Video Shot Segmentation for Implementation in OPENi NLM Search Engine

Overview:

The National Library of Medicine (NLM) owns a large collection of medical publications, much of which has been digitalized and uploaded for public access. This digital library employs an open access biomedical image search engine, OPENi, which is able to rank publications in order of relevance of matching strings, image type, subset, collection, license type, specialties, or in text citations.

A facet of OPENi that is not yet well-established is the incorporation of video information from video clips often found in digital medical publications into said queries.

This project aims to build a collection of video segmentation algorithm that would optimize the user query experience by efficiently extracting the synopsis, or "key information" of video files found in these publications and displaying them in a user-friendly and easily-recognized manner. These video segmentation algorithms are derived from subsets of image processing and audio processing algorithms that detect pivotal changes in the content of the video clips. The corresponding video frames in these points of pivotal content change are then extracted and displayed with the search results so that a user may gauge for himself if a query is relevant.

Successful implementation of this project into the OPENi search engine would improve the ease-of-use of said search engine, and yield results that are more relevant to the user.

Background:

The information in a video clip is usually stored as distinct video and audio streams, each of which is assigned an integer index. In most instances, there is one video stream and one audio stream. Under these circumstances, the video stream index is 0 and the audio stream index is 1. However, this information can all be checked with the VideoInfo() class.

The video stream is compressed into a collection of still frames by an encoded algorithm such that less memory is needed. There are three types of frames used in video compression: I-frame, P-frame, and B-frame. Frames I-frames are the least compressible but don't require other video frames to decode. P-frames can use data from previous frames to decompress and are more compressible than I-frames. B-frames can use both previous and forward frames for data reference to get the highest amount of data compression. Most video compression algorithms use the sequence, I-P-B-I…., to encode the video information. For the purposes of this project, only I-frames are ever used as they are entirely independent from the information of other frames and are essentially stand-alone image files. For the purposes of this project and much of image processing in the context of video manipulation in general, I-frames are the point of focus.

The library, Xuggle, which will be introduced in the next section, is very compatible with the purposes of this project as it has specialized classes with a naming scheme starting with "I" (IContainer, IStream, etc.) that only work with I-frames.

Project Specifications:

This project was coded entirely in JAVA with the well-known IDE, Eclipse. It referenced many libraries designed for video and audio manipulation, including Xuggle, VLCJ (JAVA wrapper for VLC player), APACHE, SLF4J, JNA, EZMORPH, and JACKSON Annotations.

**Note: OpenCV appears to be an obvious library to use when doing image processing on video frames. However, it has many bugs in dealing with stored video files and does not have the capability to do frame-by-frame extraction of images. It also does not store any information about the time element in frames. Furthermore, OpenCV does not have classes that exclusively interact with I-frames such as in the case of Xuggler as mentioned in the previous section. Thus, it is largely inferior to Xuggle for use here.

All programs were run on a computer running the Windows 7 operating system and containing an Intel® Xeon® CPU. It has a clock speed of 2.67 GHz and 12 GB RAM.

Outline of Packages (Beta API):

<u>Audio</u>

Methodology:

This package is designed to extract the key frames in which there is a pivotal change in audio information. This will be accomplished by converting the audio information into a String object and then searching for key phrases in the String object.

Xuggle is employed to strip the audio stream from the video clip in question and save it as a .wav file. This file is then sent as a byte stream via an Http POST request to the online speech-to-text service, "Bing Voice Recognition," available through the Microsoft Azure Marketplace. The service will then send back the information as a String object.

The String object is run through a key phrase detector (programmed by Suchet) and points of pivotal information change will be recorded. The corresponding points in the byte array are marked and thus the key frames are indexed and ready for extraction.

AudioToString() //extracts the audio data from the video file, sends it to Bing Voice Recognition service as a .wav file, reads back String response

1. extractAudio() //extracts the audio stream from the video file and saves it as a .wav file
    a. 'fileAddress' is the address of the video file to be analyzed. 'fileTo' is the destination of where the String information returned from Bing Voice Recognition Service will be stored
    b. Constructs a reader for the file found at 'fileAddress'
    c. Extracts the audio data from the primary audio stream from the file found at 'fileAddress'
    d. Writes the audio data as a .wav file to the address 'fileTo'
    e. Closes the reader (garbage collecting)

2. executePost() //sends the .wav file of audio data to Bing Voice Recognition service as an HTTP POST request and retrieves the translated String
    a. Sets up an HttpURLConnection with the parameters and headers as specified by the Bing Voice Recognition API: https://onedrive.live.com/view.aspx?resid=9A8C02C3B59E575!106&ithint=file%2cdocx&app=Word&authkey=!AIEJaNeh8CcDTjU
    b. Requests an access token held by the class, 'AdmAuthentication'
    c. Sends the .wav file as a byte stream and reads back the response from the input stream of the URL connection and saves it in a StringBuffer

AdmAuthentication() //sends a POST request to Bing Voice Recognition service to retrieve an access token so that a HTTP Connection can be made to send/retrieve files to/from the service

1. Constructor
    a. 'DatamarketAccessUri' is given in the Bing Voice Recognition API
    b. Token is refreshed every ten minutes
    c. 'clientId' and 'clientSecret' are the ID and password set by the user when registering the project – NOT the ID and password to the Microsoft Azure Marketplace account
    d. 'accessTokenRenewer' is a timer object that is called every tenth minute (every nine minute span)
2. RenewAccessToken() //refreshes the access token
    a. Sends a new Http POST request for an access token every time accessTokenRenewer records the tenth minute
    b. Replaces the current token with the new one
3. OnTokenExpiredCallback(Object stateInfo) //in case the connection times out, renew the token and timer object
    a. If the access token is not refreshed in time because 'accessTokenRenewer" and the Bing Voice Recognition internal timer have a gap in communication or some other reason, reset the 'accessTokenRenewer,'
    b. Make a new Http POST request
    c. Attempt to reset the token.
4. HttpPost(String DatamarketUri) //makes the Http POST request to get access tokens from Bing Voice Recognition service/Microsoft Azure Marketplace
    a. Sets the parameters and headers according to the Bing Voice Recognition sample program: https://onedrive.live.com/?cid=09a8c02c3b59e575&id=9a8c02c3b59e575%21115&ithint=file,cs&authkey=!AOE9yiNn9bOskFE
    b. Sends the request through a DataOutputStream to Bing Voice Recognition
    c. Reads the returned token as a String via DataInputStream
    d. Use a buffered reader to extract the token information from the String and use it to make a new AdmAccessToken object

microsoft.hawaii.hawaiiClientLibraryBase.Identites

1. AdmAccessToken() //Class that organizes the different information found in an access token
    a. Distributed by Microsoft, this class is a logical organizational scheme for the information found in a String-based access token. Used to store the access token sent from Microsoft Azure Marketplace Cloud for the Bing Voice Recognition service.

<u>VideoManipulation</u>

Methodology:

There are three algorithms contained in this package for detecting pivotal changes in image information derived from the video frames.

Before the implementation of these algorithms, the 'FrameCollector()' class is used to extract a frame every 0.5 seconds in a specified video clip and save it as a .png image files in a specified folder.

` The first algorithm, KeyFrameFinder_Xuggler_Default(), uses a built-in "seekKeyFrame()" method that is part of the Xuggle library. The algorithm simply runs through every single extracted image file and saves it to another folder if it is determined to be a key frame. The algorithm takes around one minutes to run for a nine minute video.

The other two algorithms, KeyFrameFinder_Xuggler_Method1() and KeyFrameFinder_Xuggler_Method2(), are built around the HSV histogram comparison technique. HSV data is extracted from all frames by utilizing kernels and sliding them through the image. Subsequent HSV data values of respective coordinates in the image are compared, and an overall normalized correlation coefficient is computed for the comparison between two consecutive frames and between a frame and a calculated average of a local cluster of frames, for the first and second algorithms, respectively. If this coefficient exceeds a certain global threshold (may be customized for different types of videos: X-rays, documentaries, cell divisions, etc.), then that frame is taken to be "pivotal" in information change and captured.

FrameCollector() //Extracts a frame every 0.5 seconds elapsed in a specified video clip

1. Constructor
    a. 'frames' is an ArrayList that will hold all the extracted frames as BufferedImage objects
    b. 'mVideoStreamIndex' stores the index number of the video stream.
    c. 'container' is an IContainer type object (part of the Xuggle library) which can load multiple media files (such as a video clip).
    d. The IStream, IStreamCoder, IMediaReader, and other classes with similar naming schemes are all classes that are able to either mount a media file or extract some form of information from a given media file. All of these classes are well documented on the Xuggle API.
    e. 'mLastPtsWrite' holds the integer value of the last frame that was indexed to be extracted.
2. ImageSnapListener() //runs through the entire video clip and takes a snapshot to be saved every 0.5 seconds

   a. The indexed images are first saved in the ArrayList ‘frames’ as BufferedImage objects and then are extracted and written to a separate folder as .png files.

FrameExtractor() //writes the indexed image files to a specified location – DEPRECATED

1. processFrame(IVideoPicture, BufferedImage)
   a. Uses ImageIO class to write the BufferedImage specified by the IVideoPicture object to a specified location
2. Start() //does the same thing as ImageSnapListener but less efficient

KeyFrameFinder_Xuggler_Default() //a key-frame-finding algorithm that utilizes the method, seekKeyFrame(), found in the Xuggle library, to detect and extract video key frames

1. testSeekKeyFrameCheckIndex()
   a. Global.setFFmpegLoggingLevel() sets the level of admin access that FFmpeg, the software on which Xuggle is based, is allowed to access. The corresponding level of control to each integer input is given in the Xuggler API.
   b. Runs through all the collected frames from a video clip in chronological order and checks each frame to see if it is counted as a “key frame.” If it is, the index of the frame is saved in ‘timestamps’
   c. Runs through all the frames again and writes the frames with indices stored in ‘timestamps’ to a specified location.

KeyFrameFinder_Xuggler_Method1() //the first custom algorithm for finding video key frames

1. findKeys() //indexes all of the key frames
   a. Every video frame is divided into 400 blocks in which HSV data is averaged.
   b. ‘hsb1’ and ‘hsb2’ contain arrays for the average hue, saturation, and brightness values for the two frames that are being compared.
   c. Two consecutive frames are compared at once. A kernel-like algorithm is employed to slide through all of the 400 corresponding blocks from the two frames. If the HSV value averages are above a specified threshold (can be optimized for different kinds of video clips, such as X-ray scan vs documentary) then the latter frame is indexed as a keep frame and the index is saved in ‘actualKeys.’
   d. After the comparison between the two frames, ‘hsb2 ‘is set equal to ‘hsb1’ and ‘hsb1’ will take on values of the frame after the previous two frames. In this way, ‘hsb2’ is a trailer variable for ‘hsb1.’
2. exportFrames() //saves the key frames as .png files to a specified address
   a. Runs through the ArrayList, ‘actualKeys,’ and writes the corresponding frame at each of the recorded indices to a specified index.

KeyFrameFinder_Xuggler_Method2() //the second custom algorithm for finding video key frames

1. findKeys() //indexes all of the key frames
   a. Every video frame is divided into 400 blocks in which HSV data is averaged.
   b. ‘frameColors’ is an ArrayList of 2D float arrays which carry the index of the current block of inspection (out of 400 possible positions, in which the top left block is index = 0, and indices increase toward the right and then downwards such that and the top right block

is index = 19 and the bottom right block is index = 399) and the average HSV of the current block.

   c. ‘avgColors’ holds parallel indices as ‘frameColors’ but holds the average HSV values of the fifteen frames before and after the current frame of inspection. If there do not exist fifteen frames before or after said frame, then it takes the average of as many as possible.
   d. Each frame is compared to its surrounding 30 frames. A kernel-like algorithm is employed to slide through all of the 400 corresponding blocks from ‘frameColors’ and ‘avgColors’. If the HSV value averages are above a specified threshold for any given block (can be optimized for different kinds of video clips, such as X-ray scan vs documentary), the counter variable ‘hits’ is incremented.
   e. If ‘hits’ for a specified frame is greater than a specified threshold (can be optimized for different kinds of videos) then the frame is recorded as a key frame and its index is stored into the ArrayList ‘actualKeys.’
2. exportFrames() //saves the key frames as .png files to a specified address
   a. Runs through the ArrayList, ‘actualKeys,’ and writes the corresponding frame at each of the recorded indices to a specified index.

Runner() //used to run any of the classes in the class VideoManipulation

1. When ran, two applet windows will pop up in sequence. The user must first choose the location of the VLC library folder and then the location of the video clip to be analyzed. The VLC library address must be either hard-coded or found manually because there is no .jar file for this library.

VideoInfo //Gives the user comprehensive information regarding the video clip specifications

1. Using commands found in the Xuggler library, the class is able to give the user information regarding the video
   a. Number of information streams
   b. Duration (milliseconds)
   c. File size (bytes)
   d. Bit Rate
   e. Index of individual streams
   f. Video format
   g. Dimensions of frames
   h. Lag time (at the start of video)
   i. Timebase

VLCPlayer //Custom video player that can be used to capture individual frames in a video manually

1. JAVA built-in AWT and SWING classes are used to construct an applet in which a video can be played through the VLC player library.
2. Three JButtons allow the user to rewind 10 seconds, fast forward ten seconds, or pause the video.
3. ‘Pause’ button at the present will also capture the closest I-frame and save it as a .png file in a specified location.

Strings

CombinedSearch – deprecated

Search() //searches abstracts to see if they contain specified key words

1. Uses a scanner object to read a String input (such as an abstract to a research paper)
2. Checks each word in the String input to see if it is found in the ArrayList 'keywords'
3. Increments the counter variable 'hits' each time there is a keyword found
4. Returns 'hits'

Malfunctions:

- KeyFrameFinder_Xuggler_Method2() will only export the beginning or ending key frames found in a video clip. This might be a problem in memory. Upon adjusting the memory allowed for JVM to 16GB from 2GB, a previous problem with the storage of HSV values was solved. This may be a continuation of that problem.
- The HTTP Post request sent to access Bing Voice Recognition returns the error message: "java.io.IOException: Server returned HTTP response code: 400 for URL: https://datamarket.accesscontrol.windows.net/v2/OAuth2-13" which according to stackoverflow users, is most often caused by an error in input of the clientId or clientSecret variables. However, this was not the case here. It is most likely a corruption or mismatch of data types when writing to the website and reading back from it.

Future Development:

The next step to this project would be to consolidate the three algorithms in the VideoManipulation package. All algorithms export under 160 frames for a nine minute video consisting of roughly 18,000 frames. This means that all algorithms already successfully isolate the most important 1% of information. However, for this project to be practical in the long run and be applied to OPENi, the goal to be able to secure around 10 to 20 frames that can accurately tell the storyboard a video. This is a reasonable estimate for the threshold at which the produced video storyboard will be user-friendly.

After the issues with Bing Voice Recognition are resolved, the two different groups of key frames found from video information analysis and audio information analysis also need to be consolidated. This should further reduce the true number of "pivotal" frames.