag{3}$$

$$\mathbf{d} = \left[0 \quad \sum_{rows} \left(|\mathbf{X}_{2:N} - \mathbf{X}_{1:(N-1)}| \odot |\mathbf{X}_{2:N} - \mathbf{X}_{1:(N-1)}|\right)\right]^T \tag{4}$$

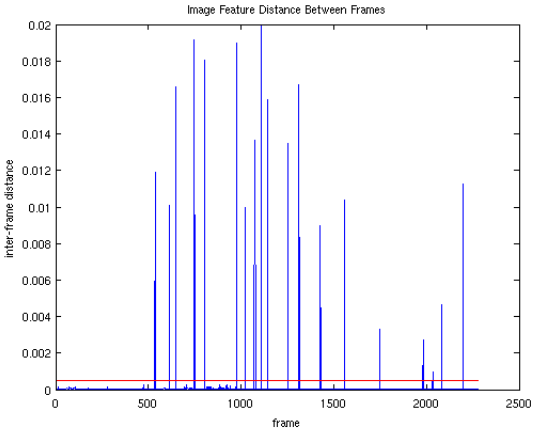
In this case, the circle-dot operator refers to element-by-element multiplication (.* in Matlab). You can compute this distance for ALL n in exactly 1 line of code in Matlab. It is inadvisable to loop over the frames and compute this norm iteratively.

Shot Boundary Threshold Determination

1. This is really up to you.
2. One way to figure out a good metric to use for the thresholds during the "fine-tuning" phase is to use the ginput() function in Matlab. Essentially, this will allow you to get the (x,y) coordinates of where you click your mouse. This in turn allows you to set boundaries by mouse-clicks by setting boundary = y. For the final implementation that you turn in, though, make sure you have established where the boundaries should be (it should run on it's own with nothing required on my part).
3. You can also create rules for determining thresholds (in my case, I used a linear combination of two statistics of the distance signal over all frames). Feel free to come up with rules similar to that, or completely different than that (again, up to you). The only requirement for setting the threshold is that it makes sense given signal processing intuition or experimental data.
4. Whatever threshold you use, be sure to include the threshold itself (or the rule itself) and your process for coming up with said rule.

Shot Boundary and Inter-frame Distance Example

The plot below is an example of the inter-frame distance signal, with a threshold line drawn in.



Determining Accuracy

1. You can determine accuracy by counting the frames returned and seeing which were true-positives, which were missed detections, and which were false positives. The number of true positives divided by the total number of true transitions is the accuracy. The way I did it was the following:
   1.1 Make Matlab calculate this for you by listing the number of true transitions in a matrix that is $N \times 2$ where N is the number of transitions.
   1.2 The first column is the start point of the transition, the second column is the end point
   1.3 If the error between the transition index returned by your algorithm and the row-wise mean of your matrix is below some threshold (generally M/2 for the transition in question where M is the width of the transition) the return a true positive, otherwise return nothing.
   1.4 The number of returned positive hits divided by the total number of true transitions is the accuracy

Some Checkpoints You Should Be Able to Meet

1. Because of the structure of the first video, it should be easy to get 100% accuracy for the image-based boundary detection.
2. Likewise, the second video will be more difficult for your algorithm to parse. You should not expect 100% accuracy. Something greater than 75% would be acceptable
3. I will leave the other results to your imagination. Design your algorithm within the bounds given thus far to acheive maximal results.

Deliverables

1. The report
   1.1 The report should cover your design implementation. Specifically, how did you design your thresholding rules, how did you structure your algorithm, what frames did you chose as the ”truth” for your transition boundaries, how did you write an algorithm to determine accuracy, etc.
   1.2 In addition, it should include an intelligent discussion of the results themselves (accuracy, etc) as well as your interpretation of the results (which of the three methods is best, why, and under what conditions).
   1.3 Your report should be submitted as a pdf
2. Your Code - You should have all the neccessary functions and a single main wrapper program that runs everything. Your code should work ”out of the box”, so to speak, with zero intervention required on my part.
3. Your Readme.txt file - Your readme file should list any implementation details that are relevant to running your code. If I can’t figure out exactly how to run your code and reproduce your results from your readme file, then you will lose points.

Coding Guidelines

1. The first way you think of to accomplish a task in code may or may not (and probably isn't) the best way.
2. For each task, try to think about how to program it in such a way that you minimize lines of code (function calls for repeated code), maximize accuracy (have your code actually perform the correct operations), and minimize overhead time (program in a way that is optimal for the paradigm of the programming language in question).
3. Remember that Matlab works best on vectorized operations, so vectorize every operation you possibly can (minimize calculations iterated through for loops unless the net operation is cheap and fast). Most of the operations for this MP can be done in one or two lines.
4. If your code takes more than 3 minutes to execute, then you **must** re-think your approach. Programs with excessive run-times will lose points.
5. Don't hard code things if you can avoid it
6. Use functions to break up your code, it will make your code more readable and is ultimately easier to use from an implementation perspective

Suggested Functions

Remember that built-in Matlab functions that trivialize the problem at hand or functions that are otherwise not written by you are banned. Here are the functions that I needed to do this problem.

1. cell, ones, zeros, dir, pwd
2. numel, size, length, ceil, floor
3. imread, double, audioread
4. fft, ifft, log, abs, hamming
5. sum, sign, hist3

Some other functions that might be useful are:

1. cellfun, bsxfun, struct, norm, mean, max, median, min, mode, var, ginput

In other words, you don't really need to much outside of the usual basic functions to do this machine problem.